

Automatic Firmware Generation for Spaceflight Hardware based on Schematics

Bachelorarbeit
von

Christopher Besch

am Karlsruher Institut für Technologie (KIT)
Fakultät für Informatik
Institut für Technische Informatik (ITEC)
Chair for Embedded Systems (CES)
für DLR Institute of Space Systems

Erstgutachter:	Prof. Dr. Jörg Henkel
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuer:	Dr. Hassan Nassar (CES), Janis Sebastian Häseker (DLR)

Tag der Anmeldung:	03.11.2025
Tag der Abgabe:	10.02.2026

Chair for Embedded Systems



Erklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Die verwendeten Quellen und Hilfsmittel sind im Literaturverzeichnis vollständig aufgeführt.

Karlsruhe, den 10.02.2026

Christopher Besch

Summary

Any satellite's Power Conditioning and Distribution Unit (PCDU) fulfills the same purpose, providing the subsystems with electrical power. However, because each satellite has a unique set of subsystems, differing PCDUs must realize differing power distributions. How do you efficiently develop firmware for such similar hardware?

One approach is to group components and their PCB layout into a snippet, e.g., an Analog to Digital Converter (ADC) with supportive components. Each PCDU can reuse the same snippets arranged differently to create different power distributions. While this drastically speeds up the hardware design process, previous works failed at translating this speedup to the firmware development process.

We propose the Group Netlist, a novel representation of the logical functions of hardware. This machine-readable file format enables automatic firmware generation while remaining agnostic to both programming-language and schematics editor. The electrical engineer annotates the schematics to produce not just the PCB layout but also the Group Netlist. Contrary to initial expectations, we show that the Group Netlist has uses beyond firmware generation. This includes but is not limited to harness specification and snippet based design analysis. We provide the reference implementation `kicad_firmware_generation` as a ready-to-use product for different projects even outside space-exploration.

Contents

Contents	1
1 Introduction	5
2 Background	7
2.1 Payload Under Test Orbiter (PLUTO)	7
2.1.1 Mission Control Center (MCC)	7
2.1.2 Telemetry and Telecommand (TMTC)	7
2.1.3 Latching Current Limiter (LCL)	8
2.1.4 Serial Peripheral Interface (SPI)	8
2.1.5 Analog to Digital Converter (ADC)	8
2.1.6 Electrical Power Subsystem (EPS)	8
2.1.7 Power Conditioning and Distribution Unit (PCDU)	8
2.2 Electronics Design Process	9
2.2.1 KiCad	9
2.2.2 Schematic Editor Hierarchical Structure	11
2.3 Firmware Development Process	12
2.4 Harness Specification Process	12
2.4.1 Harness Designer (HaDes)	12
2.5 Snippet Based Design	12
2.5.1 Snippet based Design Analysis	14

2.6	sed	15
2.7	Jinja2	15
3	Related Works	17
3.1	DLR Institute of Space System's Pando	17
3.2	C/C++ Stepper Motor Drivergeneration from Schematics	18
3.3	Generate pin definition header file from KiCad schematic	18
3.4	Generating Platform Configuration from Netlists	19
3.5	Safe and Reusable Approach for Pin-to-Port Assignment in Multiboard FPGA Data Acquisition and Control Designs	19
4	Design Goals and Specification	21
4.1	Group Netlist	23
4.1.1	Group Netlist XML Specification	25
4.1.2	Group ID	28
4.1.3	Group Glob	28
4.1.4	Group Netlist Python Library	29
4.2	Extracting Group Netlists from KiCad using kicad_group_netlister	29
4.2.1	Grouping KiCad Components to Groups	30
4.2.2	Pin Naming	30
4.2.3	Illegal KiCad Schematics	32
4.2.4	kicad_group_netlister Implementation	32
4.3	Generating C++ Code with code_gen	33
4.4	Merging multiple Group Netlists with group_netlist_merger	34
4.5	Generating a CSV with netlist_to_csv	36
5	Development History	39
5.1	Renaming Snippet to Group	40

5.2	Choosing the <code>kicadxml</code> Netlist	41
5.2.1	KiCad Netlist Limitations	43
5.3	Choosing to implement multiple Programs	43
5.3.1	One-to-many Group Map	45
5.4	Structuring Groups into a Hierarchy with Group Path	45
5.5	Python Library	46
5.6	Group Netlist	46
5.7	Many-to-many Group Map and CSV	46
5.8	Merging multiple Schematis	47
5.9	Transitive Group Connections	48
5.10	Publishing as Open-Source	49
6	Deployment and Evaluation	51
6.1	Integration in DLR's PCPU Design Workflow	51
6.1.1	Annotating PLUTO PCPU Schematics	52
6.1.2	Generating partial PLUTO PCPU Firmware	52
6.1.3	Generating a Harness Specification	54
6.1.4	Snippet based Design Analysis	55
6.2	Benchmark	55
6.3	Future Work: Modifying KiCad to Simplify Annotations	55
7	Conclusion	57
	List of figures	59
	Bibliography	61

Chapter 1

Introduction

The DLR Avionics Systems designs the PCDU for each satellite using modular Snippets (circuit blocks implementing some function with well defined interfaces; see section 2.5). Because different PCDUs reuse the same Snippets, this modularization greatly accelerates the hardware development process. The firmware development, however, does not benefit in the same way. Even though each PCDU's firmware shares many similarities, the firmware is different enough so that the software engineer has to write the firmware by hand every time. This thesis' goal is to develop tooling to accelerate the software engineer process. The tooling should deduce what Snippets a PCDU uses and how they connect from its KiCad schematics. Afterwards the tooling should automatically generate C++ firmware.

Additionally, the harness specification process also requires much manual labor (see section 2.4). It could benefit from schematics based automation, too. Furthermore, Snippet based Design Analysis is a prospective research topic, which requires information from the schematics (see subsection 6.1.4). While generating C++ firmware is our main focus, we extend our scope to include harness specification and Snippet based Design Analysis.

Because the hardware's schematics contain the entire hardware specification, it also contains much irrelevant information. A major challenge to our work is to filter the information relevant for our use-cases. This is difficult, firstly, because of KiCad schematics limitations and, secondly, because the use-cases' requirements remain unclear.

Therefore, this thesis is the first in-depth experimentation in what requirements our use-cases have. We propose the Group Netlist (see section 4.1), a file format representing an abstracted view of the hardware. Furthermore, we propose to add annotations to the schematics during the hardware design phase. It is the electrical engineer's responsibility to both design the hardware and annotate the schematics for further use.

We implement four programs, `kicad_group_netlist` (see section 4.2), `code_gen` (see section 4.3), `group_netlist_merger` (see section 4.4) and `netlist_to_csv` (see section 4.5). These create, combine and use the Group Netlist solving our use-cases.

Chapter 2

Background

2.1 Payload Under Test Orbiter (PLUTO)

PLUTO (Payload Under Test Orbiter) is a satellite in the 6U CubeSat form factor. Its mission is to verify novel technologies in low earth orbit. This, e.g., includes a deployable 100W solar array. The DLR Avionics Systems develops, maintains and monitors PLUTO's mission [27].

While the hardware of any spacecraft might exist multiple times, often including an engineering model that remains on earth, there is only one flight model. At the time of writing we expect PLUTO's flight model to launch into space in Q2 of 2026.

2.1.1 Mission Control Center (MCC)

A Mission Control Center (MCC) is a ground-based facility monitoring and commanding the mission of a spacecraft.

2.1.2 Telemetry and Telecommand (TMTC)

Telemetry and Telecommand (TMTC) is the basic data flow between a spacecraft and its MCC. Spacecraft's send telemetry to its MCC to relay its state and environmental conditions. Telecommands command the spacecraft to perform some action, e.g., powering a subsystem up or down. The ground station's uplink carries telecommands from the MCC to the spacecraft while the downlink returns telemetry from the spacecraft back to the MCC [12].

Both the MCC and the satellite must adhere to the same TMTC protocol. See section 3.1 on how the DLR Avionics Systems ensures this.

2.1.3 Latching Current Limiter (LCL)

A Latching Current Limiter (LCL) is a load switch with overcurrent protection when placed in between a power source and load. When the load draws too much current, the LCL initially lowers the voltage until it completely latches off the power[10]. The LCL then awaits for a latch-on command. Retrigger LCLs (RLCLs) autonomously latch on after a specified time duration.

An LCL may consist of multiple components (see the PLUTO Electrical Power Subsystem in subsection 2.1.6).

2.1.4 Serial Peripheral Interface (SPI)

The Serial Peripheral Interface (SPI) is a digital interface allowing a controller to send and receive data from multiple peripherals. It uses five pins, Controller In Periphery Out (CIPO, formerly MISO), Controller Out Periphery IN (COPI, formerly MOSI), Source Clock (SCK) and Chip Select (CS).

2.1.5 Analog to Digital Converter (ADC)

An Analog to Digital Convert (ADC) reads an analog potential and transmits a corresponding digital value. The ADC128S102, for example, transmits the measured potentials via SPI. Figure 6.1 shows the schematics of an ADC128S102 with surrounding components.

2.1.6 Electrical Power Subsystem (EPS)

The Electrical Power Subsystem (EPS) is the powerhouse of the satellite. It facilitates the electrical power requirements of all satellite subsystems. Consequently, it is a critical part of any spacecraft. The famous Apollo 13 Main B Bus Undervolt incident shows what dangers a compromised EPS poses to the spacecraft's mission (see the CSM EECOM channel at Nasa [21] starting at mission time 55:54).

2.1.7 Power Conditioning and Distribution Unit (PCDU)

The Power Conditioning and Distribution Unit (PCDU) is part of an EPS. A PCDU receives input power and distributes it among the spacecraft's subsystems. In contrast, the EPS also consists of the battery and power source (i.e., fuel cell, radioisotope thermoelectric generator (RTG) or solar array). This thesis concerns itself with PLUTO's PCDU.

PLUTO's PCDU consists of two Printed Circuit Boards (PCBs), the Distribution and SOLBAT boards, which connect using a 60-pin connector. The Distribution board houses the PCDU's controller, an STM32. PLUTO's on-board computer receives commands

via an S-Band uplink from the MCC on earth. The PCDU's controller receives relevant telecommands through a CAN interface. Those most importantly include requests to power subsystems up or down. It does so through a series of LCLs on the Distribution board, one for each subsystem.

Furthermore, the controller collects telemetry using ADCs on both the Distribution and SOLBAT board. The ADCs connect to relevant potentials across both boards. The controller sends the telemetry back to the on-board computer through the CAN interface, which in-turn performs the downlink to earth. Figure 4.2 shows a simplified view of PLUTO's PCDU.

Importantly, the PCDU for different satellites developed by the DLR Avionics Systems performs the same purpose. The DLR Avionics Systems only adapts them to the specific needs of any satellite's subsystems. Both the LCLs and ADCs consist of multiple components.

2.2 Electronics Design Process

Electronics hardware, including PLUTO's PCDU, consist of multiple components, e.g., resistors, microcontrollers and capacitors. These components only perform their purpose if the electrical engineer combines them into an electrical circuit. For this purpose, the electrical engineer designs a PCB and places the components on it (see Figure 2.1a) [3]. Each component exposes pins, which the PCB connects to other components' pins. The PCB consists of multiple layers (see Figure 2.1c) with conductive and insulating sections that form the desired electrical connection. Therefore, the engineer designs the PCB by deciding which sections should conduct or insulate. Figure 2.1b shows how a PCB connects the pins of a component with a trace, a conductive section on the PCB. Typically, the electrical engineer does not directly design the PCB but creates schematics first. The schematics show what components should connect through which pins (see subsection 2.2.1).

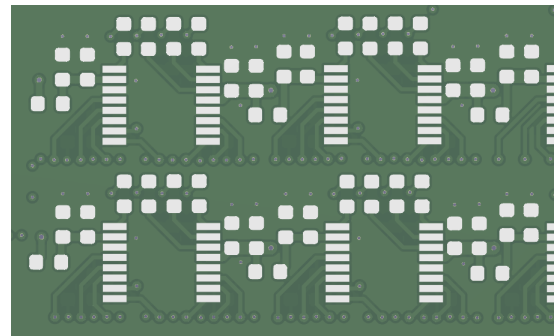
2.2.1 KiCad

KiCad is a cross-platform and open-source PCB design suite [16]. It offers two main editors: the Schematic Editor and the PCB Editor. The electrical engineer starts with the Schematic Editor. Here the electrical engineer designs the schematics, i.e., defines what components should be placed on the PCB and through which pins they should connect. The electrical engineer does so by placing symbols on a virtual sheet of paper and drawing wires. Each symbol represents a component on the final PCB and has a unique ref. If two symbol's pins overlap with the endpoints of connected wires, both pins belong to the same net. Figure 2.2a, for example, shows how a resistor with ref R10012 connects to another component through wires. Therefore, the resistor's pin and the pin on the other component belong to the same net. Generally, a net is a set of component pins that should electrically connect on the PCB.

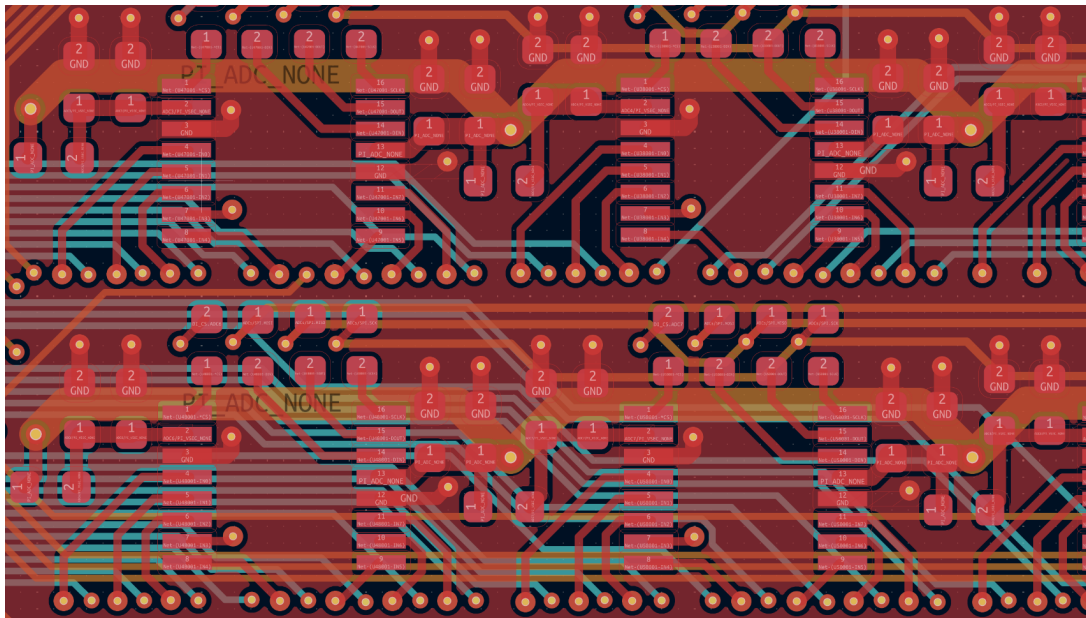
There also exist other ways of connecting pins to a net, including buses and labels: Buses represent multiple parallel electrical connections with a single line and labels connect



(a) An assembled PCB. Notice that this PCB is unrelated to PLUTO.

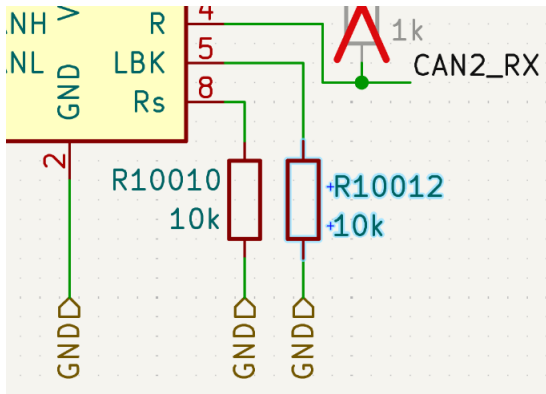


(b) PCB closeup without components. The conductive sections are in a lighter green tone than the insulating sections. The thin light-green lines are traces.

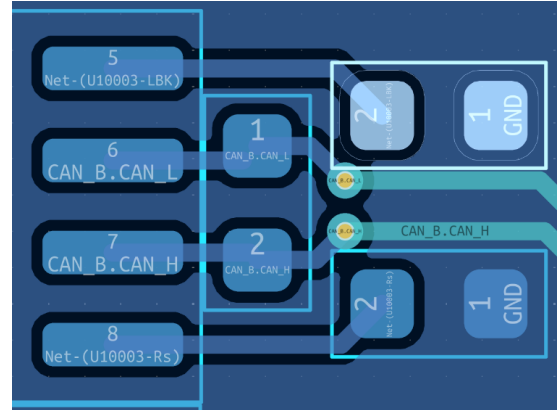


(c) The same closeup as Figure 2.1b but in KiCad's PCB Editor. Notice how the traces overlap as they belong to different layers. We do not show all layers in this figure.

Figure 2.1: PCBs.



(a) The resistor highlighted in the Schematic Editor connected to another component on the left and the GND label with wires.



(b) The resistor highlighted in the PCB Editor connected to another component with a trace on the left and the blue GND plane on the right.

Figure 2.2: A resistor connected to another component and the GND net.

wires without drawing any lines between them. We explain a specific type of label, the hierarchical label, in subsection 2.2.2.

Notice that the electrical engineer does not define where the components and traces should be. Instead, the electrical engineer does so using the PCB Editor once she completed the schematic. The PCB Editor receives a list of all nets, called netlist, from the Schematic Editor and allows the electrical engineer to decide how to realize the desired connections. Figure 2.2b shows the same resistor from Figure 2.2a in the PCB Editor. Finally, the PCB Editor produces manufacturing files printing the PCB requires.

2.2.2 Schematic Editor Hierarchical Structure

PLUTO's PCDU contains the same group of components multiple times. Thus, the schematics need to contain these structures more than once. To avoid tedious manual repetition, the Schematic Editor offers hierarchical sheets and hierarchical symbols, shown in Figure 2.3c: There always is a single root sheet, which may include other sheets, called hierarchical sheets (see Figure 2.3a). In the root sheet there may be hierarchical symbols, shown in Figure 2.4a, representing the included hierarchical sheets. A hierarchical symbol exposes pins, which connect to the hierarchical labels in the associated hierarchical sheet, shown in Figure 2.4b. The hierarchical sheets, in turn, may include further hierarchical sheets.

The root sheet and all other sheets are stored in their own `.kicad.sch` file on the file system. Multiple hierarchical symbols may link to the same file, removing redundant files. Therefore, the same file may appear on multiple pages in the entire schematic. While two pages from the same file have exactly the same structure, the Schematic Editor adjusts the component's refs so that they remain globally unique.

Furthermore, like all symbols every hierarchical symbol has a name. Joining these names with slashed (/) forms a sheet path. The root sheet has the path /. If a sheet with arbitrary path /X/ contains a hierarchical symbol of name Y, the sheet path of the included sheet is

/X/Y/. Every sheet path has / as a prefix and suffix. Notice, however, that two hierarchical symbols may have the same name. Therefore, sheet paths are counterintuitively not unique. Furthermore, hierarchical sheet names may itself include the character /. This is an issue we explain in subsection 4.2.3.

2.3 Firmware Development Process

The PCDU controller runs firmware to perform its job (see subsection 2.1.7). While the electrical engineer creates the hardware, the software engineer writes the PCDU's firmware. The most relevant parts of firmware for this thesis include

- board definitions, which state what controller pin performs which action,
- LCL driver code, which controls the LCL (i.e., switches an LCL up or down), and
- housekeeping code, which reads telemetry through the ADCs or digital status information directly via the microcontroller (see subsection 2.1.2).

2.4 Harness Specification Process

A spacecraft consists of multiple subsystems, like EPS, Attitude and Orbit Control Subsystem (AOCS) and science payload. The systems engineer specifies what subsystems there are and what interfaces they each have. The wire harness is the electrical framework connecting the respective interfaces together. Every subsystem uses connectors to realize its interfaces. The harness attaches to those connectors [11].

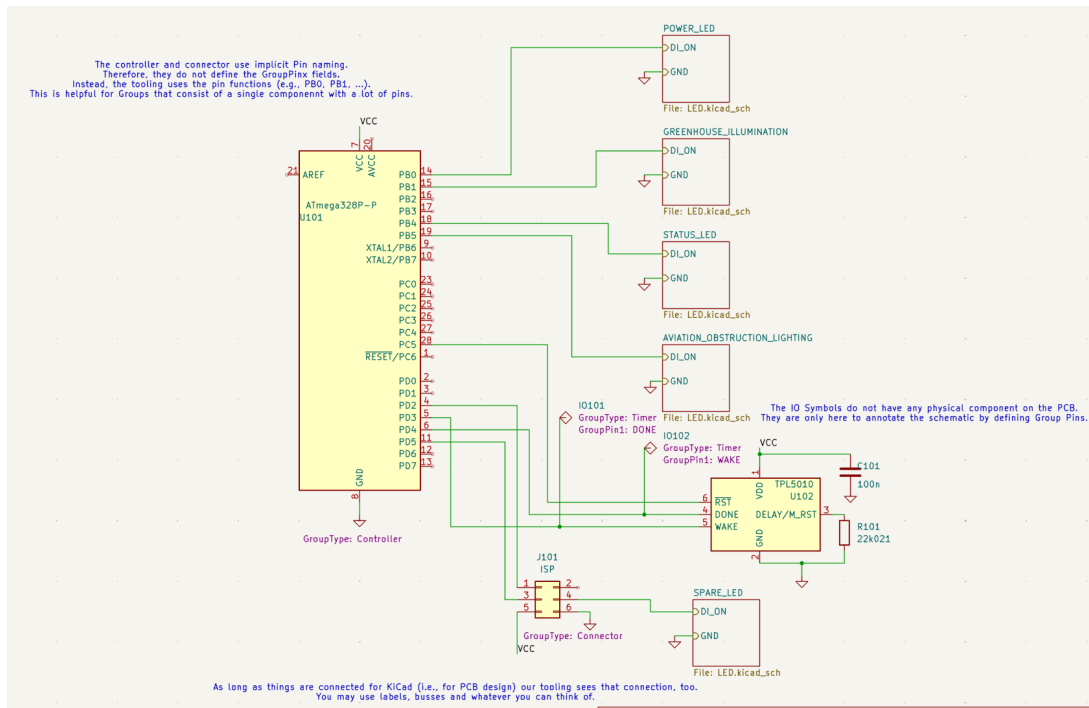
The harness engineer creates the harness specification, which, among others, specifies what cables connect which connectors of what length. Therefore, the harness engineer must understand both the system design and what connectors with what pins each subsystem has.

2.4.1 Harness Designer (HaDes)

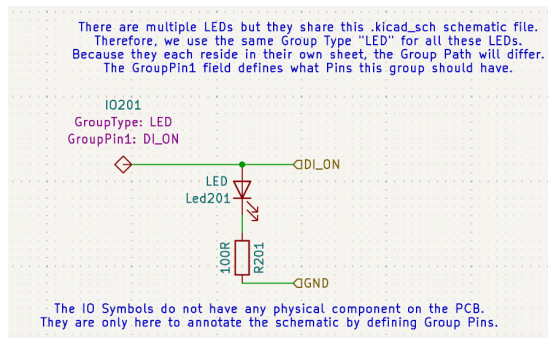
The Harness Designer (HaDes) is an application for managing a spacecraft's harness. The harness engineer uses HaDes to create the harness specification. For this purpose she inputs the system design into HaDes, i.e., what subsystems there are with what connectors.

2.5 Snippet Based Design

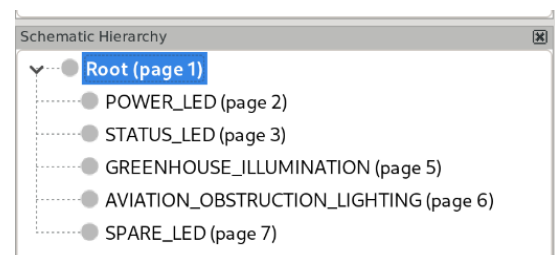
As we explain in subsection 2.2.2 the PCDU uses a hierarchy of schematic sheets. The DLR Avionics Systems goes even further and forms their entire PCDU electrical engineering



(a) The schematics of a synthetic example KiCad Project. The controller on the root schematic connects to multiple LEDs, each on their own hierarchical schematic sheet.

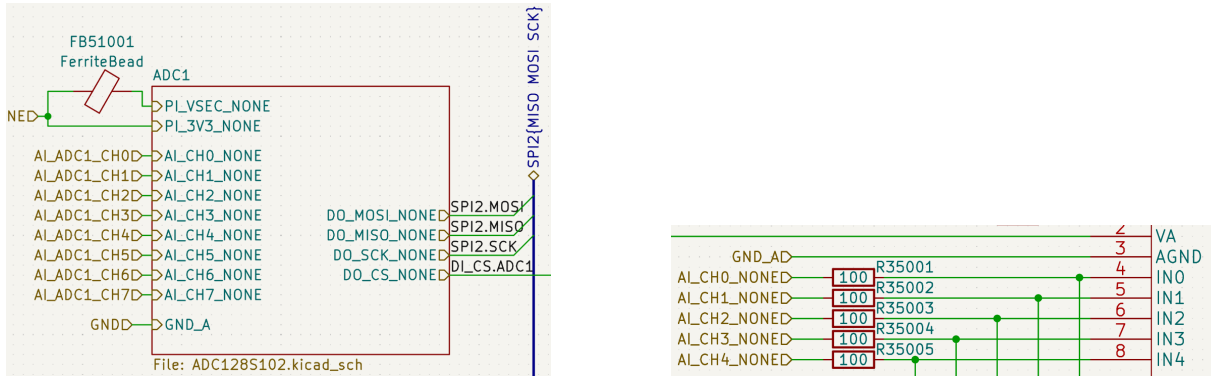


(b) The schematics of the hierarchical sheet, which the root sheet in Figure 2.3a shows. Notice while there are multiple LED sheets, there is only one .kicad_sch file for them.



(c) The sheet hierarchy of the synthetic project. Figure 2.3a shows the root sheet. Figure 2.3b shows the LED sheet. Notice that there are only two .kicad_sch files, one for the root and one for the LEDs.

Figure 2.3: Synthetic KiCad schematics. This example is unrelated to the PCDU.



(a) A hierarchical symbol with ref ADC1 includes schematic file ADC128S102.kicad_sch.

(b) An extract from the schematic file ADC128S102.kicad_sch showing some of its hierarchical labels on the left.

Figure 2.4: Hierarchical sheet and symbol. Notice that the hierarchical labels in (b) connect the hierarchical symbol's pins in (a).

around the idea of reusing groups of components, snippets [1]. A Snippet's electrical design consists of exactly one `.kicad_sch` file and the PCB layout for the constituent components. Figure 6.1 shows an example Snippet.

Snippet Based Design provides multiple advantages. Firstly, this accelerates the PCDU's electrical engineering process. The electrical engineer does not need to connect individual components together. Instead, she identifies what power distributions the PCDU must provide and selects a corresponding set of Snippets. Furthermore, the DLR Avionics Systems tests each Snippet individually. This, for example, includes radiation tests, which test whether the Snippet operates correctly under the radiation environment in space. The alternative of testing the entire PCDU would be much more costly. Therefore, Snippet Based Design greatly accelerates the electrical design process of an PCDU.

Be aware of the difference between a Snippet and a Group, a concept we propose to transfer the advantages of Snippet Based Designs to firmware development (see section 5.1).

2.5.1 Snippet based Design Analysis

While Snippet Based Design already accelerates the PCDU's electrical design process, the concept of Snippets provides further potential. Snippet based Design Analysis enhances Snippets with electrical specifications. This includes maximum ratings, e.g., how much current may flow through an LCL. Snippet based Design Analysis is the idea of simulating the power flow from the power source to the load based on how Snippets connect. Snippet based Design Analysis aims at reading the Snippet topology from the schematics. The relevant information are:

- What Snippets are there?
- How do the Snippets connect to each other? Similar to firmware development, Snippet based Design Analysis does not use any information on the constituent components. However, In contrast to firmware development, Snippet based Design

```
{% for group in glob_groups("***") %}
    {{ group.group_type }}
    {{ group.group_path }}

{% endfor %}
```

Figure 2.5: Figure 2.6 shows the output this template produces. Our tool `code_gen` defines the function `glob_groups` (see section 4.3). Furthermore, our Python library defines the `Group` class this template uses (see subsection 4.1.4).

Analysis focuses on electrical characteristics. Therefore, the digital control pins are not of interest while the power inputs and outputs do matter in the context of a PCDU.

With this information Snippet based Design Analysis hopes to answer, among others,

- which type of Snippet would best suits its needs and
- if all values remain within their maximum ratings.

2.6 sed

`sed`, a stream editor, is a terminal application which performs transformations on input streams, including files [14]. We use `sed` to rename words in large amounts of files (see section 5.1) and refactor code to abide by a new nomenclature (see subsection 6.1.2).

2.7 Jinja2

Jinja2 is a template engine for use with Python [23]. Figure 2.5 shows an example Jinja2 template file. While Jinja2's main purpose is web templating, it receives large use beyond generating HTML, CSS and JavaScript. Ansible, for example, uses Jinja2 to generate arbitrary system configuration files [2]. Pando also uses Jinja2 to generate C++ firmware (see section 3.1).

Based on GitHub stars, Jinja2 is the most popular template engine for Python [25].

Most importantly to our work, Jinja2 templates may call Python functions and perform operations on variables. In some situation, including ours, this allows the user to not create a separate Python file. Furthermore, Jinja2 template files may include other template files. This allows splitting large templates across multiple files and reusing sections of the template in multiple places.

```
StatusLED
/Dist_12V/DCDC1/

SomeDCDCType
/Dist_12V/DCDC1/

STM32
/Controller/

SomeADCType
/ADC1/

SomeLCLType
/DIST_12V/LCL1/

SomeLCLType
/DIST_12V/LCL2/
```

Figure 2.6: The output code_gen produces using Jinja2 and Figure 2.5 as input. Figure 4.2 shows the data the output reflects.

Chapter 3

Related Works

Several works already try to automate parts of the hardware and firmware development process by reading information from hardware schematics or netlists. Another work already generates C++ firmware code and is in use by the DLR. Therefore, we briefly explain the relevant aspects of each work and compare how they relate to our contribution.

3.1 DLR Institute of Space System's Pando

TMTC consists of sending telemetry from the satellite to the Mission Control Center and telecommands back to the satellite (see subsection 2.1.2). Therefore, both the Mission Control Center and the PLUTO PCDU should adhere to the same protocol. The DLR Avionics Systems defines this protocol in model XML files. Such a model includes what packets form the TMTC protocol and what data they contain in which order. The DLR Avionics Systems developed Pando to generate C++ firmware based on the model XML files. Pando uses Jinja2 templates to generate the C++ firmware.

Most importantly, Pando does not know the hardware specification. Hence, it cannot generate firmware that depends on this information, e.g., pin definition headers (files defining which of the Microcontroller's pins does what). Instead, the software engineer must manually search for this information in the KiCad schematics. Our tooling, however, does use the hardware specification the schematics provide (see section 4.2). Therefore, our tooling fills this gap in firmware generation, saving the software engineer from manually searching through the schematics (see subsection 6.1.2).

Furthermore, there are two parts of Pando, pando-core and pando-satellite. While pando-satellite is unpublished because of export restrictions pando-core is published as Open-Source software [26]. The PLUTO PCDU firmware uses both. Our tooling, however, allows writing logic in Jinja2 template. This allows keeping the Jinja2 templates private while publishing our entire tooling as Open-Source software (see section 5.10).

3.2 C/C++ Stepper Motor Driver generation from Schematics

Brinz [5] is a related work that generates C++ driver firmware for stepper motors from KiCad schematics. It does so by reading the symbols' pins' xy coordinates from the `.kicad_sch` file with the kicad skip library [8]. At those coordinates the related work searches for an endpoint of a wire and traces it to a label. If two pins connect to labels with the same name, the related work considers them connected. This is a reimplementaion of KiCad's netlist generator, among others, without support for buses and nets that use multiple transitive labels. The PLUTO PCDU schematics, however, use these features. Furthermore, the related work fails to provide a method of ensuring parity between the tool's netlist generator and KiCad's. While the electrical engineer expects Brinz [5] to consider pins connected iff KiCad does so as well, the related work does not ensure that. Our `kicad_firmware_generation` tooling uses the KiCad Netlist, instead, (see section 5.2). This ensures parity between KiCad and our tooling and fulfills the electrical engineer's expectations.

Similar to our tooling, Brinz [5] uses Jinja2 to generate firmware. However, the related work limits the use to generating firmware for stepper motors. Our proposed Group Netlist (see 4.1) goes further and unlocks firmware generation for arbitrary hardware and other use-cases like harness generation and snippet based design analysis.

3.3 Generate pin definition header file from KiCad schematic

Deshmukh [9] is another related work that aims at generating pin definition header files. It is at a very early stage in development without documentation and has remained abandoned for seven years. Nevertheless, the related work serves as a proof-of-concept for using the KiCad netlist as input, similar to our tooling.

In contrast to our tooling, the related work does not group components and does not use schematic annotations in general. This is a problem for the PLUTO PCDU, because it realizes some functions with multiple components, not just a single one. For example, there is a resistor between each ADC128S102's input channel and the analog signal to measure (see Figure 6.1). In that case Deshmukh [9] could only find the resistor and not the ADC behind it. With our `kicad_firmware_generation` tooling the user can group the resistor and component together. That way, the generated firmware contains the required information.

Furthermore, we need more than just a pin definitions header file; we need to generate the entire driver for the PLUTO PCDU's LCLs, for example. While a pin definition header file explains what components directly connect to the microcontroller, not all components do directly connect to the microcontroller. E.g., the PLUTO PCDU's LCLs connect through ADCs to the Microcontroller. The related work cannot realize this transitive connection in the generated firmware.

Additionally, the related work only generates firmware. Our proposed Group Netlist (see 4.1), however, goes further and unlocks more use-cases like harness generation and snippet based design analysis.

Lastly, the source-code hardcodes the input file paths and the user cannot easily use the related work on different schematics. We conclude that while Deshmukh [9] serves as a proof-of-concept for using the KiCad netlist, we deem it unsuitable for our needs.

3.4 Generating Platform Configuration from Netlists

Wehrli [31] partially generates platform configuration files or checks for errors in existing files. Platform configuration files describe how to configure a Field Programmable Gate Array (FPGA). The related work uses the netlist of Altium Designer projects as input. Similarly to our tooling, the related work supports connecting multiple boards together. However, there are some connections the related work cannot identify. In these cases the user must manually inspect the schematics and write the missing platform configuration by hand. Our tooling, however, supports KiCad schematics and extracts all information, which generating firmware requires. We propose the Group Netlist file format (see section 4.1) to store this information allowing other use-cases than firmware generation, too.

Furthermore, Wehrli [31] aims at reading existing netlists as they are. Our tooling, however, requires annotations in the schematics allowing to group components together and specify their function. Therefore, our tooling forgoes the complicated component identification process the related work requires. Additionally, the related work requires separate handwritten files with information the netlist does not contain. To our work the netlist alone suffices. We argue that annotating schematics in the schematics editor, which the electrical engineer is used to, is easier and less error-prone.

We conclude that Wehrli [31] is a proof-of-concept showing that a system's netlist may aid in finding errors in an FPGAs pin assignments. Our tooling, however, is a ready-to-use product allowing anyone to use it on other projects.

3.5 Safe and Reusable Approach for Pin-to-Port Assignment in Multiboard FPGA Data Acquisition and Control Designs

[17] generates constraint files and reports on errors in systems comprising multiple FPGAs. The related work supports Vivado and Quartus to understand the hardware description of any FPGA design. With this information it checks if the engineers properly connected the FPGAs together. Our work, however, supports KiCad to understand the connections between components on a PCB and, among others, generates firmware.

Furthermore, [17] focuses on how multiple subsystems connect. In comparison, our tooling describes how the components inside a subsystem connect. While we do generate a harness specification, this specification only describes a single subsystem and aids in designing the harness (see subsection 6.1.3). The related work's goal, however, is to better understand the existing connections between subsystems and identify errors.

Chapter 4

Design Goals and Specification

We propose extracting information from the Schematic Editor to generate the hardware's C++ firmware using automated tooling. `kicad_firmware_generation` [4] is our reference implementation for such 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).

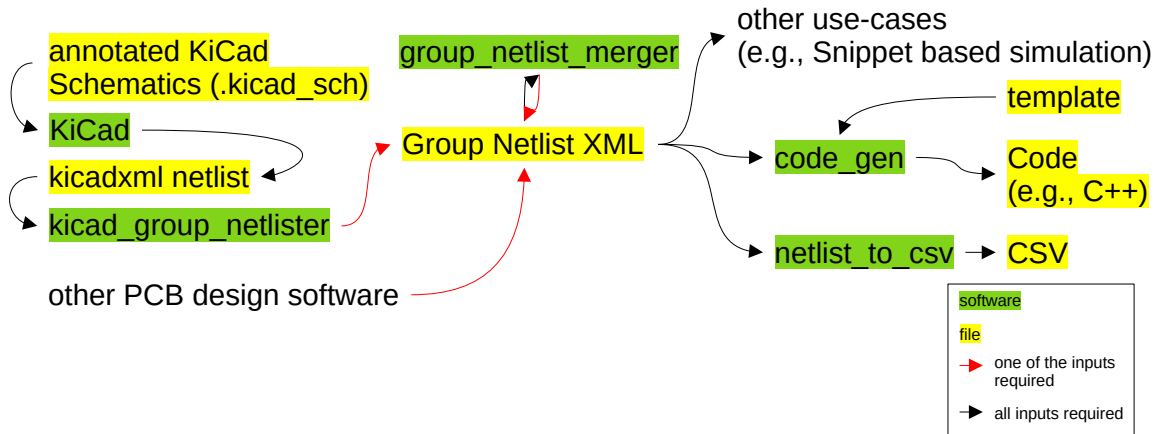


Figure 4.1: Overview of the final generator design. Notice how data flows from left to right, from the electrical engineering tools into the software engineering tools. Our proposed Group Netlist XML is the common interface inbetween. The group_netlist_merger takes multiple Group Netlists and combines 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 PCDU 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 and, thus, more maintainable program (see DG4). Analogously, we anticipate the Group

Netlist specification to be simpler, being independent of both KiCad and the firmware. A simpler specification, especially a human-readable one (see DG8) with a stable specification (see DG7), should further help create 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 **Groups** 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 alphanumeric string including underscore, slash, minus, plus and space with `/` as a prefix and suffix).

Every Group has multiple **Pins**, 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 firmware driver. For this, we support **Group Types**. 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.

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).

Additionally, to represent that some points are electrically connected there are **Nets**. 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.

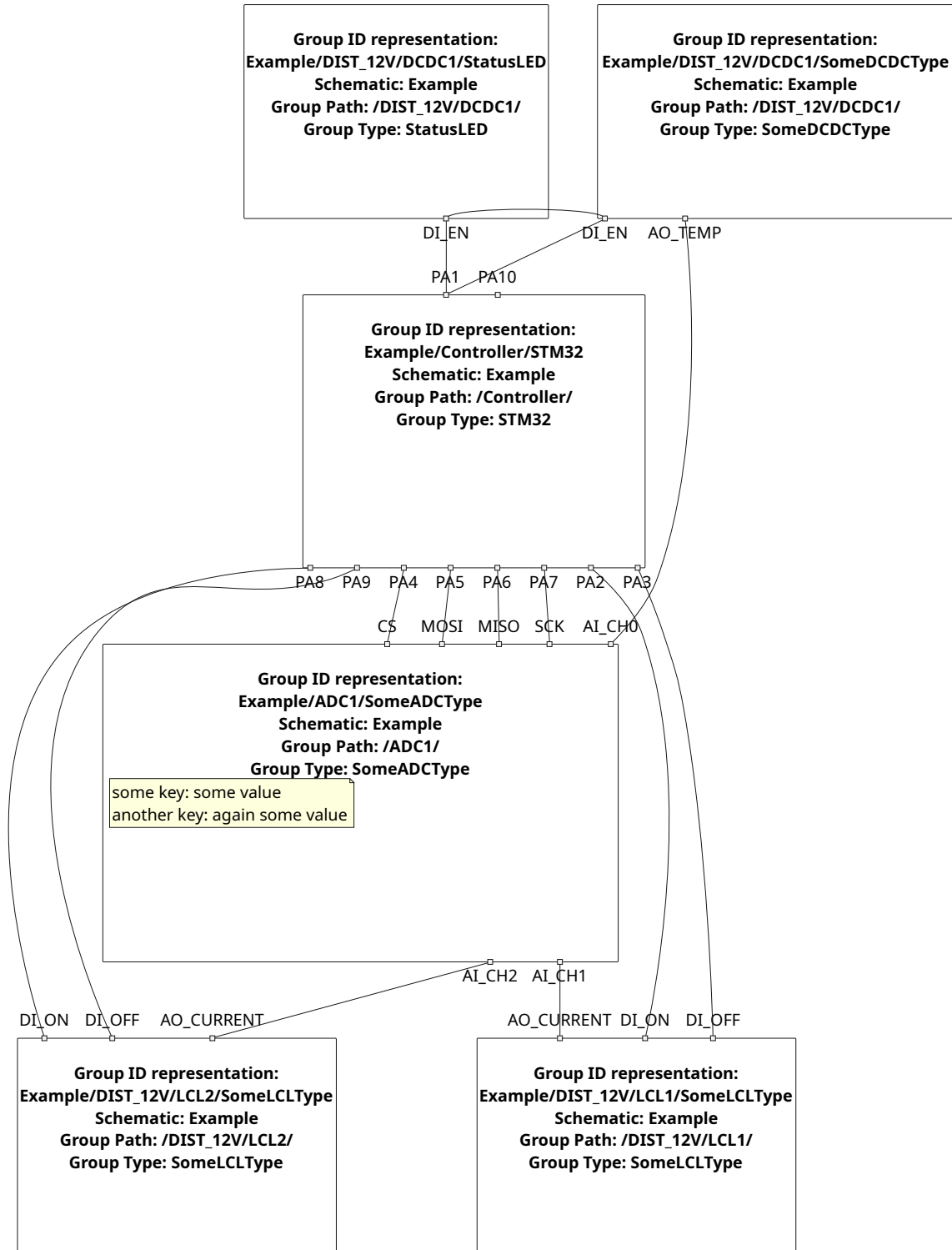


Figure 4.2: Example Group Netlist Visualization of Figure 4.3. Every box represents a Group with the string representation of the Group ID explained in subsection 4.1.2. All small squares represent Pins and lines between represent Nets. Notice that the two bottommost Groups belong to the same Group Type, “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.

```

<?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_netlist 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.

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 Avionics Systems code generator Pando already uses XML input files, easing the maintenance and adoption of 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 Fields** 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.

```

<!-- -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.

```

<!-- -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.

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 standard 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 select 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 Globbs are based on UNIX' globs for pathname pattern expansion, specifically the implementation in the Python standard library [18]. This explains the limited amount of characters a Group ID may consist of. However, Group Globbs 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 Groups with **Example/ADC1/SomeADCType** or **Example/DIST_12V/LCL1/SomeLCLType** as Group ID representation.
- ****/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 serializing 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 carries 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 and graphical user interface allow converting a KiCad schematic (`.kicad.sch` file) into a netlist in the `kicadxml` format [15]. `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_netlist` 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 (see Figure 4.6a and Figure 4.6c). 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

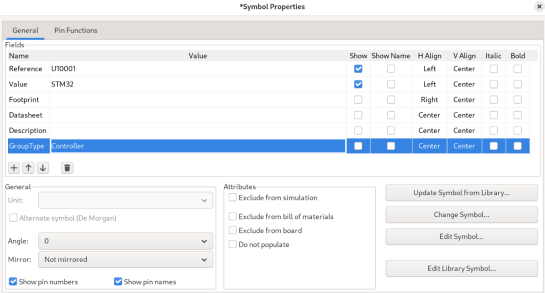
A KiCad component may have multiple pins, each with a pin number. Additionally, pins may have a pin function, a string unique for the symbol (see Figure 4.6b and Figure 4.6d). As multiple KiCad components form a Group and some of those components may have the same pin number and/or function, choosing Pin Names for the Group is not trivial. `kicad_group_netlist` implements two naming methods. For each group it makes an individual decision on which naming method to use.

1. **Entirely explicit pin 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 pick Group Pin Names. However, the user must give every pin an explicit name using an associated field.

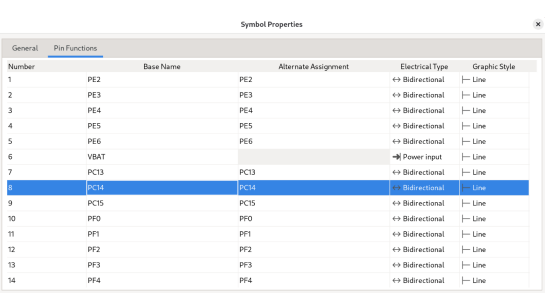
This allows creating a `Group_IO` symbol without any electrical function and a single pin. For example, in some situations the user is only interested in some pins of a collection of components. In that case the user can add the `Group_IO` symbol only on the pins that she cares about. This allows abstracting complex circuitry as a Group with some Pins.

`kicad_group_netlist` uses this mode iff any of the Group's KiCad components have a `GroupPinX` field set. See Figure 4.6c and Figure 4.6d for an example.

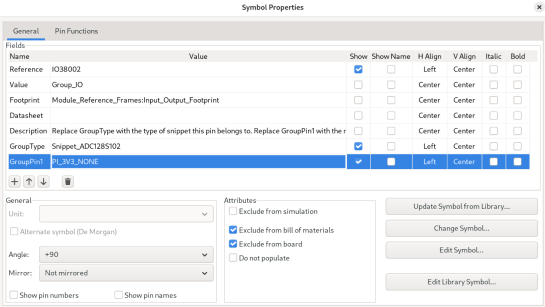
2. **Implicit pin naming:** In other situations there might only be a single component with many pins, for example a microcontroller. The user does not want to label all those pins, especially when the pin functions of the symbol are already unique. Then the user simply does not provide any explicit `GroupPinX` fields and `kicad_group_netlist` uses the pin functions as Pin Names for the Group. `kicad_group_netlist` enforces that all component's pin functions are set and unique. See Figure 4.6a and Figure 4.6b for an example.



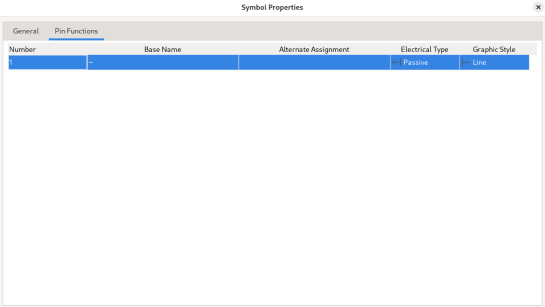
(a) The **General** tab in KiCad’s **Symbol Properties** window. It shows the microcontroller. The **GroupType** field defines this Group’s Type. Notice how there is no **GroupPinX** field as this Group uses implicit pin naming.



(b) The **Pin Functions** tab in KiCad’s **Symbol Properties** window. It shows the microcontroller symbol. The **Base Name** column shows the relevant pin functions. Those pin functions are the Group’s Pin Names, because this Group uses implicit pin naming.



(c) The **General** tab in KiCad’s **Symbol Properties** window. It shows a **Group_IO** symbol with a single pin belonging to an ADC. The **GroupType** field defines this Group’s Type. The **GroupPin1** field defines the single Pin’s Name, using explicit pin naming.



(d) The **Pin Functions** tab in KiCad’s **Symbol Properties** window. It shows the **Group_IO** symbol, which uses explicit pin naming. Therefore, kicad_group_netlist ignores all pin functions. Notice that multiple of these **Group_IO** symbols belong to an ADC. Therefore the pin functions would not be unique, making implicit pin naming impossible.

Figure 4.6: KiCad’s symbol properties interface for two example symbols; one with implicit and one with explicit pin naming.

4.2.3 Illegal KiCad Schematics

Unfortunately, the KiCad user interface allows creating schematics that `kicad_group_netlist` 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_netlist` 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 difficulty concerning Group Paths that `kicad_group_netlist` cannot properly differentiate: 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. However, 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_netlist` 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_netlist` Implementation

This section explains how `kicad_group_netlist` works in detail. We intend it for readers interested in replicating our implementation. Consider reading the source code [4] alongside it.

`kicad_group_netlist` uses a pipeline of six steps:

1. `kicad_group_netlist` 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_netlist` considers belonging 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 reference identifier to that component's Group's ID. The

KiCad pin number and component reference identifier 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_netlist` 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). If a Group uses implicit pin naming, there is no entry in this lookup. `kiCad_group_netlist` 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_netlist` 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 fourth 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_netlist` 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.
5. The fifth stage creates a Group from each Raw Group, removing the KiCad-specific information and adding the Pins.
6. Lastly, `kiCad_group_netlist` 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).

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 2.7). `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 4.7 shows an example Jinja2 template producing Figure 4.8. 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

```

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 %}
            {% 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 %}
            typedef modm::platform::GpioAnalogOutput{{ gpio_pin.pin[1:] }}
                {{ pascal_case(pin) }}Pin;
        {% endif %}
    {% endif %}
{% endfor %}
}

```

Figure 4.7: Jinja2 template for C++ modm pin mapping [22]. The main Jinja2 template file sets the `group` variable to a `Group` to generate C++ code for. Afterwards, it includes this template file.

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_netlist` converts exactly one KiCad schematic into one Group Netlist and `code_gen` consumes exactly one Group Netlist. The PLUTO PCDU's 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 PCDU 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**

```

namespace PlutoEPS
{
// [...]
namespace Dist12vBLclExt1
{
// snippet type: [Redacted]
// [Handle to describe this LCL.]
typedef void Handle;
// [The pin through which the controller turns this LCL on.]
typedef modm::platform::GpioOutputE2 DiOnNomPin;
// [The pin through which the controller turns this LCL off.]
typedef modm::platform::GpioOutputE1 DiOffNomPin;
// [This snippet has an analog output pin]
// [transmitting the current draw.]
// [This type specifies through which ADC and]
// [which channel the controller may read that value.]
typedef Adc1::Ch3 AoCurrentNoneAdc;
} // namespace Dist12vBLclExt1
// [...]
} // namespace PlutoEPS

```

Figure 4.8: C++ code output of Figure 4.7 using the same highlighting. Notice that Figure 4.7 only produces the `Dist12vBLclExt1` namespace. Furthermore, we added comments with square braces that the template did not produce.

Mapper. Some connectors, for example, have a female and a male version. When they mate, pins of the same name connect electrically. The PLUTO PCDU, 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`.

This paragraph explains how `group_netlist_merger` works in detail. We intend it for readers interested in replicating our implementation. Consider reading the source code [4] alongside it. 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 4.9a 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 in a second, inner loop. If another net should be merged, it unionizes the Nodes of both Nets, adds the new Net to the Group Netlist and removes the two old Nets. Figure 4.9b shows an example final Group Netlist.

Notably the Matching Groups do not need to be spread across multiple schematics. This

way, for example, a user 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`

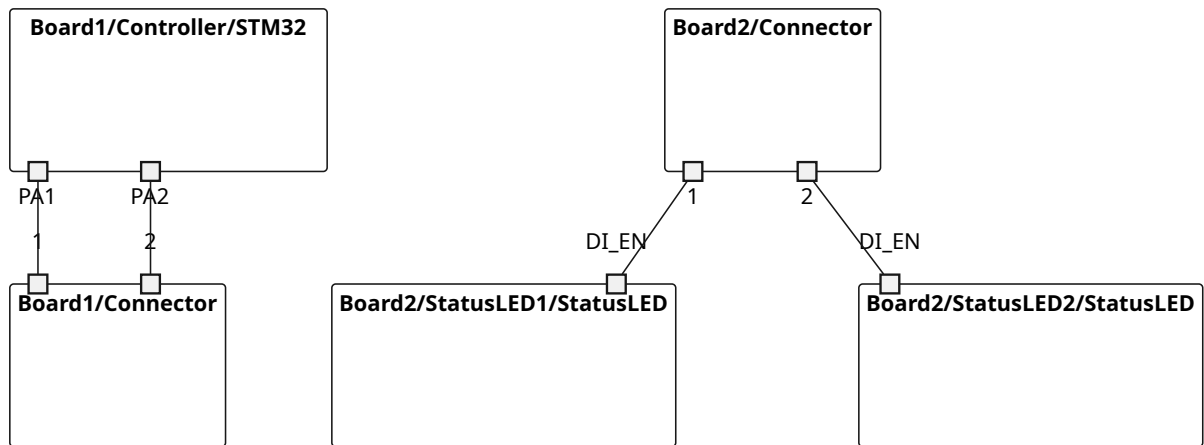
`netlist_to_csv` receives the path of a Group Netlist XML and two string arguments: `--root-group-glob` and `--simplify-pins`. The output CSV contains for every Pin a list of all other Pins that connect to it.

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.

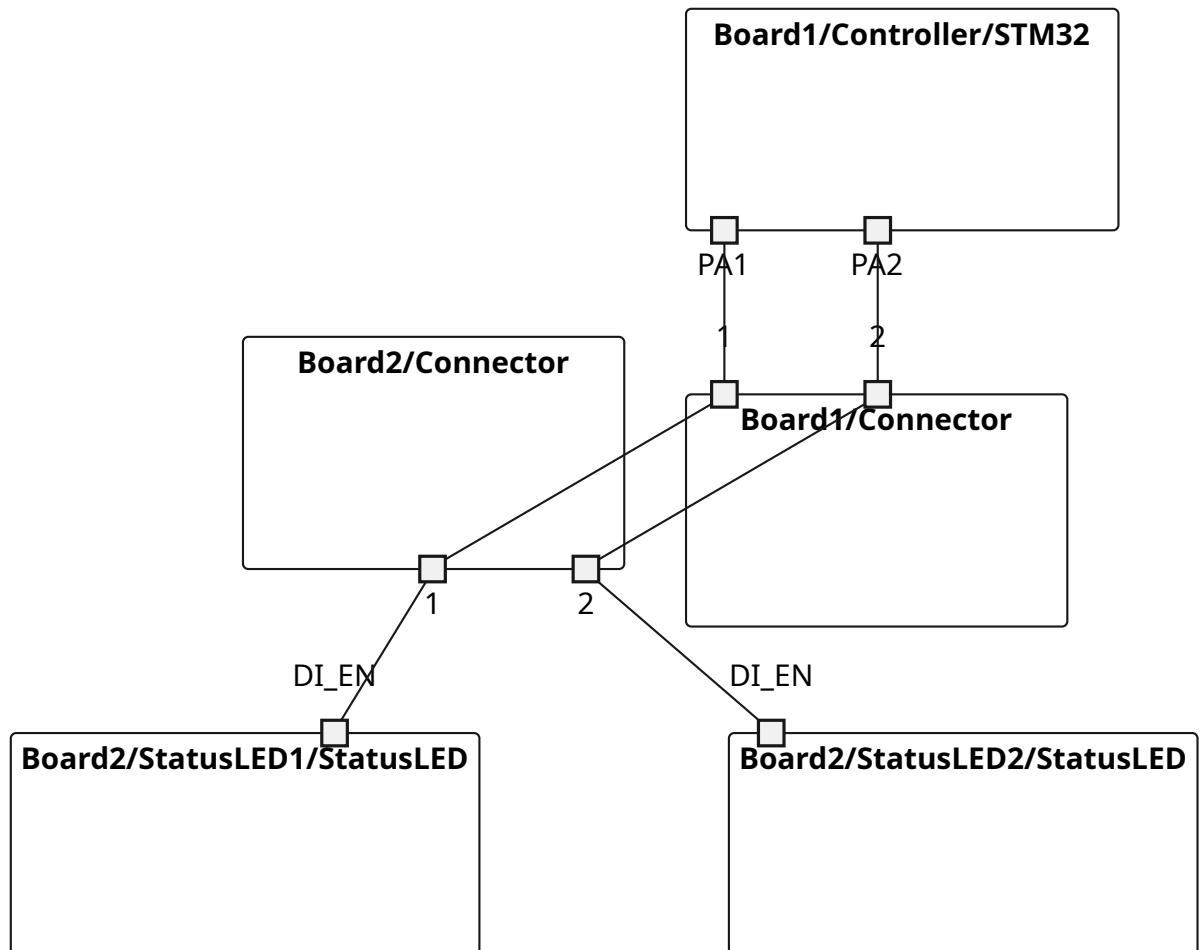
Additionally, the user may only care about certain Groups. When the user provides `--root-group-glob` the CSV only contains Groups that match this Group Glob. Furthermore, the CSV does not contain any connections between Groups that match `--root-group-glob`. Only connections from matching Groups to 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.

This paragraph explains how `netlist_to_csv` works in detail. We intend it for readers interested in replicating our implementation. Consider reading the source code [4] alongside it. `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. `netlist_to_csv` takes special care to keep the pins of the same connector next to each other in the CSV. Furthermore, it orders the pins



(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 4.9a 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 4.9: Example Group Netlist in different stages of `group_netlist_merger`.

by their number, if they have a number. This is not trivial, as the Pin Name may contain a non-numerical prefix or postfix. Our `netlist_to_csv` solves this by searching for any number within the Pin Name and ordering by that.

Chapter 5

Development History

The original goal was to generate the C++ firmware for the PLUTO PCDU (see subsection 2.1.7) from its KiCad schematics. We chose Python and Jinja2 (section 2.7) for implementing this KiCad schematics to firmware generator. Firstly, DLR Avionics Systems Pando (see section 3.1) already uses Python and Jinja2 causing less friction for the software development workflow (see DG2). Additionally, the DLR Avionics Systems 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 [13] and Pallets [23]. 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 PCDU 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, the exact requirements such use-cases impose remained unknown. 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 reimplementaion in ,e.g., Rust will be considerably easier. That is because this thesis already explains the requirements, working and discarded high-level implementations.

Because the software development workflow uses Makefiles for compilation, we 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 either to stdout or a file according to the `--output` parameter. 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 decided to heavily rely on Python type hints, checked by mypy [24]. 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 serialize 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 serializing function and that the serialize 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 explain 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 advise 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 select “Snippet” to align with the Snippets from Snippet Based Design (section 2.5). However, at a later stage we realized a difference between “Snippets” (later “Groups”) and actual 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_netlist` 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 [7] appeared promising because the KiCad contributors officially support it. Unfortunately, the KiCad IPC API it uses does not support the Schematic Editor [6]. Additionally, the requirement of having a running KiCad instance would have made `kicad_group_netlist` 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 [30] is a third-party library claimed to parse KiCad files including schematics. However, we did not 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_netlist`'s own code is maintainable but also its dependencies'.

KiCad Schematic API [20] 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` [19] and `kicad_skip` [8] 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, [5] uses [8] 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_netlist` guarantees parity, any changes to KiCad's netlist generator would require a change to `kicad_group_netlist`. This, in turn, would make `kicad_group_netlist` 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 [5] and discarded the idea of reading the `.kicad_sch` file directly. Instead, we decided 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 selected 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

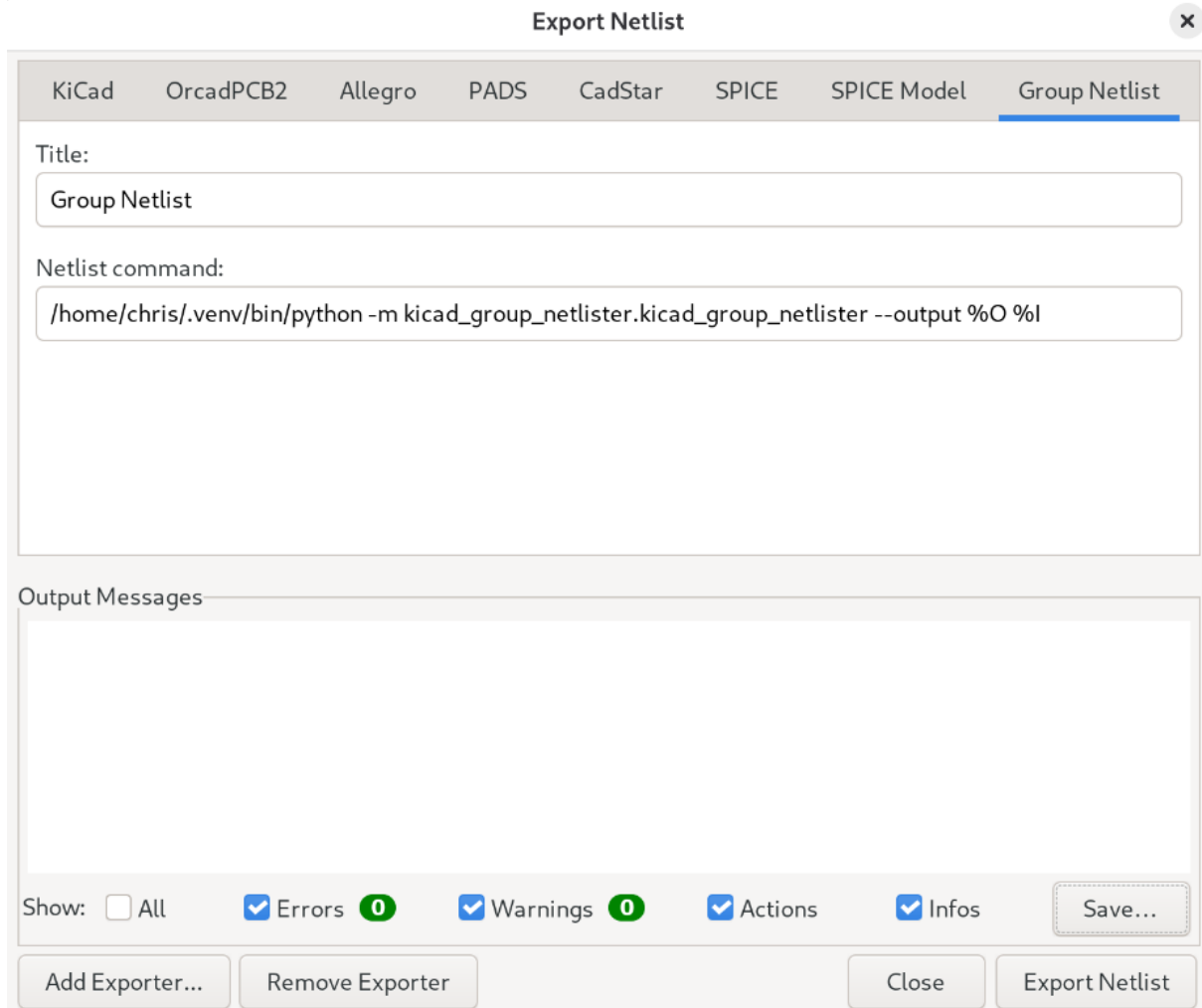


Figure 5.1: The KiCad user interface showing a custom netlist exporter. It uses `kicad_group_netlist` to generate a Group Netlist from the schematics the user opened. KiCad replaces `%O` with the desired output file path and `%I` with the path to the `kicadxml` netlist [15]. Notice how the netlist exporter does not have access to the `.kicad.sch` schematics file but does have access to the KiCad netlist.

includes `kicad_group_netlist`, because we picked the `kicadxml` format. While the software engineer can use `kicad_group_netlist` 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. Figure 5.1 shows such a netlist exporter. The electrical engineer can use this exporter to check her schematic annotations produce her desired Group Netlist, see section 4.2. Furthermore, the alternative `kicadsexpr` format requires using a third-party library like `kinparse` [29] or `kicad skip` [8], which reduces maintainability (see DG4). `kicadxml`, however, can be parsed by the python standard library, again, making `kicad_group_netlist` 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_netlist` 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_netlist` 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 of 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 Snippets. Therefore, the Snippet's Pins are the hierarchical sheet labels. The `.kicad_sch` schematics file does contain this information. Therefore, a theoretical implementation of `kicad_group_netlist` 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 5.2. 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 decided 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 [15]. We propose modifying the KiCad netlist specification to include all label names for each net. This could allow a `kicad_group_netlist` implementation that solves the above problem without reading the `.kicad_sch` file. Because KiCad is Open-Source software, future work could implement this feature.

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 5.3 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

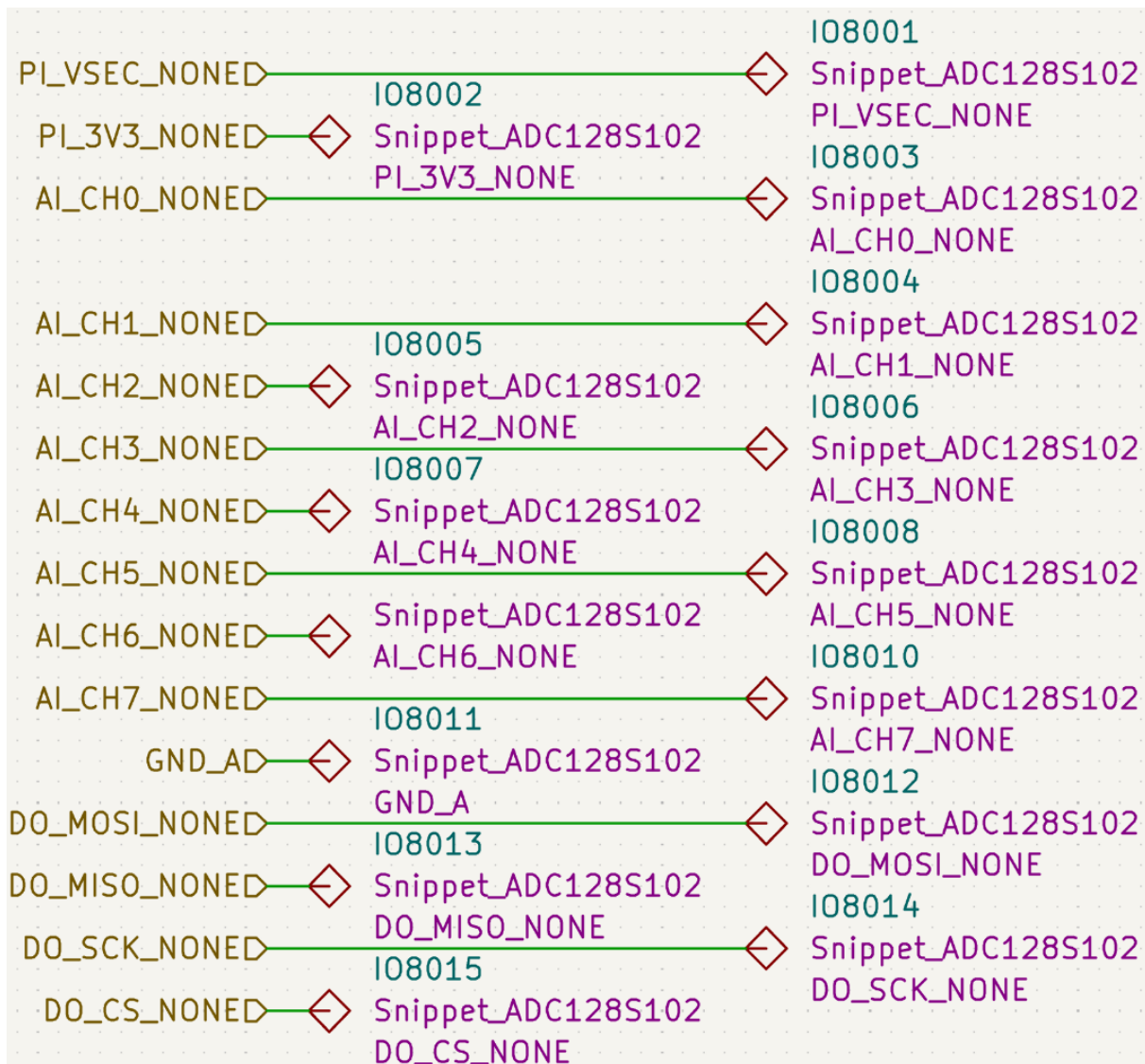


Figure 5.2: Annotating KiCad net labels (on the left) with `Group_IO` symbols (on the right). A `kicad-group-netlist` implementation reading the `.kicad.sch` file directly or modification to KiCad could remove the need for this. Figure 6.1 shows the snippet's entire schematics.

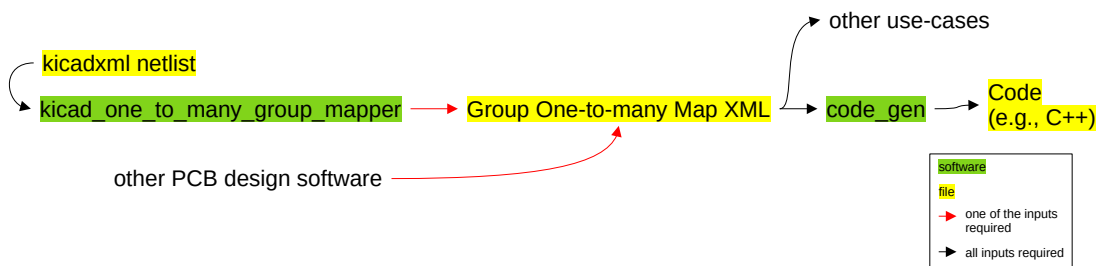


Figure 5.3: Overview of the first generator design. We hide some of the external tools shown in Figure 4.1.

`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 microcontroller 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 microcontroller's Group, which runs the firmware. Then the One-to-many Group Map only represents connections to the Root Group, e.g., microcontroller. 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 microcontroller.

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 Names** 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 PCDU 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 what 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 PCDU's template (see section 6.1) shows this.

5.5 Python Library

Initially, `kicad_one_to_many_group_mapper` directly included XML serializing 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.

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 (see section 4.1). 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_netlist` and `one_to_many_group_mapper`. The interface between was the Group Netlist XML file. Figure 5.4 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 (see section 2.4). 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.

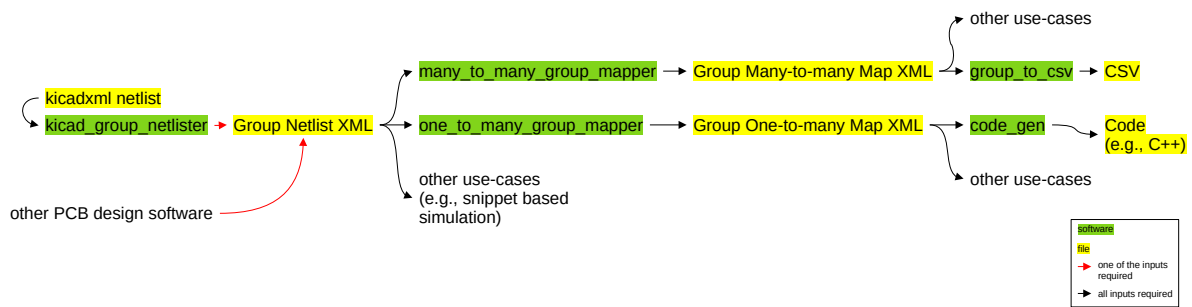


Figure 5.4: Overview of the second generator design. We hide some of the external tools shown in Figure 4.1. The Group Netlist and Maps are XML documents. Notice how there are three file formats we would have exposed to other use-cases. In the final design there is only one, the Group Netlist XML.

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 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 5.4. We understood that three file formats is too many to maintain (see DG4). Nevertheless, at this point the requirements of other use-cases were not well known. Therefore, we chose not to combine any programs or remove a format at that time.

5.8 Merging multiple Schematis

The PLUTO PCDU does not consist of a single PCB but a stack of two: the distribution motherboard and SOLBAT daughterboard. At first, we only considered extracting information from the distribution board’s schematics. However, there are multiple compo-

nents on the daughterboard of the microcontroller’s interest. These components connect through connectors to the distribution board. 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_netlist` 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 5.4 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 microcontroller and how? This proved incorrect because there are **Transitive Group Connections**: Sometimes a Group does not directly connect to the microcontroller 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 `A0_TEMP` Pin does not directly connect to the microcontroller but to the ADC, which in turn does connect to the microcontroller. 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 5.4). 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 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).

5.10 Publishing as Open-Source

Because of our goal to create highly adaptable tooling (see DG3) we created `code_gen` as a generic tool (see section 4.3). Therefore, the PLUTO PCDU-dependent logic resides entirely in Jinja2 templates and is independent of `code_gen`'s Python code. Consequently, we could publish our entire tooling except for the Jinja2 PLUTO PCDU templates. The MIT Open-Source license allows anyone to use, extend and modify our tooling, which we made accessible on GitHub [4].

This is different to Pando of which there are two parts, one unpublished and one Open-Source (see section 3.1). Because both parts contain Python code, users outside the DLR Avionics Systems cannot use the logic present in the unpublished part of Pando. Furthermore, this eases maintenance (see DG4) because maintainers can clearly and securely differentiate the unpublished templates from the Open-Source Python code. Notice, however, that before publishing we waited until narrowing down all file specifications to a single one, the Group Netlist. Otherwise, other users could have created software relying on now deprecated file formats.

Chapter 6

Deployment and Evaluation

6.1 Integration in DLR's PCDU Design Workflow

Our `kicad_firmware_generation` tooling reads the KiCad netlist of one or more KiCad schematic files. These schematics specify what components the hardware consists of and how they are connected. The firmware we generate with this tooling, however, is only interested in what functionality the hardware has and how to control them. Because multiple components may fulfill a single functionality, the firmware should treat those components as a single unit, a group of components. Therefore, our tooling requires annotations in the KiCad schematics. These annotations, among others, explain what components belong together (see section 4.2).

While one can add these annotations to existing schematics, the DLR's planned use for `kicad_firmware_generation` is different. The DLR Avionics Systems intends to integrate `kicad_firmware_generation` in the development process for a new PCDU using snippet based design (see section 2.5).

We argue the most relevant quality metrics for our tooling are the following:

1. Did we fulfill the design goals laid out in chapter 4?
2. How much handwritten firmware can `kicad_firmware_generation` replace?
3. How much time and effort does it take to integrate generated firmware into existing firmware?
4. How much time and effort does it take to adjust the schematics design process to facilitate Group Netlist generation?

Firstly, we argue in chapter 5 that we have indeed fulfilled all design goals. Secondly, we would have liked to apply our tooling to a new PCDU and directly measure how the development workflow changes when switching to our tooling. Unfortunately, there is currently no PCDU in the electrical design phase at the DLR. Therefore, we cannot directly measure those metrics.

While PLUTO (see section 2.1) remains to be launched into orbit, its PCDU hardware and firmware is already complete. Nevertheless, we chose to use it as an example project and approximate the quality of our tooling. For this purpose we annotated the snippet schematics the PLUTO PCDU uses. Afterwards, we replace the manually written firmware with the firmware `kicad_firmware_generation` generates.

6.1.1 Annotating PLUTO PCDU Schematics

The PLUTO PCDU uses ten different snippets, which future PCDU will reuse. Therefore, we started by converting each snippet into a Group. For each snippet we created a Group with Pins reflecting the hierarchical sheet labels the respective schematic file has. The hierarchical sheet labels are the PCDU schematics' interface to the snippet. Therefore, the Group Netlist has the same perspective on snippets as the PCDU schematics has.

The only inconvenience was that `kicad_group_netlist` cannot read those hierarchical net labels. Instead, we had to use `Group_IO` labels for each hierarchical label as Figure 5.2 shows. Though, because there are only ten different snippets, the effort remained low. Figure 6.1 shows a fully annotated snippet.

We used implicit Pin naming for the STM32 microcontroller (see subsection 4.2.2). This kept us from manually annotating all 144 pins of the STM32.

Furthermore, we utilized the ability to have multiple Groups on the same schematics file. For example, the connectors share a schematic sheet. We annotated each connector as a Group to aid in harness specification (see subsection 6.1.3).

Similarly, we annotated the SPI, CAN and One_wire interfaces as Groups. Interestingly, these Groups do not contain any components and only annotate nets these interfaces use. Lastly, we had to rename some hierarchical symbol names because they did not abide by any consistent nomenclature. This is necessary to simplify the C++ firmware generation. Lastly, the PLUTO PCDU consists of two boards, the main distribution board and the solar battery board. We annotated both and combined them the two Group Netlists using `group_netlist_merger` by specifying the Group IDs of each board's respective connector. This was a very quick process.

We conclude that the electrical engineer can annotate schematics relatively quickly, especially when the schematics use consistent naming. For snippet based design the electrical engineer only needs to annotate schematics when creating new snippets. Therefore, the electrical engineer does not need to annotate the schematics when creating new PCDU schematics. This fulfills DG2.

6.1.2 Generating partial PLUTO PCDU Firmware

After having annotated the PLUTO PCDU schematics, we generated its Group Netlist and created Jinja2 templates for `code_gen`. Similar to annotating the schematics the main hurdle was nomenclature (see subsection 6.1.1). The nomenclature of the existing firmware's variables did not resemble the names the schematics used. Mapping from schematic names to firmware names is tedious and prone to errors. Therefore, we chose to refactor the firmware and adopt a consistent nomenclature across both the schematics and the firmware. We performed the renaming using `sed`. Producing the templates themselves

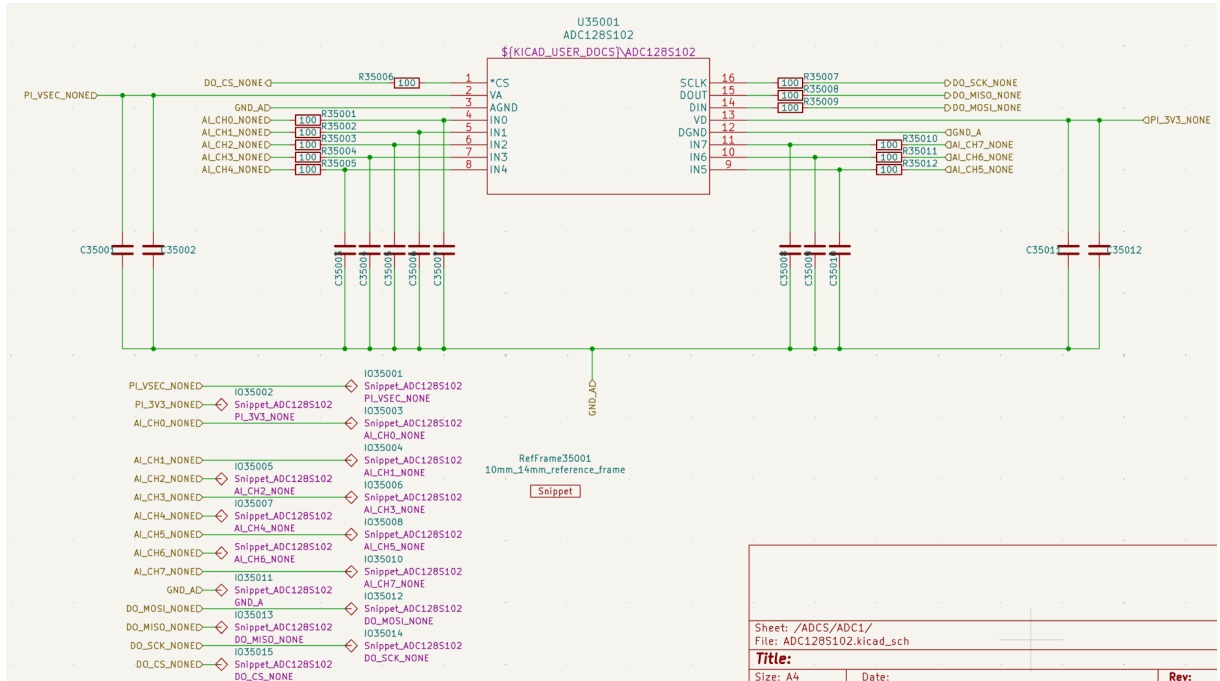


Figure 6.1: The schematics of the ADC snippet.

was an unsophisticated and quick process. Therefore, we argue that in combination with subsection 6.1.1 we fulfill DG2.

We wrote templates for the board definition file and the LCL driver. Unfortunately, the firmware performing housekeeping using the ADCs requires not just the information of what ADCs there are and what they connect to. It also needs the TMTC packet specification to know how to transmit the telemetry to earth. However, to write the housekeeping C++ code the software engineer does not require the schematics. That is because we extended the board definition file to include what groups connect to which ADC. Combined with the TMTC packet specification this is all the information the software engineer requires. Thus, we argue that we successfully fulfilled DG1.

Figure 4.7 shows the generic template we use for all snippets to generate the board definition. Figure 4.8 shows an abbreviated form of the output. The generated files exactly replace existing files that were previously manually written. Therefore, the integration was quick, too. Additionally, because all output our tooling produces is deterministic, version control and identifying differences between outputs is easy. Lastly, we run ClangFormat [28] after generating C++ files to adhere to coding styling.

We argue that this process reduces the risk of human errors during firmware development, e.g., accidentally specifying the wrong pin in the pin definitions. Furthermore, we detected a faulty electrical connection in the schematics using our tooling: The PLUTO PCPU's LCLs each have four control pins (On, On Redundant, Off and Off Redundant). An LCL on the PCPU SOLBAT board, however, only had a connection to On and On Redundant. It should have been On and Off. Therefore, this LCL could not shut off. Our tooling detected the fault by throwing an error during C++ driver code generation. The error stated that the microcontroller did not connect to any Off pin on this LCL. After noticing the error the electrical engineer had to manually fix the error on the PLUTO's flight model.

	A	B	C	D	E
1	schematic	group_path	group_type	pin_name	other_pins
2	ICA_EPS_Distribution	/	Connector_1	MountPin	
3	ICA_EPS_Distribution	/	Connector_1	Pin_1	
4	ICA_EPS_Distribution	/	Connector_1	Pin_2	This_was/Simplified/Away/GND
5	ICA_EPS_Distribution	/	Connector_1	Pin_3	
6	ICA_EPS_Distribution	/	Connector_1	Pin_4	
7	ICA_EPS_Distribution	/	Connector_1	Pin_5	ICA_EPS_Distribution/Controller/Controller/PB4
8	ICA_EPS_Distribution	/	Connector_1	Pin_6	
9	ICA_EPS_Distribution	/	Connector_1	Pin_7	This_was/Simplified/Away/GND
10	ICA_EPS_Distribution	/	Connector_1	Pin_8	This_was/Simplified/Away/GND
11	ICA_EPS_Distribution	/	Connector_1	Pin_9	ICA_EPS_Distribution/Controller/Controller/PA13
12	ICA_EPS_Distribution	/	Connector_1	Pin_10	ICA_EPS_Distribution/Controller/Controller/PA14
13	ICA_EPS_Distribution	/	Connector_1	Pin_11	ICA_EPS_Distribution/Controller/Controller/PB3
14	ICA_EPS_Distribution	/	Connector_1	Pin_12	ICA_EPS_Distribution/Controller/Controller/PA15
15	ICA_EPS_Distribution	/	Connector_1	Pin_13	ICA_EPS_Distribution/Controller/Controller/NRST
16	ICA_EPS_Distribution	/	Connector_1	Pin_14	
17	ICA_EPS_Distribution	/	Connector_1	Pin_15	ICA_EPS_Distribution/Controller/Controller/PA10 ICA_EPS_Distribution/Controller/USART1/RX
18	ICA_EPS_Distribution	/	Connector_1	Pin_16	ICA_EPS_Distribution/Controller/Controller/PA9 ICA_EPS_Distribution/Controller/USART1/TX
19	ICA_EPS_Distribution	/APR Interface/	Connector_2		1 ICA_EPS_Distribution/DIST_3V3_AUX/RLCL_AUX/Snippet_RLCL/PI_MAIN_NONE
20	ICA_EPS_Distribution	/APR Interface/	Connector_2		2 ICA_EPS_Distribution/DIST_3V3_AUX/RLCL_AUX/Snippet_RLCL/PI_MAIN_NONE
21	ICA_EPS_Distribution	/APR Interface/	Connector_2		3 This_was/Simplified/Away/GND

Figure 6.2: An extract of the CSV `netlist_to_csv` generates for harness specification of the PLUTO PCDU.

This human oversight likely happened because the LCL on the SOLBAT board only indirectly connects to the PCDU Distribution board, where it connects to the microcontroller. Consequently, to comprehend this electrical connection the engineers must manually trace wires across five KiCad sheets, which is prone to such errors. Our tooling, instead, performs this tracing automatically and is not prone to this error.

We conclude that the software engineer can quickly adapt to our `kiCad_firmware_generation` tooling. Once the firmware uses our tooling, the software engineer saves time and has a chance of catching errors. This is especially true when the variable names already match the nomenclature the schematics use.

6.1.3 Generating a Harness Specification

After having created a Group Netlist for the PLUTO PCDU, we evaluated how the information in the Group Netlist can enhance the harness specification process (see section 2.4). We used `netlist_to_csv` with the `root-group-glob` parameter set to `**/Connector*` and the `simplify-pins` parameter to `GND` (see section 4.5). This removed the excessive ground nets and focuses on the connectors, which the harness connects to. We prefixed the Group Types that represent such a connector with `Connector_` in the schematics.

This generates a CSV answering four important questions to the harness engineer: What connectors of what type does the subsystem have, what pins have they and what are they connected to?

Previously, the harness engineer had to manually search through the PCDU schematics to answer these questions. Figure 6.2 shows an extract of such an CSV. With the CSV generated by our tooling the harness engineer does not require the schematics anymore. Instead, she can focus on the interaction between subsystems, e.g., what the PCDU connects to. This saves time and removes chances for human error.

Future work could go further and generate the entire harness specification from a Group Netlist. A possible approach is to transfer the information from the Group Netlist into the system design in HaDes. HaDes uses the system design to generate the harness specification (see subsection 2.4.1). Consequently, we argue that we fulfill DG3.

6.1.4 Snippet based Design Analysis

While Snippet based Design Analysis (see subsection 2.5.1) remains an unproven idea, our work and specifically the Group Netlist abets research in this direction. A few dozen lines of Python code transform a Group Netlist into the graph data structure Snippet based Design Analysis requires. The only hurdle was producing unique names for each Net. The Group Netlist purposefully leaves out such names for reasons of simplicity. A possible solution is to hash each Net and use that hash as its unique name.

6.2 Benchmark

The PLUTO PCDU project serves not just to verify the above qualitative metrics but also to perform a quantitative benchmark. We measured the time it takes to

- create the KiCad netlist for both boards using the KiCad-CLI,
- convert both KiCad netlists to Group Netlists using `kicad_group_netlister`,
- merge both Group Netlists into a combined Group Netlist using `group_netlist_merger`,
- generate two firmware files, a combined 1153 lines, using `code_gen` and
- generate a CSV for harness specification using `netlist_to_csv`.

Our benchmark ran on an Intel Core Ultra 7 255H and we measured an average runtime of 1.7s. Compared to compiling the firmware, which takes roughly 40s, we argue this to be negligible. The PLUTO PCDU uses consists of two six-layer PCBs a combined 2081 components. The combined Group Netlist contains 62 Groups and 402 Nets. Therefore, we conclude that even though we explicitly did not optimize for runtime efficiency our tooling is adequately fast.

6.3 Future Work: Modifying KiCad to Simplify Annotations

As we have explained in section 5.2 we chose not to use the `.kicad_sch` file directly. Instead, we use the KiCad netlist in the `kicadxml` format. This ensures that the Groups in the Group Netlist connect iff the respective components connect in the KiCad schematic. However, our tooling cannot read the net label names. Therefore, our tooling requires manually annotating each relevant net label as Figure 5.2 shows.

Future work could change KiCad and modify it in a way that stores this information in the KiCad netlist. This would make annotating snippets trivial.

Chapter 7

Conclusion

We have shown that our proposed Group Netlist can replace considerable manual labor in the spacecraft PCDU development process. This includes not just the initially planned C++ firmware generation (see subsection 6.1.2) but also harness specification (see subsection 6.1.3) and Snippet based Design Analysis (see subsection 6.1.4). Furthermore, we showed that our tooling can generate Group Netlists not only for the PLUTO PCDU's Distribution board. Instead, our `group_netlist_merger` enables combining information from multiple boards (see section 4.4). This unlocks crucial insights across large systems, identifying errors in the schematics. We used this ability to identify and fix an error in PLUTO's flight model, which is expected to be in orbit this year (see subsection 6.1.2). Lastly, separating confidential PLUTO information from our tooling allowed us to publish the tooling as Open-Source software (see section 5.10). Therefore, we believe that our contribution may serve as an invaluable accelerant in other projects, too, even outside the space sector. Future work could add support for different PCB design suites, automatically generate technical documentation or reach otherwise unrelated areas like generating accurate project art.

Prior to our contribution, the requirements for an abstract hardware specification remained unexplored. Our rapid iteration enabled experimenting with different concepts before zeroing in on our proposed Group Netlist (see section 4.1). We believe figuring this out without such experiments to not have been feasible.

Our initial decision to use a pipeline architecture and key technologies, like the `kiCadxml` netlist and `Jinja2`, however, remained unchanged throughout the development.

Concluding, our constant strive towards simple solutions lead to a feature-complete, ready-to-use but easy to understand and maintain product (see [4]).

List of Figures

2.1	PCBs.	10
2.2	A resistor connected to another component and the <code>GND</code> net.	11
2.3	Synthetic KiCad schematics. This example is unrelated to the PCDU. . . .	13
2.4	Hierarchical sheet and symbol. Notice that the hierarchical labels in (b) connect the hierarchical symbol's pins in (a).	14
2.5	Figure 2.6 shows the output this template produces. Our tool <code>code_gen</code> defines the function <code>glob_groups</code> (see section 4.3). Furthermore, our Python library defines the <code>Group</code> class this template uses (see subsection 4.1.4). . .	15
2.6	The output <code>code_gen</code> produces using Jinja2 and Figure 2.5 as input. Figure 4.2 shows the data the output reflects.	16
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 inbetween. The <code>group_netlist_merger</code> takes multiple Group Netlists and combines them into a single one (see section 5.8).	22
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 bottommost 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.	24
4.3	Example Group Netlist XML visualized in Figure 4.2. Figure 4.4 shows the full <code>groups</code> tag and Figure 4.5 the full <code>nets</code> tag.	25

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.	26
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.	27
4.6	KiCad’s symbol properties interface for two example symbols; one with implicit and one with explicit pin naming.	31
4.7	Jinja2 template for C++ modm pin mapping [22]. The main Jinja2 template file sets the group variable to a Group to generate C++ code for. Afterwards, it includes this template file.	34
4.8	C++ code output of Figure 4.7 using the same highlighting. Notice that Figure 4.7 only produces the Dist12vBLc1Ext1 namespace. Furthermore, we added comments with square braces that the template did not produce.	35
4.9	Example Group Netlist in different stages of <code>group_netlist_merger</code>	37
5.1	The KiCad user interface showing a custom netlist exporter. It uses <code>ki-cad_group_netlister</code> to generate a Group Netlist from the schematics the user opened. KiCad replaces %0 with the desired output file path and %I with the path to the <code>ki-cad.xml</code> netlist [15]. Notice how the netlist exporter does not have access to the <code>.ki-cad.sch</code> schematics file but does have access to the KiCad netlist.	42
5.2	Annotating KiCad net labels (on the left) with Group_I0 symbols (on the right). A <code>ki-cad_group_netlister</code> implementation reading the <code>.ki-cad.sch</code> file directly or modification to KiCad could remove the need for this. Figure 6.1 shows the snippet’s entire schematics.	44
5.3	Overview of the first generator design. We hide some of the external tools shown in Figure 4.1.	44
5.4	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.	47
6.1	The schematics of the ADC snippet.	53
6.2	An extract of the CSV netlist_to_csv generates for harness specification of the PLUTO PCDU.	54

Bibliography

- [1] Niklas Aksteiner, Rene Schulz, and Janis Sebastian Häseker. “Snippet Based Electronics Design for Spacecraft Avionic Controllers.” In: *Journal of Evolving Space Activities*. Journal of Evolving Space Activities 1 (2023). ISSN: 2758-1802. DOI: [10.57350/jesa.31](https://elib.dlr.de/197563/). URL: <https://elib.dlr.de/197563/>.
- [2] *Ansible*. Accessed: 2026-01-23. URL: <https://www.redhat.com/en/ansible-collaborative>.
- [3] Graham Keeth; Jon Evans; Glenn Peterson; David Jahshan; Phil Hutchinson; Fabrizio Tappero; Christina Jarron; Melroy van den Berg. *Introduction to KiCad 9.0*. 90da21fb. Accessed: 2025-11-03. Free Software Foundation. Oct. 2025. URL: https://docs.kicad.org/9.0/en/getting_started_in_kicad/getting_started_in_kicad.html.
- [4] Christopher Besch. *kicad_firmware_generation*. Accessed: 2026-01-22. Jan. 2026. URL: https://github.com/DLR-RY/kicad_firmware_generation.
- [5] Norbert Brinz. “C/C++ Treibergenerierung eines Steppermotors aus Schaltplänen.” May 2025.
- [6] KiCad Contributors. *KiCad IPC API*. Accessed: 2025-11-05. Free Software Foundation. Apr. 2025. URL: <https://dev-docs.kicad.org/en/apis-and-binding/ipc-api/>.
- [7] KiCad Contributors. *KiCad Library Utils*. Accessed: 2025-11-05. Oct. 2025. URL: <https://gitlab.com/kicad/libraries/kicad-library-utils>.
- [8] Pat Deegan. *kicad skip*. Accessed: 2025-11-05. Feb. 2024. URL: <https://github.com/psychogenic/kicad-skip>.
- [9] Malhar Deshmukh. *Generate pin definition header file form KiCAD schematic*. Accessed: 2025-11-05. Nov. 2019. URL: <https://github.com/DeshmukhMalhar/sch2header>.
- [10] ECSS. *latching current limiter (LCL)*. Accessed: 2026-01-28. URL: https://ecss.nl/item/?glossary_id=911.
- [11] esa. *Anatomy of a Spacecraft*. Accessed: 2026-01-27. URL: https://www.esa.int/Science_Exploration/Space_Science/Anatomy_of_a_spacecraft.
- [12] esa. *Telemetry & Telecommand*. Accessed: 2026-01-28. URL: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Onboard_Computers_and_Data_Handling/Telemetry_Telecommand.

- [13] Python Software Foundation. *Python*. Accessed: 2026-01-10. URL: <https://www.python.org>.
- [14] The Free Software Foundation. *sed*. Accessed: 2026-01-28. URL: <https://www.gnu.org/software/sed/manual/sed.html>.
- [15] Jean-Pierre Charras; Fabrizio Tappero; Wayne Stambaugh; Graham Keeth. *KiCad 9.0 Reference Manual*. 90da21fb. Accessed: 2025-11-03. Free Software Foundation. Oct. 2025. URL: <https://docs.kicad.org/9.0/en/eeschema/eeschema.html>.
- [16] KiCad. *KiCad*. Accessed: 2025-11-03. 2025. URL: <https://www.kicad.org>.
- [17] Michał Kruszewski and Wojciech Marek Zabołotny. “Safe and Reusable Approach for Pin-to-Port Assignment in Multiboard FPGA Data Acquisition and Control Designs.” In: *IEEE Transactions on Nuclear Science* 68.6 (2021), pp. 1186–1193. DOI: [10.1109/TNS.2021.3074530](https://doi.org/10.1109/TNS.2021.3074530).
- [18] The Python Standard Library. *glob — Unix style pathname pattern expansion*. Accessed: 2026-01-07. Jan. 2026. URL: <https://docs.python.org/3/library/glob.html>.
- [19] Marvin Mager. *kiutils*. Accessed: 2025-11-05. May 2024. URL: <https://github.com/mvnmgrx/kiutils>.
- [20] Shane Mattner. *KiCad Schematic API*. Accessed: 2025-11-05. Nov. 2025. URL: <https://github.com/circuit-synth/kicad-sch-api>.
- [21] Nasa. *Apollo 13 Mission Audio*. Accessed: 2026-01-27. URL: <https://apolloinrealtime.org/13>.
- [22] modm authors Niklas Hauser. *modm*. Accessed: 2026-01-09. URL: <https://modm.io>.
- [23] Pallets. *Jinja2*. Accessed: 2026-01-08. URL: <https://jinja.palletsprojects.com>.
- [24] the mypy project. *mypy*. Accessed: 2026-01-10. URL: <https://www.mypy-lang.org/>.
- [25] Shashank Shailabh. “Template Engines by Language.” Accessed: 2025-11-05. Jan. 2026. URL: <https://github.com/sshailabh/awesome-template-engine>.
- [26] DLR Institute of Space Systems. *pando-core*. Accessed: 2026-01-23. May 2017. URL: <https://github.com/DLR-RY/pando-core>.
- [27] DLR Institute of Space Systems. *PLUTO*. Accessed: 2026-01-23. 2026. URL: <https://www.dlr.de/en/irs/research-transfer/missions-and-projects/pluto>.
- [28] The Clang Team. *ClangFormat*. Accessed: 2026-01-26. 2026. URL: <https://clang.llvm.org/docs/ClangFormat.html>.
- [29] Dave Vandembout. *kinparse*. Accessed: 2025-11-05. June 2025. URL: <https://github.com/devbisme/kinparse>.
- [30] Steffen W. *KiCadFiles*. Accessed: 2025-11-05. Oct. 2025. URL: <https://github.com/Steffen-W/KiCadFiles>.
- [31] Georg Wehrli. “Generating Platform Configuration from Netlists.” May 2024. URL: <https://doi.org/10.3929/ethz-b-000684943>.