Contents

Chapter 4

Design Goals and Specification

We propose extracting information from the Schematic Editor to generate C++ code using automated tooling. This tooling integrates firmware development in a snippet based hardware design process. Our design goals are the following:

DG1 Functional: The software engineer who programs the firmware should not require the schematics any longer. Instead, the generated C++ code should include all needed information in the form a software engineer prefers working with. Additionally, the tooling should produce all the information required for harness specification.

DG2 Easily Usable: Both the electrical and the software engineer will interface with the new tooling. With the new tooling their workflows should remain as unchanged as possible. The new workflow should be easy to learn and adapt to.

DG3 Adaptable: As many projects as possible should be able to use this tooling. Therefore, this tooling should be easily adaptable to different design software, programming languages and completely different use-cases.

DG4 Maintainable: The tooling should be easily maintainable. This includes extending the tooling to new use cases and adapting to changes in the workflow, e.g., new versions of KiCad.

We chose to split the tooling into four programs and a file format, the Group Netlist, visualized in Figure 4.1.

1. kicad_group_netlister extracts information from the schematics and generates the Group Netlist, which we specify in subsection 4.1.1.
2. group_netlist_merger combines multiple Group Netlists from different schematics into a single one (explained in section 4.4).
3. code_gen generates C++ code from a Group Netlist and a project-specific template (explained in section 4.3).

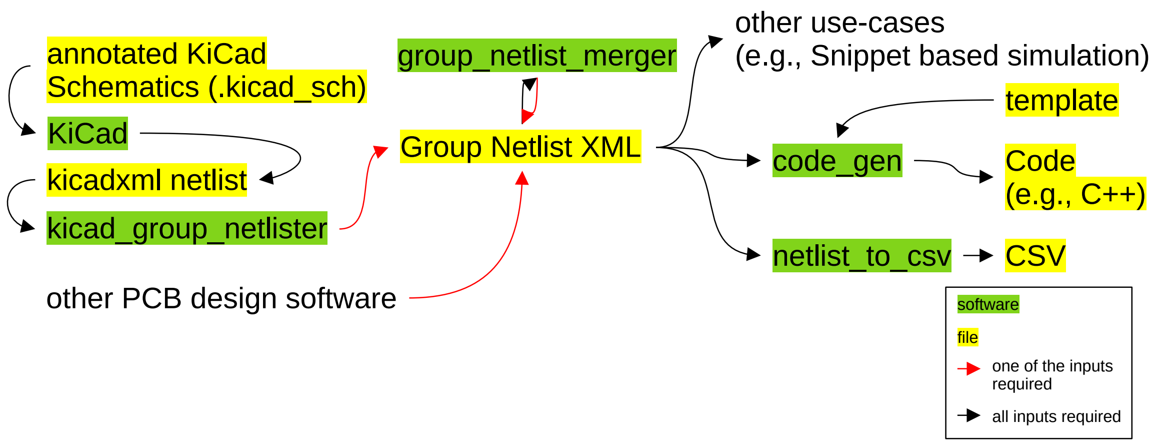


Figure 4.1: Overview of the final generator design. Notice how data flows from left to right, from the electrical engineering tools into the software engineering tools. Our proposed Group Netlist XML is the common interface between. The group_netlist_merger takes multiple Group Netlists and combines them into a single one (see section 5.8).

4. netlist_to_csv converts a Group Netlist into a CSV file meant for harness specification (explained in section 4.5).

With this structure we extend our goals:

DG5 code_gen, netlist_to_csv and group_netlist_merger Independent of KiCad: code_gen, netlist_to_csv and group_netlist_merger should not use the schematics, relying entirely on the Group Netlist data that is unrelated to the schematics. Thus, code_gen, netlist_to_csv and group_netlist_merger should be independent of KiCad.

DG6 kicad_group_netlister and group_netlist_merger Independent of specific Usage: The Group Netlist specification should not be limited to C++, the PLUTO EPS firmware and harness specification. Instead, kicad_group_netlister and group_netlist_merger should be independent of how the Group Netlist will be used.

DG7 Stable Group Netlist Specification: The tooling maintainer should not have to change the Group Netlist specification in the future.

DG8 Human Readable Group Netlist: A user should understand a Group Netlist without accompanying documentation.

DG5 allows implementing a program that creates a Group Netlist for PCB design software that is not KiCad. Thus, users could also utilize any Group Netlist consuming tools with other PCB design software. Similarly, DG6 allows implementing Group Netlist consuming tools for different purposes. This, for example, includes generating code for a different firmware project or in a different language. Consequently, DG5 and DG6 aid in fulfilling the desired adaptability DG3.
Furthermore, we expect limiting the input data for code_gen (see DG5) to result in simpler

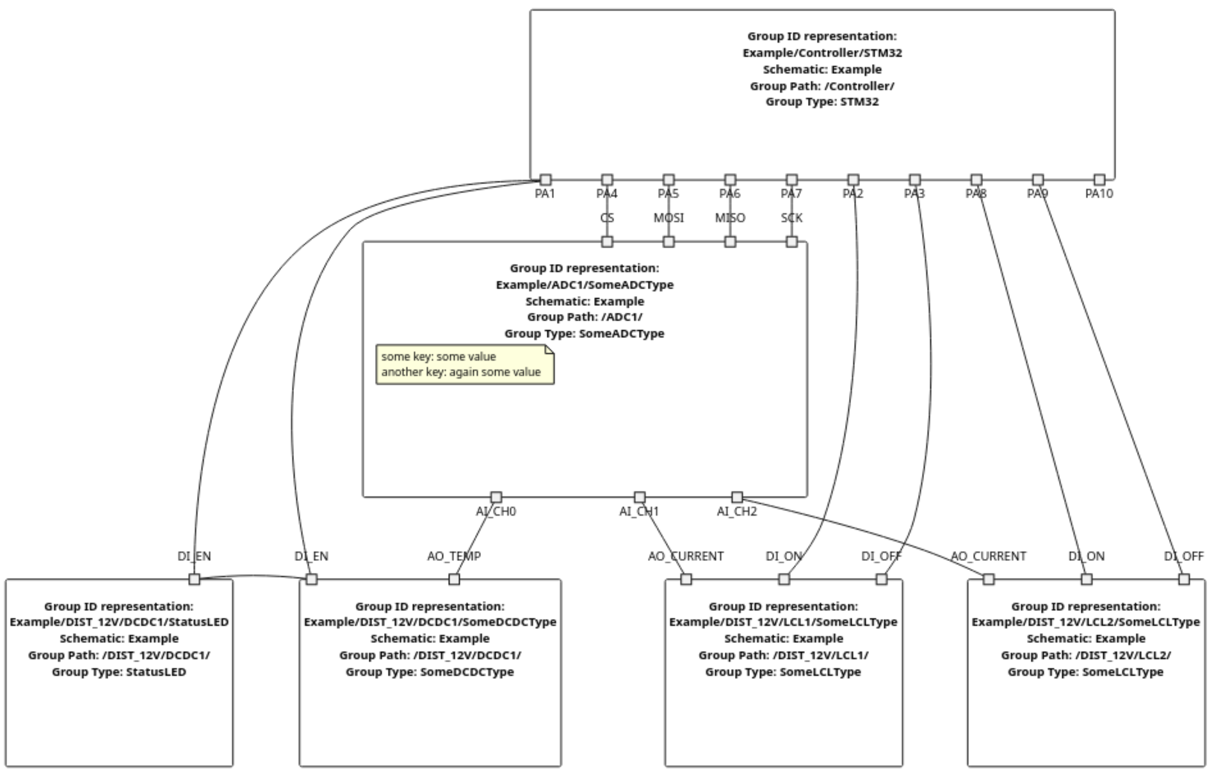


Figure 4.2: Example Group Netlist Visualization of Figure 4.3. Every box represents a Group with the string representation of the Group ID explained in subsection 4.1.2. All small squares represent Pins and lines between represent Nets. Notice that the two bottom right Groups belong to the same Group Type, "SomeLCLType". Also notice that there are two Groups with the same Schematic and Group Path but different Group Type; only the combination of the three is uniquely identifying. Additionally, while each Group may consist of multiple hardware components, the components are not relevant for this perspective and, thus, not shown.

and, thus, more maintainable program (see DG4). Analogously, we anticipate the Group Netlist specification to be simpler, being independent from both KiCad and the firmware. A simpler specification, especially a human readable one (see DG8) with a stable specification (see DG7), should further help creating maintainable tooling DG4.

4.1 Group Netlist

The **Group Netlist** is the common interface between the schematics and generated code or other use-cases. Notice that we capitalize terms introduced by us. Figure 4.2 shows an example Group Netlist. Schematics consist of components and wires to specify the electrical layout of the hardware. The firmware, however, only considers logical groups of hardware: Each group fulfills a single task using multiple components. The firmware controls it as a single unit, ignoring its constituents. Therefore, we group components into so-called **Group**s and only represent the connections between these Groups. Each Group has a **Group Path**, allowing structuring Groups.
The Group Path must match the (Python) regex `/([a-zA-Z0-9_/\-\+ ]+/|)` (an alphanu-

```
<?xml version='1.0' encoding='utf-8'?>
<groupNetlist>
    <netlist>
        <sources>
            <source>/path/to/Example.kicad_sch</source>
        </sources>
        <date>2026-01-05T14:56:19.655613</date>
        <tool>kicad_group_netlister v0.1.0</tool>
    </netlist>
    <groups>
        <!-- -snip- -->
    </groups>
    <nets>
        <!-- -snip- -->
    </nets>
</groupNetlist>
```

Figure 4.3: Example Group Netlist XML visualized in Figure 4.2. Figure 4.4 shows the full `groups` tag and Figure 4.5 the full `nets` tag.

meric string including underscore, slash, minus, plus and space with `/` as a prefix and suffix).
Every Group has multiple **Pin**s, via which Groups connect. Each Pin has a name, unique in its Group.
Furthermore, the firmware typically categorizes Groups into types, each with their own driver code. For example, there could be multiple identical LCLs, each individually controlled but with a shared driver. For this, we support **Group Type**s. In the above example all LCL Groups could have the "SomeLCLType" Group Type. Every Group has exactly one Group Type but every Group Type may be used by multiple Groups. Groups of the same Group Type should have the same Pin Names.
Finally, some hardware splits its components over multiple schematics (i.e., multiple interconnected PCBs). Therefore, every Group must belong to exactly one **Schematic**. The Schematic, Group Path and Group Type uniquely identify a Group and, thus, when combined form the **Group ID** (see subsection 4.1.2). Consequently, a Pin is globally unique through the Pin's Group ID and the **Pin Name** (i.e., there are no two Pins in the entire Group Netlist with the same Group ID and Pin Name). We call the Group ID, Pin Name tuple **Global Pin ID**. The Schematic, Group Type and Pin Name must match the (Python) regex `[a-zA-Z0-9_\-\+ ]+` (an alphanumeric string including underscore, minus, plus and space).
Lastly, to represent that some points are electrically connected there are **Net**s. Each Net consists of a list of all Pins that are electrically connected. Pins are called **Node** in the context of Nets.
Finally, the Group Netlist consists of a list of Groups and a list of Nets. This is all the information the firmware generation or other use-cases require.

4.1.1 Group Netlist XML Specification

Aiming for maintainable tooling (see DG4) we chose XML to represent a Group Netlist. Firstly, XML is a human readable format (see DG8) with an established standard with support in many programming languages, including Python. Secondly, the DLR code generator Pando already uses XML input files, easing the maintenance and adoption of

```
<!-- -snip- -->
<groups>
    <group schematic="Example" path="/Controller/" type="STM32">
        <groupMapFields />
        <pins>
            <pin name="PA1" />
            <pin name="PA2" />
            <pin name="PA3" />
            <pin name="PA4" />
            <pin name="PA5" />
            <pin name="PA6" />
            <pin name="PA7" />
            <pin name="PA8" />
            <pin name="PA9" />
            <pin name="PA10" />
        </pins>
    </group>
    <group schematic="Example" path="/ADC1/" type="SomeADCType">
        <groupMapFields>
            <groupMapField name="some key">some value</groupMapField>
            <groupMapField name="another key">again some value</groupMapField>
        </groupMapFields>
        <pins>
            <pin name="MISO" />
            <pin name="MOSI" />
            <pin name="SCK" />
            <pin name="CS" />
            <pin name="AI_CH0" />
            <pin name="AI_CH1" />
            <pin name="AI_CH2" />
        </pins>
    </group>
    <group schematic="Example" path="/DIST_12V/DCDC1" type="StatusLED">
        <groupMapFields />
        <pins>
            <pin name="DI_EN" />
        </pins>
    </group>
    <group schematic="Example" path="/DIST_12V/DCDC1" type="SomeDCDCType">
        <groupMapFields />
        <pins>
            <pin name="DI_EN" />
            <pin name="AO_TEMP" />
        </pins>
    </group>
    <group schematic="Example" path="/DIST_12V/LCL1" type="SomeLCLType">
        <groupMapFields />
        <pins>
            <pin name="DI_ON" />
            <pin name="DI_OFF" />
            <pin name="AO_CURRENT" />
        </pins>
    </group>
    <group schematic="Example" path="/DIST_12V/LCL2" type="SomeLCLType">
        <groupMapFields />
        <pins>
            <pin name="DI_ON" />
            <pin name="DI_OFF" />
            <pin name="AO_CURRENT" />
        </pins>
    </group>
</groups>
<!-- -snip- -->
```

Figure 4.4: `groups` tag of the example Group Netlist in Figure 4.3. Notice how one Group has `groupMapField` tags and others do not; these fields are optional. Furthermore, notice that Groups of the same Group Type have the same Pins.

```xml
<!-- -snip- -->
<nets>
    <!-- Unconnected Pin on Controller -->
    <net>
        <node schematic="Example" path="/Controller/" type="STM32" pin="PA10" />
    </net>
    <!-- For ADC1 -->
    <net>
        <node schematic="Example" path="/Controller/" type="STM32" pin="PA6" />
        <node schematic="Example" path="/ADC1/" type="SomeADCType" pin="MISO" />
    </net>
    <net>
        <node schematic="Example" path="/Controller/" type="STM32" pin="PA5" />
        <node schematic="Example" path="/ADC1/" type="SomeADCType" pin="MOSI" />
    </net>
    <net>
        <node schematic="Example" path="/Controller/" type="STM32" pin="PA7" />
        <node schematic="Example" path="/ADC1/" type="SomeADCType" pin="SCK" />
    </net>
    <net>
        <node schematic="Example" path="/Controller/" type="STM32" pin="PA4" />
        <node schematic="Example" path="/ADC1/" type="SomeADCType" pin="CS" />
    </net>
    <!-- For DCDC1 and StatusLED -->
    <net>
        <node schematic="Example" path="/Controller/" type="STM32" pin="PA1" />
        <node schematic="Example" path="/DIST_12V/DCDC1" type="StatusLED" pin="DI_EN" />
        <node schematic="Example" path="/DIST_12V/DCDC1" type="SomeDCDCType" pin="DI_EN" />
    </net>
    <net>
        <node schematic="Example" path="/ADC1/" type="SomeADCType" pin="AI_CH0" />
        <node schematic="Example" path="/DIST_12V/DCDC1" type="SomeDCDCType" pin="AO_TEMP" />
    </net>
    <!-- For LCL1 -->
    <net>
        <node schematic="Example" path="/Controller/" type="STM32" pin="PA2" />
        <node schematic="Example" path="/DIST_12V/LCL1" type="SomeLCLType" pin="DI_ON" />
    </net>
    <net>
        <node schematic="Example" path="/Controller/" type="STM32" pin="PA3" />
        <node schematic="Example" path="/DIST_12V/LCL1" type="SomeLCLType" pin="DI_OFF" />
    </net>
    <net>
        <node schematic="Example" path="/ADC1/" type="SomeADCType" pin="AI_CH1" />
        <node schematic="Example" path="/DIST_12V/LCL1" type="SomeLCLType" pin="AO_CURRENT" />
    </net>
    <!-- For LCL2 -->
    <net>
        <node schematic="Example" path="/Controller/" type="STM32" pin="PA8" />
        <node schematic="Example" path="/DIST_12V/LCL2" type="SomeLCLType" pin="DI_ON" />
    </net>
    <net>
        <node schematic="Example" path="/Controller/" type="STM32" pin="PA9" />
        <node schematic="Example" path="/DIST_12V/LCL2" type="SomeLCLType" pin="DI_OFF" />
    </net>
    <net>
        <node schematic="Example" path="/ADC1/" type="SomeADCType" pin="AI_CH2" />
        <node schematic="Example" path="/DIST_12V/LCL2" type="SomeLCLType" pin="AO_CURRENT" />
    </net>
</nets>
<!-- -snip- -->
```

Figure 4.5: `nets` tag of the example Group Netlist in Figure 4.3. Notice how some nets contain only a single (unconnected) Pin and others contain more (in this example up to three). The comments are purely for presentational purposes and not part of the specification.

our tooling (see DG2 and DG4).
The Group Netlist XML format (Figure 4.3 shows an example) allows passing a Group Netlist between programs and storage for future use. The root tag is `groupNetlist`, which contains a list of all Groups in the `groups` tag and Nets in `nets`. Additionally, it contains the `netlist` tag, which holds metadata in the `source`, `date` and `tool` tags. These show the path to the input schematic file, the date of Group Netlist generation and version of tool used, respectively.
The `groups` tag (Figure 4.4 shows an example) contains a `group` tag for each Group. These have a `schematic` attribute for the Schematic, `path` for the Group Path and `type` for the Group Type. Furthermore, every `group` tag contains a set of `pin` tags inside the `pins` tag. Each `pin` tag has a `name` for the Pin Name. Finally, the `groupMapField` tag inside the `group` tag may contain a list of `groupMapField` tags with the key in the `name` attribute and the value as the tag's text. This specification does not stipulate any meaning for any Group Map Field and leaves its meaning to future extensions. **Group Map Field**s allow annotating the schematics for specific firmware projects and/or programming languages without breaking legacy tools expecting the current specification.
The `nets` tag (Figure 4.5 shows an example) contains a `net` tag for each Net. These, in turn, hold at least one `node` tag for each Node. Similar to the `group` tag, each `node` tag has a `schematic`, `path` and `type` attribute. Though, it extends these with a `pin` tag containing the Pin Name. Consequently, a `node` tag holds all information required to globally and uniquely identify a Pin.

4.1.2 Group ID

The Group Path consists of path nodes concatenated with the character `/` (slash), similar to UNIX file systems. This allows grouping Groups together in a hierarchy. Furthermore, the Group Path must have `/` as a prefix and suffix.
As the Group ID is the combination of Schematic, Group Path and Group Type, there is no canonical string representation for it. Therefore, we specify representing a Group ID by simply concatenating the Schematic, Group Path and Group Type in that order (see Figure 4.3 for examples). Consequently, the Group Type and Schematic may not contain the character `/` (slash). Otherwise retrieving the Group ID from its string representation would be ambiguous. One can interpret the Group ID representation as a UNIX path with the first path node being the Schematic and the last the Group Type. Everything in between is the Group Path (including the leading and trailing `/`).

4.1.3 Group Glob

There are many situations in which the user must selects multiple Groups. For example, in section 4.4 we present a method of connecting Groups from different schematics. subsection 4.1.2 shows how every group can be identified using a string akin to a UNIX path. To select a set of Groups, the user may input a **Group Glob**. Group Globs are based on UNIX' globs for pathname pattern expansion, specifically the implementation in the Python standard library [10]. This explains the limited amount of characters a Group ID may consist of. However, Group Globs extend the Python glob in one way: multiple globs may be combined with commas. Iff any of the sub-globs matches a Group, the entire Group Glob does. These are a few examples.

- `**` matches all Groups.
- `Example/ADC1/SomeADCType` matches only the Group with Schematic `Example`, Group Path `/ADC1/` and Group Type `SomeADCType`.
- `Example/ADC1/SomeADCType,Example/DIST_12V/LCL1/SomeLCLType` matches only the Group with Group ID representation `Example/ADC1/SomeADCType` and `Example/DIST_12V/LCL1/SomeLCL`
- `**/Connector*` matches all Groups with a Group Type prefixed with `Connector`. Any Schematic and Group Path matches.
- `Example/**/Connector*` only matches Groups with a Group Type prefixed with `Connector`. The Schematic must be `Example`. Any Group Path matches.
- `Example*/**/Connector*` only matches Groups with a Group Type prefixed with `Connector`. The Schematic must be prefixed with `Example`. Any Group Path matches.
- `Example*/Connector*` only matches Groups with a Group Type prefixed with `Connector`. The Schematic must be prefixed with `Example`. Any Group Path must be `/`.
- The empty string matches no Group.

4.1.4 Group Netlist Python Library

We implement a Python library easing the adoption of the Group Netlist standard. The tools we implemented also use this library. Firstly, it contains an in memory representation of a Group Netlist through the `GroupNetlist` and `Group` classes. Secondly, there is the `parse_group_path` and `stringify_group_netlist` function for parsing and serialising a Group Netlist into the XML format we specify in subsection 4.1.1. It ensures that the XML file is deterministic. I.e., the same Group Netlist always creates exactly the same XML document. This is important for version control and testing. Thirdly, it provides many helper functions for checking if a string fulfills the requirements of Schematic, Group Path, Group Type and Pin Name but also matching Group Globs and Group Path handling. Lastly, it implements the `GroupNetlistWithConnections` class, which does not contain any nets. However, it caries that information in a list of `GroupWithConnection` class instances through the `GroupWithConnection` class. The `GroupWithConnection` class differs from the `Group` class in one major way: It does not simply have a list of Pins Names. Instead, it has a list of Pins carrying information: Every Pin holds a list of the Global Pin IDs. Those are the Pins that connect to the Pin this list belongs to. Consequently, the `GroupNetlistWithConnections` class does not need to hold any information about the nets.
The library converts a `GroupNetlist` instance into a `GroupNetlistWithConnections` instance by looping over each net. For every node in a net it adds all other nodes in that net to the Group this Pin belongs to. Notice that a `GroupNetlistWithConnections` instance requires quadratic memory compared to the corresponding `GroupNetlist`.

The `GroupWithConnection` class most importantly has the `get_single_pin_to_glob` member function. `get_single_pin_to_glob` receives a Pin Name and Group Glob. It goes through all other Pins that connect to the provided Pin Name and Group. If the other Pin's Group matches the Group Glob, it returns its Global Pin ID. `get_single_pin_to_glob` only ever returns a single Pin, throwing an error when there are multiple matching Pins. This function allows finding if and how any Group is connected to other Groups, i.e., any ADCs. The templating in code_gen makes heavy use of this function, see section 4.3.

4.2 Extracting Group Netlists from KiCad using kicad_group_netlister

Both the KiCad CLI graphical user interface allow converting a KiCad schematic (`.kicad_sch` file) into a netlist in the `kicadxml` format [8]. kicad_group_netlister uses this KiCad netlist as input and produces a Group Netlist XML, which we specify in subsection 4.1.1.
We only explain the KiCad netlist's contents that kicad_group_netlister consumes. The KiCad netlist contains some metadata, a list of all components and a list of all nets. A component consists of a ref, which uniquely identifies it, a sheet path, the path to the component's sheet, and fields, a key-value map. Every net is a list of nodes, which contain a pin name, pin function and the pin's component's ref.

4.2.1 Grouping KiCad Components to Groups

A Group is a logical unit performing some task. It does so by comprising multiple physical components shown in the KiCad schematic. Therefore, to generate a Group Netlist kicad_group_netlister needs to group all KiCad components that belong to any Group. It considers all components to belong to a Group iff they have the same `GroupType` field and are on the same KiCad schematic sheet. The components' `GroupType` field defines the Group Type and the components' sheet path the Group Path. This works because sheet paths have the same structure as Group Paths (see subsection 2.2.2). The Schematic of the Group is the name of the root sheet (i.e., the name of the file the user opens to read the entire schematic without the `.kicad_sch` extension). Notice that some KiCad schematics may not fulfill the restrictive naming conventions we specify in section 4.1, see subsection 4.2.3.

4.2.2 Pin Naming

A KiCad component may have multiple pins, each with a pin number. Additionally, pins may have a pin function, a string unique for the group. As multiple KiCad components form a Group and some of those components may have the same pin number and/or function, choosing Pin Names for the Group is not trivial. kicad_group_netlister implements two naming methods. For each group it makes an individual decision on which naming method to use.

1. **Entirely explicit naming:** The user may set the `GroupPinX` field, where X is a pin number, and choose a unique string to be used in the Group's Pin Name. The advantage is that the user does not have to change the symbol to choose Group Pin Names. However, the user must give every pin an explicit name using an associated field.
   This allows creating a Group_IO symbol without any electrical function and a single pin. For example, in some situations the user is only interested in some pins of a collection of components. In that case the user can add the Group_IO symbol only on the pins that she cares about. This allows abstracting complex circuitry as a Group with some Pins.
   kicad_group_netlister uses this mode iff any of the Group's KiCad components have a `GroupPinX` field set.

2. **Implicit naming:** In other situations there might only be a single component with many pins, for example a controller. The user does not want to label all those pins, especially when the pin functions of the symbol are already unique. Then the user simply does not provide any explicit `GroupPinX` fields and kicad_group_netlister uses the pin functions as Pin Names for the Group. kicad_group_netlister enforces that all component's pin functions are set and unique.

4.2.3 Illegal KiCad Schematics

Unfortunately, the KiCad user interface allows creating schematics that kicad_group_netlister cannot convert into a Group Netlist. For example, the root schematics file may use characters the specification for Schematic forbids, e.g., `Ä`. kicad_group_netlister checks for almost all such illegal KiCad schematics and throws a helpful error message. This allows the user to correct her KiCad schematic quickly.
However there is a problem concerning Group Paths that kicad_group_netlister cannot identify: While the sheet path in a KiCad schematic has a similar structure to a Group Path, there are two issues: Firstly, the name of a hierarchical symbol, which forms the nodes of a sheet path (see subsection 2.2.2), may itself include a `/`. Take for example a hierarchical symbol of name `A/B`. Then, if the user places that symbol in the root sheet, the sheet path to the hierarchical symbol is `/A/B/`. Group Netlist consuming tools could not differentiate such a Group Path from symbol `B` inside `A` inside the root sheet. This is a problem because the user expects every node of the Group Path to represent a hierarchical symbol. If a hierarchical symbol includes a `/`, code_gen and any other Group Netlist consuming program cannot find this symbol.
Secondly, two hierarchical symbols on the same sheet may have the same name. Thus sheet paths may not be unique. This is a problem for Group Paths, which must be unique. kicad_group_netlister only has a list of all sheet paths. Unfortunately, this is not sufficient information to unambiguously identify the above issues. For example, as we have shown above, the sheet paths `/A/B/` could originate from either be a legal or illegal KiCad schematic. The user must ensure not to create such an illegal KiCad schematic:

1. Do not name two hierarchical symbols the same on the same sheet.

2. Do not use `/` inside a hierarchical symbol name.

4.2.4 kicad_group_netlister Implementation

kicad_group_netlister works using a pipeline of four steps:

1. kicad_group_netlister starts by parsing the KiCad netlist XML file and creating an in-memory representation of the interesting information.

2. The second step is to group components into Groups. See subsection 4.2.1 on what components kicad_group_netlister considers to belong to a Group. The output of this step is a lookup from Group ID to **Raw Group**. A Raw Group is an intermediary step between KiCad components and Groups. It contains a list of all KiCad components (explained in section 4.2) and all Group Map Fields that belong to this Group. Additionally, this step produces a reverse Group ID lookup; a lookup from KiCad component ref to that component's Group's ID. The KiCad pin number and component ref form the **Global KiCad Pin ID**, which uniquely identifies a KiCad pin in the entire schematic.
   This stage also performs some checks to throw an error in case of an illegal KiCad schematic as we have specified in subsection 4.2.3.

3. Thirdly, kicad_group_netlister extracts all explicit pin namings (see subsection 4.2.2). This stage's output is a lookup from Group ID and Global KiCad Pin ID to explicitly chosen Pin Name (for the Group). Notice that if a Group uses implicit pin naming, there is no entry in this lookup. kicad_group_netlister produces this lookup by only using the Raw Group lookup from the previous stage. Notice that the Raw Group lookup does not contain the information from the nets in the KiCad Netlist. Therefore and because the representation kicad_group_netlister uses for a component does not contain its pins, this stage does not have any information about implicitly. However, by looping over every component in each Raw Group and looking through their respective fields this stage manages to find all explicit Pin Names.

4. The last stage is the only stage that reads the nets from the KiCad netlist. Furthermore, it uses the reverse Group ID lookup and a list of all Raw Groups. Firstly, this stage loops over all nets and converts each KiCad net into a Net for the Group Netlist. To perform the conversion this stage loops over every node in the KiCad net that belongs to a Raw Group. The reverse Group ID lookup provides the Group ID for each such node. If the explicit pin lookup contains an entry for that Group ID and KiCad Pin ID, the respective value provides the Pin Name. Otherwise, the KiCad node's pin function does (meaning implicit Pin naming). This is the place where kicad_group_netlister ensures that the user did not mix explicit and implicit Pin naming. Now the Group Id and Pin Name form a Node in the final Net. This stage adds a Net to the Group Netlist if it contains at least one Node.
   Secondly, this stage creates a Group from each Raw Group, removing the KiCad-specific information and adding the Pins.
   Lastly, it passes the schematic filename from the KiCad netlist's metadata to the Group Netlist. This concludes the Group Netlist generation, which the Group Netlist library stringifies into an XML file (see subsection 4.1.4).

```
namespace {{ pascal_case(group.path) }} {
// snippet type: {{ group.group_type }}
typedef void Handle;
{% for pin in group.pins.keys() %}
    {% if pin.startswith("DI_") %}
        {% set gpio_pin = group.get_single_pin_to_glob(pin, root_glob) %}
        {% if gpio_pin is not none %}
            typedef modm::platform::GpioOutput{{ gpio_pin.pin[1:] }} {{ pascal_case(pin) }}Pin;
        {% endif %}
    {% endif %}
    {% if pin.startswith("DO_") %}
        {% set gpio_pin = group.get_single_pin_to_glob(pin, root_glob) %}
        {% if gpio_pin is not none %}
            typedef modm::platform::GpioInput{{ gpio_pin.pin[1:] }} {{ pascal_case(pin) }}Pin;
        {% endif %}
    {% endif %}
    {% if pin.startswith("AO_") %}
        {% set adc_pin = group.get_single_pin_to_glob(pin, adc_glob) %}
        {% set gpio_pin = group.get_single_pin_to_glob(pin, root_glob) %}
        {% if adc_pin is not none %}
            {# TODO: this is quite ugly. #}
            {% set adc_name = pascal_case(adc_pin.group_id[1])[4:] %}
            typedef {{adc_name}}::{{ pascal_case(adc_pin.pin.split("_")[1]) }} {{ pascal_case(pin) }}Adc;
        {% endif %}
        {% if gpio_pin is not none %}
            {# TODO: check if this name is correct #}
            typedef modm::platform::GpioAnalogOutput{{ gpio_pin.pin[1:] }} {{ pascal_case(pin) }}Pin;
        {% endif %}
    {% endif %}
{% endfor %}
}
```

Figure 4.6: Jina2 template for C++ modm pin mapping [13]. The main Jinja2 template file sets the `group` variable to a Group to generate C++ code for. Afterwards, it includes this template file.

4.3 Generating C++ Code with code_gen

code_gen receives the path to a Jinja2 template file and a Group Netlist XML file (see section 3.3). code_gen parses the Group Netlist and creates a `GroupNetlistWithConnections` instance using the Group Netlist library, see subsection 4.1.4. Lastly, it hands the Jinja2 template the Group Netlist alongside the `glob_groups` function. `glob_groups` returns a list of matching Groups for any Group Glob. The output of the Jinja2 template is code_gen's output.
The Jinja2 template file may use the Group Netlist library, most importantly the `get_single_pin_to_glob` member function. Figure 4.6 shows an example Jinja2 template. The template line `group.get_single_pin_to_glob(pin, adc_glob)` returns the Global Pin ID of any ADCs pin that connects to this Pin or None when there is no such ADC. This retrieves the information needed to control this snippet, via the ADC.

4.4 Merging multiple Group Netlists with group_netlist_merger

kicad_group_netlister converts exactly one KiCad schematic into one Group Netlist and code_gen consumes exactly one Group Netlist. The PLUTO EPS' firmware, however,

controls hardware located on multiple PCBs. There is an individual KiCad schematic for each PCB and, thus, an associated Group Netlist for each. These PCBs connect via connectors, each of which are a Group in their respective Group Netlist. As stated above, code_gen can only read a single Group Netlist.
Therefore, we introduce group_netlist_merger, a command line program which combines multiple Group Netlists into a single one. Besides the paths to the input Group Netlists the user provides a Group Glob (see subsection 4.1.3) matching all Groups that will be electrically connected. In the case of the PLUTO EPS the user selects the Group of the connector on each PCB. We call all Groups that match the Group Glob **Matching Groups**. They must have the same Pin Names.
Lastly, the user specifies how Matching Groups will be connected, by choosing a **Pin Mapper**. Some connectors, for example, have a female and a male version. When they mate, pins of the same name connect electrically. The PLUTO EPS, however, uses gender-neutral connectors, where a connector mates with an identical, 180°rotated connector. In this case pin 1 connects with pin 2 on the other connector, pin 3 with pin 4 and so on. Therefore, group_netlist_merger supports two Pin Mappers, `equal` and `even_odd`.

There are two steps the tool performs:

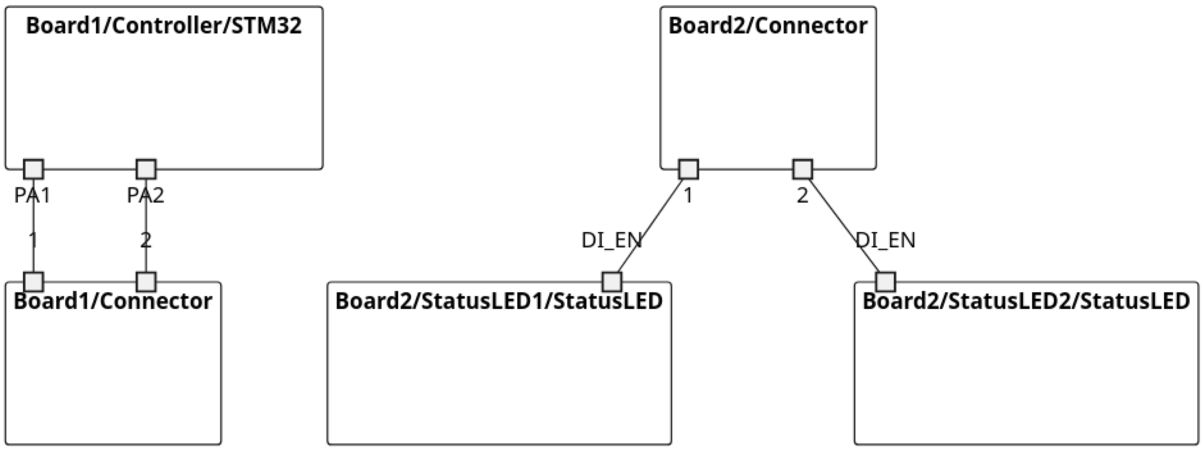
1. It creates the union of all input Group Netlists' Groups and Nets respectively. This generates a new intermediary Group Netlist, in which there are no connections between the Groups of the input Group Netlists. Figure 4.7a shows an example intermediary Group Netlist.

2. In this step it firstly finds all Matching Groups.
   Secondly, in the intermediary Group Netlist there are Nets, which should be combined. Two Nets should be combined iff both Nets contain a Pin belonging to different Matching Groups and those Pin Names should be connected according to the selected Pin Mapper. The tool defines a predicate that returns true for any two Nets iff they should be connected.
   Lastly, it loops over all Nets and looks for a second Net that should be merged according to the predicate. group_netlist_merger does this is a second, inner loop. If another net should be merged, it unions the Nodes of both Nets, adds the new Net to the Group Netlist and removes the two old Nets. Figure 4.7b shows an example final Group Netlist.

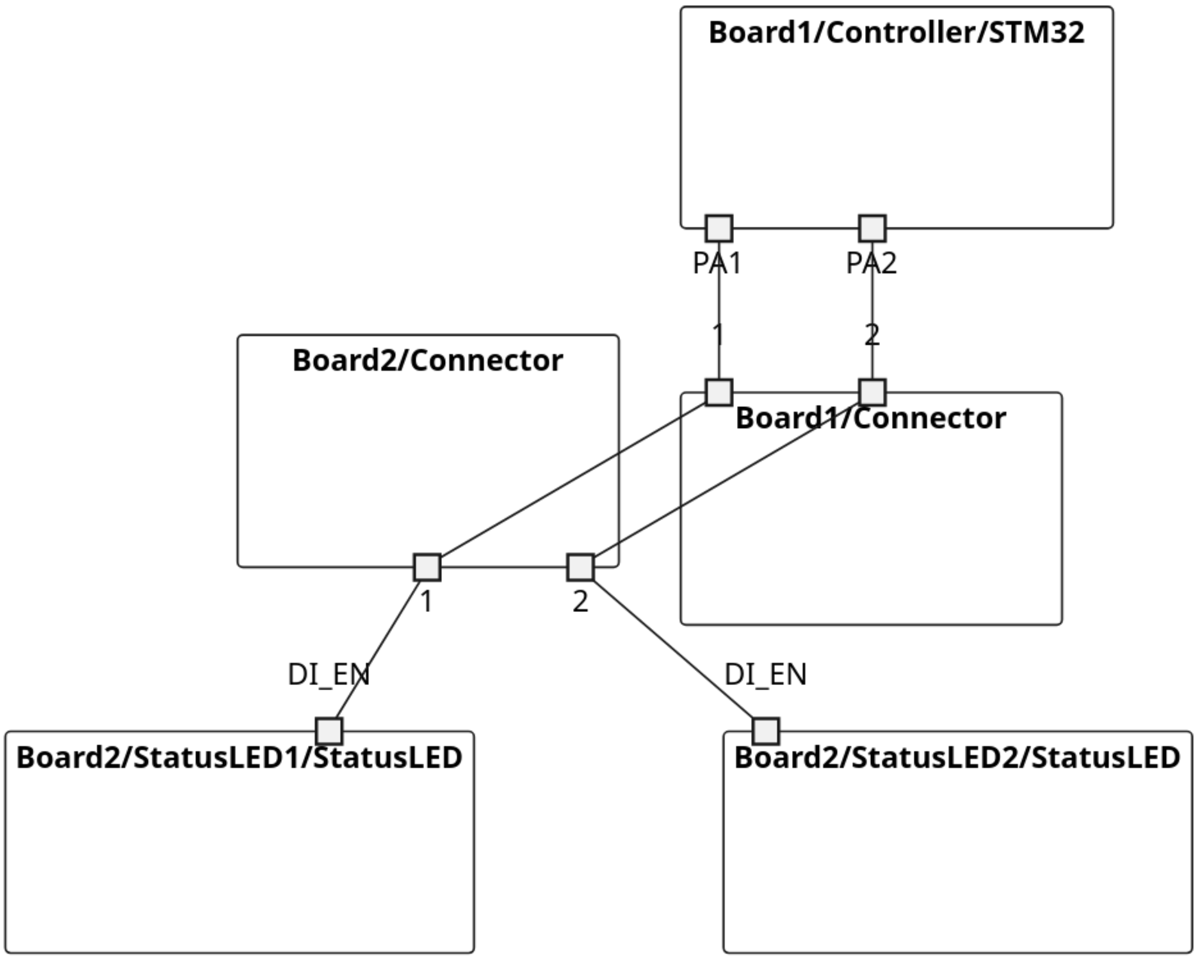
Notably the Matching Groups do not need to be spread across multiple schematics. This way, for example, a use can add jumper cables that are not part of the schematic. Furthermore, this supports having multiple connectors in parallel. The user can simply run the group_netlist_merger multiple times with different Group Globs for the different parallel connectors.

4.5 Generating a CSV with netlist_to_csv

netlist_to_csv receives the path of a Group Netlist XML and two string arguments: `--root-group-glob` and `--simplify-pins`. The output CSV contains for every Pin a



(a) Example intermediary Group Netlist directly after Step 1 of group_netlist_merger. Notice how there are two unconnected clusters of Groups.



(b) The same example as in Figure 4.7a but after completing Step 2 of group_netlist_merger using the Group Glob `Board*/Connector`. Notice how the two connector Groups connect through all their pins.

Figure 4.7: Example Group Netlist in different stages of group_netlist_merger.

list of all other Pins that connect to it.
In some cases this is a problem. For example the `GND` Net, which may span hundreds of Groups, could create a CSV which is very difficult to read by a human. Therefore, netlist_to_csv allows simplifying such Nets: `--simplify-pins` specifies a comma-separated list of Pin Names. netlist_to_csv simplifies every Net with a Pin of such a Name.
Additionally, the user may only care about certain Groups. When the user provides `--root-group-glob` the CSV only contains Groups that match this Group Glob. Furthermore, the CSV does not contain any connections between Groups that match `--root-group-glob`. Only connections from matching Groups to not matching Groups are in the CSV. This is useful, e.g., to create a harness specification: From the harness' perspective only connectors and the meaning of their pins matter. The hardware behind a connector is unimportant. Also, connections through the hardware from one connector to another connector is not interesting. Only the function of each connector pin matters.

netlist_to_csv performs four steps:

1. At first netlist_to_csv parses the Group Netlist and creates a `GroupNetlistWithConnections` instance.
2. netlist_to_csv then goes through every Group (i.e., `GroupWithConnection` instances) and their Pins. Remember that every Pin in a `GroupWithConnection` contains all other Pins it connects to. If a Pin connects to an other Pin in `--simplify-pins`, netlist_to_csv removes all other Pins from this Pin. Lastly it adds a synthetic Pin back: It belongs to the synthetic Group of ID `This_was/Simplified/Away` and has the matching element in `--simplify-pins` as Pin Name, e.g. `GND`. This way the user knows why a net is simplified.
Notice that this step returns a correct Group Netlist.
3. This step linearly filters every Group's Pin connections and the Groups themselves according to `--root-group-glob`. This step, too, returns a correct Group Netlist.
4. Lastly, netlist_to_csv creates a CSV with the columns: Schematic, Group Path, Group Type, Pin Name and `other_pins`. `other_pins` is a list of Group Pin IDs of Pins that connect to any given Pin.

Chapter 5

Development History

We chose Python and Jinja2 (section 3.3) for implementing the KiCad schematics to code generator. Firstly, DLR Pando already uses Python and Jinja2 causing less friction for the software development workflow (see DG2). Additionally, the DLR has developers for these technologies, simplifying maintenance (see DG4). Secondly, Jinja2 and Python's standard library includes all functionality we need. Therefore, all the technology we use is well maintained by the Python Software Foundation [7] and Pallets [14]. This also makes for easier maintenance (see DG4).
Thirdly, we do not expect the performance advantages of other programming languages to matter for this tooling. Our Python implementation takes three seconds for the PLUTO EPS project whenever there is a hardware change. This is acceptable.
Lastly, while C++ code generation is the target use-case for our tooling, we quickly realized there are many more applications, e.g., harness specification generation. However, we did not understand the exact requirements such use-cases impose. Therefore, we chose to rapidly iterate and quickly implement prototypes for each use-case. Python allowed us to focus on those details rather than low-level technicalities. Hence, we explored the design space exhaustively and can propose a stable specification (see DG7). If at some point the speed does become a problem, a reimplementation in ,e.g., Rust will be considerably easier. That is because this thesis already explains the requirements, working and failed high-level implementations.

Because the software development workflow uses Makefiles for compilation, we chose to create a terminal application without a graphical user interface. This eases adapting to the new workflow (see DG2) and keeps the source code simpler (see DG4). Additionally, command line parameters suffice for user input.
Every program in our generator receives the path to one or multiple input files, some parameters and prints the output to stdout. stderr carries errors and warnings to the user and we do not use stdin. This makes for a simple user interface that Makefiles easily automate.

We chose to heavily rely on Python type hints, checked by mypy [15]. While adding type hints takes time, we argue consequently using type hints sped up development. For example, we do not use the type `str` directly. Instead, we create new types, e.g., `GroupType`

`= NewType("GroupType", str)`. Firstly, this prevents accidentally using a Group Type in a place which, e.g., expects a Pin Name. Consequently, we argue that strict typing leads to more correct software. Secondly, we often started refactorings by changing or deleting types. For example, in section 5.4 we deleted the `GroupName` type and added the `GroupPath` type. All places that needed adjustments to this change created type errors, speeding up the refactoring.

Similarly, we heavily use asserts whenever there is an invariant that must hold. There are a total of 70 asserts and errors and 9 warnings in 1162 lines of code. In a correct implementation that invariant always holds. An incorrect implementation, however, would trigger the assert. Knowing which assert triggers, rather than only receiving wrong output, makes it easier to locate incorrect source code. Again, while implementing asserts takes time, they greatly speed up debugging. Furthermore, we accept doubling the execution time for some asserts. For example, whenever our tooling parses an XML file, we immediately serialise the returned in-memory representation and compare the output XML to the input XML. This ensures that both our XML parsing is the exact inverse of our serialising function and that the serialise function is deterministic. A deterministic XML output makes version control (e.g., with git) easier for the user.
Therefore, asserts and typing decisions lead to easier to maintain and adapt tooling (see DG4 and DG3).

Expectedly, the rapid iteration lead to many discarded implementations. Consequently, the final tooling we explain in chapter 4 is only the last stage of a long history. In this chapter we explains what high-level, architectural changes form this history. We chose to omit any development changes that naturally extend the previous implementation and do not require a refactor or redesign. Furthermore, we advice the reader to understand the final tooling first, as we write this chapter from that perspective. Unless we specifically mention a different ordering, we lay out the changes in chronological order.

5.1 Renaming Snippet to Group

For most of the development we used the term **Snippet** to describe a collection of KiCad components, instead of the final term **Group**. We chose this name to align with the Snippets from Snippet Based Design (section 2.5). However, at a very late stage we realised a difference between Groups and a Snippets: A Snippet is a circuit with a clear function and PCB layout. It always has its own hierarchical KiCad schematic sheet. A Group, however, is the unit in which Group Netlist consuming programs understand the schematic. Therefore, a Snippet may but does not have to be a Group. A Snippet is not a Group for example when that Snippet is not relevant for simulation or firmware generation. Also, a Group may be a Snippet but does not have to be. A connector, for example, could be a Group but not a Snippet because it is a single component not requiring a reusable PCB Snippet. There may be multiple connectors on a single hierarchical sheet.
We settled for "Group" after considering excruciatingly flamboyant constructs, backronyms and acronyms; sometimes and for good measure of the recursive kind. In comparison, performing the source code renaming with sed was surprisingly quick.
The following sections use the new term, Group, for reasons of presentation. Even though,

they describe a point in time at which we used the term Snippet and composite terms like Snippet Netlist or Snippet One-to-one/Many-to-one Map.

5.2 Choosing the kicadxml Netlist

kicad_group_netlister receives a hardware project's KiCad schematics files and generates a Group Netlist XML document, see subsection 4.1.1. At first, we considered extracting information from a running KiCad process. For this purpose, The KiCad Library Utils [4] appeared promising because the KiCad contributors officially support it. Unfortunately, the KiCad IPC API it uses does not support the Schematic Editor [3]. Additionally, the requirement of having a running KiCad instance would have made kicad_group_netlister difficult to run headlessly. However, DG2 requires a headless tool because the software engineer's workflow uses Makefiles running CLI programs.
Therefore, we considered extracting the information from the `.kicad_sch` file. KiCadFiles [17] is a third-party library claimed to parse KiCad files including schematics. However, we failed to find any public uses of this library and consider it unmaintained as its development lasted only a single month. DG4, however, requires not just that kicad_group_netlister's own code is maintainable but also its dependencies'.
KiCad Schematic API [12] is an alternative library for extracting information from the `.kicad_sch` file, which currently does receive development and maintenance. Though, it is largely written by an LLM, thus, we question its code quality. While kiutils [11] and kicad skip [5] are further alternatives that are unmaintained for a year, they have received some use. Because of that, we considered them more closely.

As described in 3, [2] uses [5] to create a custom netlist generator. We explained that the main drawback of this approach is guaranteeing parity between the custom netlist generator and KiCad's own netlist generator. We decided that proving this is out of scope for this thesis. Furthermore, even if kicad_group_netlister guarantees parity, any changes to KiCad's netlist generator would require a change to kicad_group_netlister. This, in turn, would make kicad_group_netlister laborious to maintain (see DG4).
We conclude that implementing a custom fully-featured netlist generator is difficult, complex and prone to errors. Alternatively, we could have implemented only a subset of KiCad netlist generator features and guaranteed parity for those. However, this would, again, require the electrical engineer to drastically change her workflow (see DG2).

For the above reasons, we decided against all presented libraries including the related work [2] and discarded the idea of reading the `.kicad_sch` file directly. Instead, we chose to use the KiCad CLI to headlessly generate a netlist file with KiCad's own netlist generator. The KiCad CLI can export multiple netlist formats, including the KiCad specific `kicadsexpr` and `kicadxml` formats but also formats for other software, like SPICE. We chose the `kicadxml` format. Firstly, the `kicadxml` format is the input for so called Netlist Exporters, which convert the netlist into documents for other software. The Schematic Editor allows adding these Netlist Exporters to the user interface, which includes kicad_group_netlister, because we chose the `kicadxml` format. While the software engineer can use kicad_group_netlister from the command line, the electrical engineer can also use it directly from the Schematic Editor's user interface as a Netlist Exporter, see

DG3. We expect this to be useful as the electrical engineer may want to check the schematic annotations kicad_group_netlister requires, see section 4.2. Furthermore, the alternative `kicadsexpr` format requires using a third-party library like kinparse [16] or kicad skip [5], which reduces maintainability (see DG4). `kicadxml`, however, can be parsed by the python standard library, again, making kicad_group_netlister more easily maintainable (see DG4). Lastly, we chose to only use the `kicadxml` netlist as input and not also use the `.kicad_sch` file. This means that kicad_group_netlister both receives and generates only an XML document, resulting in a simpler program (see DG4).

5.2.1 KiCad Netlist Limitations

The major disadvantage of kicad_group_netlister only using the KiCad netlist and not the `.kicad_sch` schematics file is the lack of net labels. Unfortunately, the KiCad netlist does not contain the name all net labels.
The use case of hierarchical sheets and Snippets exemplifies this problem: A hierarchical sheet uses hierarchical labels to specify what nets the surrounding circuitry may connect to. Specifically, hierarchical sheets represent Snippet. Therefore, the Snippet's Pins are the hierarchical sheet labels. The `.kicad_sch` schematics files does contain this information. Therefore, a theoretical implementation of kicad_group_netlister using the `.kicad_sch` file could use the hierarchical labels as Group Pins. Our implementation of kicad_group_netlist, however, does not have this information and, thus, cannot use hierarchical sheet labels as Group Pins.
Instead, it requires the user to annotate symbols with the `GroupType` field to define what Pins a Group has.
The canonical workaround is to introduce the `Group_IO` symbol shown in Figure 5.1. It has a single pin and the `GroupType` and `GroupPin1` fields set. Because it is a symbol without any component, it is not part of the hardware and only exists in the schematic. This approach is prone to errors as the user must ensure that the hierarchical label names exactly match the `Group_IO` Pin Names.

While the `.kicad_sch` file does contain all label names, linking those names with the information in a KiCad Netlist is not trivial. That is because `.kicad_sch` only contains the position and name of each label. The KiCad netlist, however, does not contain the position information. Therefore, it is not possible to determine which label connects to which net without reimplementing the netlist generation. In section 5.2 we argue why we chose against doing so.

An alternative solution is to change the KiCad netlist file specification to include all label names. As of KiCad 9.0.2 every net in a KiCad netlist has a single name. While many labels might connect to this net, KiCad assigns one of the label's names to the net according to fixed rules [8]. We propose modifying the KiCad netlist specification to include all label names for each net. This could allow a kicad_group_netlister implementation that solves the above problem without reading the `.kicad_sch` file. Because KiCad is Open-Source software, future work could implement this feature.

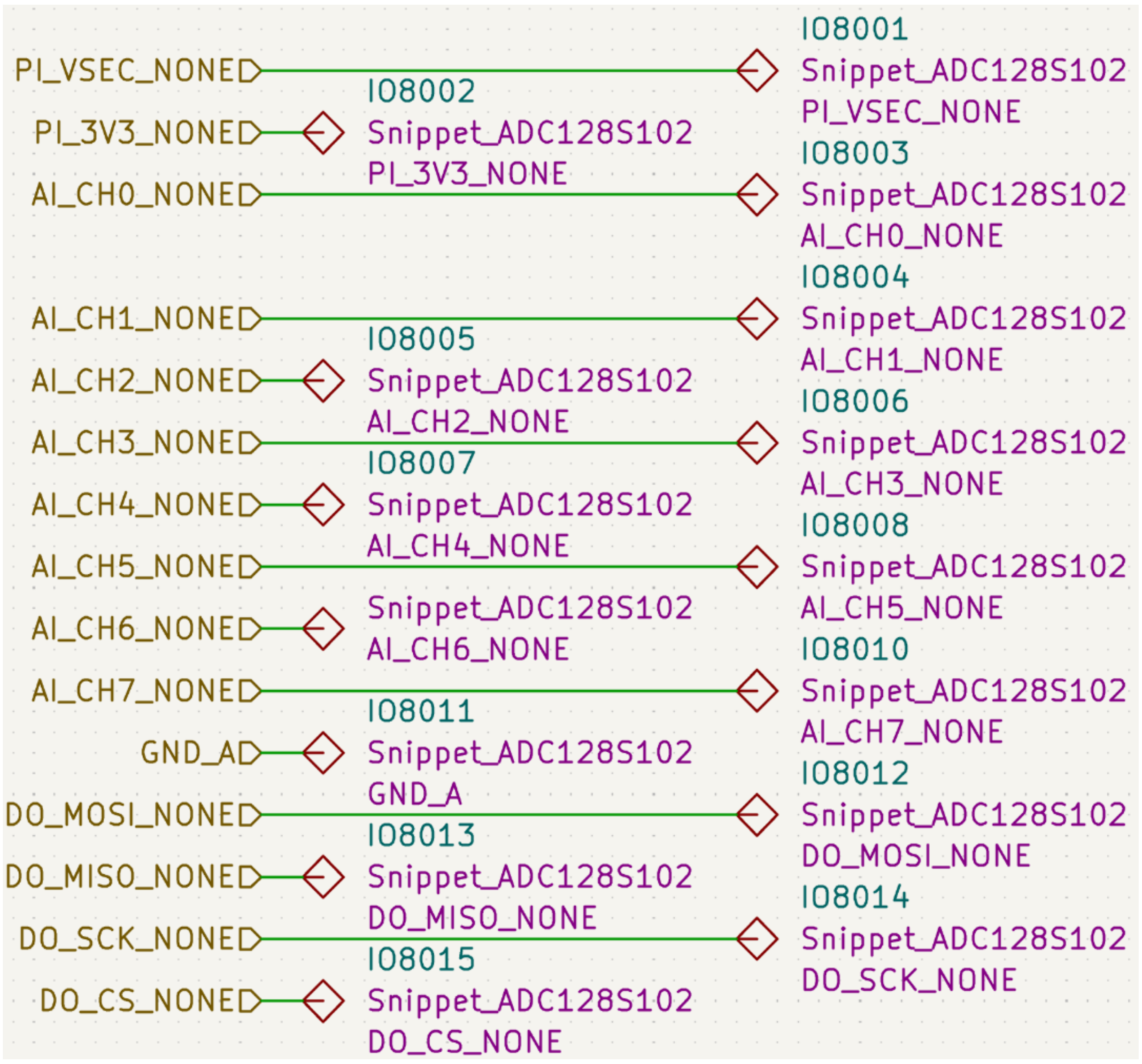

PI_VSEC_NONE
PI_3V3_NONE
AI_CH0_NONE
AI_CH1_NONE
AI_CH2_NONE
AI_CH3_NONE
AI_CH4_NONE
AI_CH5_NONE
AI_CH6_NONE
AI_CH7_NONE
GND_A
DO_MOSI_NONE
DO_MISO_NONE
DO_SCK_NONE
DO_CS_NONE
IO8001 Snippet_ADC128S102 PI_VSEC_NONE
IO8002 Snippet_ADC128S102 PI_3V3_NONE
IO8003 Snippet_ADC128S102 AI_CH0_NONE
IO8004 Snippet_ADC128S102 AI_CH1_NONE
IO8005 Snippet_ADC128S102 AI_CH2_NONE
IO8006 Snippet_ADC128S102 AI_CH3_NONE
IO8007 Snippet_ADC128S102 AI_CH4_NONE
IO8008 Snippet_ADC128S102 AI_CH5_NONE
Snippet_ADC128S102 AI_CH6_NONE
IO8010 Snippet_ADC128S102 AI_CH7_NONE
IO8011 Snippet_ADC128S102 GND_A
IO8012 Snippet_ADC128S102 DO_MOSI_NONE
IO8013 Snippet_ADC128S102 DO_MISO_NONE
IO8014 Snippet_ADC128S102 DO_SCK_NONE
IO8015 Snippet_ADC128S102 DO_CS_NONE

Figure 5.1: Annotating KiCad net labels with `Group_IO` symbols. A kicad_group_netlister implementation reading the `.kicad_sch` file directly or modification to KiCad could remove the need for this.

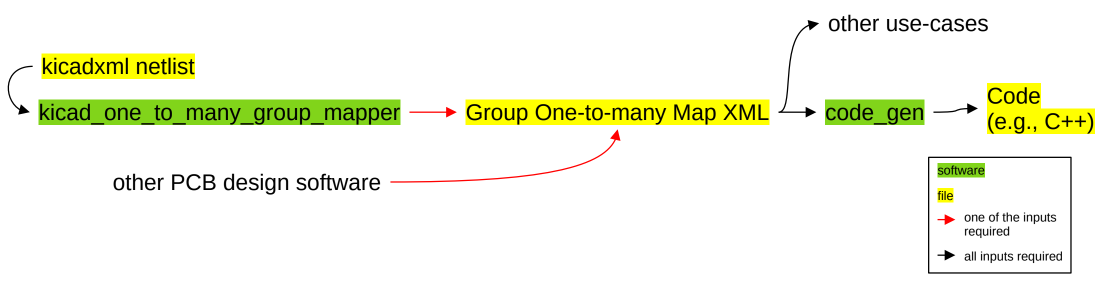

kicadxml netlist
kicad_one_to_many_group_mapper
Group One-to-many Map XML
other PCB design software
other use-cases
code_gen
Code (e.g., C++)
software
file
one of the inputs required
all inputs required

Figure 5.2: Overview of the first generator design. We hide some of the external tools shown in Figure 4.1.

5.3 Choosing to implement multiple Programs

Early in development we chose to split the firmware generator into two separate programs: kicad_one_to_many_group_mapper and code_gen. The interface between was the **One-to-many Group Map**, which we specify in subsection 5.3.1. Figure 5.2 shows the pipeline architecture at this stage in development. In our final tooling we replace the One-to-many Group Map with the Group Netlist but keep the pipeline architecture, see Figure 4.1.
This split allowed us to complete a prototype of kicad_one_to_many_group_mapper before starting the implementation of code_gen. The prototype, in turn, allowed us to quickly test kicad_one_to_many_group_mapper's design decisions. We were quickly confident that the design will fulfill the functionality (see DG1) and does not impede the electrical engineer's workflow (see DG2). This aided in our goal to rapidly iterate.

5.3.1 One-to-many Group Map

During early development we expected code_gen to only require information to answer the question: Which Groups connect directly to the controller and how? This proved incorrect as we explain in section 5.9.
While we used the Group Netlist and One-to-many Group Map for the same purpose, as a common interface, they contain different information: To generate a One-to-many Group Map the user provides the ID of a special Group, the **Root Group**. In the case of firmware generation the Root Group is the controller's Group, which runs the firmware. Then the One-to-many Group Map only represents connections to the Root Group, e.g., controller. For example, the Group Netlist shown in Figure 4.2 also expresses how the ADC's channels (`AI_CH0`, `AI_CH1`, `AI_CH2`) connect to other Groups. A One-to-many Group Map would not express those connections because they do not connect to the Root Group, the controller.
This design leads to a simpler specification (see DG8) and incorrectly expected the limitation to not matter to code_gen.

Like with the Group Netlist we chose to serialize the One-to-many Group Map using XML. Nevertheless, while the final Group Netlist XML represents connections through a list of Nets, the One-to-many Group XML does not use the concept of nets. Instead, the

One-to-many Group Map contains a list of all Groups, similar to how the Group Netlist represents Groups. However, for every Group it does not only provide the Group ID, Group Fields and Pin Names. Instead, for every Group's Pin that connects directly to the Root Group there is extra information. This information specifies which Pin of the Root Group it connects to.
Furthermore, the One-to-many Group Map XML contains the information of which Group is the Root Group. The Root Group is not part of the list of other Groups but has its own XML tag. Notably, we chose to serialize the Root Group exactly the same as any other Group. This created an easier to comprehend file (see DG8) specification and eases maintenance (see DG4).

5.4 Structuring Groups into a Hierarchy with Group Path

The initial version of the One-to-many Group Map used **Group Name**s instead of Group Paths. While kicad_one_to_many_group_mapper already used KiCad's sheet path, our specification treated it like an opaque data type. However, we realized that code_gen needs to understand the structure of Groups. The PLUTO EPS schematics, for example, uses a hierarchy of sheets. E.g., there is a sheet for the 3,3V distribution, which contains a sub-sheet for each LCL and DCDC Snippet. code_gen needs to be able to understand whats Groups belong to one power distribution. Nevertheless, simply allowing to group Groups will reach its limits, too (see DG3). Therefore, we chose to replace the Group Name with the Group Path. This allows creating a hierarchy of Groups, which we explain in 4.1.2).

code_gen intermittently relied more heavily on this Group hierarchy. In this discarded implementation the template had to walk the path tree to reach any Group. This provides an advantage when the user wants to reuse templates for widely different hierarchies. However, differing hierarchies also mean widely differing hardware designs. Such differences also require separate templates. Therefore, we do not expect templates to benefit from such a path walk to warrant its complexity. Instead, using Group Globs to retrieve Groups (see subsection 4.1.3) is much less of an ordeal than an over-engineered tree walk. The PLUTO EPS' template (see section 5.10) shows this.

5.5 Python Library

Initially, kicad_one_to_many_group_mapper directly included XML serialising and code_gen included XML parsing functions. However, we realized that there will be other Python tools using the shared XML format. Therefore, we refactored the source-code to form an independent library including the KiCad-independent in-memory representations and XML handling.

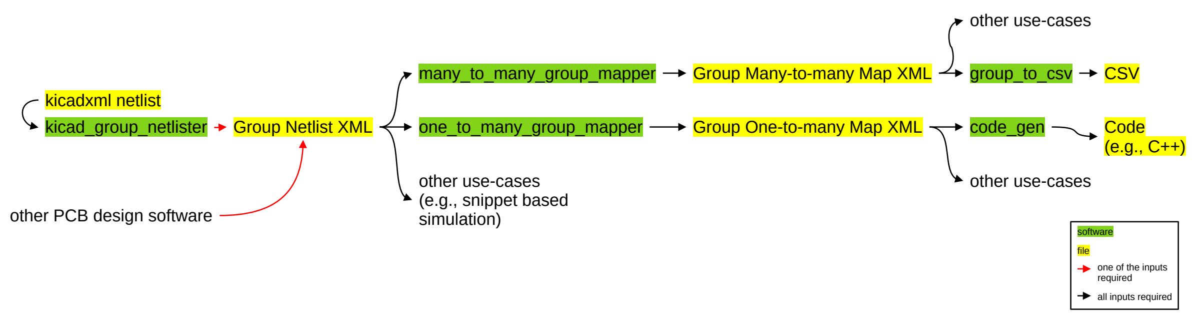


Figure 5.3: Overview of the second generator design. We hide some of the external tools shown in Figure 4.1. The Group Netlist and Maps are XML documents. Notice how there are three file formats we would have exposed to other use-cases. In the final design there is only one, the Group Netlist XML.

5.6 Group Netlist

We realized that our tooling has another use-case besides C++ code generation: Snippet-based simulation (see subsection 2.5.1). Snippet-based simulation requires a netlist of all Snippets, not a One-to-many Group Map. The kicad_one_to_many_group_mapper's intermediate in-memory representation of a schematic already resembled a netlist. Therefore, extracting this information in the form of an abstract netlist was simple and we specified the Group Netlist XML format. We considered extending kicad_one_to_many_group_mapper to output both a Group Netlist and One-to-many Group Map. However, we decided against this approach and, instead, extended the pipeline: We replaced kicad_one_to_many_group_mapper with kicad_group_netlister and one_to_many_group_mapper. The interface between was the Group Netlist XML file. Figure 5.3 shows this pipeline and other programs we have not explained yet. The decision to extend the pipeline kept the individual programs small and simple.

5.7 Many-to-many Group Map and CSV

After extending the use-cases from only C++ code generation to code and Group Netlist generation, we found another use-case: harness specification. The PLUTO satellite consists of multiple subsystems, each with their own PCB or stack of PCBs. These subsystems have connectors allowing a harness to connect the subsystems together. The harness engineer creates a specification for each harness by reading the schematics of each subsystem. While the harness engineer did not design the schematics, she must understand what each connector pin's function is. Currently the harness engineer must manually trace the cables in the schematic and understand what components lie behind the connectors. Our tooling already extracts that information, in the form of a Group Netlist. Because the harness engineer needs the connectors' perspective, which comprises of multiple Groups, the One-to-many Map proves useless. Therefore, we introduced the **Many-to-many Group Map**, which exactly matches the `GroupNetlistWithConnections` class we explain

in section 4.5. We implement many_to_many_group_mapper for this. It also took care of simplifying and filtering the Many-to-many Group Map to only include the information the harness engineer requires, which we explain in section 4.5. Lastly and because the harness engineer works with spreadsheets, we implemented group_to_csv to convert the Many-to-many Group Map XML into a CSV.
Though, we used the term One-to-many Group Map in this chapter before, we only named it that after introducing the Many-to-many Group Map. Before, we simply called it "Group Map", not expecting another type of Group Map. Also, notice that while the Many-to-many Group Map contains the same information as a Group Netlist, the One-to-many Group Map typically only contains a subset. Thus, converting a Many-to-many Group Map back into a Group Netlist is possible (if the user did not filter or simplify) but the same is typically impossible with a One-to-many Group Map.
At this point we had three different XML formats, see Figure 5.3. We understood that three file formats is too many to maintain (see DG4). Nevertheless, at this point we did not understand the requirements of other use-cases well enough to combine any programs and remove a format.

5.8 Merging multiple Schematis

The PLUTO EPS does not consist of a single PCB but a stack of two: the motherboard and SOLBAT daughterboard. At first we only considered extracting information from the motherboard's schematics. However, there are multiple components on the daughterboard of the controller's interest. These components connect through connectors to the motherboard. In order for code_gen to create firmware for those components, the Group Netlist must contain the Groups those components form. Therefore, we implemented group_netlist_merger (see section 4.4).
We considered using multiple KiCad netlists instead of Group Netlists as input to the merging but ultimately decided against it. Firstly, decoupling the merging of multiple schematics from the KiCad specific kicad_group_netlister means other PCB design software can use group_netlist_merger, too (see DG3). Secondly, extracting information from KiCad netlists and performing actions on that data are conceptually different operations. Separating them creates a much more loosely coupled system aiding in maintenance (see DG4).
Furthermore, merging the One-to-many or Many-to-many Group Map would not permit all use-cases. That is because as Figure 5.3 shows we have not implemented a tool to convert any Group Map back into a Group Netlist. Implementing those conversions would be impossible or unnecessarily complex.
Lastly, we initially implemented the merging inside the many_to_many_group_mapper but moved it into its own program for the above reasons.

After a merge there are Groups from different schematics in the same Group Netlist. Because the previous Group ID contains only the Group Path and Group Type, there was the risk of having two Groups with the same ID. Therefore, we introduced the Schematic. The Schematic together with the Group Path and Group Type forms the final Group ID (see subsection 4.1.2), retaining uniqueness after merging.
Alternatively, group_netlist_merger could rename Groups when their Group IDs are not

unique after merging. This, however, would be more complex and error-prone than ensuring all Groups have unique Group IDs across schematics and projects; even if they never merge.

5.9 Transitive Group Connections

During early development we expected code_gen to only require information to answer the question: Which Groups connect directly to the controller and how? This proved incorrect because there are **Transitive Group Connections**: Sometimes a Group does not directly connect to the controller Group running the firmware. Instead, there is an intermediary Group. An example is the connection between the DCDC and ADC in Figure 4.2. The DCDC's `AO_TEMP` Pin does not directly connect to the controller but to the ADC, which in turn does connect to the controller. As we explain in subsection 5.3.1 the One-to-many Group Map does not express this connection. However, the firmware needs to understand if a component connects to what ADC through which channel. Therefore and because the Many-to-many map already existed, we reimplemented code_gen to use the Many-to-many instead of the One-to-many Group Map.

At this point we would introduce three file formats and five programs (see Figure 5.3). That amount would be unnecessarily large and difficult to maintain (see DG4). Therefore, we performed a set of simplifications:

1. At this point there remained no program to use the One-to-many Group Map. Consequently, we assume future tooling to not need this file format, either, and dropped the One-to-many XML file format and one_to_many_group_mapper.

2. We implemented the `GroupNetlistWithConnections` class by moving logic from many_to_many_group_mapper into into our common Python library. This removed the need for the Many-to-many Group Map because code_gen and the CSV generation can read the Group Netlist directly and use the common Python library. Therefore, we removed the Many-to-many Group Map.
   Because the Many-to-many Group Map requires quadratic memory compared to the Group Netlist, the corresponding XML is very large. Removing it conserves resources.

This lead to the final implementation with one file format and four programs, which we explain in chapter 4 and show in Figure 4.1. Most importantly, there is only one file we specify. Therefore, any future tooling must use this file format creating a simpler, easier maintain and extend ecosystem (see DG4, DG3 and DG7).