Device-driver toolkit

A toolkit to write better device drivers, faster.

This book aims to guide you to write your own device drivers using the device-driver toolkit. It is not a replacement of the docs though. The documentation describes all the small details while this book is concerned with more big-picture concepts and the description of the DSL and manifest (JSON, YAML and TOML) inputs.

> **Note**
>
> Definitions are important!
> The name `device-driver` consists of two parts:
>
> - `driver` : Code to enable the use of hardware.
> - `device` : A chip or peripheral you can talk to over a bus.
>
> Examples of good targets for using this toolkit:
>
> - An I2C accelerometer
> - A SPI radio transceiver
> - A screen/display with parallel bus
>
> The driver is usable in any no-std context and can be made to work with the `embedded-hal` crate or any custom interfaces.

(In theory this toolkit can be used for memory-mapped peripherals too, but there are likely better crates to use for that like `svd2rust` and `chiptool`. The major difference is that this toolkit assumes device interfaces to be fallible.)

▸ Sneak peak of yaml register definition

```
SYNT:
  type: register
  address: 0x05
  size_bits: 32
  reset_value: 0x42162762
  fields:
    PLL_CP_ISEL:
      base: uint
      start: 29
      end: 32
      description: Set the charge pump current according to the XTAL
frequency (see Table 37. Table 34).
    BS:
      base: bool
      start: 28
      description: |
        Synthesizer band select. This parameter selects the out-of loop
        divide factor of the synthesizer:
        - false: 4, band select factor for high band
        - true: 8, band select factor for middle band
        (see Section 5.3.1 RF channel frequency settings).
    SYNT:
      base: uint
      start: 0
      end: 28
      description: The PLL programmable divider (see Section 5.3.1 RF channel
frequency settings).
```

Book overview:

- The intro chapter describes the goal of the toolkit, what it does for you and why you may want to use it instead of writing the driver manually.
- After the intro are chapters about how to generate and then import the driver code into your project. This can be done during compilation through a proc-macro or ahead of time with the CLI and `include!`.
- Next is a chapter about creating a driver interface where you'll see how to implement the right traits so the generated driver can talk with your device.
- Then the actual definition of the driver is covered. These chapters teach what options there are for defining registers, commands, buffers and more using either the DSL or a manifest language like YAML.

The addendum contains more things that mostly provide useful background information.

> **Caution**
>
> It's hard to keep book like this up-to-date with reality. Small errors might creep in despite my best effort.
> If you do find something out of place, missing or simply wrong, please open an issue,

> even if it's just for a typo! I'd really appreciate it and helps out everyone.

Known drivers using the toolkit:

It's nice to have examples:

- S2-LP radio
- Nordic nPM1300 Power Management IC
- iqs323 inductive/capacitive sensing controller
- VCNL36825T proximity sensor
- AXP192 Power Management IC
- ONSEMI FUSB302B USB-PD PHY
- iC-Haus iC-MD 48bit quadrature counter

Feel free to add to this list!

Intro

> **Important**
>
> We deserve better drivers. Rust has shown that we don't need to stick to old principles and that we as an industry can do better.

Device-driver is a Rust toolkit that generates safe, documented interfaces for hardware devices, handling bit-packed registers and device commands through an expressive macro DSL or config file.

While the Rust language provides many opportunities to improve the way we write drivers, it doesn't mean those are easy to use. There are two issues:

1. Creating good datastructures to represent the driver is hard
2. Writing the definitions by hand takes a lot of thankless work

By using this toolkit, you get both 1 and 2 solved.

Number one is solved by getting the datastructures as part of this toolkit which has seen over 5 years of iteration and improvements. The second issue is solved by using code generation so you only need to manually take care of the things that make your driver unique.

Together, this delivers a really tight and precise way of authoring your device driver:

```
device_driver::create_device!(
    device_name: MyDevice,
    dsl: {
        config {
            type RegisterAddressType = u8;
        }
        /// This is the Foo register
        register Foo {
            const ADDRESS = 0;
            const SIZE_BITS = 8;

            /// This is a bool at bit 0!
            value0: bool = 0,
            /// Integrated enum generation
            value1: int as enum GeneratedEnum {
              A,
              /// Variant B
              B,
              C = default,
            } = 1..4,
            /// This is a 4-bit integer
            value2: uint = 4..8,
        },
    }
);

let mut device = MyDevice::new(device_interface);
device.foo().write(|reg| reg.set_value_1(GeneratedEnum::B)).unwrap();
```

Instantly we get a nice and familiar API that is well documented. There's a bunch more features to discover like using YAML as the input and a bunch of analysis steps, so read on!

The goal

When you're writing a driver, you just want to implement it and be done with it. Most of the time developing a driver is boring and repetitive.

To help you do less of the boring work and to create a higher quality driver at the same time, the goals are:

1. Get a great driver for minimal effort
2. Get a driver that is correct and hard to misuse
    - (assuming the input spec is correct)
3. Get a driver that is well documented
    - (assuming the input spec gives docs)

These goals are met by:

- Using a dense and precise input language
- Having many options do deal with byte and bit ordering
- Having analysis steps to decrease the chance of common mistakes
- Separating the interface to the device from the definitions
- Allowing you to put docs on pretty much anything

How to continue

Simply read the rest of the book!

Looking at existing drivers and examples can also be very helpful.

Using the macro

The macro is the main way of generating a driver. It is defined in the `device-driver-macros` crate which is re-exported in the `device-driver` crate by default. You don't have import the macros crate yourself.

The macro can be used in two forms.

Inline DSL

The first form is for writing the register definitions using the DSL right in the source of your project.

```
device_driver::create_device!(
    device_name: MyTestDevice,
    dsl: {
        // DSL code goes here
    }
)
```

It consists of two parts:

- `device_name` : This will be the name of the root block that will take ownership of the device interface.
    - The name must be provided in PascalCase
    - If you're going to distribute this as the main part of your driver, then it's recommended to use the name of the chip this driver is for. For example: 'Lis3dh'
    - If you're going to write a higher level wrapper around it, then it's recommended to name it something appropriate for a low level layer. For example: 'Registers' or 'LowLevel'
- `dsl` : This selects the option to write DSL code in the macro

Using the DSL in this way allows for nice error messages and keeps the definitions close to your code.

Manifest file

The second form uses an external manifest file.

```
device_driver::create_device!(
    device_name: MyTestDevice,
    manifest: "driver-manifest.yaml"
)
```

You can provide an absolute path or a relative path to the file. If it's relative, then the base path is the value of the `CARGO_MANIFEST_DIR` environment variable. This is the same directory as your `Cargo.toml` is in.

The extension of the file determines which parser is used.

The options are:

- yaml
- json
- toml
- dsl

Output

> **Tip**
>
> The generated code is placed exactly where the macro is invoked. This means you can decide to contain everything in its own module. This is recommended to do, but not required.

> **Caution**
>
> Code in the same module as the generated code is able to access the private API of the generated code. It is discouraged to make use of the private API since it's not considered as part of the SemVer guarantees and it's designed in a way where you shouldn't need to.

> **Note**
>
> If you feel part of the private API should be stabilized, then please open an issue to discuss it. If you really need to access the private API, consider pinning the exact device-driver versions and make sure to pin the sub crates too, including the generation and the macros crate.

Optimizing compile times

The device-driver crate has features for turning on the json, yaml and toml parsers. These are enabled by default for your convenience.

Once you've settled on a format, you can optimize the compile times for you and your dependents by disabling the default features and adding back the features you need.

Suggestions:

- When using the DSL (inline or as manifest)
    - `default-features = false`
    - `features = ["dsl"]`
- When using yaml
    - `default-features = false`
    - `features = ["yaml"]`
- When using json
    - `default-features = false`
    - `features = ["json"]`
- When using toml
    - `default-features = false`
    - `features = ["toml"]`

> **Tip**
>
> With these steps the compile times should be acceptable. However, they can be further optimized by getting rid of the macro altogether. This is explained in the cli chapter.

Using the cli

The cli is there to optimize compile times for your driver users. Instead of having to compile the device-driver macros and run them, you can generate the code ahead of time and then `include!` or make a module out of it.

> **Tip**
>
> During development using the proc macro will be lots easier since the code generation won't go out of sync with the driver definitions. Then once the development is done, you may want to use the CLI as an optimization.

Installation

The cli can be installed using cargo:

```
cargo install device-driver-cli
```

This always supports all input formats.

Usage

The CLI is written with clap and has a minimal and simple interface.

To see all options, use:

```
device-driver-cli --help
```

To do the code generation three things are required:

- `-m` or `--manifest` : The path to the manifest file
- `-o` or `--output` : The path to the to be generated rust file
- `-d` or `--device-name` : The name the toolkit will use for the generated device. This must be specified in PascalCase

Using the output

Exactly how you include the generated rust file is up to you. You could generate it into

your `/src` folder and declare it a module, which would be nice for Rust analyzer but forces the generated code to be its own module. Or to include it in an existing module you can use the `include!` macro.

However you choose to include it, don't forget to track the file in your git repo.

The generated code still depends on the device-driver crate, but since we don't depend on the proc macro anymore we can turn off the default features. So in your `Cargo.toml` you can now import the toolkit as:

```
device-driver = { version = <VERSION>, default-features = false }
```

This makes it so all unused dependencies are gone.

Writing an interface

> **Important**
>
> The device-driver crate and the generated code don't know anything about how to talk to your device. This means we need to teach it about the interface it has!

Let's first create our device:

```
device_driver::create_device!(
    device_name: MyDevice,
    dsl: {
        // ...
    }
);
```

This generates a top-level block `MyDevice` which has a `new` function that takes ownership of an interface. We have to create our own interface type that we can pass into it. This type will implement the logic to communicate with the device.

In this example, let's assume a register 'foo' was defined and see what happens:

```
/// Our interface struct that owns the bus.
pub struct MyDeviceInterface<BUS> {
    pub bus: BUS,
}

fn try_out() {
    // Initialize the bus somehow. Your HAL should help you there
    let bus = init_bus();
    // Create our custom interface struct
    let interface = MyDeviceInterface { bus };

    // Create the device driver based on the interface
    let mut my_device = MyDevice::new(interface);

    // Try to read the foo register. This results in an error
    let _ = my_device.foo().read();
    // ERROR:               ^^^^ method cannot be called due to unsatisfied
trait bounds
    //
    // note: the following trait bounds were not satisfied:
    //       `DeviceInterface: RegisterInterface`
}
```

This example doesn't compile and outputs an error. Luckily the compiler tells us what's wrong. The problem is that we provided a device interface that doesn't provide a way to read or write registers, but we ask the driver to read a register.

The error tells us the device interface should implement the `RegisterInterface` trait.

> **Important**
>
> Every kind of operation has its own trait. Find the up-to-date docs of them on docs.rs.
>
> There's an interface for register, command and buffer.

Of each of the traits there is an async version too. When implemented the async versions of the operations can be used. They've got the same name as the normal operations, except they end with `_async`. The register `.read()` then becomes `.read_async()`.

Let's make our example complete by implementing the `RegisterInterface`:

```
pub struct MyDeviceI2cInterface<BUS> {
    pub bus: BUS,
}

// See the docs of the traits to get more up-to-date information about how
and what to impl
impl<BUS: embedded_hal::i2c::I2C> device_driver::RegisterInterface for
MyDeviceI2cInterface<BUS> {
    // ...
}

// For the async I2C we can implement the async register interface
impl<BUS: embedded_hal_async::i2c::I2C> device_driver::AsyncRegisterInterface
for MyDeviceI2cInterface<BUS> {
    // ...
}

fn try_out_sync() {
    let bus = init_sync_bus(); // Implements the I2c trait
    let interface = MyDeviceI2cInterface { bus };

    let mut my_device = MyDevice::new(interface);

    let _ = my_device.foo().read();
}

async fn try_out_async() {
    let bus = init_async_bus(); // Implements the async I2c trait
    let interface = MyDeviceI2cInterface { bus };

    let mut my_device = MyDevice::new(interface);

    let _ = my_device.foo().read_async().await;
}
```

We've now covered how to create an interface type and implement the interface trait you need on it.

Some chips can have multiple interfaces, like both SPI and I2C or SPI and QSPI. You can choose to support them in one type or make separate types for them.

> **Tip**
>
> You can make your interface type(s) as complex or as simple as you need. It depends on your chip and your requirements what it should look like. It is good practice, though, to inform the driver users of this with docs and examples.

Defining the device

This toolkit brings three different kinds of concepts you can use to build various aspects of your driver.

- The register is some memory located at an address on the device. It contains fields, may have a reset value and could be restrictive in its read and write access.
- The command can model multiple things. It can be an event to send to the device so it changes state or it could be an RPC-like call. It can send data and receive back an answer.
- The buffer is anything that you'd like to have a `Write` or `Read` interface to. A good example is a fifo buffer in a radio chip.

The registers, commands and buffers can be grouped into blocks.

Except for buffers all of them can be repeated and ref'ed. Repeats take the same object and repeat them for a repeat count with an address stride. A 'ref' object copies another object and allows to override some values like the address and access.

The registers, commands, buffers, blocks and refs are all called 'objects' in this project.

To configure the driver, there's the global config. In it you can define the address types, various defaults for e.g. byte ordering and the method used for name normalization.

These concepts and how you can use them in your driver are described in more detail in their own chapter.

Global config

The global config exists to house three kinds of configs:

1. Required
2. Defaults
3. Transformations

> **Note**
>
> A driver can only have one global config.

Below is a short overview for the DSL format and the manifest format of the global config and their defaults (or `_` for no default). The last chapters describe the options in more detail.

> **Tip**
>
> Use the available default values to your advantage to cut back having to specify things on each individual register, command or buffer.

DSL

```
config {
    type DefaultRegisterAccess = RW;
    type DefaultFieldAccess = RW;
    type DefaultBufferAccess = RW;
    type DefaultByteOrder = _;
    type DefaultBitOrder = LSB0;
    type RegisterAddressType = _;
    type CommandAddressType = _;
    type BufferAddressType = _;
    type NameWordBoundaries = [
        Underscore, Hyphen, Space, LowerUpper,
        UpperDigit, DigitUpper, DigitLower,
        LowerDigit, Acronym,
    ];
    type DefmtFeature = "my-feature";
}
```

Manifest

> **Note**
>
> Example is written in json, but works for yaml and toml too when literally translated.

```
"config": {
    "default_register_access": "RW",
    "default_field_access": "RW",
    "default_buffer_access": "RW",
    "default_byte_order": "_",
    "default_bit_order": "LSB0",
    "register_address_type": "_",
    "command_address_type": "_",
    "buffer_address_type": "_",
    "name_word_boundaries": [
        "Underscore", "Hyphen", "Space", "LowerUpper",
        "UpperDigit", "DigitUpper", "DigitLower",
        "LowerDigit", "Acronym"
    ],
    "defmt_feature": "my-feature"
}
```

Required

`register_address_type`

Specifies the integer type used to represent the address of a register. It is required once a register has been defined.

The value is a string in manifest form or an integer type in DLS form.

Options are: `u8`, `u16`, `u32`, `u64`, `i8`, `i16`, `i32`, `i64`

`command_address_type`

Specifies the integer type used to represent the address of a command. It is required once a command has been defined.

The value is a string in manifest form or an integer type in DLS form.

Options are: `u8`, `u16`, `u32`, `u64`, `i8`, `i16`, `i32`, `i64`

`buffer_address_type`

Specifies the integer type used to represent the address of a buffer. It is required once a buffer has been defined.

The value is a string in manifest form or an integer type in DLS form.

Options are: `u8`, `u16`, `u32`, `u64`, `i8`, `i16`, `i32`, `i64`

Defaults

`default_register_access`

Provides a default to the access type of registers. Any register can override this.

The value is a string in manifest form or written 'as is' in the DSL.

Options are: `RW` (default), `ReadWrite`, `RO`, `ReadOnly`, `WO`, `WriteOnly`

`default_field_access`

Provides a default to the access type of fields. Any field can override this.

The value is a string in manifest form or written 'as is' in the DSL.

Options are: `RW` (default), `ReadWrite`, `RO`, `ReadOnly`, `WO`, `WriteOnly`

`default_buffer_access`

Provides a default to the access type of buffers. Any buffer can override this.

The value is a string in manifest form or written 'as is' in the DSL.

Options are: `RW` (default), `ReadWrite`, `RO`, `ReadOnly`, `WO`, `WriteOnly`

`default_byte_order`

Sets the global byte order. This is used for the register and command fieldsets. Any command or register can override it.

The value is a string in manifest form or written 'as is' in the DSL.

Options are: `LE`, `BE`

`default_bit_order`

Sets the global bit order. This is used for the register and command fieldsets. Any command or register can override it.

The value is a string in manifest form or written 'as is' in the DSL.

Options are: `LSB0` (default), `MSB0`

Transformations

`name_word_boundaries`

All object, field, enum and enum variant names are converted to the correct casing for where it's used in the generated code. This is because some of them have dual use like the object names which are used as struct names (PascalCase) and function names (snake_case).

This also aids when copying names from datasheets since they're often weird, inconsistent, wrong or all three in regards to casing.

> **Important**
>
> To do proper casing, it must be known when a new word starts. The transition from one word to the next is called a boundary.

The conversions are done using the `convert_case` crate. With this config option you can specify the boundaries the crate uses to do the conversions.

Options are: `[Boundary]` or `string`

The available boundaries can be found in the docs of the crate. The boundary names should be specified as strings in the manifest and 'as is' in the DSL.

The string is converted to an array of boundaries using this function which is a really easy way to define it.

The default value is also provided by the crate from this function.

`defmt_feature`

When defined the generated code will have defmt implementations on the types gated behind the feature configured with this option. The feature gate looks like: `#[cfg(feature = "<VALUE>")]`. This allows you, the driver author, to optionally include defmt support.

The value is a string in manifest form and also written as a string in the DSL.

Registers

A register is a piece of addressable memory stored on the device.

It is accessed as a function on the block it's part of. The function returns a `RegisterOperation` which can be used to read/write/modify the register.

Example usage:

```
let mut device = MyDevice::new(DeviceInterface::new());

device.foo().write(|reg| reg.set_bar(12345)).unwrap();
assert_eq!(device.foo().read().unwrap().bar(), 12345);
```

Below are minimal and full examples of how registers can be defined. Only one field is shown, but more can be added. Details about the fields can be read in their own chapter.

- Registers
  - DSL
  - Manifest
  - Required
    - `address`
    - `size_bits`
    - `type` (manifest only)
  - Optional
    - `cfg` or `#[cfg(...)]`
    - `description` or `#[doc = ""]`
    - `access`
    - `byte_order`
    - `bit_order`
    - `reset_value`
    - `repeat`
    - `allow_bit_overlap`
    - `allow_address_overlap`
    - `fields` (manifest only)

DSL

Minimal:

```
register Foo {
    const ADDRESS = 3;
    const SIZE_BITS = 16;

    value: uint = 0..16,
}
```

Full:

```
/// Register docs
#[cfg(feature = "bar")]
register Foo {
    type Access = WO;
    type ByteOrder = LE;
    type BitOrder = LSB0;
    const ADDRESS = 3;
    const SIZE_BITS = 16;
    const RESET_VALUE = 0x1234;  // Or [0x34, 0x12]
    const REPEAT = {
        count: 4,
        stride: 2
    };
    const ALLOW_BIT_OVERLAP = false;
    const ALLOW_ADDRESS_OVERLAP = false;

    value: uint = 0..16,
}
```

> **Tip**
>
> `type` or `const`, which one is it?
> It's `type` if it's overriding a global config and `const` if it's not.

Manifest

> **Note**
>
> The biggest differences with the DSL are the additional `type` field to specify which type of object this is and the `fields` field that houses all fields.

Minimal (json):

```
"Foo": {
    "type": "register",
    "address": 3,
    "size_bits": 16,
    "fields": {
        "value": {
            "base": "uint",
            "start": 0,
            "end": 16
        }
    }
}
```

Full (json):

```
"Foo": {
    "type": "register",
    "cfg": "feature = \"foo\"",
    "description": "Register docs",
    "access": "WO",
    "byte_order": "LE",
    "bit_order": "LSB0",
    "address": 3,
    "size_bits": 16,
    "reset_value": 4066, // Or [52, 18] (no hex in json...)
    "repeat": {
        "count": 4,
        "stride": 2
    },
    "allow_bit_overlap": false,
    "allow_address_overlap": false,
    "fields": {
        "value": {
            "base": "uint",
            "start": 0,
            "end": 16
        }
    }
}
```

Required

`address`

The address of the register.

Integer value that must fit in the given address type in the global config and can be negative.

`size_bits`

The size of the register in bits.

Positive integer value. No fields can exceed the size of the register.

`type` (manifest only)

The type of the object.

For registers this field is a string with the contents `"register"`.

Optional

`cfg` or `#[cfg(...)]`

Allows for cfg-gating the register.

In the DSL, the normal Rust syntax is used. Just put the attribute on the register definition. Only one attribute is allowed.

In the manifest it is configured with a string. The string only defines the inner part: `#[cfg(foo)] = "cfg": "foo",`.

> **Warning**
>
> Check the chapter on cfg for more information. The cfg's are not checked by the toolkit and only passed to the generated code and so there are some oddities to be aware of.

`description` or `#[doc = ""]`

The doc comments for the generated code.

For the DSL, use the normal doc attributes or triple slash `///`. Multiple attributes get concatenated with a newline (just like normal Rust does).

For the manifest, this is a string.

The description is added as normal doc comments to the generated code. So it supports markdown and all other features you're used to. The description is used on the generated

register struct and on the function to access the register.

`access`

Overrides the default register access.

Options are: `RW`, `ReadWrite`, `WO`, `WriteOnly`, `RO`, `ReadOnly`.
They are written 'as is' in the DSL and as a string in the manifest.

Anything that is not `ReadWrite` will limit the functions you can call for the registers. `.write` is only available when the register has write access, `.read` only when the register has read access and `.modify` only when the register has full access.

> **Note**
>
> This only affects the capability of a register being read or written. It does not affect the `access` specified on the fields.
>
> This means you can have a register you cannot write, but does have setters for one or more fields.
> That won't be harmful or break things, but might look weird.

`byte_order`

Overrides the default byte order.

Options are: `LE`, `BE`.
They are written 'as is' in the DSL and as a string in the manifest.

When the size of a register is > 8 bits (more than one byte), then either the byte order has to be defined globally as a default or the register needs to define it.

`bit_order`

Overrides the default bit order. If the global config does not define it, it's `LSB0`.

Options are: `LSB0`, `MSB0`.
They are written 'as is' in the DSL and as a string in the manifest.

`reset_value`

Defines the reset or default value of the register.

Can be a number or an array of bytes.

> **Warning**
>
> When specified as an array, this must be formatted as the bytes that are returned by the `RegisterInterface` implementation. This means that when the register has little endian byte order, the reset value number `0x1234` would be encoded as `[0x34, 0x12]` in the array form.
> The same concern is there for the bit order.

It is used in the `.write` function. To reset a register to the default value, it'd look like `.write(|_|())`. When a zero value is desired instead of the default, you can use the `.write_with_zero` function instead.

`repeat`

Repeat the register a number of times at different addresses.

It is specified with two fields:

- Count: unsigned integer, the amount of times the register is repeated
- Stride: signed integer, the amount the address changes per repeat

The calculation is `address = base_address + index * stride`.

When the repeat field is present, the function to do a register operation will have an extra parameter for the index.

`allow_bit_overlap`

Allow field addresses to overlap.

This bool value is false by default.

`allow_address_overlap`

Allow this register to have an address that is equal to another register address. This calculation is also done for any repeat addresses.

Only exact address matches are checked.

This bool value is false by default.

`fields` (manifest only)

The fields of the register.

A map where the keys are the names of the fields. All values must be fields.

Commands

A command is a call to do something. This can be to e.g. change the chip state, do an RPC-like call or to start a radio transmission.

It is accessed as a function on the block it's part of. The function returns a `CommandOperation` which can be used to dispatch the command.

Example usage:

```
let mut device = MyDevice::new(DeviceInterface::new());

device.foo().dispatch().unwrap();
// Commands can carry data too
let result = device.bar().dispatch(|data| data.set_val(1234)).unwrap();
assert_eq!(result.xeno(), true);
```

Below are minimal and full examples of how commands can be defined. Only one field is shown, but more can be added. Details about the fields can be read in their own chapter.

> **Note**
>
> A command can have only input, only output, both input and output, or no fields.
>
> - When input fields are defined, the dispatch function will have a closure parameter to set up the input value.
> - When output fields are defined, the dispatch function returns the data that was read back from the device.

- `in` (dsl) or `fields_in` (manifest)
- `out` (dsl) or `fields_out` (manifest)

DSL

Minimal without fields (with address 5):

```
command Foo = 5,
```

Minimal with in and out fields:

```
command Foo {
    const ADDRESS = 5;
    const SIZE_BITS_IN = 8;
    const SIZE_BITS_OUT = 16;

    in {
        value: uint = 0..8,
    },
    out {
        value: uint = 0..16,
    }
},
```

Full:

```
/// Foo docs
#[cfg(feature = "blah")]
command Foo {
    type ByteOrder = LE;
    type BitOrder = LSB0;
    const ADDRESS = 5;
    const SIZE_BITS_IN = 8;
    const SIZE_BITS_OUT = 16;
    const REPEAT = {
        count: 4,
        stride: 2
    };
    const ALLOW_BIT_OVERLAP = false;
    const ALLOW_ADDRESS_OVERLAP = false;

    in {
        value: uint = 0..8,
    },
    out {
        value: uint = 0..16,
    }
},
```

> **Tip**
>
> `type` or `const`, which one is it?
> It's `type` if it's overriding a global config and `const` if it's not.

Manifest

> **Note**
>
> The biggest difference with the DSL is the additional `type` field to specify which type of object this is and the absence of the super short hand.

Minimal with no fields (json):

```
"Foo": {
    "type": "command",
    "address": 5
}
```

Minimal (json):

```
"Foo": {
    "type": "command",
    "address": 5,
    "size_bits_in": 8,
    "fields_in": {
        "value": {
            "base": "uint",
            "start": 0,
            "end": 8
        }
    },
    "size_bits_out": 16,
    "fields_out": {
        "value": {
            "base": "uint",
            "start": 0,
            "end": 16
        }
    },
}
```

Full (json):

```
"Foo": {
    "type": "command",
    "cfg": "feature = \"blah\"",
    "description": "Foo docs",
    "byte_order": "LE",
    "bit_order": "LSB0",
    "address": 5,
    "repeat": {
        "count": 4,
        "stride": 2
    },
    "allow_bit_overlap": false,
    "allow_address_overlap": false,
    "size_bits_in": 8,
    "fields_in": {
        "value": {
            "base": "uint",
            "start": 0,
            "end": 8
        }
    },
    "size_bits_out": 16,
    "fields_out": {
        "value": {
            "base": "uint",
            "start": 0,
            "end": 16
        }
    },
}
```

Required

`address`

The address of the command.

Integer value that must fit in the given address type in the global config and can be negative.

`size_bits_in` & `size_bits_out`

The size of the command in bits for their respective field sets.

Positive integer value. No fields can exceed the specified size.

Only required when their respective field sets are defined.

`type` (manifest only)

The type of the object.

For commands this field is a string with the contents `"command"`.

Optional

`cfg` or `#[cfg(...)]`

Allows for cfg-gating the command.

In the DSL, the normal Rust syntax is used. Just put the attribute on the command definition. Only one attribute is allowed.

In the manifest it is configured with a string. The string only defines the inner part: `#[cfg(foo)] = "cfg": "foo",`.

> **Warning**
>
> Check the chapter on cfg for more information. The cfg's are not checked by the toolkit and only passed to the generated code and so there are some oddities to be aware of.

`description` or `#[doc = ""]`

The doc comments for the generated code.

For the DSL, use the normal doc attributes or triple slash `///`. Multiple attributes get concatenated with a newline (just like normal Rust does).

For the manifest, this is a string.

The description is added as normal doc comments to the generated code. So it supports markdown and all other features you're used to. The description is used on the generated command input and output structs and on the function to access the command.

`byte_order`

Overrides the default byte order.

Options are: `LE`, `BE`.
They are written 'as is' in the DSL and as a string in the manifest.

When the size of a command input or output is > 8 bits (more than one byte), then either the byte order has to be defined globally as a default or the command needs to define it.

The value is applied to both the input and output fieldsets.

`bit_order`

Overrides the default bit order. If the global config does not define it, it's `LSB0`.

Options are: `LSB0`, `MSB0`.
They are written 'as is' in the DSL and as a string in the manifest.

The value is applied to both the input and output fieldsets.

`repeat`

Repeat the command a number of times at different addresses.

It is specified with two fields:

- Count: unsigned integer, the amount of times the command is repeated
- Stride: signed integer, the amount the address changes per repeat

The calculation is `address = base_address + index * stride`.

When the repeat field is present, the function to do a command operation will have an extra parameter for the index.

`allow_bit_overlap`

Allow field addresses to overlap.

This bool value is false by default.

`allow_address_overlap`

Allow this command to have an address that is equal to another command address. This calculation is also done for any repeat addresses.

Only exact address matches are checked.

This bool value is false by default.

`in` (dsl) or `fields_in` (manifest)

The input fields of the command.

- For the dsl, a list of fields.
- For manifest, a map where the keys are the names of the fields All values must be fields.

`out` (dsl) or `fields_out` (manifest)

The output fields of the command.

- For the dsl, a list of fields.
- For manifest, a map where the keys are the names of the fields All values must be fields.

Field sets

A field set is a collection of fields that make up the data of a register, command input or command output.

Each field set generates to a struct where each of the fields are accessible through functions with the names of the fields.

A field set can be created using the `new` function and will be initialized with the reset value (or zero if there is no reset value). When it's desired to get an all-zero version of the field set, you can call `new_zero`.
When a ref object overrides the reset value, the field set will have an extra constructor `new_as_<ref name>` that will use the reset value override for the initial value.

> **Note**
>
> As a user you should not have to construct your field sets manually in normal use. But it's available to you for special cases in the generated `field_sets` module.

Example usage:

```
use field_sets::MyFieldSet;

let mut reg = MyFieldSet::new();
reg.set_foo(1234);
let foo = reg.foo();
```

Field sets also implement all bitwise operators for easier manipulation. These operations are done on *all* underlying bits, even ones that are not part of a field.

There's also an `Into` and `From` implementation to the smallest byte array that can fit the entire field set.

Example usage:

```
use field_sets::MyFieldSet;

let all_ones = !MyFieldSet::new_zero();
let lowest_byte_set = MyFieldSet::from([0xFF, 0x00]);
let lowest_byte_inverted = all_ones ^ lowest_byte_set;
```

Below are minimal and full examples of how fields can be defined. There are three major variants:

- Base type
- Converted to custom type

- Converted to generated enum

The conversions can be fallible or infallible. When the fallible `try` option is used, reading the field will return a result instead of the type directly. For generated enums, even though they might not be generally infallible when converted from their base type, the toolkit uses extra range information to see if it can safely present an infallible interface regardless.

DSL

Simple (base type only):

```
foo: uint = 0..5,
bar: bool = 5,
zoof: int = 6..=20,
```

With attributes and access specifier:

```
/// Field comment!
#[cfg(blah)]
foo: WO uint = 0..5,
```

With conversion to custom type:

```
foo: uint as crate::MyCustomType = 0..16,
bar: int as try crate::MyCustomType2 = 16..32,
```

With conversion to generated enum:

```
foo: uint as enum GeneratedEnum {
    A,
    B = 5,
    /// Default value
    C = default,
    D = catch_all,
} = 0..8,
```

Manifest

Simple (base type only) (json):

```
{
  "foo": {
    "base": "uint",
    "start": 0,
    "end": 5
  },
  "bar": {
    "base": "bool",
    "start": 5,
  },
  "zoof": {
    "base": "int",
    "start": 6,
    "end": 21
  }
}
```

With attributes and access specifier:

```
{
  "foo": {
    "cfg": "blah",
    "description": "Field comment!",
    "access": "WO",
    "base": "uint",
    "start": 0,
    "end": 5
  }
}
```

With conversion to custom type:

```
{
  "foo": {
    "base": "uint",
    "conversion": "crate::MyCustomType",
    "start": 0,
    "end": 16
  },
  "bar": {
    "base": "int",
    "try_conversion": "crate::MyCustomType2",
    "start": 16,
    "end": 32
  }
}
```

With conversion to generated enum:

```
{
  "foo": {
    "base": "uint",
    "conversion": {
      "name": "GeneratedEnum",
      "A": null,
      "B": 5,
      "C": {
        "description": "Default value",
        "value": "default"
      },
      "D": "catch_all"
    },
    "start": 0,
    "end": 8
  }
}
```

Required

`base`

The base type denotes the primitive type used to convert the bits in the address range to a value.

Options:

- uint - unsigned integer
- int - two's complement signed integer
- bool - low or high, only available for 1 bit values

The integer options will generate to the smallest signed or unsigned Rust integers that can fit the value. So a 10-bit uint will become a `u16`.

The value is specified as a string in the manifest format and is written 'as is' in the DSL.

`start`, `end` & address range

Every field must specified the bitrange it covers. The way this is done differs a bit between the DSL and the manifest but boil down to the same.

The DLS uses `= <ADDRESS>` as the syntax. Valid options for the address are:

- Exclusive range: `0..16`
- Inclusive range: `0..=16`
- Single address: `0`
    - Only in combination with bool base types

The manifest has two fields `start` and `end`, both containing unsigned integers:

- The `start` is the starting bit of the field
- The `end` is the exclusive end bit of the field
    - Not required for bool base types

The address must lie fully within the size of the defining object and no fields may overlap unless the defining object has the `AllowBitOverlap` property set to true.

Optional

`cfg` or `#[cfg(...)]`

Allows for cfg-gating the command.

In the DSL, the normal Rust syntax is used. Just put the attribute on the field definition. Only one attribute is allowed.

In the manifest it is configured with a string. The string only defines the inner part: `#[cfg(foo)] = "cfg": "foo",`.

> **Warning**
>
> Check the chapter on cfg for more information. The cfg's are not checked by the toolkit and only passed to the generated code and so there are some oddities to be

> aware of.

`description` or `#[doc = ""]`

The doc comments for the generated code.

For the DSL, use the normal doc attributes or triple slash `///`. Multiple attributes get concatenated with a newline (just like normal Rust does).

For the manifest, this is a string.

The description is added as normal doc comments to the generated code. So it supports markdown and all other features you're used to. The description is used on the generated field getter and setter.

`access`

Overrides the default field access.

Options are: `RW`, `ReadWrite`, `WO`, `WriteOnly`, `RO`, `ReadOnly`.
They are written 'as is' in the DSL and as a string in the manifest.

If the specified access can do read, a getter is generated with the name of the field. If the specified access do write, a setter is generated with the `set_` prefix followed by the name of the field.

Conversion

If the base type of a field is an integer, the value can be converted to a further higher level type. There are two options for this:

- Conversion to an existing type
- Conversion to an inline defined enum value

The conversion can be specified as infallible or fallible. When infallible, the field getter will call on the `From<INTEGER>` trait to convert the base value to the conversion value after which the value is returned. When fallible, the field getter will use the `TryFrom<INTEGER>` trait instead and will return the result value from it.

In the DSL the conversion is specified using the `as <TARGET>` or `as try <TARGET>` keywords for the infallible and fallible variants respectively.

The manifest has two possible fields `conversion` and `try_conversion` for the infallible

and fallible variants respectively.

To existing type

When a type path is given as the DSL `<TARGET>` or as string in the manifest `conversion` field, the conversion will be done using the specified type.

The type path is used as is in the generated code, so you need to make sure that the type is in scope. Due to how the generated modules are structured, the specified paths get `super::` prepended to them. To be able to still use extern crates and absolute paths this isn't done when the path starts with `::` or `crate`.

Furthermore the type must implement the `From<INTEGER>` or `TryFrom<INTEGER>` traits for the infallible or fallible conversions respectively when the field has read access. When the field has write access, the type must implement the `Into<INTEGER>` trait.

> **Tip**
>
> The existing type can also be a enum generated by the toolkit defined in another place by just using the name of that enum.
>
> This has an added bonus that the toolkit still has the information for accepted input which means it can use the infallible conversion method instead of the `try` fallible one. This creates a nicer and cleaner API.

To generated enum

Instead of a custom type, the toolkit can also generate an enum inline.

In the DSL the format for `<TARGET>` is:

```
enum Foo {
    A,
    B = 5, // Also supports bit and hex specification
    /// Comment
    C
}
```

The enum is written pretty much as a normal Rust enum including setting the value of every variant and writing docs on every variant. In this example, the number value of `C` would be 6.

The generated enum will have the same docs as the field (if any).

In the manifest, the same enum would be specified like so:

```
"conversion": {
  "name": "Foo",
  "description": "Enum docs", // In manifest, enum can be separately
documented
  "A": null,
  "B": 5,
  "C": {
    "description": "Comment",
    "value": null
  }
}
```

The values for each variant can be the following:

- Empty or null
  - Use auto counting starting at 0 for the first variant and one higher than the previous variant
- Signed integer
  - To manually specify the value
- `default`
  - To specify a default value
  - When the conversion is of a number that doesn't match any variant, the default variant will be returned
  - In DSL specified 'as is'
  - In manifest specified as a string
  - Also implements the `Default` trait for the enum
- `catch_all`
  - Similar to default, but makes the variant contain the raw value (like `Catch(u8)`)
  - When the conversion is of a number that doesn't match any variant, the catch all will be returned with the raw value
  - In DSL specified 'as is'
  - In manifest specified as a string

When an enum contains both a catch all and a default, the catch all value is used to return unknown numbers.

A generated enum can be used infallibly when any of these properties hold:

- Any bitpattern of the field is covered by an enum variant
- The enum has a default value
- The enum has a catch all value

Buffers

A buffer is used to represent an stream of bytes on a device. This could for example be a fifo for a radio. It's quite a simple construct and thus is limited in configuration options.

It is accessed as a function on the block it's part of. The function returns a BufferOperation which can be used to read and write from/to the buffer. This operation type also implements the embedded-io traits.

Example usage:

```
let mut device = MyDevice::new(DeviceInterface::new());

device.foo().write_all(&[0, 1, 2, 3]).unwrap();
let mut buffer = [0; 8];
let len = device.bar().read(&mut buffer).unwrap();
```

Below are minimal and full examples of how buffers can be defined.

DSL

Minimal:

```
buffer Foo = 5,
```

Full:

```
/// A foo buffer
#[cfg(bar)]
buffer Foo: WO = 5,
```

Manifest

Minimal:

```
"Foo": {
    "type": "buffer",
    "address": 5
},
```

Full:

```
"Foo": {
    "type": "buffer",
    "cfg": "bar",
    "description": "A foo buffer",
    "access": "WO",
    "address": 5
},
```

Required

`address`

The address of the buffer.

Integer value that must fit in the given address type in the global config and can be negative.

`type` (manifest only)

The type of the object.

For buffers this field is a string with the contents `"buffer"`.

Optional

`cfg` or `#[cfg(...)]`

Allows for cfg-gating the buffer.

In the DSL, the normal Rust syntax is used. Just put the attribute on the buffer definition. Only one attribute is allowed.

In the manifest it is configured with a string. The string only defines the inner part: `#[cfg(foo)] = "cfg": "foo",`.

> **Warning**
>
> Check the chapter on cfg for more information. The cfg's are not checked by the toolkit and only passed to the generated code and so there are some oddities to be aware of.

`description` or `#[doc = ""]`

The doc comments for the generated code.

For the DSL, use the normal doc attributes or triple slash `///`. Multiple attributes get concatenated with a newline (just like normal Rust does).

For the manifest, this is a string.

The description is added as normal doc comments to the generated code. So it supports markdown and all other features you're used to. The description is used on the generated buffer struct and on the function to access the buffer.

`access`

Overrides the default buffer access.

Options are: `RW`, `ReadWrite`, `WO`, `WriteOnly`, `RO`, `ReadOnly`.
They are written 'as is' in the DSL and as a string in the manifest.

Blocks

A block is a collection of other objects. This can be great to e.g. pool related objects together.

Blocks have their own address offset which is applied to all child objects. With this repeated and ref blocks are supported and can be used to great effect.

> **Tip**
>
> The generated code has one implicit root block with the name of the device that acts as the entry point of the driver. The only difference with other blocks is that it takes ownership of the interface instance and always has address offset 0.

It is accessed as a function on the parent block it's part of.

All objects are generated globally so child objects still need a globally unique name and are not generated in a module.

Example usage:

```
// MyDevice is the root block
let mut device = MyDevice::new(DeviceInterface::new());

let mut child_block = device.foo();
child_block.bar().dispatch().unwrap();
// Or in one go
device.foo().bar().dispatch().unwrap();
```

Below are minimal and full examples of how blocks can be defined. There's one child object defined as example.

- Blocks
  - DSL
  - Manifest
  - Required
    - `type` (manifest only)
  - Optional
    - `cfg` or `#[cfg(...)]`
    - `description` or `#[doc = ""]`
    - `address_offset`
    - `repeat`
    - `objects` (manifest only)

DSL

Minimal:

```
block Foo {
    buffer Bar = 0,
}
```

Full:

```
/// Block description
#[cfg(not(blah))]
block Foo {
    const ADDRESS_OFFSET = 10;
    const REPEAT = {
        count: 2,
        stride: 20,
    };

    buffer Bar = 0,
}
```

Manifest

Minimal:

```
"Foo": {
    "type": "block",
    "objects": {
        "Bar": {
            "type": "buffer",
            "address": 0
        }
    }
}
```

Full:

```
"Foo": {
    "type": "block",
    "cfg": "not(blah)",
    "description": "Block description",
    "address_offset": 10,
    "repeat": {
        "count": 2,
        "stride": 20,
    },
    "objects": {
        "Bar": {
            "type": "buffer",
            "address": 0
        }
    }
}
```

Required

`type` (manifest only)

The type of the object.

For blocks this field is a string with the contents `"block"`.

Optional

`cfg` or `#[cfg(...)]`

Allows for cfg-gating the block.

In the DSL, the normal Rust syntax is used. Just put the attribute on the block definition. Only one attribute is allowed.

In the manifest it is configured with a string. The string only defines the inner part: `#[cfg(foo)] = "cfg": "foo",`.

> **Warning**
>
> Check the chapter on cfg for more information. The cfg's are not checked by the toolkit and only passed to the generated code and so there are some oddities to be aware of.

`description` or `#[doc = ""]`

The doc comments for the generated code.

For the DSL, use the normal doc attributes or triple slash `///`. Multiple attributes get concatenated with a newline (just like normal Rust does).

For the manifest, this is a string.

The description is added as normal doc comments to the generated code. So it supports markdown and all other features you're used to. The description is used on the generated block struct and on the function to access the block.

`address_offset`

The address offset used for all child objects specified as a signed integer.

The offset is applied to all addresses of the children. So when the offset is 5 and a child specifies address 7, then the actual used address will be 12.

If the offset is not specified, it is default 0.

`repeat`

Repeat the block a number of times at different address offsets.

It is specified with two fields:

- Count: unsigned integer, the amount of times the block is repeated
- Stride: signed integer, the amount the address offset changes per repeat

The calculation is `offset = base_offset + index * stride`.

When the repeat field is present, the function to access a block will have an extra parameter for the index.

`objects` (manifest only)

A map that contains all the child objects.

For the DSL the children are defined in the block directly.

Refs

A ref is a copy of another object where parts of that object are overridden with a new value.

For example, you may have two different registers that have the same fields but reside at different addresses. You may not want to use a repeat if they are not logically repeated.

Refs can target registers, commands and blocks. Buffers can't be reffed because they're so simple there's nothing worth overriding. You also can't ref other refs since that would open the gates of hell in the toolkit implementation.

> **Note**
>
> Using a ref is exactly the same as using the original, just with the new name. The only difference in API is that if the reset value of a field set is overridden, that fieldset gets an extra constructor with which you can initialize it with the overridden reset value.
>
> The possible overrides are all of the object properties that *don't* specify things about the field set. For example, `size_bits`, `fields`, `byte_order` and more can't be overridden.

Below are minimal and full examples of how refs can be defined. The examples all override a register and its address.

DSL

Minimal:

```
register Foo {
    const ADDRESS = 3;
    const SIZE_BITS = 16;

    value: uint = 0..16,
},
ref Bar = register Foo {
    const ADDRESS = 5;
},
```

Full:

```
register Foo {
    const ADDRESS = 3;
    const SIZE_BITS = 16;

    value: uint = 0..16,
},
/// This is a copy of Foo, but now with address 5!
#[cfg(feature = "bar-enabled")]
ref Bar = register Foo {
    const ADDRESS = 5;
},
```

Manifest

Minimal:

```
"Foo": {
    "type": "register",
    "address": 3,
    "size_bits": 16,
    "fields": {
        "value": {
            "base": "uint",
            "start": 0,
            "end": 16
        }
    }
},
"Bar": {
    "type": "ref",
    "target": "Foo",
    "override": {
        "type": "register",
        "address": 3,
    }
}
```

Full:

```
"Foo": {
    "type": "register",
    "address": 3,
    "size_bits": 16,
    "fields": {
        "value": {
            "base": "uint",
            "start": 0,
            "end": 16
        }
    }
},
"Bar": {
    "type": "ref",
    "target": "Foo",
    "description": "This is a copy of Foo, but now with address 5!",
    "cfg": "feature = \"bar-enabled\"",
    "override": {
        "type": "register",
        "address": 3,
    }
}
```

Required

`target` (manifest only)

The (string) name of the reffed object.

`type` (manifest only)

The type of the object.

For refs this field is a string with the contents `"ref"`.

`override` or `{ .. }`

Contains the override fields of the ref.

This is formatted as an object normally is, but some fields will be rejected.

Optional

`cfg` or `#[cfg(...)]`

Allows for cfg-gating the ref.

In the DSL, the normal Rust syntax is used. Just put the attribute on the ref definition. Only one attribute is allowed.

In the manifest it is configured with a string. The string only defines the inner part: `#[cfg(foo)] = "cfg": "foo",`.

> **Warning**
>
> Check the chapter on cfg for more information. The cfg's are not checked by the toolkit and only passed to the generated code and so there are some oddities to be aware of.

`description` or `#[doc = ""]`

The doc comments for the generated code.

For the DSL, use the normal doc attributes or triple slash `///`. Multiple attributes get concatenated with a newline (just like normal Rust does).

For the manifest, this is a string.

The description is added as normal doc comments to the generated code. So it supports markdown and all other features you're used to. The description is used on the generated ref struct and on the function to access the ref.

Dsl syntax

> **Caution**
>
> This doc is written manually. The implementation may differ. If it does, then either this doc is wrong or the implementation is wrong. In any case, them disagreeing is a bug. Please file an issue!

> **Warning**
>
> While something may be valid to be parsed, it may not be valid as a construct and may generate an error deeper down.

Top-level item is *Device*.

- '*' is used to signal 0 or more instances.
- '?' is used to signal 0 or 1 instances.
- '|' is used as an 'or'. One of the options in the chain can be used.
- '( )' is used to group things together.
- Any `keyword` or brackets in the grammar use backticks just like word 'keyword' on this line.

This doesn't map perfectly on the YAML and JSON inputs, but they should be made as close as possible.

Device:

GlobalConfigList
ObjectList

GlobalConfigList:

(`config` `{` *GlobalConfig** `}`)?

GlobalConfig:

(`type` `DefaultRegisterAccess` `=` *Access* `;`)
| (`type` `DefaultFieldAccess` `=` *Access* `;`)
| (`type` `DefaultBufferAccess` `=` *Access* `;`)
| (`type` `DefaultByteOrder` `=` *ByteOrder* `;`)

| (`type DefaultBitOrder =` *BitOrder* `;`)
| (`type RegisterAddressType =` *IntegerType* `;`)
| (`type CommandAddressType =` *IntegerType* `;`)
| (`type BufferAddressType =` *IntegerType* `;`)
| (`type NameWordBoundaries =` *NameWordBoundaries* `;`)
| (`type DefmtFeature =` *String* `;`)

NameWordBoundaries: This specifies the input, not the output. Only applies to object and field names.

[*Boundary**]
| *String*

ObjectList:

(*Object*(`,` *Object*)* `,` ?)?

Object:

Block
| *Register*
| *Command*
| *Buffer*
| *RefObject*

RefObject: An object that is a copy of another object. Any items in the object are overrides.

AttributeList `ref` *IDENTIFIER* `=` *Object*

AttributeList:

---

*Attribute**

Attribute: Used for documentation and conditional compilation

---

(`#[doc =` *STRING* `]`)

| (`#` `[` `cfg` `(` *ConfigurationPredicate* `)` `]`)

Block:

AttributeList
`block` *IDENTIFIER* `{` *BlockItemList ObjectList* `}`

BlockItemList:

---

*BlockItem**

BlockItem:

(`const` `ADDRESS_OFFSET` `=` *INTEGER* `;`)
| (`const` *Repeat*)

Register:

AttributeList
`register` *IDENTIFIER* `{` *RegisterItemList FieldList* `}`

RegisterItemList:

---

*RegisterItem**

RegisterItem:

```
( type Access = Access ; )
| ( type ByteOrder = ByteOrder ; )
| ( type BitOrder = BitOrder ; )
| ( const ADDRESS = INTEGER ; )
| ( const SIZE_BITS = INTEGER ; )
| ( const RESET_VALUE = INTEGER | U8_ARRAY ; )
| ( const Repeat)
| ( const ALLOW_BIT_OVERLAP = BOOL ; )
| ( const ALLOW_ADDRESS_OVERLAP = BOOL ; )
```

Access:

```
( ReadWrite | RW )
| ( ReadOnly | RO )
| ( WriteOnly | WO )
```

ByteOrder:

```
LE | BE
```

BitOrder:

```
LSB0 | MSB0
```

FieldList:

```
(Field ( , Field)* , ?)
```

Field:

```
AttributeList
IDENTIFIER : Access? BaseType FieldConversion? = FieldAddress
```

FieldConversion:

(`as` `try` ? *TYPE_PATH*)
| (`as` `try` ? `enum` *IDENTIFIER* `{` *EnumVariantList* `}`)

EnumVariantList:

EnumVariant(`,` *EnumVariant*)* `,` ?

EnumVariant:

AttributeList
IDENTIFIER (`=` *EnumValue*)?

EnumValue:

INTEGER| `default` | `catch_all`

FieldAddress:

INTEGER
| (*INTEGER* `..` *INTEGER*)
| (*INTEGER* `..=` *INTEGER*)

BaseType:

`bool` | `uint` | `int`

Command:

AttributeList
`command` *IDENTIFIER CommandValue*?

CommandValue:

(`=` *INTEGER*)

| (`{` *CommandItemList* (`in` `{` *FieldList* `}` `,`?)? (`out` `{` *FieldList* `}` `,`?)? `}`)

CommandItemList:

---

*CommandItem**

CommandItem: Commands have data going in and out, so they need two separate data field types. If no in fields, then no data is sent. If no out fields, then no data is returned.

(`type` `ByteOrder` `=` *ByteOrder* `;`)
| (`type` `BitOrder` `=` *BitOrder* `;`)
| (`const` `ADDRESS` `=` *INTEGER* `;`)
| (`const` `SIZE_BITS_IN` `=` *INTEGER* `;`)
| (`const` `SIZE_BITS_OUT` `=` *INTEGER* `;`)
| (`const` *Repeat*)
| (`const` `ALLOW_BIT_OVERLAP` = *BOOL* `;`)
| (`const` `ALLOW_ADDRESS_OVERLAP` = *BOOL* `;`)

Repeat:

`REPEAT` `=` `{` `count` `:` *INTEGER* `,` `stride` `:` *INTEGER* `,`? `}` `;`

Buffer:

AttributeList
`buffer` *IDENTIFIER*(`:` *Access*)? (`=` *INTEGER*)?

Manifest syntax

> **Caution**
>
> This doc is written manually. The implementation may differ. If it does, then either this doc is wrong or the implementation is wrong. In any case, them disagreeing is a bug. Please file an issue!

> **Warning**
>
> While something may be valid to be parsed, it may not be valid as a construct and may generate an error deeper down.

Top-level item is *Device*.

Anything marked like *this* denotes its own type specification.

These are the pre-defined types:

- bool
- uint
- int
- float
- string
- array
  - Using `[]` brackets.
  - If inner types are restricted, then signaled as e.g. `[float]`
- map
  - Using `{}` brackets.
  - The keys are always text/string.
  - Restrictions can be signaled as required by `?`
  - Restriction syntax: `{ foo?, bar?: float, xen: bool, *: bool }
    - Optional field `foo` without type restriction
    - Optional field `bar` with float restriction
    - Required field `xen` with bool restriction
    - `0..N` fields with any name with bool restriction

Further restriction can be denoted using `oneof()`, for example: `int oneof(1, 2, 3, 4)` or `oneof(bool, int)`

Device: The key of the object will become the name of it

```
{
    config?: _GlobalConfig_,
    *: _Object_
}
```

GlobalConfig:

```
{
    default_register_access?: _Access_,
    default_field_access?: _Access_,
    default_buffer_access?: _Access_,
    default_byte_order?: _ByteOrder_,
    default_bit_order?: _BitOrder_,
    register_address_type?: _IntegerType_,
    command_address_type?: _IntegerType_,
    buffer_address_type?: _IntegerType_,
    name_word_boundaries?: _NameWordBoundaries_
    defmt_feature?: string
}
```

Access:

```
string oneof("ReadWrite", "RW", "ReadOnly", "RO", "WriteOnly", "WO")
```

ByteOrder:

```
string oneof("LE", "BE")
```

BitOrder:

```
string oneof("LSB0", "MSB0")
```

IntegerType:

```
string oneof("u8", "u16", "u32", "i8", "i16", "i32", "i64")
```

NameWordBoundaries:

```
oneof([_Boundary_], string)
```

Object:

```
oneof(
  _Block_,
  _Register_,
  _Command_,
  _Buffer_,
  _RefObject_
)
```

RefObject:

```
{
    type: string oneof("ref"),
    cfg?: string,
    description?: string,
    target: string,
    override: _Object_,
}
```

Block:

```
{
    type: string oneof("block"),
    cfg?: string,
    description?: string,
    address_offset?: int,
    repeat?: _Repeat_,
    objects?: {
        *: _Object_
    }
}
```

Repeat:

```
{
    count: uint,
    stride: int
}
```

Register:

```
{
    type: string oneof("register"),
    cfg?: string,
    description?: string,
    access?: _Access_,
    byte_order?: _ByteOrder_,
    bit_order?: _BitOrder_,
    address: int,
    size_bits: int,
    reset_value?: oneof(int, [uint]),
    repeat?: _Repeat_,
    allow_bit_overlap?: bool,
    allow_address_overlap?: bool,
    fields?: {
        *: _Field_
    }
}
```

Field:

```
{
    cfg?: string,
    description?: string,
    access?: _Access_,
    base: _BaseType_,
    conversion?: _FieldConversion_,
    try_conversion?: _FieldConversion_,
    start: int,
    end?: int,
}
```

BaseType:

```
string oneof("bool", "int", "uint")
```

FieldConversion:

```
oneof(
    string,
    {
        name: string,
        description?: string,
        *: _EnumVariant_
    }
)
```

EnumVariant:

```
oneof(
    _EnumValue_,
    {
        cfg?: string,
        description?: string,
        value?: _EnumValue_
    }
)
```

EnumValue:

```
oneof(
    null, int, string oneof("default", "catch_all")
)
```

Command:

```
{
    type: string oneof("command"),
    cfg?: string,
    description?: string,
    byte_order?: _ByteOrder_,
    bit_order?: _BitOrder_,
    address: int,
    repeat?: _Repeat_,
    allow_bit_overlap?: bool,
    allow_address_overlap?: bool,
    size_bits_in?: int,
    fields_in?: {
        *: _Field_
    },
    size_bits_out?: int,
    fields_out?: {
        *: _Field_
    },
}
```

Buffer:

```
{
    type: string oneof("buffer"),
    cfg?: string,
    description?: string,
    access?: _Access_,
    address: int,
}
```

Cfg

Pretty much anywhere you can put docs/description, you can also put some cfg. They use the same syntax as the inside of the cfg attribute, e.g. `feature = "blah"`.

> **Important**
>
> The cfg's have no impact on the code generation other than forwarding the cfg's as attributes on items.

This presents a couple of challenges:

- It's quite hard to check whether a driver can compile with any combination of cfg's.
- The cfg's are resolved after code generation, so the toolkit can't check anything.
- It's hard to predict how the cfg attributes on various items interact.

So what does this all mean?

> **Caution**
>
> - The support for cfg's are best effort only. Expect things to be weird or something to work against you.
> - Some analysis may not be done on objects with cfg which can lead to weird errors in the generated code since problems are not caught beforehand.

> **Warning**
>
> - If you use cfg's, check the generated code to see if everything looks alright.
> - Use cfg's only sparingly.
> - Test all realistic cfg combinations, preferably even in CI.

If there is a problem and the toolkit can do better, then please make an issue!

Memory

Memory is quite easy. But assigning meaning to it is where all complexity comes from. This page describes all the different levels of memory and what this crate does. The goal is to leave you with a better understanding of how memory is handled and to serve as a quick reference if or when confusion ensues.

Concepts

Byte order

Also known as endianness. It describes what the first byte is in an array of bytes. There are two options generally:

- Little endian (LE)
  - The smallest or first byte is at the front
  - I.E. `[10, 11, 12, 13]` where indexing at 0 would yield `10`
  - Lower indices are in lower memory addresses, higher indices are in higher memory addresses
- Big endian (BE)
  - The smallest or first byte is at the back
  - I.E. `[10, 11, 12, 13]` where indexing at 0 would yield `13`
  - Lower indices are in higher memory addresses, higher indices are in lower memory addresses

Bit order

There is also order on the bit level. We get to decide which bit is the smallest of the 8 in a byte. There's two options:

- Least significant bit 0 (LSB0)
  - The bit at index 0 is the lowest bit
  - I.E. the number `1` is coded as `0b0000_0001` or `0x01`
- Most significant bit 0 (MSB0)
  - The bit at index 0 is the highest bit
  - I.E. the number `1` is coded as `0b1000_0000` or `0x80`

Together

> **Important**
>
> Together, the bit and byte order determine where a given bit is in an array of bytes.

Bit `0` is defined as the `0th` bit on the `0th` byte.
Bit `10` is defined as the `2nd` bit on the `1st` byte.

Here are the options for when only bit 0 is high in a 2-byte array:

```
LE, LSB0:
  [0b0000_0001, 0b0000_0000] or [0x01, 0x00]
             ^            ^  <- Bits 0
   ^^^^^^^^^^               <- Byte 0

LE, MSB0:
  [0b1000_0000, 0b0000_0000] or [0x80, 0x00]
     ^            ^          <- Bits 0
   ^^^^^^^^^^               <- Byte 0

BE, LSB0:
  [0b0000_0000, 0b0000_0001] or [0x00, 0x01]
             ^            ^  <- Bits 0
                ^^^^^^^^^^  <- Byte 0

BE, MSB0:
  [0b0000_0000, 0b1000_0000] or [0x00, 0x80]
     ^            ^          <- Bits 0
                ^^^^^^^^^^  <- Byte 0
```

Here are the options for when only bit 10 is high in a 2-byte array:

```
LE, LSB0:
  [0b0000_0000, 0b0000_0100] or [0x00, 0x04]
            ^            ^    <- Bits 2
                 ^^^^^^^^^^  <- Byte 1

LE, MSB0:
  [0b0000_0000, 0b0010_0000] or [0x00, 0x20]
       ^            ^        <- Bits 2
                 ^^^^^^^^^^  <- Byte 1

BE, LSB0:
  [0b0000_0100, 0b0000_0000] or [0x04, 0x00]
            ^            ^    <- Bits 2
   ^^^^^^^^^^               <- Byte 1

BE, MSB0:
  [0b0010_0000, 0b0000_0000] or [0x20, 0x00]
       ^            ^        <- Bits 2
   ^^^^^^^^^^               <- Byte 1
```

The memories of device-driver

> **Important**
>
> Here's the tricky part. The data of a register can be present in three places:
>
> - On the device
> - On the transport bus (e.g. while writing/reading it over SPI)
> - In RAM on your microcontroller

The first two we don't have any influence over. But we can create our own model with this crate that fits the device.

Let's do some examples for real existing devices. If there's a device that does things a bit different, feel free to PR this file!

> **Tip**
>
> If all registers have the same behaviour (which is the case usually), you can set the bit and byte orders in the global config too so it applies to all registers that don't explicitly have it set.

Example LIS3DH - Multi-register LE, LSB0

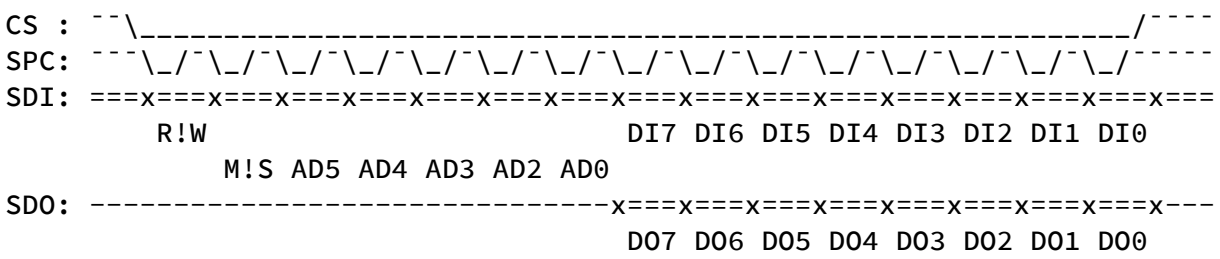
The LIS3DH accelerometer is one byte per register, but there are some multi-register

values that we may want to model as one two-byte register. This can be done because the address will auto increment after any reads/writes.

In the datasheet we find the registers:

Name	Access	Address	Value
OUT_X_L	ro	0x28	`X [0..8]`
OUT_X_H	ro	0x29	`X [8..16]`

And the transport schema:



Let's analyze:

Byte order

- We will make one register out of the two starting at address 0x28
- The first byte will be from `OUT_X_L` and the second byte will be from `OUT_X_H`
- So, low index is low byte and high index is high byte
- Thus this combined register is **little endian (LE)**

Bit order

- Depends on the hardware settings of the SPI. We set it to most significant bit first to match the datasheet.
- The 0th bit is the last and least significant one of the byte
- Thus this is **Least Significant Bit 0 (LSB0)**

And so we get our register definition:

```
register OutX {
    const ADDRESS = 0x68; // Including bit for multi-register ops
    const SIZE_BITS = 16;
    type ByteOrder = LE;
    type BitOrder = LSB0;

    value: int = 0..16,
}
```

Example s2-lp - Multi-register BE, LSB0

This is a radio chip and just like the LIS3DH can combine multiple registers in one read/write.

In the datasheet we find the registers:

Name	Address	Bits	Value
SYNT3	05	7:5	PLL_CP_ISEL
		4	BS
		3:0	SYNT[27:24]
SYNT2	06	7:0	SYNT[23:16]
SYNT1	07	7:0	SYNT[15:8]
SYNT0	08	7:0	SYNT[7:0]

And the transport schema (for writes):

Let's analyze:

Byte order

- We will make one register out of this starting at address 0x05
- The first byte will contain `SYNT[27:24]` and the last byte will contain `SYNT[7:0]`
- So, low index is high byte and high index is low byte
- Thus this combined register is **big endian (BE)**

Bit order

- Depends on the hardware settings of the SPI. We set it to most significant bit first to match the datasheet.
- The 0th bit is the last/least significant one
- Thus this is **Least Significant Bit 0 (LSB0)**

```
register OutX {
    const ADDRESS = 0x05;
    const SIZE_BITS = 32;
    type ByteOrder = BE;
    type BitOrder = LSB0;

    synt: uint = 0..=27,
    bs: bool = 28,
    pll_cp_isel: uint = 29..=31
}
```

Example DW1000 - Single-register LE, LSB0

This chip doesn't have multi register reads, but it does have registers bigger than a byte. So even a single register must take care of byte ordering.

Luckily for us, the user manual spells out the modes (along to the diagrams):

- Note: The octets of a multi-octet value are transferred on the SPI interface in octet order beginning with the low-order octet.

- Diagram example: Register `0x00` contains `0xDECA0130` and is sent as `[0x30, 0x01, 0xCA, 0xDE]`
  - Thus **little endian (LE)**

- Note: The octets are physically presented on the SPI interface data lines with the high order bit sent first in time.

  - Depends on the hardware settings of the SPI. We set it to most significant bit first to match the datasheet.
  - Thus **Least Significant Bit 0 (LSB0)** (assuming your SPI master also sees the first bit as the LSB)

```
register DevId {
    const ADDRESS = 0x00;
    const SIZE_BITS = 32;
    type ByteOrder = LE;
    type BitOrder = LSB0;

    r_id_tag: uint = 16..32,
    model: uint = 8..16,
    ver: uint = 4..8,
    rev: uint = 0..4
}
```