Automating Firmware Generation from Schematics for Microcontroller based Designs

Christopher Besch*§, Janis S. Häseker§, Hassan Nassar*, Norbert Tóth§, Jörg Henkel*
*Chair for Embedded Systems, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
§Institute of Space Systems, German Aerospace Center (DLR), Bremen, Germany
mail@chris-besch.com, janis.haeseker@dlr.de, hassan.nassar@kit.edu, norbert.toth@dlr.de, henkel@kit.edu

This work is a preprint of our paper published in IEEE-ESL. We will make the author version freely available by October 2027.

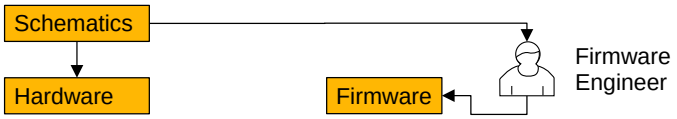
***Abstract*—The interaction between hardware design and firmware implementation requires strict consistency in embedded systems. Firmware must exactly match the schematic-level component configuration and connectivity. In current workflows, this consistency is largely ensured through manual inspection and extensive testing. In this work, we introduce a schematics-aware approach to automate firmware generation. We propose the *Group Netlist*, a design- and PCB tool-agnostic abstraction of hardware. Using a reference implementation, we demonstrate our proposal for the open-source PCB design suite KiCad. As added benefit our approach serves for hardware verification purposes that we demonstrate through a fault-injection analysis. Considering relevant cases, our tool detects 75% of otherwise undetected injected faults and requires negligible overhead (less than 0.2% codebase addition and on average a 4.8% runtime increase). Lastly, we verify our tooling by successfully using it in the development workflow of an embedded satellite module.**

I. Introduction

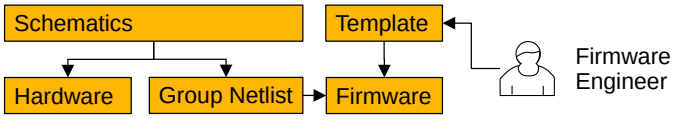
The development of embedded devices requires coordination across multiple engineering domains. A single product may combine power and high-speed digital electronics, mechanical and thermal constraints, as well as software components and interfaced modules [1]. Therefore, well-defined design processes are necessary to ensure consistency between these aspects.

From an electronics design perspective, methodologies such as snippet based design [2] are becoming increasingly common. Hardware designs compose reusable snippets, representing groups of components with well-defined functionality and Printed Circuit Board (PCB) layouts. The reuse of verified circuits and the reduction of design effort to establishing inter-snippet connectivity enable rapid prototyping, particularly for bespoke devices. However, this abstraction provides limited support for the accompanying firmware development. Electrical engineers can efficiently derive new hardware configurations. Firmware engineers, however, must still manually reconstruct the system configuration from schematics, see Figure 1a. This includes the microcontroller pin assignments but also the channel usage of Analog to Digital Converters (ADCs).

This work was partially supported by BMFTR (DI-EDAI - 16ME0990K).



(a) The state-of-the-art workflow of developing embedded devices. Notice how there the communication from the hardware to the firmware development side is entirely manual; the firmware developer must laboriously read the schematics.



(b) Our proposed workflow. Instead of just defining the hardware, the schematics enable automatic firmware generation. The firmware engineer writes templates, omitting the details from the schematics to be automatically provided.

Fig. 1: Workflow transformation.

Previous works on embedded design generation has explored generating embedded designs from higher-level specifications [3], [4], but the schematic-to-firmware consistency problem remains largely manual in practical PCB workflows. Our approach eliminates this manual translation step through an automated schematics-aware process, generating like in Figure 2. Firmware development shifts from inspecting schematics to specifying Jinja2 [5] templates. These templates represent firmware source files containing placeholders that our tooling automatically populates with connectivity information extracted from schematics. Additionally, template mechanisms such as loops and function calls enable the generation of complex firmware structures. The template might be reused together with the snippet's schematics and PCB layout for numerous embedded devices, reducing the required effort.

To support this workflow, we introduce a novel abstract representation of hardware, the *Group Netlist*, see Figure 1b. The proposed abstraction is not limited to direct pin assignments but also captures transitive connectivity relationships; for example when analog signals are routed through ADCs before reaching the microcontroller. Figure 3 shows how the STM32 uses an ADC to interpret the temperature of a DCDC.

II. *Group Netlist* and Tooling for KiCad

We propose *Groups*, collections of components implementing well-defined functions. Each *Group* exposes *Pins*,

```
namespace Dist12VDCDC {
typedef modm::platform::GpioOutputA1 EnablePin;
typedef Adc2::Ch0 Temperature;
}  // namespace Dist12VDCDC

namespace Adc2 {
typedef Spi3::Handler Spi;
typedef modm::platform::GpioOutputA4 CSPin;
typedef void Ch0;
typedef void Ch1;
}  // namespace Adc2

namespace Spi3 {
typedef modm::platform::SpiMaster3 Handler;
typedef modm::platform::GpioA7::Sck SckPin;
typedef modm::platform::GpioA6::Miso MisoPin;
typedef modm::platform::GpioA5::Mosi MosiPin;
}  // namespace Spi3
```

Fig. 2: Excerpt from our tooling output using modm for the satellite module in Figure 5. It uses C++ types: Dist12VDCDC uses Adc2::Ch0 to specify the ADC and channel for its temperature reading. Adc2 uses Spi3::Handler and modm::platform::GpioA4 to select the SPI interface and chip-select pin. Thus, the generated code documents the entire transitive connection.

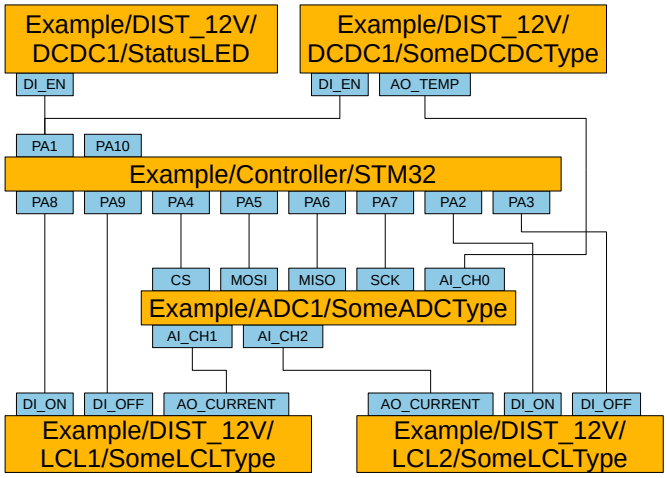


Fig. 3: Extracted and simplified visualization of the satellite module's (see Figure 5) *Group Netlist*. Each box is a *Group*; small blue rectangles are *Pins*, with lines indicating their connections. Unlike traditional netlists, the *Group Netlist* emphasizes logical connections from the firmware's perspective. Individual hardware components within a Group are irrelevant here and therefore omitted.

representing selected pins of its constituent components. As such, a *Group* abstracts internal structure and hides irrelevant connectivity behind a defined interface. A snippet may consist of one or more *Groups*.

The tuple of *Group Type*, *Group Path* and *Group Schematic* identify any *Group*. The *Group Type* should be the same for *Groups* with the same hardware. The *Group Path* differentiates between *Groups* of the same type on a board and the *Group Schematic* allow identifying *Groups* across large designs spanning multiple boards. This simple approach is, therefore, effortlessly scalable to large and complex designs. A *Group Netlist* is an `XML` file specifying which *Groups* compose a hardware design and through which *Pins* they interconnect. Figure 3 visualizes an example *Group Netlist*. To facilitate creation, transformation, and consumption of the *Group Netlist*, we provide a Python library[1].

For KiCad, we provide a reference implementation, which

[1]We provide our ready-to-use tooling as Open-Source software under: https://github.com/DLR-RY/kicad_firmware_generation

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

{% endfor %}
```

(a) An example Jinja2 template. Notice that the *Group Glob* "**" matches all *Groups*. This simple template generates a list of all *Groups* printing their *Group Schematic*, *Group Path* and *Group Type*

```
Example
/Dist_12V/DCDC1/
StatusLED

Example
/Dist_12V/DCDC1/
SomeDCDCType

Example
/Controller/
STM32

Example
/ADC1/
SomeADCType

Example
/DIST_12V/LCL1/
SomeLCLType

Example
/DIST_12V/LCL2/
SomeLCLType
```

(b) The output of using the template in Figure 4a with the *Group Netlist* in Figure 3. Notice that the *Group Schematic* is all "Example" because this example uses a single board.

Fig. 4: Template example

generates *Group Netlists* from kicadxml netlists. The KiCad-CLI already exports netlists in the kicadxml format, which captures full component-level connectivity but is too verbose and unstructured for firmware generation. Hence, we implement a program translating it into a *Group Netlist*; leveraging the kicadxml netlist guarantees consistency between schematic connectivity and the resulting *Group Netlist* abstraction. Furthermore, KiCad's schematic hierarchy conveniently maps to *Group Path* and *Group Schematic*, when an entire schematic sheet constitutes a *Group*. Consequently, the electrical engineer must only declare what group of components or entire schematic sheets are *Groups* and define useful *Group Types*. When designs span multiple PCBs, there typically are multiple schematic projects. We provide an additional program to merge the corresponding *Group Netlists*, supporting cross-project system analysis. The *Group Schematic* identifies *Groups* across multiple boards.

To utilize the *Group Netlist* for firmware development, we provide a third program, interfacing the abstraction with Jinja2 templates. Figure 4a shows an example template. This mechanism enables firmware generation across arbitrary programming languages. By automating a previously manual translation step, the approach reduces the likelihood of human-induced inconsistencies. Beyond firmware generation, related engineering workflows impose similar consistency requirements. For instance, cable harness specification depends on the connector-level pin assignments rather than microcontroller pins alone. Therefore, we provide a fourth program converting the *Group Netlist* into a CSV pinout table for all

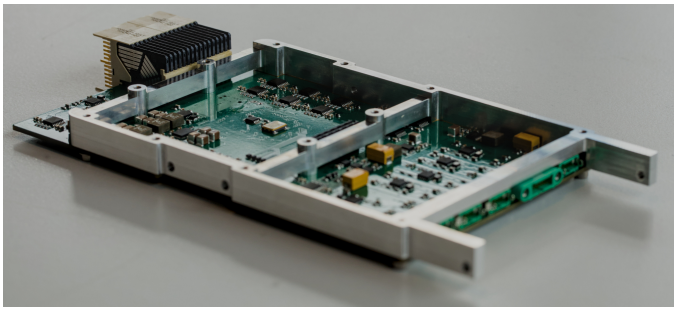


Fig. 5: Image of the satellite module for our case study.

connectors. This automation eliminates repetitive schematic inspection and reduces the potential for manual errors. These examples illustrate broader applicability of the *Group Netlist* abstraction. Finally, since both our reference implementation and the KiCad-CLI are deterministic terminal-based programs, the entire workflow is completely reproducible. Therefore, it naturally integrates into continuous integration pipelines. Such automation enables agile hardware/software codesign, in which firmware artifacts adapt continuously to evolving hardware designs.

III. Practical Evaluation

To practically evaluate our tooling, we integrated it into the development process of an STM32-based satellite module, see Figure 5. This module is part of the electrical power system (EPS), conditioning and distributing power to all the satellites' systems. The module consists of 67 functional blocks with 19 unique circuit configurations counting 1914 components. Together with 41 external interfaces this results in an overall complex electronics design.

Our tooling stack generates firmware directly from schematics based on templates. This firmware, in turn, uses the barebone embedded library generator modm [6]. Figure 2 shows an extract from that generated code. This eliminates the need for recurrent manual schematic inspection to determine pin assignments. The initial template creation required only a few hours of development effort. Once defined, templates adapt automatically to subsequent hardware modifications. Since the satellite module follows a snippet based design methodology, we introduced annotations once per snippet. Consequently, new designs reusing the same snippets require no additional annotation effort. While a snippet based design already constrains human errors to inter-snippet connectivity, our tooling explicitly targets this remaining source of inconsistencies. Driver firmware inherently imposes connectivity requirements on the controlled hardware components. Missing or incorrect connections, therefore, manifest as violations during firmware generation. Our tooling demonstrated its verification capability in the case study, where it detected an incorrectly wired component, lacking a required digital pin connection. Such schematic-level faults may otherwise remain undetected until functional verification through testing. Therefore, we observe that our tooling effectively contributes to design verification without incurring extra cost.

IV. Fault Injection Analysis

To quantify the verification capability of our tool, we perform a fault-injection analysis on the satellite module used in the case study. We systematically inject one schematic modification at a time using the kicad skip library [7]. The injected modifications cover three classes of schematic changes: swapping net labels, replacing labels with no-connect symbols, and deleting wires.

For each injected change, we run the unmodified firmware generation and compilation workflow. We refer to this complete process as building. If building fails, we count the injected change as a detected fault. However, not every injected modification represents a functional design fault. For example, swapping two ADC channels is not necessarily faulty in our setup. Because the firmware is generated from the resulting schematic, it adapts to the selected ADC channel. In such case, the modified design always produces correct firmware.

Manually classifying all 8885 injected changes is prohibitively expensive. Therefore, we manually inspect a random sample of 300 injected changes and label whether each change represents an actual design fault. When the effect of a change is unclear, we conservatively label it as a real fault to avoid over-claiming the detection capability of our tool. None of the sampled changes classified as non-faulty caused a build failure. We therefore assume that build failures correspond to actual schematic faults in this analysis.

Figure 6 summarizes the results. Our tool detects almost no faults internal to *Groups*, which is expected. This is because the *Group Netlist* intentionally abstracts away the internal component-level structure of each *Group*. Instead, it captures the interfaces and interconnections between *Groups*, which are the parts of the design that must be reflected correctly in firmware. This explains the higher detection rate for *Group Pin* swaps of 33%, where the injected change directly affects the connectivity visible at the firmware interface. Across non-mechanical faults outside *Groups*, our tool detects 15% of injected faults.

This behavior matches the intended use of our abstraction. We expect individual *Groups* to be reviewed and tested as reusable building blocks before being integrated into larger designs. Since the internal structure of a *Group* remains stable across designs, effort spent on validating it can be reused whenever the same *Group* appears again. In contrast, inter-*Group* connectivity is design-specific: it changes with each new board or system configuration, so manual validation must be repeated for every design. Our tool therefore complements conventional schematic checking by bringing the firmware perspective into schematic validation. KiCad's Electrical Rules Checker (ERC), for example, focuses on electrical consistency and is better suited for detecting low-level electrical faults inside *Groups*. However, ERC cannot detect firmware-relevant connectivity faults such as swapped CIPO and COPI lines, or the use of a microcontroller pin that does not expose the required peripheral function, such as SPI. In our tool, such inconsistencies can cause invalid firmware types or compilation errors, enabling early detection during firmware generation rather than post-fabrication testing.

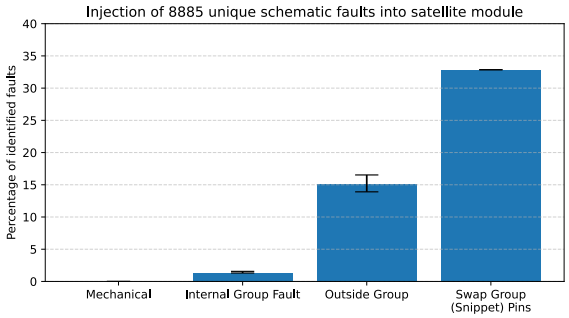


Fig. 6: Results of a fault injection benchmark, grouped by fault type. Error bars show the 95% confidence interval. Since here we consider all *Group Pin* swaps as actual faults, the rightmost column has no error. Our tool detects almost no mechanical faults or faults internal to the *Groups*, but it is effective at detecting faults outside *Groups*, especially those involving *Group Pins*.

To estimate how many faults are detected only by our tool, we manually label the random sample with respect to whether the injected change would be detected by ERC. For this analysis, we only consider changes that ERC would not classify as real faults. In the important case of *Group Pin* swaps, our tool detects **75%** of otherwise undetected faulty swaps. These results show that the proposed tooling is not a replacement for electrical rule checking, but a complementary verification mechanism that detects schematic faults from the firmware integration perspective. This verification capability is obtained without additional effort from the electrical or firmware engineer, as it follows directly from the same *Group Netlist*-based firmware generation tool.

V. Performance Analysis

To obtain quantified performance measurements, we constructed a benchmark comprising 120 test cases. We selected Open-Hardware designs referenced by KiCad's official repository [8]. These range in complexity from simple dev kits to entire x86 laptops. Since these schematics do not contain the required annotations, we introduced synthetic annotations using kicad skip. Each sheet received eight *Group Types*, with 17% (mimicking the satellite module) of all components assigned randomly to one of these *Group Types* alongside unique *Pin Names*. For each design we generated three independent random annotation sets. Additionally, we excluded erroneous designs or ones using outdated formats. Figure 7 plots the runtime of our tooling as a function of schematic size. The runtime of the KiCad CLI generating the kicadxml netlist dominates the overall execution time. Our tool requires only 4.8% of overall execution time. Nevertheless, the combined runtime remains well below one second for typical designs. Even the substantially larger satellite module design requires less than 1.7 seconds. As with the above benchmark, the runtime is bottlenecked by the existing KiCad-CLI. Consequently, we conclude that our tooling introduces negligible runtime overhead. This justifies its usage in even larger and more complex designs.

Finally, while KiCad contains 1.4 million lines of C++ code, our reference implementation comprises only 1312 lines of Python code. That is amounts to a negligible 0.2% codebase addition. Hence, we observe that our proposed approach achieves its functionality with minimal implementation complexity.

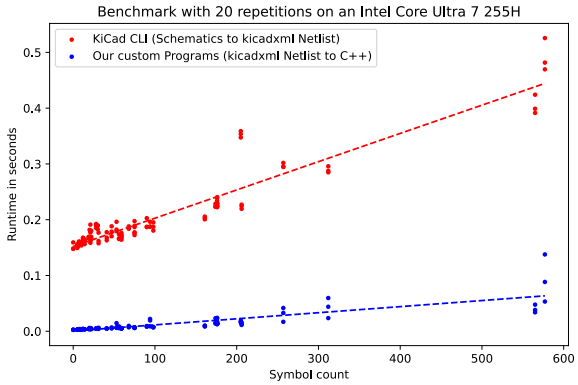


Fig. 7: Runtime results of a synthetic benchmark. Notice how KiCad's netlist generator, not our custom tooling, bottlenecks the runtime. KiCad's netlist generator's overhead is already part of the existing hardware design process.

VI. Conclusion

In this work, we introduce a novel abstract representation of schematics for automated firmware generation. Furthermore, we provide a ready-to-use implementation for KiCad-based designs. It transforms a traditionally laborious process into a lightweight step within the firmware integration pipeline. With a systematically constructed performance benchmark we demonstrate our tooling's feasibility in complex embedded design workflows. We show that the proposed approach achieves low implementation complexity while maintaining negligible runtime overhead. We quantitatively measure our tools verification capabilities using a fault injection analysis. Our approach identifies faults, which other tools would have left undetected. It does so without requiring additional effort. Lastly, we investigate a real-world case study from the space sector The tooling exposed a previously undetected hardware fault, illustrating its capability to contribute to the verification of embedded systems. Our approach integrated smoothly in the existing development process, enabling firmware generated by our tool to be in orbit within a year.

References

[1] S. Safari et al., "A survey of fault-tolerance techniques for embedded systems from the perspective of power, energy, and thermal issues," *IEEE Access*, vol. 10, pp. 12 229–12 251, 2022.

[2] N. Aksteiner, R. Schulz, and J. S. Häseker, "Snippet based electronics design for spacecraft avionic controllers," *Journal of Evolving Space Activities*, Journal of Evolving Space Activities, vol. 1, 2023, ISSN: 2758-1802.

[3] R. Ramesh et al., "Turning coders into makers: The promise of embedded design generation," in *Proceedings of the 1st Annual ACM Symposium on Computational Fabrication*, ser. SCF '17, Cambridge, Massachusetts: Association for Computing Machinery, 2017, ISBN: 9781450349994.

[4] C.-Y. Huang et al., "Netlistify: Transforming circuit schematics into netlists with deep learning," in *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)*, Santa Cruz, CA, USA: IEEE Press, 2025, pp. 1–8.

[5] Pallets, *Jinja2*, 2025. [Online]. Available: https://jinja.palletsprojects.com.

[6] N. Hauser and modm authors, *Modm*, 2025. [Online]. Available: https://modm.io.

[7] P. Deegan, *Kicad skip*, 2024. [Online]. Available: https://github.com/psychogenic/kicad-skip.

[8] KiCad. "Made with kicad." Accessed: 2026-02-27.