Holistic Data Extraction From Technical Documentation For Generating Embedded Software

Master Thesis

Niklas Hauser

The present work was submitted to the

Chair of Communication and Distributed Systems

RWTH Aachen University, Germany

Advisor:

Jan Pennekamp, M. Sc.

Examiners:

Prof. Dr.-Ing. Klaus Wehrle
Prof. Dr.-Ing. Stefan Kowalewski

Registration date: January 11, 2022
Submission date: July 11, 2022

Zentrales Prüfungsamt/Central Examination Office

RWTH AACHEN UNIVERSITY

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Hauser, Niklas	
Name, Vorname/Last Name, First Name	Matrikelnummer (freiwillige Angabe) Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende ~~Arbeit~~/~~Bachelorarbeit~~/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present ~~paper~~/~~Bachelor thesis~~/Master thesis* entitled

Holistic Data Extraction From
Technical Documentation For
Generating Embedded Software

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Aachen, 11.07.2022	
Ort, Datum/City, Date	Unterschrift/Signature

*Nichtzutreffendes bitte streichen
*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Aachen, 11.07.2022	
Ort, Datum/City, Date	Unterschrift/Signature

Abstract

With an ever expanding selection of embedded hardware comes the challenge of porting hardware-dependent software (HdS) to thousands of new devices. Vendors typically provide their HdS libraries only in the C programming language, which cannot be reused if a project uses new programming languages or needs a specialized HdS stack. In these cases, the effort to manually port HdS to new devices is significant, but can be reduced with the use of code generators. However, existing data sources to use for model-driven software engineering are strictly limited to what the vendor chooses to publish in machine-readable formats. In contrast, the technical documentation contains much more data, but is difficult to access due to the print-oriented nature of the portable document format. In this thesis, we design and implement a modular data processor for extracting data from the technical documentation using table processing. To achieve the highest quality and coverage, we then merge this data with machine-readable sources into a single knowledge graph of embedded hardware descriptions. Our evaluation demonstrates the resulting dataset to be of a very high quality and consistency, proving the usefulness of our processor design for future embedded software projects.

Kurzfassung

Die ständig wachsende Auswahl an eingebetteter Hardware bringt die Herausforderung mit sich, hardwareabhängige Software (HdS) auf Tausende von neuen Geräten zu portieren. Hersteller stellen ihre HdS-Bibliotheken in der Regel nur in der Programmiersprache C zur Verfügung, die aber nicht wiederverwendet werden können, wenn ein Projekt andere Programmiersprachen verwendet oder eine spezielle HdS benötigt. In diesen Fällen ist der Aufwand für die manuelle Portierung der HdS auf neue Geräte beträchtlich, kann aber durch den Einsatz von Code-Generatoren reduziert werden. Die vorhandenen Datenquellen sind jedoch streng auf das beschränkt, was der Hersteller in maschinenlesbaren Formaten veröffentlicht. Im Gegensatz dazu enthält die technische Dokumentation viel mehr Daten, ist aber aufgrund des druckorientierten Charakters des Portable Document Formats schwer zugänglich. In dieser Arbeit entwerfen und implementieren wir einen modularen Datenprozessor zur Extraktion von Daten aus dieser Dokumentation mittels Table Processing. Um eine möglichst hohe Qualität und Abdeckung zu erreichen, führen wir diese Daten dann mit maschinenlesbaren Quellen zu einem einzigen Wissensgraphen von Beschreibungen eingebetteter Hardware zusammen. Unsere Evaluierung zeigt, dass der resultierende Datensatz von sehr hoher Qualität und Konsistenz ist, was die Nützlichkeit unseres Designs für zukünftige Embedded-Software-Projekte beweist.

Acknowledgments

I would like to thank my supervisor Jan Pennekamp for his steady guidance throughout this thesis. Our many productive discussions made sure I stayed on schedule and focused on the next steps. His frequent and precise feedback helped me improve my academic writing skills and always pointed me in the right direction regarding the structure and content of this thesis.

Furthermore, I would like to thank the COMSYS team for all the great lectures that got me acquainted with the chair and highly interested in their teaching and research topics. I also want to express my thanks to Prof. Dr.-Ing. Klaus Wehrle for the opportunity to write my thesis on such an interesting and personal topic. I am further thankful to Prof. Dr.-Ing. Stefan Kowalewski for kindly agreeing to take on the task of second examination.

My thanks go out to the many members of the Roboterclub Aachen e.V. for getting me interested in embedded software and always pushing me to find solutions to tricky problems in our libraries their tooling. I also would like to thank my co-maintainers of the modm project, Raphael Lehmann and Christopher Durand, for sharing a vision to improve the state of embedded software development and steadily working to bring it to reality.

Finally, I would like to thank my family and friends for their support during this thesis and especially my parents for their unwavering encouragement and care during these challenging times.

Contents

1 Introduction

With an ever expanding product catalog of embedded hardware comes the challenge of porting the corresponding hardware-dependent software (HdS) stack to thousands of devices [EMD09]. Hardware vendors typically provide a HdS implementation in the C programming language only, since it is de-facto standard for writing embedded software [EMD09]. However, newer compiled languages [modm09, rust17a] and optimized dynamic language runtimes [mpy14, ZB21] bring new programming paradigms and features to resource-limited embedded systems that are simply not supported by C. Unless these new projects can reuse the vendor's HdS stack, they have to write their own, which can be a very time consuming task [Kor18, EMD09].

Porting HdS is mostly a manual process, where a software engineer consults the technical documentation of the device to inform design and implementation decisions [EMD09]. In addition, device-specific hardware description data must be extracted from the documentation and converted into code [EMD09]. However, the technical documentation is often only available in portable document format (PDF), which complicates the extraction of structured data [EHLN06, Ras17] due to its print-oriented content model, rather than a semantic one [pdf08]. As a result, the porting process requires a lot of manual labor [EMD09], which slows down these new projects.

To alleviate these limitations, some projects make use of model-driven software engineering [modm09, rust17a, rust20, rust16, ada15] to code generate large parts of their HdS stack. The data required to feed the generators is extracted from machine-readable sources such as standardized formats [svd15b], proprietary databases from tooling [stm08], or provided by manually curated datasets [i2c11, Fel20]. However, the scope and fidelity of this data is limited to what the vendor publishes, which is usually significantly less than what is available in the technical documentation [modm16, rust20].

Existing work in the field of document information extraction focuses almost exclusively on universal inputs using heuristic approaches to recognize tables in a variety of document formats [EHLN06, KLU15]. Most existing solutions ignore vector graphics in the PDF and instead focus only on the whitespace analysis of text to detect structured content such as tables and figures, which can lower their accuracy [RCVF03, CF04, RPS16, SAM^{+}18, RPS^{+}18]. Since the style and formatting of technical documentation from a specific vendor is known beforehand, a tuned parser guided by both the vector graphics and text can be easier to implement and yield more accurate results [RCVF03, CF04, CD16]. Once the technical documentation content is accessible, it needs to be understood semantically and converted into a useful data format. For generic input, the domain is usually not known in advance, thus the semantics are derived either from the document itself using heuristics [Ras17, AS13], informed by an external ontology [CHCG15], or manually defined [EHLN06]. However, the results can be much less accurate [EHLN06] than when a domain expert guides the content interpretation.

To address these challenges, we design and implement a data processor that accesses technical documentation using table processing and text mining in several modular pipelines. The documentation data is converted using bespoke algorithms written by a domain expert and then merged with machine-readable sources to create the most complete and accurate dataset possible. By combining multiple sources with different device resolutions and data fidelity, we compensate for both while detecting and arbitrating conflicts in a controlled strategy. The resulting data is then encoded unambiguously as a knowledge graph with a custom ontology describing the embedded hardware semantics and a lightweight software wrapper to provide simple data access to code generators. We use our pipeline to extract the interrupt vector table, package and pinout, pin functions, and register descriptions for almost 3000 microcontrollers from STMicro.

We evaluate the performance, implementation effort, and data quality of our technical documentation processor by comparison with the machine-readable data sources. Our implementation extracts and merges data from over 124 thousand PDF pages of documentation in about 2.5 h on consumer hardware while achieving an average of 96.5% data similarity compared to the machine-readable sources. We give a detailed analysis for the remaining data conflicts informed by an in-depth inspection of the sources to formulate the best strategy for arbitrating these conflicts. For the register description data, we successfully apply majority voting to resolve 45–64% of conflicts automatically. These results demonstrate that our pipeline extracts data from the technical documentation with a high quality and resolution, thus providing the necessary foundation to enable HdS porting via code generation. Future work can build on our processor design to support many more embedded software use cases that depend on data only available in the technical documentation.

This thesis is organized as follows. In Chapter 2, we present the background knowledge that is required for the understanding of the remainder of this thesis. In particular, we introduce the PDF format, table processing paradigms, define the HdS stack with standardized formats and tooling, and approaches to knowledge modeling. Chapter 3 contains a description of related work on document information extraction, existing data pipelines, and code generators for embedded software. In Chapter 4, we introduces our scenario of porting HdS to a new microcontroller and discuss why the related work is not sufficient to address the associated challenges. Chapter 5 describes our pipeline design. We explain the implementation of our design in Chapter 6. Afterward, we evaluate the performance, implementation effort, and resulting data quality of our pipeline in Chapter 7. Chapter 8 concludes this thesis.

2 Background

Our work investigates the quality of hardware description data required for creating embedded software and tries to improve on it. Therefore, we provide an introduction into these topics in this chapter. First, in Section 2.1, we give an overview of the available technical documentation and its data format. Subsequently, in Section 2.2, we introduce table processing paradigms that can extract tabular data from such documentation. Then, in Section 2.3, we discuss the functionality of the hardware-dependent software stack and show which data sources already exists that describe the low-level hardware. We conclude this chapter in Section 2.4 with a look at knowledge modeling specifically through knowledge graphs in the context of the semantic web.

2.1 Technical Documentation

When designing a product with electronic components, hardware engineers typically start with a rough sketch of the required components and their interaction and perhaps even build a prototype with any hardware they have at hand [Kul17]. As the requirements become more clear and a custom design is started, the specific components need to be picked [Kul17]. In this manual process, the engineer consults the technical documentation of a set of chosen plausible parts to evaluate whether they satisfy the design criteria [Kul17].

For simple mechanical or passive electronic components such as resistors, capacitors, and connectors, a datasheet with only a few pages is sufficient to describe the electric characteristics, such as voltage and current requirements under various conditions, and physical properties, such as shapes, dimensions, pinouts, and footprints, as well as fulfillment of relevant standards in the electronic space [Kul17]. However, for complex active electronic components like microcontrollers, documents with thousands of pages each are common to describe the specialized internal functionality and their interaction with the external components via (de-facto) standardized interfaces [Kul17]. Such large documents can be split up into multiple parts to reduce

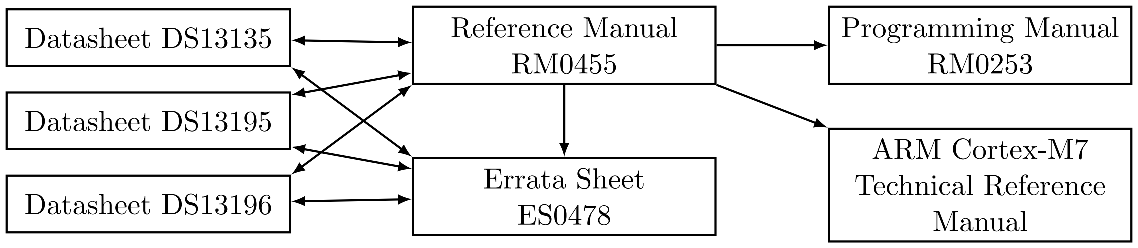


Figure 2.1 This directed graph show the references within a group of related documents describing the STM32H7A3/B0/B3 microcontroller family, with the Reference Manual [RM0455] alone accounting for almost 3000 out of the almost 4000 total pages [DS13139, DS13195, DS13196, ES0478, PM0253]. The technical documentation for all STM32 microcontrollers is split up into multiple such related files per device grouping.

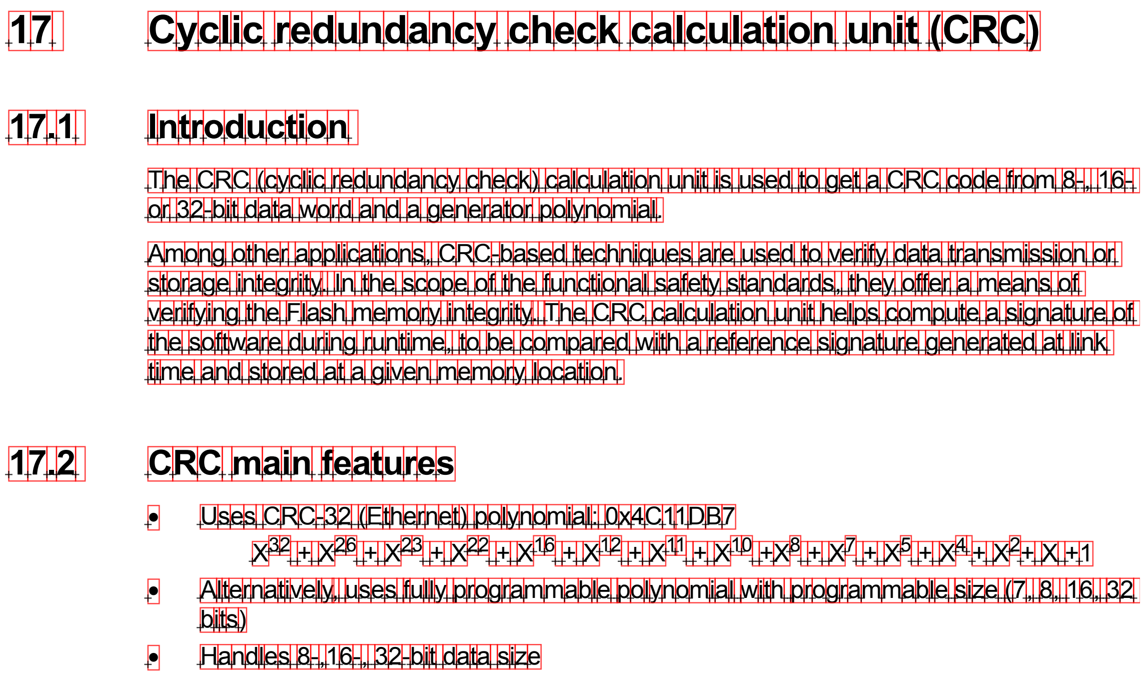
duplication, especially since complex electronic products typically share and reuse a lot of common hardware [Kul17]. For example, STM32 microcontrollers manufactured by STMicro are based on the ARM Cortex-M microprocessor described in the *Programming Manual* and *ARM Cortex-M Technical Reference Manual*, with a set of common peripherals detailed in the *Reference Manual*. Devices are arranged in a number of combinations and packages as described in the *Datasheet*, with any flaws in their design listed in the *Errata Sheet*, as visualized in Figure 2.1.

Hardware vendors publish the technical documentation of their products almost exclusively as portable document format (PDF) files accessible without limitations through the vendor (e.g. st.com) or distributor (e.g. digikey.com) website. PDF documents are a print-oriented format and describe their content as a stream of graphical and textual object that are placed at precise positions inside the document canvas [pdf08]. As a result, documents look and print the same on all platforms, however, all semantic and hierarchical information is lost, which makes it difficult to parse and convert automatically [Ras17]. In this work, we look at PDF documentation from STMicro, whose style is limited to specific formatting building blocks, three of which we describe in more detail next: text, figures, and tables.

Figure 2.2 shows a chapter introduction of the cyclic redundancy check (CRC) peripheral, shortly describing its functionality in textual form and then listing its features as bullet points. Notice how the superscript for $\texttt{X}^{32}\texttt{+X}^{26}\texttt{...}$ is created by placing smaller letters higher in this line of text. Similarly, the heading characters differ only in font size and boldness and the list uses an explicitly placed • character with an indent to denote a new (multi-line) list item. While the structural semantics of this short excerpt may seen obvious to humans, it is difficult to accurately infer the specific formatting of PDF documents using a generalized, heuristic approach, due to its lack of explicit semantic hints [Ras17, SAM$^+$18].

The STMicro technical documentation also contains a lot of graphical figures that supplement the text and tables with additional understanding. While Figure 2.3 is easy for humans to interpret as a function block diagram, the variety of figure styles, each mixing vector graphics and text differently, can make it very difficult for an algorithm to convert such figures into a structured form [CD16].

These format limitations are particularly noticeable for tabular data, which is rendered using a combination of vector graphics to draw the table cells and individually placed text characters as visualized in Figure 2.4.



17 Cyclic redundancy check calculation unit (CRC)

17.1 Introduction

The CRC (cyclic redundancy check) calculation unit is used to get a CRC code from 8-, 16- or 32-bit data word and a generator polynomial.

Among other applications, CRC-based techniques are used to verify data transmission or storage integrity. In the scope of the functional safety standards, they offer a means of verifying the Flash memory integrity. The CRC calculation unit helps compute a signature of the software during runtime, to be compared with a reference signature generated at link time and stored at a given memory location.

17.2 CRC main features

- Uses CRC-32 (Ethernet) polynomial: 0x4C11DB7
  $X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^5 + X^4 + X^2 + X + 1$
- Alternatively, uses fully programmable polynomial with programmable size (7, 8, 16, 32 bits)
- Handles 8-,16-, 32-bit data size

Figure 2.2 This reference manual excerpt shows a chapter heading followed by a section heading with two paragraphs of text. After the next section heading, a bullet point list begins. The bounding boxes of text characters are shown in red with their origins marked by a black cross [RM0432].

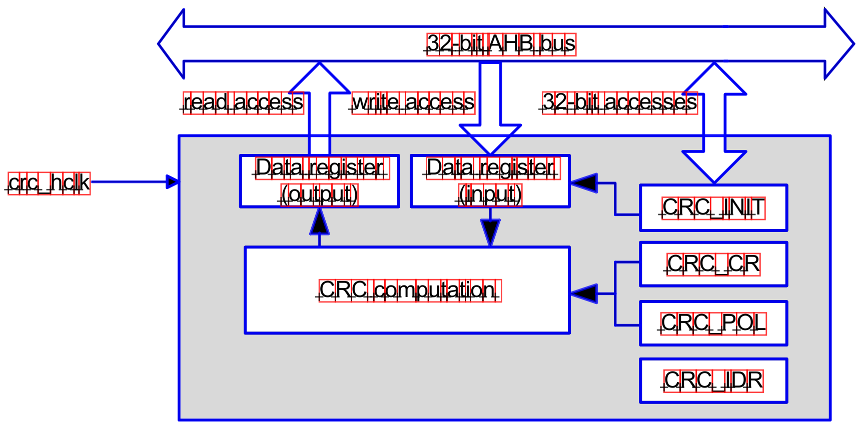


Figure 2.3 The function block diagram of the CRC peripheral showing the relation of the bus and registers. The bounding boxes of the glyphs are shown in red with their origin marked with a black cross, while the graphics are shown in blue. Note the different styles and sizes of arrows in this diagram, which are all explicitly drawn using vector lines [RM0432].

When trying to extract data from such tables without taking the graphics layer into consideration, inferring table layout relies on heuristics applied to the text positions only, which can limit accuracy severely [Ras17, SAM⁺18, CF04]. In contrast, deterministic algorithms primarily guided by the graphics layer produce very accurate and reliable results [RCVF03].

The list in Figure 2.2 can also be interpreted as a table with an implicit header row `CRC main features` and underneath a row for every bullet point. However, the list is heavy on text, requiring a semantic analysis on the grammar itself to make its content accessible to an algorithm [LKM01, CHCG15].

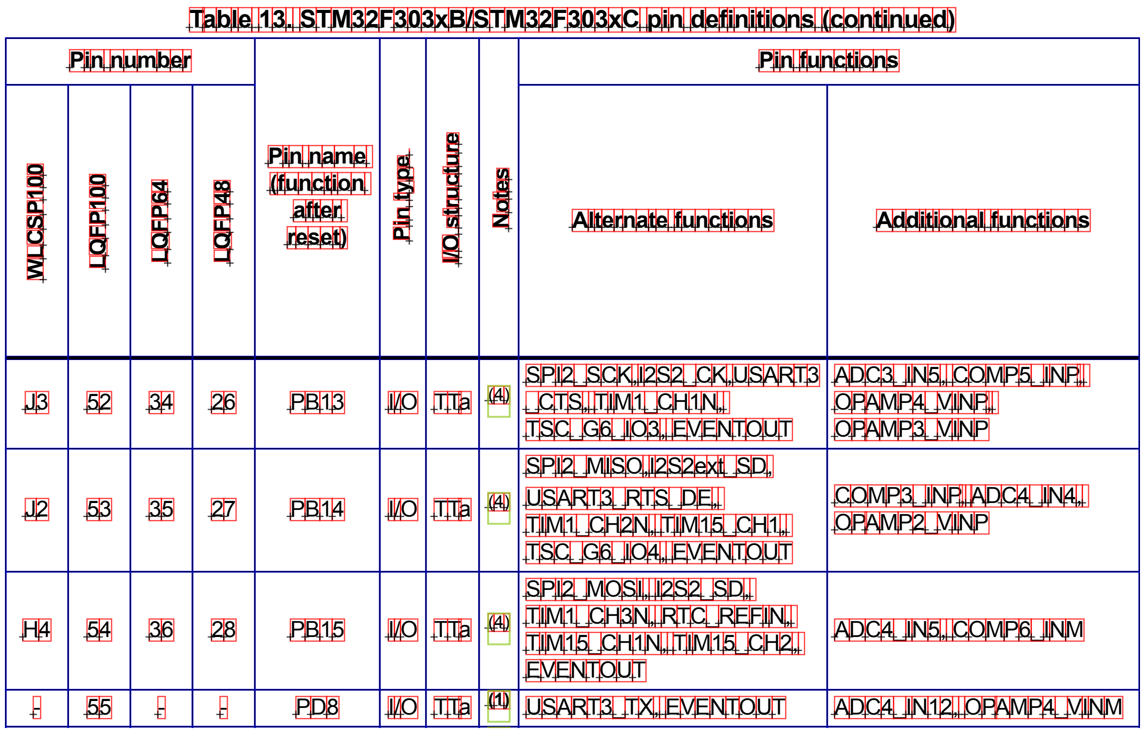


Table 13. STM32F303xB/STM32F303xC pin definitions (continued)

Pin number				Pin name (function after reset)	Pin type	I/O structure	Notes	Pin functions	
WLCSP100	LQFP100	LQFP64	LQFP48					Alternate functions	Additional functions
J3	52	34	26	PB13	I/O	TTa	(4)	SPI2_SCK, I2S2_CK, USART3_CTS, TIM1_CH1N, TSC_G6_IO3, EVENTOUT	ADC3_IN5, COMP5_INP, OPAMP4_VINP, OPAMP3_VINP
J2	53	35	27	PB14	I/O	TTa	(4)	SPI2_MISO, I2S2ext_SD, USART3_RTS_DE, TIM1_CH2N, TIM15_CH1, TSC_G6_IO4, EVENTOUT	COMP3_INP, ADC4_IN4, OPAMP2_VINP
H4	54	36	28	PB15	I/O	TTa	(4)	SPI2_MOSI, I2S2_SD, TIM1_CH3N, RTC_REFIN, TIM15_CH1N, TIM15_CH2, EVENTOUT	ADC4_IN5, COMP6_INM
-	55	-	-	PD8	I/O	TTa	(1)	USART3_TX, EVENTOUT	ADC4_IN12, OPAMP4_VINM

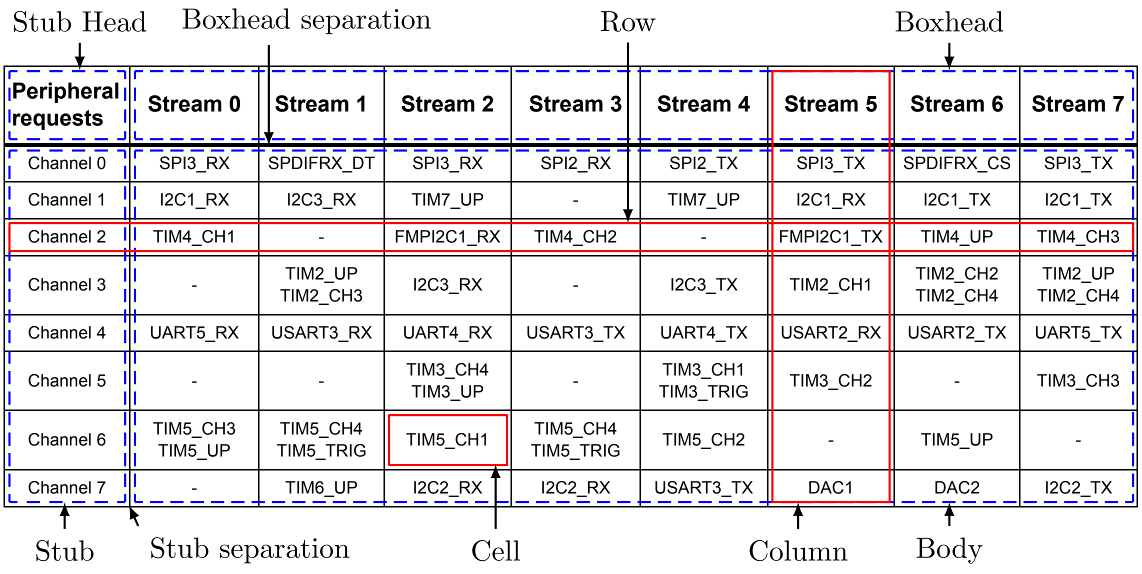
Figure 2.4 This table excerpt shows the bounding boxes of the individual glyphs in red with their origin marked with a black cross. Inter-document links like the footnote markers in the `Notes` column are marked with a green box. The actual table is drawn in the graphics layer with individual paths marked in blue. Note that the table caption contains `(continued)`, since this particular table is very long and is therefore split over multiple document pages [DS9118].

Rather than a full understanding of all unstructured textual descriptions in the STMicro technical documentation, which enters the domain of text mining, we instead focus more on extracting and processing data from tables next.

2.2 Table Processing

Table processing exists at the intersection of several decades-old research communities that deal with extracting data from untagged, but semi-structured input to edit, convert, and format it into semantically valuable information [EHLN06]. Tables present multi-dimensional information in a two-dimensional rendering whose semantic interpretation requires additional information usually present in the context of the table [Hur00].

In their simplest form, tables can be rendered as a row-column structure of cells representing an array of data [Wan96, Hur00]. However, tables express and amend information presented in text form and therefore include implicit hierarchical information as part of their formatting [Hur00, EHLN06]. The visual rendering of tables includes using different text, separator, and border styles, spanning cells spread over multiple rows and/or columns, cells with multi-line content, and even the splitting of the entire table into multiple parts to help fit into the presentation medium dimensions, usually a printable page or a digital display [Wan96, Ras17]. For clarity, we describe table formatting with the terminology defined in Figure 2.5.



Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX	SPDIFRX_DT	SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX	SPDIFRX_CS	SPI3_TX
Channel 1	I2C1_RX	I2C3_RX	TIM7_UP	-	TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1	-	FMPI2C1_RX	TIM4_CH2	-	FMPI2C1_TX	TIM4_UP	TIM4_CH3
Channel 3	-	TIM2_UP TIM2_CH3	I2C3_RX	-	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
Channel 5	-	-	TIM3_CH4 TIM3_UP	-	TIM3_CH1 TIM3_TRIG	TIM3_CH2	-	TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	-	TIM5_UP	-
Channel 7	-	TIM6_UP	I2C2_RX	I2C2_RX	USART3_TX	DAC1	DAC2	I2C2_TX

Figure 2.5 The terminology for the structural parts of a row-column table according to Wang [Wan96]. This DMA trigger table maps peripheral events in the body to streams in the boxhead and channels in the stub [RM0390]. Note that the stub column and boxhead row are chosen depending on the logical table structure, rather than the physical locations of left and top respectively. Long tables may choose to repeat the boxhead row, while others may place the stub in the middle or duplicate it again on the right.

Figure 2.4 depicts a table whose boxhead is indicated by bold text and a bold border, called the boxhead separator, and is organized hierarchically for the pin number and pin functions. However, the stub head of the table is actually the pin name, instead of the pin number on the left, since the pin number can be empty for some packages, as indicated by "-" for the `LQFQ48` package in the fourth row. Additional context is required to know that `TTa` is an abbreviation for `3.3V tolerant analog I/O structure` and that `(1)` and `(4)` refer to footnotes that are rendered as a numbered list underneath the table. Note the use of multi-line cells to fit the list of alternate pin functions, however, with an inconsistent use of line breaks even within words. The caption of the table includes `(continued)` since the table is very long and needed to be split up across multiple PDF pages and the caption and header row is duplicated to aid with comprehension. A conversion algorithm for these tables would first have to merge them back into one big table, removing the duplicated headers, then map the pin name to its pin number and a list of pin functions, while also interpreting the footnotes correctly.

To work with tables more comfortably, an abstraction that separates the logical structure from its layout structure is needed. The Wang abstract model introduces a topology defining the composition of the table headers, an abstract indexing relation for accessing the table content, a set of editing operations, and optional formatting attributes describing the presentation [Wan96].

To describe the logical table topology, Wang [Wan96] defines a *labeled domain* as a labeled empty set or a labeled set of labeled domains. The labeled domain can be visualized by a tree of labels, where each node is called an *item* that can be uniquely identified by the *label sequence* of its path from the root. A leaf node in this tree is called *frontier label* and identifies a labeled domain with an empty set. Figure 2.6 exemplifies these relationships for the previously mentioned table in Figure 2.4.

Boxhead labeled domains:

$$\begin{aligned}
&(Number, \{(WLCSP100, \emptyset), (LQFP100, \emptyset), (LQFP64, \emptyset), (LQFP48, \emptyset)\}),\\
&(Function, \{(Alternate, \emptyset), (Additional, \emptyset)\}),\\
&(Structure, \emptyset),\\
&(Type, \emptyset),\\
&(Notes, \emptyset).
\end{aligned}$$

Stub labeled domain:

$$(Name, \{(PB13, \emptyset), (PB14, \emptyset), (PB15, \emptyset), (PD8, \emptyset)\}).$$

Figure 2.6 The simplified labeled domains for Figure 2.4 split between boxhead and stub. Notice how the labels are independent from the order of the rows and columns of the table, it only describes the logical structure.

Using this topology, Wang [Wan96] defines the indexing relation as a partial function δ that maps from a set of frontier items from different labeled domains to a table entry, also called *attribute-value* pairs in later works [Hur00, EHLN06]. While the abstract model can only describe tables with a regular, multi-dimensional logical structure [Wan96], and therefore is not generally applicable to more exotic table formatting options, all the tables in the STMicro technical documentation conform to such a regular logical structure.

Listing 2.1 describes the first row of the table in Figure 2.4 via such attribute-value pairs. Note that the content of the cells is not modified, as such the line break inside `USART3_CTS` is maintained as well as the superscript and the link of the footnote.

A subsequent algorithm would clean up this data by removing the line breaks correctly, splitting up the pin function string into a list, and resolving the footnotes before converting this information into another representation. For example, the Python dictionary from Listing 2.2 removes the "`-`" Number entries for `PD8`, resolves the footnotes into text, and splits and merges the values of `Structure` into `Type`.

However, any further interpretation and transformation of this data requires more than just a structural understanding informed by the table layout, it requires domain knowledge about the *content* of the table. Therefore, we introduce the topic of hardware-dependent software next.

$$\begin{aligned}
\delta(\{Name.PB13, Number.WLCSP100\}) &= \texttt{J3};\\
\delta(\{Name.PB13, Number.LQFP100\}) &= 52;\\
\delta(\{Name.PB13, Number.LQFP64\}) &= 34;\\
\delta(\{Name.PB13, Number.LQFP48\}) &= 26;\\
\delta(\{Name.PB13, Type\}) &= \texttt{I/O};\\
\delta(\{Name.PB13, Structure\}) &= \texttt{TTa};\\
\delta(\{Name.PB13, Notes\}) &= {}^{(4)} \text{ (including reference)};\\
\delta(\{Name.PB13, Function.Alternate\}) &= \texttt{SPI2\_SCK,I2S2\_CK,USART3}\\
& \texttt{\_CTS, TIM1\_CH1N,}\\
& \texttt{TSC\_G6\_IO3, EVENTOUT};\\
\delta(\{Name.PB13, Function.Additional\}) &= \texttt{ADC3\_IN5, COMP5\_INP,}\\
& \texttt{OPAMP4\_VINP,}\\
& \texttt{OPAMP3\_VINP};
\end{aligned}$$

Listing 2.1 The partial function δ maps the attribute-value pairs for the first row of Figure 2.4.

```
pins = {
    "PB13": {
        "Number": {
            "WLCSP100": "J3",
            "LQFP100": 52,
            "LQFP64": 34,
            "LQFP26": 26,
        },
        "Type": ["I/O", "3.3V", "Analog"],
        "Notes": ["Fast ADC Channel"],
        "Functions": {
            "Alternate": ["SPI2_SCK", "I2S2_CK",
                          "USART3_CTS", "TIM1_CH1N",
                          "TSC_G6_IO3", "EVENTOUT"],
            "Additional": ["ADC3_IN5", "COMP5_INP",
                           "OPAMP4_VINP", "OPAMP3_VINP"],
        },
    },
    "PD8": {
        "Number": {"LQFP100": 55},
        "Type": ["I/O", "3.3V", "Analog"],
        "Functions": {
            "Alternate": ["USART3_TX", "EVENTOUT"],
            "Additional": ["ADC4_IN12", "OPAMP4_VINM"],
        },
    }
}
```

Listing 2.2 A possible Python dictionary of data derived from Figure 2.4.

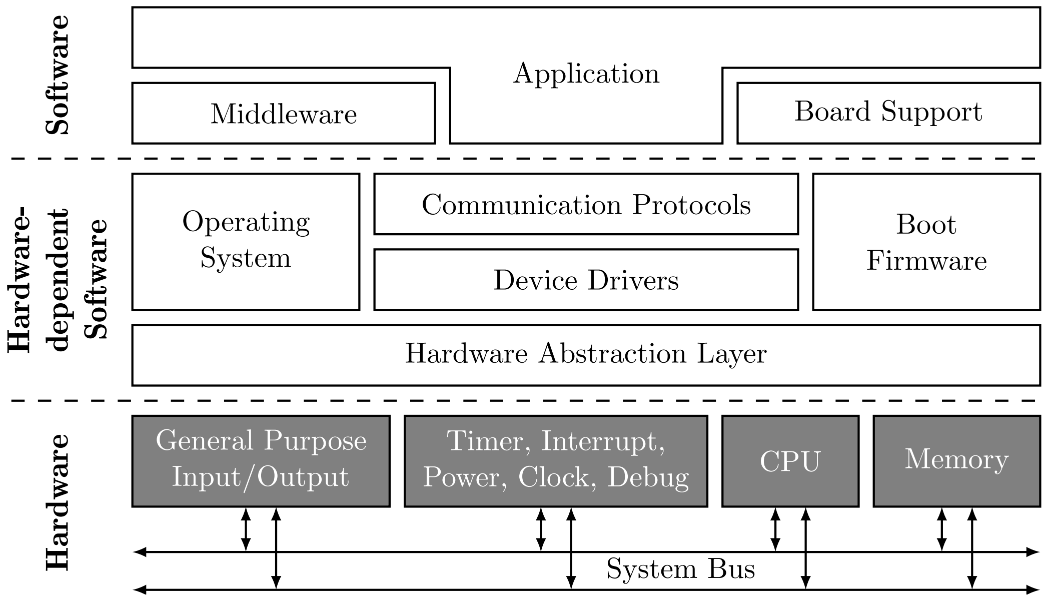


Figure 2.7 The simplified software stack of a typical hardware-dependent software (HdS) architecture [EMD09]. This thesis focuses on the hardware description, hardware abstraction layer, boot firmware, device drivers and board support.

2.3 Hardware-dependent Software

As the name suggests, embedded system requirements derive from the embedding system [EMD09]. Therefore, in practice, the variety in embedded software is rather large and can include a lot of different hardware and software configurations [EMD09]. In this thesis we focus on software running on microcontrollers only, especially on the ARM Cortex-M architecture due to its large feature set and immense popularity. However, our work applies to other microcontroller architectures likes AVR, MSP430, Xtensa and RISC-V, and, to a lesser extend, also to more fully featured embedded designs like the ARM Cortex-A-based Raspberry Pi.

Hardware-dependent software (HdS) consists of the lowest layers in an embedded system that directly interact with the underlying hardware and provide an abstraction to the application software ideally via a portable interface that is the same on different hardware [BBA17, Kor18]. In doing so, the HdS can only implement a system functionality *together* with the underlying hardware and would loose their utility without this dependence [EMD09]. Figure 2.7 depicts layers of varying degree of hardware specialization of a conceptual and simplified HdS architecture. Embedded software running on microcontrollers typically do not implement every layer as the abstraction can be too expensive for smaller devices or simply unnecessary for the scope of the application [BBA17, Kor18]. However, the more capable the hardware and the more complex the application becomes, the more abstraction layers can help reuse large parts of the stack [EMD09, BBA17, Kor18]. The common layers in a HdS software stack are the following:

The **hardware abstraction layer** (HAL) is where the knowledge of the technical documentation is converted into embedded software for the first time and the design decisions of this abstraction layer greatly influence the rest of the system [EMD09]. It provides language bindings for accessing register and internal memories and functional shims for small differences in hardware implementations [EMD09].

Microcontroller **boot firmware** does not generally conform to standardized application programming interfaces (APIs), such as the basic I/O system (BIOS) or the more recent unified extensible firmware interface (UEFI), and instead delegates them to the application firmware called directly by the reset interrupt to prepare the environment before jumping into the `main()` function [EMD09]. If a bootloader is used, its feature set usually varies greatly from simple unauthenticated wired connections to complex, authenticated over-the-air firmware update [boot17].

Device drivers provide access to internal peripherals and external hardware through a set of standardized functions to configure the device and read/write data from/to it [EMD09, BBA17]. These drivers depend on the HAL and can vary between different microcontroller projects, since there is no standard defined for them [EMD09, BBA17, Kor18].

Communication protocols are built on top of the device drivers and typically conform to complex external standards such as TCP/IP over Ethernet [EMD09] or industry protocols built on top of specialized peripherals such as the controller area network (CAN). These stacks benefit dramatically from a deep integration of specialized hardware for media access control (MAC) and direct memory access (DMA) to accelerate data handling [EMD09].

The real-time **operating system** (RTOS) typically refers to a stackful threading implementation such as the very popular FreeRTOS [rtos03], which is ported to over 30 microcontroller architectures. Using an RTOS is considered good practice for complex embedded applications as it provides well known communication and resource sharing primitives between threads to help with the reuseability of existing embedded software [EMD09, BBA17, Kor18]. However, simple polling- or interrupt-based scheduling implementations can also be very effective on small devices, making the use of an RTOS optional as well [rust17b, EMD09].

Finally, the **board support** packages configure the HdS layers for the specific hardware setup, while the **middleware** provides application-specific services as an adapter layer between the RTOS and the **application**, which implements the overall functionality of the embedded system [EMD09].

In this thesis we focus on generating parts of the HdS stack, specifically the hardware description, the HAL, boot firmware, device drivers, and board support using data derived from the technical documentation via table processing. In order to understand the type of data required for this task, we introduce how the HdS accesses the underlying hardware in the next section.

2.3.1 Accessing Hardware in Software

Modern microcontrollers such as the STM32 series from STMicro are built around the ARM Cortex-M microprocessors by adding specialized memories and hardware peripherals to a standardized internal bus connection at a specific address in a unified 32-bit address space. This modular approach maps the configuration of the peripherals as registers into the address space of the microprocessor, hence the name memory-mapped input/output (MMIO) register access. The same load and store instructions that can access internal memories can also access peripheral registers regardless of their number, type, and feature set [EMD09]. An example of a complex microcontroller with many peripherals accessible via MMIO registers is visualized in Figure 2.8.

For example, the previously discussed Figure 2.2 provides a short textual description of the cyclic redundancy check (CRC) peripheral, which is visualized as the function block diagram in Figure 2.3 with two data and four configuration registers all connected to a 32-bit bus interface. To access these registers we need to know their specific addresses, for which we need to combine the CRC peripheral boundary address `0x40023000` found in the table from Figure 2.9 with the register offsets in the summary view from Figure 2.10. The specific register layout, initial value, and functional description of each bit is documented separately again. For example, Figure 2.11 describes the control register `CRC_CR`.

Combining this knowledge, we can interpret the reset values of the `CRC_INIT` = `0xFFFFFFFF` and `CRC_POL` = `0x04C11DB7` as a default configuration for CRC-32 (Ethernet). Writing a 8-bit value into `CRC_IDR` will update the CRC computation and output the result in `CRC_DR`. To change the default CRC configuration to CRC-16-CCITT, we first write its polynomial 0x1021 as a 32-bit value to `CRC_POL` at address `0x40023000 + 0x14`, then update the polynomial size to 16-bit and reset the peripheral using a read-modify-write operation on `CRC_CR` at address `0x40023000 + 0x08` to preserve the value of the reversal options. Listing 2.3 achieves MMIO register access in plain C by casting the address to a volatile pointer of the correct width and then de-referencing it for read or write access.

Even though MMIO registers are a very simple and elegant hardware solution that can be natively implemented even in "high-level" languages such as C without requiring special treatment, having to manually compute all the register addresses, correct types, and bit-offsets and translate them into C code is not very user-friendly. Therefore, vendors publish generated language bindings that provide these register definitions as C header files, that follow a format standardized by ARM, which we introduce next.

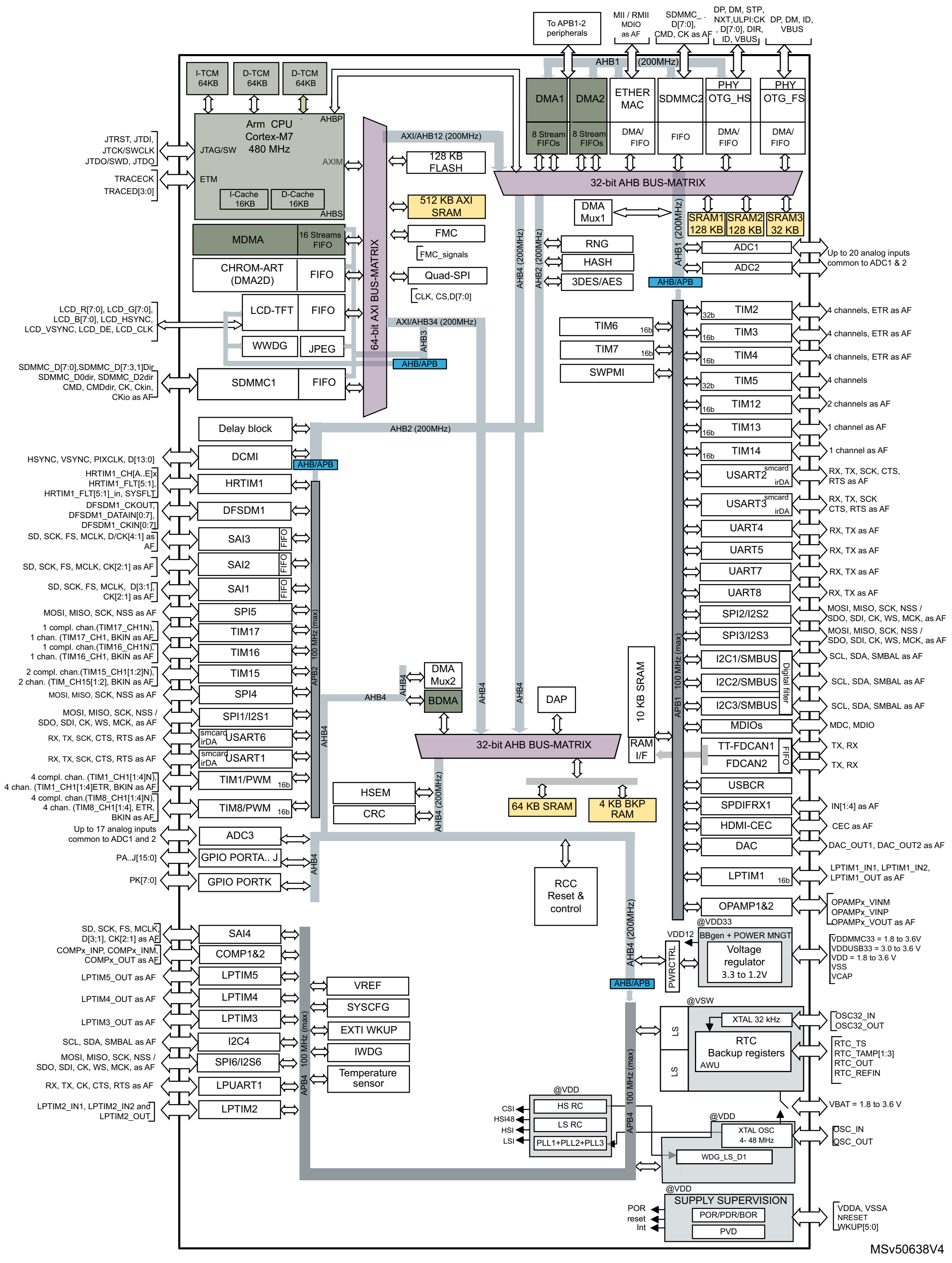


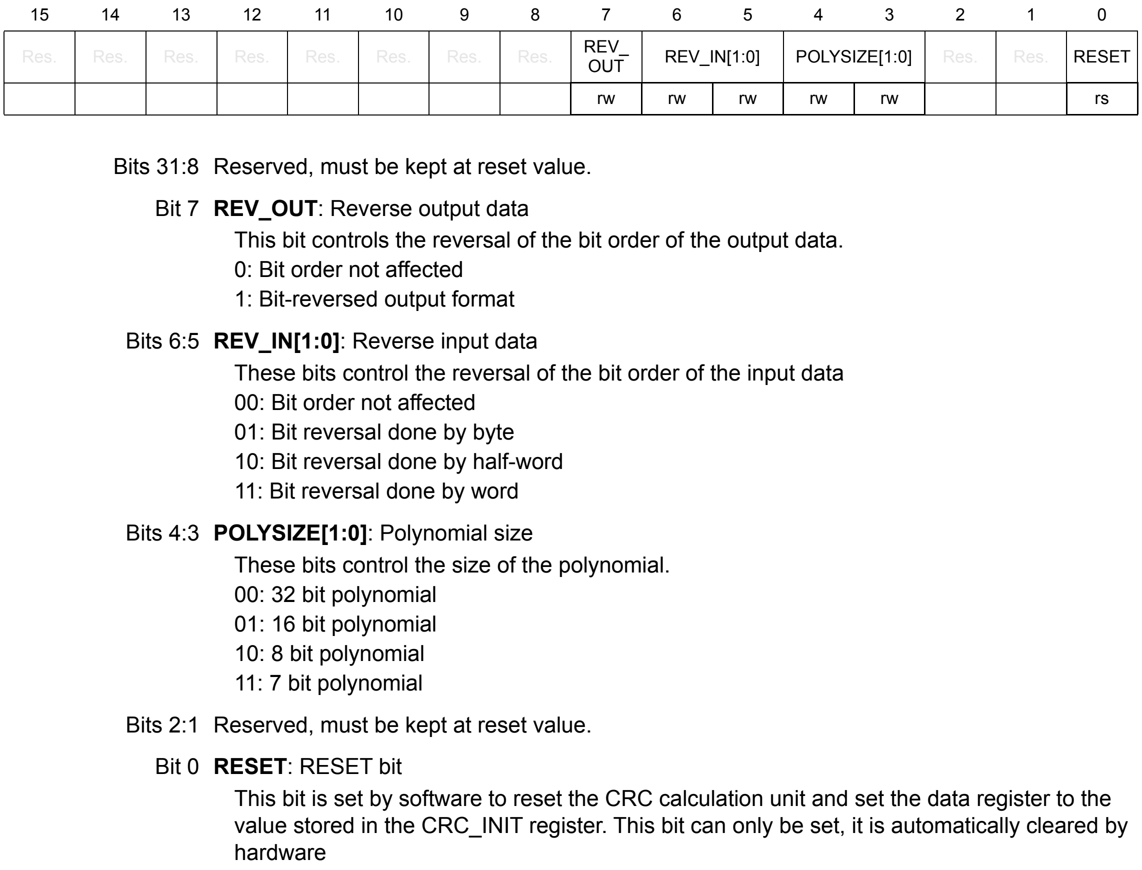
Figure 2.8 The block diagram of the high-performance STM32H750 microcontroller showing all specialized hardware blocks with their external signals connected to the internal peripheral APB bus colored dark gray connected via bus bridges in blue to the faster AHB bus in light gray. Only bus masters like the ARM Cortex-M7 in the top left and the DMA peripherals in green are allowed to initiate accesses to these busses. Notice how the internal volatile memories colored in yellow and DMA peripherals are connected to many different busses to enable parallel distributed operation [DS12556].

Bus	Boundary address	Size (bytes)	Peripheral	Peripheral register map
AHB1	0x4002 F000 - 0x47FF FFFF	~127 MB	Reserved	-
	0x4002 C000 - 0x4002 EFFF	1KB	GFXMMU	*Section 14.5.11: GFXMMU register map*
	0x4002 BC00 - 0x4002 BBFF	1 KB	Reserved	-
	0x4002 B000 - 0x4002 BBFF	3 KB	DMA2D	*Section 13.5.23: DMA2D register map*
	0x4002 4400 - 0x4002 AFFF	26 KB	Reserved	-
	0x4002 4000 - 0x4002 43FF	1 KB	TSC	*Section 31.6.11: TSC register map*
	0x4002 3400 - 0x4002 3FFF	1 KB	Reserved	-
	0x4002 3000 - 0x4002 33FF	1 KB	CRC	*Section 17.4.6: CRC register map*
	0x4002 2400 - 0x4002 2FFF	3 KB	Reserved	-
	0x4002 2000 - 0x4002 23FF	1 KB	FLASH registers	*Section 3.7.18: FLASH register map*
	0x4002 1400 - 0x4002 1FFF	3 KB	Reserved	-
	0x4002 1000 - 0x4002 13FF	1 KB	RCC	*Section 6.4.34: RCC register map*
	0x4002 0800 - 0x4002 0FFF	2 KB	Reserved	-
	0x4002 0400 - 0x4002 07FF	1 KB	DMA2	*Section 11.6.7: DMA register map*
	0x4002 0800 - 0x4002 0BFF	1 KB	DMAMUX1	*Section 12.6.7: DMAMUX register map*
	0x4002 0C00 - 0x4002 0FFF	1 KB	Reserved	-
	0x4002 0000 - 0x4002 03FF	1 KB	DMA1	*Section 11.6.7: DMA register map*

Figure 2.9 The memory map shows the register boundary address of the CRC peripheral on bus AHB1 [RM0432]. All addresses are aligned to $1\,\mathrm{kB}$ to trade a simpler address decoding in hardware for a sparse memory map with plenty of `Reserved` address space [PM0253].

Offset	Register name	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	CRC_DR	DR[31:0]																															
	Reset value	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0x04	CRC_IDR	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	IDR[7:0]							
	Reset value																									0	0	0	0	0	0	0	0
0x08	CRC_CR	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	REV_OUT	REV_IN[1:0]		POLYSIZE[1:0]		Res.	Res.	RESET
	Reset value																									0	0	0	0	0			0
0x10	CRC_INIT	CRC_INIT[31:0]																															
	Reset value	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0x14	CRC_POL	POL[31:0]																															
	Reset value	0	0	0	0	0	1	0	0	1	1	0	0	0	0	0	1	0	0	0	1	1	1	0	1	1	0	1	1	0	1	1	1

Figure 2.10 The register map summary of the CRC peripheral [RM0432]. An unused 32-bit register at offset 0x0C is not rendered and unused bits are marked with `Res`.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	REV_OUT	REV_IN[1:0]		POLYSIZE[1:0]		Res.	Res.	RESET
								rw	rw	rw	rw	rw			rs

Bits 31:8 Reserved, must be kept at reset value.

Bit 7 **REV_OUT**: Reverse output data
This bit controls the reversal of the bit order of the output data.
0: Bit order not affected
1: Bit-reversed output format

Bits 6:5 **REV_IN[1:0]**: Reverse input data
These bits control the reversal of the bit order of the input data
00: Bit order not affected
01: Bit reversal done by byte
10: Bit reversal done by half-word
11: Bit reversal done by word

Bits 4:3 **POLYSIZE[1:0]**: Polynomial size
These bits control the size of the polynomial.
00: 32 bit polynomial
01: 16 bit polynomial
10: 8 bit polynomial
11: 7 bit polynomial

Bits 2:1 Reserved, must be kept at reset value.

Bit 0 **RESET**: RESET bit
This bit is set by software to reset the CRC calculation unit and set the data register to the value stored in the CRC_INIT register. This bit can only be set, it is automatically cleared by hardware

Figure 2.11 This section on the CRC control register CRC_CR contains detailed descriptions of the functionality of each bit [RM0432]. Note that the RESET bit can only be set by software (denoted rs), since it is reset automatically in hardware. Unused bits are marked with Res.

```c
// Write CRC-16-CCITT polynomial into CRC_POL
*(volatile uint32_t*)0x40023014 = 0x1021;

// Read CRC_CR as 8-bit value and configure it
uint8_t control = *(volatile uint32_t*)0x40023008;
control &= ~(0b11 << 3); // Reset POLYSIZE only
control |=   0b01 << 3;  // Set   POLYSIZE to 16-bit
control |=   0b01;       // Set   RESET
// Write configuration back to CRC_CR
*(volatile uint32_t*)0x40023008 = control;

// Write 4 bytes of data into CRC_IDR
*(volatile uint32_t*)0x40023004 = 12;
*(volatile uint32_t*)0x40023004 = 34;
*(volatile uint32_t*)0x40023004 = 56;
*(volatile uint32_t*)0x40023004 = 78;

// Read the computed CRC-16-CCITT from CRC_DR
uint16_t crc = *(volatile uint32_t*)0x40023000;
```

Listing 2.3 Configuring and computing a CRC-16-CCITT value from four data bytes via the CRC peripheral in plain C using MMIO register access.

2.3.2 Common Microcontroller Software Interface Standard

The common microcontroller software interface standard (CMSIS) is developed by ARM with the intention of providing a standard, lightweight HdS stack for C/C++ based software projects that ARM Cortex-M based microcontroller vendors can extend and specialize for their own device [arm15]. In practice, CMSIS is a HdS stack implementation with many sub-projects that can be composed together as needed. However, in this thesis, we only look at the CMSIS-SVD peripheral register descriptions.

The system view description (SVD) defines a extensible markup language (XML) format for describing the MMIO registers of peripherals and is the machine-readable equivalent of the register descriptions in the technical documentation (cf. Section 2.3.1) [arm15]. These files are intended to be used by a debugger to map addresses in the firmware to register names and documentation for better human understanding [arm15]. A simplified encoding for the `CRC_CR` register (cf. Figure 2.11)) is shown in Listing 2.4. The SVD files from most vendors including STMicro are aggregated in the cmsis-svd repository [svd15a].

The SVD files are also intended to be converted into C language bindings using a closed-source conversion program called `SVDConv` provided by ARM [svd15d]. The resulting C header files use a C struct of volatile members to describe the peripheral registers and then cast the peripherals address to this address. Unused space in the peripheral register file is filled with `RESERVED` fields which are not supposed to be accessed [svd15d]. The individual register bits are generated as C pre-processor (CPP) macros with a clear naming scheme [svd15d]. Listing 2.5 contains the C header output for the SVD definitions in Listing 2.4, which significantly simplify accessing the CRC peripheral registers in plain C. However, clearing and setting individual bits is still a manual process as can be seen in Listing 2.6.

This form of MMIO register access is the de-facto standard for all C based HALs due to the simplicity of the header files that works well even with older, proprietary compilers [arm15]. However, there are more tools than the `SVDConv` utility that try to reduce the complexity of writing embedded software, which we will discuss in the next section.

2.3.3 Configuration Tools

In addition to CMSIS, vendors also publish custom tooling for their specific products. This can range from custom, stand-alone graphical user interface (GUI) applications to proprietary integrated development environments (IDEs) with a tuned toolchain for the architecture. A corresponding example is the STM32CubeMX [stm08] tool from STMicro written in Java that allows the developers to connect peripheral signals to pins as shown in Figure 2.12, configure the clock system, estimate power consumption, and finally generate a complete HdS stack for the specific device.

```xml
<peripheral>
  <name>CRC</name>
  <baseAddress>0x40023000</baseAddress>
  <registers>
    <register>
      <name>CR</name>
      <addressOffset>0x08</addressOffset>
      <size>32</size>
      <resetValue>0x00000000</resetValue>
      <fields>
        <field>
          <name>REV_OUT</name>
          <bitOffset>7</bitOffset>
          <bitWidth>1</bitWidth>
          <access>read-write</access>
        </field>
        <field>
          <name>REV_IN</name>
          <bitOffset>5</bitOffset>
          <bitWidth>2</bitWidth>
          <access>read-write</access>
        </field>
        <field>
          <name>POLYSIZE</name>
          <bitOffset>3</bitOffset>
          <bitWidth>2</bitWidth>
          <access>read-write</access>
        </field>
        <field>
          <name>RESET</name>
          <bitOffset>0</bitOffset>
          <bitWidth>1</bitWidth>
          <access>write-only</access>
        </field>
      </fields>
    </register>
  </registers>
</peripheral>
```

Listing 2.4 An shortened excerpt of the CMSIS-SVD definitions for the `CRC_CR` register. Note the write-only `RESET` bit access type, which is reset automatically in hardware [svd15a].

The STM32CubeMX application uses an internal XML database that contains a machine-readable version of the pin definitions in the technical documentation (cf. Section 2.1) and is published by STMicro in the public repository [stm20]. For example, the alternate functions of pin PB13 from the first row in Figure 2.4 are encoded in Listing 2.7.

The STM32CubeMX tool allows for more configuration options such as power settings, internal clock configuration and middleware configurations using the same internal database. Once the user has configured the entire microcontroller as required

```c
// Register layout definition for the CRC peripheral
typedef struct
{
  volatile uint32_t DR;        // Data Register
  volatile uint32_t IDR;       // Independent Data Register
  volatile uint32_t CR;        // Control Register
           uint32_t RESERVED; // offset 0xC
  volatile uint32_t INIT;      // Initial CRC Value Register
  volatile uint32_t POL;       // Polynomial Register
} CRC_TypeDef;

// Address definition for the CRC peripheral
#define CRC    ((CRC_TypeDef*)0x40023000)

// Bit definition for CRC_CR register
#define CRC_CR_RESET         (0x1 << 0)

#define CRC_CR_POLYSIZE_0    (0x1 << 3)
#define CRC_CR_POLYSIZE_1    (0x2 << 3)
#define CRC_CR_POLYSIZE      (0x3 << 3)

#define CRC_CR_REV_IN_0      (0x1 << 5)
#define CRC_CR_REV_IN_1      (0x2 << 5)
#define CRC_CR_REV_IN        (0x3 << 5)

#define CRC_CR_REV_OUT       (0x1 << 7)
```

Listing 2.5 An simplified excerpt of the STM32 CMSIS header file definitions for the CRC_CR register file from Listing 2.4 [modm17].

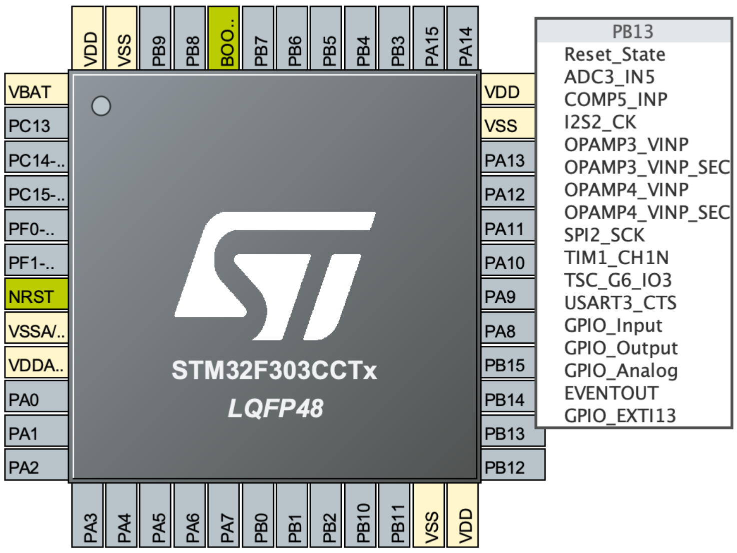
```c
// Write CRC-16-CCITT polynomial into CRC_POL
CRC->POL = 0x1021;

// Read CRC_CR and reset POLYSIZE
uint8_t control = CRC->CR & ~CRC_CR_POLYSIZE;
// Set POLYSIZE, RESET and write CRC_CR back
CRC->CR = control | CRC_CR_POLYSIZE_0 | CRC_CR_RESET;

// Write 4 bytes of data into CRC_IDR
CRC->IDR = 12;
CRC->IDR = 34;
CRC->IDR = 56;
CRC->IDR = 78;

// Read the computed CRC-16-CCITT from CRC_DR
uint16_t crc = CRC->DR;
```

Listing 2.6 This C code uses the STM32 CMSIS header files to reimplement the code from Listing 2.3 in a more readable and portable fashion [modm17].



PB13
Reset_State
ADC3_IN5
COMP5_INP
I2S2_CK
OPAMP3_VINP
OPAMP3_VINP_SEC
OPAMP4_VINP
OPAMP4_VINP_SEC
SPI2_SCK
TIM1_CH1N
TSC_G6_IO3
USART3_CTS
GPIO_Input
GPIO_Output
GPIO_Analog
EVENTOUT
GPIO_EXTI13

Figure 2.12 The STM32CubeMX graphical pinout configurator showing the alternate pin functions for pin PB13 on the STM32F303CC microcontroller as documented in the first table row from Figure 2.4. Note that pin PD8 is missing on the LQFP48 package as expected [stm08].

```
<Pin Name="PB13" Position="26" Type="I/O">
    <Signal Name="ADC3_IN5"/>
    <Signal Name="COMP5_INP"/>
    <Signal Name="I2S2_CK"/>
    <Signal Name="OPAMP3_VINP"/>
    <Signal Name="OPAMP3_VINP_SEC"/>
    <Signal Name="OPAMP4_VINP"/>
    <Signal Name="OPAMP4_VINP_SEC"/>
    <Signal Name="SPI2_SCK"/>
    <Signal Name="TIM1_CH1N"/>
    <Signal Name="TSC_G6_IO3"/>
    <Signal Name="USART3_CTS"/>
    <Signal IOModes="Analog,EVENTOUT,EXTI" Name="GPIO"/>
</Pin>
```

Listing 2.7 A simplified except of the STM32CubeMX internal database showing the pin function of PB13 encoded as a flat list. Compared to the table in Figure 2.4 the distinction between `Alternate functions` and `Additional functions` is removed and the `I/O structure` and `Notes` categories are missing, with a special `IOModes` encoding added for additional configurations [stm08].

for their embedded application, the STM32CubeMX configuration tool generates a full HdS stack in the C programming language [stm08]. While generating a HdS stack written in C is a pragmatic choice due to its popularity and history as a system programming language [EMD09, BBA17], it is also the main limitation of the STM32CubeMX tool. Therefore, we discuss what motivations exist to move beyond C for embedded software in the next section.

2.3.4 New Programming Languages

So far, we only discussed implementations of HdS in the C programming language. However, open-source GCC- or LLVM-based toolchains also support a number of newer compiled languages such as C++ and Rust, while optimized runtimes exist for interpreted languages such as Python and Go. These languages bring new programming paradigms and features to resource-limited embedded systems that are simply not supported by C, especially compile-time code execution and extending the type system.

An example of utilizing new language features is the real-time for the masses (RTFM) kernel [EHA+13], which uses the nested vector interrupt controller (NVIC) hardware to run real-time tasks with a stack resource policy on ARM Cortex-M [LFL+16]. The original implementation was targeted at C and required running a separate code generator before compilation [LLL+15]. However, a C++ [cpp17] and a Rust [rust17b] implementation can implement the kernel functionality natively and thus removes the need for a code generator.

The use of C++ is particularly interesting for embedded system, since it is backward compatible with C and the vendor-provided HdS can be reused, which allows for a more gradual entry into C++ programming where only the parts of the HdS that are important to the developer can be replaced with a C++ version [Kor18]. Therefore, even the newest C++20 features, such as native support for light-weight, stackless coroutines on ARM Cortex-M [BXHP20], can be used in a new project without breaking existing code. An example of a C++20 HdS stack using this hybrid approach is the modm embedded library [modm09], which generates a customized stack for thousands of AVR and Cortex-M devices using the data sources previously discussed in this chapter.

However, staying backward compatible to C also limits new C++ features, in particular, the lack of strict resource ownership semantics cannot be retrofitted, resulting in memory safety issues that have become a key concern for any new C and C++ projects [Wei16]. In light of these shortcomings, a new systems programming language was designed: Rust implements a borrow checker to guarantee memory safety while providing advanced features similar to C++ such as compile-time execution, type generics, and polymorphism [rust10]. As a result, it is considered a good fit for use in embedded safety-critical systems such as avionics [PCO19]. An example of a Rust HdS stack is the Rust-Embedded project [rust17a], which generates language-bindings for register access via SVD files, since it cannot reuse the CMSIS header files.

Besides compiled languages, interpreted languages such as Python and JavaScript require a runtime to execute the source code directly on the device. Beyond that are virtual machines that only run bytecode like WebAssembly or rBPF and provide security and isolation mechanisms through the use of a hypervisor [ZB21]. Even though these approaches require more resources than compiling C, C++, or Rust down to native machine code, developing a project in these high-level dynamic languages may be faster and easier. For example, MicroPython [mpy14] has been used to write CubeSat software [PL17] and internet of things (IoT) applications [GFK+20] in a memory-safe, easy-to-use, high-level language.

However, all of these new languages must access the underlying hardware with the same MMIO register mechanism described in Section 2.3.1. Therefore, they all require the same information to generate their language bindings and support tooling, regardless of what level of abstraction and convenience they provide.

2.4 Knowledge Modeling

In the previous sections, we introduced a number of data serialization formats for describing the hardware of microcontrollers, each with their own schema that is either explicitly given, e.g. by CMSIS-SVD, or implicitly derived based on the abstract table model. In contrast, a *knowledge graph* is a simple data format that does not require an initial schema [HBC+21]. Knowledge graphs model facts as edge relations between entities in a shared graph that embeds the domain-specific semantics of these edges as well [HBC+21]. Knowledge graphs can start out with only a few simple facts that over time accumulate into a large and rich graph of combined and interlinked facts, which can be used to deduct new knowledge about the domain [Jah21].

For example, the statement `STM32F303CC is available in TQFP48 package` describes a simple fact from the pinout table in Figure 2.4, which we can describe with the subject-predicate-object (SPO) triple `(STM32F303CC, package, TQFP48)`. We then add a couple of pin positions, names and signals to end up with the small knowledge graph shown in Figure 2.13 on the left. We can also describe the MMIO register map in the same format, for example, starting with the fact `STM32F303CC has peripheral USART` or `(STM32F303CC, peripheral, USART)` and extending it to include the registers and bits on the right. On this STM32 microcontroller the transmit (TX) and clear-to-send (CTS) signals need to be enabled explicitly since the universal synchronous/asynchronous receiver/transmitter (USART) peripheral can be used without both when just receiving data without flow control. We can then evolve these separate graphs into a larger one, by linking the pin signals to their corresponding enable bit in the peripheral driver.

We can query this information with the graph pattern $\texttt{PB13} \xrightarrow{\text{signal}} \texttt{CTS} \xrightarrow{\text{enable}} \texttt{?Bit} \xleftarrow{\text{bit}} \texttt{?Register} \xleftarrow{\text{register}} \texttt{?Peripheral}$ to identify the enable bit unambiguously with the tuple `(CTSE, CR1, USART)` and then feed this into a code generator to create a HAL signal connect and disconnect function demonstrated in Listing 2.8.

Significantly more complex queries are possible, especially if we define the semantics of the relations and entities in more detail [HBC+21]. For example, to provide a schema for consistency verification, we define that a pin must have an incoming $\xrightarrow{\text{pin}}$ relation from a position entity, otherwise the knowledge graph semantics are invalid. We can provide rules for relations too, for example, $\xrightarrow{\text{enable}}$ implies that the bit it points to must be set in the register to enable the entity it points from. We can also describe that the opposite action of enabling the bit equals the $\xrightarrow{\text{disable}}$ relation, even though it is not present in the data graph, which allows us to query the bit we need to clear on signal disconnection as well.

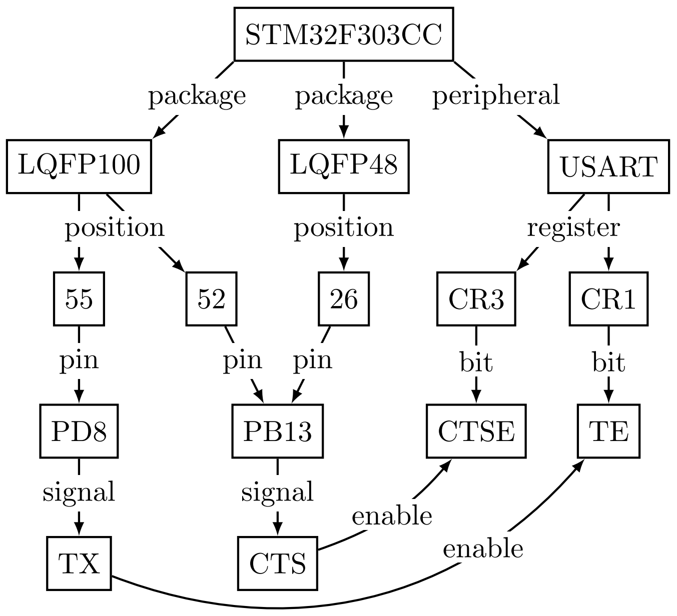


Figure 2.13 A simplified knowledge graph describing pin positions in their packages and signals connected to the USART peripheral with its control registers with the Transmit Enable (TE) and CTS Enable (CTSE) bits.

```
void connectSignal(Pin p, Signal s) {
    // Connect the pin to the signal
    setAlternateFunction(p, s);
    // Enable the signal if necessary
    if (p == Pin::PB13) {
        if (s == Signal::CTS) USART->CR1 |= USART_CR1_CTSE;
        if (s == Signal::TX)  USART->CR3 |= USART_CR3_TE;
    }
}
void disconnectSignal(Pin p, Signal s) {
    // Disconnect the pin to the signal
    resetAlternateFunction(p);
    // Disable the signal if necessary
    if (p == Pin::PB13) {
        if (s == Signal::CTS) USART->CR1 &= ~USART_CR1_CTSE;
        if (s == Signal::TX)  USART->CR3 &= ~USART_CR3_TE;
    }
}
```

Listing 2.8 Implementations for (dis-)connecting signals to pins with the data derived from the knowledge graph in Figure 2.13

The rule set and data graph together constitute a formal representation of domain-specific knowledge which is called an ontology [HBC+21]. With an appropriate query solver we can now semantically reason over this domain [Jah21]. This meta-knowledge can also be used to align the implicit schema given by the abstract table model via its labeled domain (cf. Section 2.2) with the knowledge graph and then merge the data into it [TELN03]. With a sufficiently large schema and rule set we can validate the consistency of the knowledge graph and deduct new information that can be reintegrated into the graph again [HBC+21].

We can freely define the scope and detail of this ontology depending on the extent and quality of the input data and how much additional information we want to query out of it. Therefore, a knowledge graph is easy to scale as required while still using the same query language and graph algorithms [HBC+21]. There already exist various other knowledge graphs for broad knowledge that show these properties in practice, such as DBpedia [LIJ+15] extracted from Wikipedia and the Google Knowledge Graph [Sin12] used to improve search results.

While the knowledge graph is an abstract concept, the semantic web software stack provides a concrete implementation via several standardized technologies. The core idea of the semantic web is to annotate web resources with semantic information, therefore the syntax is based on XML, so that it can be transparently integrated into hypertext markup language (HTML) content and accessed by search engines in a structured way [HBC+21].

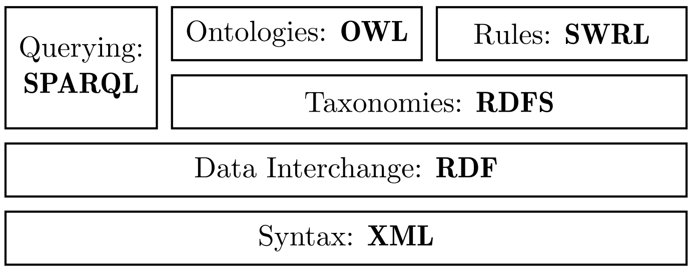


Figure 2.14 The semantic web stack showing its standardized layers.

The layers of the semantic web stack are shown in Figure 2.14. The data model for edge-labelled knowledge graphs based on SPO triples is the resource description framework (RDF), which can be extended with descriptions of semantic rules of increasing computational complexity relative to the reasoning capabilities of a solver [HBC+21]. The simplest is the RDF schema (RDFS), which provides basic vocabulary like a datatype hierarchy and pre-defined properties [HBC+21]. However, RDFS is almost entirely subsumed by the web ontology language (OWL), which generalizes such rule definitions using a description logic with well understood computational properties that allow reasoning solvers to terminate on all queries [HBC+21]. Beyond that exists the semantic web rule language (SWRL) that relaxes rule definitions at the cost of decidability and runtime complexity, therefore, solvers usually only allow a subset of this language [Jah21].

In this chapter, we gave an overview over the diverse set of topics this thesis touches on. We introduced and gave examples from the STMicro technical documentation and discussed the limitations of the PDF format that make accessing its text, figures and tables difficult. Since we are interested in extracting structured information from the documentation, we introduced the concept of table processing, which provides ways to work with tabular data in a format agnostic way. To be able to understand what structured information we need, we discussed hardware-dependent software, MMIO register functionality, hardware configuration tools, programming languages beyond C and what data requirements each topic has. Finally, to manage all of these diverse data sources and formats, we gave an overview of how knowledge modeling provides a way to store, modify, and reason over a shared representation of facts and relations. While the goal of this thesis is the combination of all these topics for the purpose of generating embedded software, they are also their own areas of research with a lot of in-depth work that we present in the next section.

3

Related Work

After introducing the background for understanding the remainder of this thesis, we now discuss the related work in the two large areas of information extraction and embedded software. First, in Section 3.1, we review the related work of data extraction, table processing, and knowledge modelling, since there is significant overlap in their problem solving approaches and limitations. Afterward, in Section 3.2, we introduce data pipeline projects that collect data from various sources for generating HdS. We conclude this chapter in Section 3.3 with an overview of previous work on the topics of embedded software and code generators and present the relevant software libraries in these areas.

3.1 Document Information Extraction

Extracting structured information from non- or semi-structured inputs is a wide area of research [EHLN06], however, in this section, we only focus on topics related to table processing, knowledge modeling, and, to a lesser extend, also text mining and web scraping. While these topics often overlap significantly, we group this section roughly into three parts: (i) an overview of different table processing paradigms, (ii) detecting and understanding tables in PDF and HTML, and (iii) converting the data into knowledge graphs.

The foundation of table processing is the abstract table model (cf. Section 2.2) [Wan96], used to decompose a table into its logical structural design, tabular arrangement, and presentation style. In this abstract state, table content can be edited with generic mathematical operations and its presentation can be changed by applying style rules [Wan96]. The modified abstraction can then be formatted back into a new table using an efficient algorithm [Wan96].

Hurst [Hur00] then applies this abstract model to tables in documents, for the purpose of extracting their information into a semantic model. Even though the abstract model already provides an implicit content hierarchy through its labeled domains, there are many different ways to display data from the same model in a table, which makes table understanding difficult [Hur00]. Hurst first catalogs all these different table formats and then provides the simple table relation as a more generic and simpler alternative to the Wang's labeled domains, which is used to derive a semantic model of the table content [Hur00].

Summarizing these foundational texts and more related work, Embley et al. [EHLN06] gives an even more thorough enumeration of table input formats, presentation styles and table processing paradigms. The survey classifies tables into four categories:

Plain text tables are encoded using only characters in a monospace font, using ASCII or Unicode symbols to create spacing, newlines and display horizontal and vertical rules. These tables are therefore limited to simple geometric layouts and plain text cell content, however, may not always have a clearly defined layout, especially when not all cell borders are explicitly included to reduce rendering verbosity [EHLN06].

Symbolic tables are unambiguously encoded using markup languages such as HTML or XML that separate table layout from cell content. Their visualization is performed as a separate step, which allows rendering the same table in different styles [EHLN06].

Vector tables are found in PDFs (cf. Section 2.1) and scalable vector graphics (SVG) and encode the table layout and cell content separately using different representations: text instructions for rendering glyphs and graphics instructions rendering line art [EHLN06].

Rasterized tables are encoded as a bitmap, typically from a scanned source or camera image, and contain no layout or content annotations at all. Unfavorable lighting conditions, low image resolution, and geometric skew make reliable table recognition and extraction a difficult problem [EHLN06].

For all categories the processing requires first detecting and locating the table and then understanding the tables structure and content [EHLN06, KLU15]. In this thesis, we only interact with vector tables in PDFs and symbolic tables in HTML, therefore we will summarize the related work for detecting and understanding tables for these two formats next.

3.1.1 Table Detection

For plain text and symbolic tables, detecting a table can be as simple as matching on special strings in the content stream [CTT00]. However, for vector tables, two or more content streams need to be evaluated to accurately identify a table location [EHLN06] and is, in general, not a solved problem for all inputs [KLU15]. Reliably detecting rasterized tables requires the use of optical character recognition (OCR) and machine learning [ZSJY20, LWX$^+$21].

A common input source of symbolic tables are websites, which use the HTML tags `<table>`, `<tr>` (row), `<th>` (header cell), and `<td>` (data cell) to encode the table layout [CTT00]. However, table detection in HTML is complicated when the `<table>` tag is used for website structure layout, where a table cell can represent a menu, form, or contain nested tables [CTT00]. Additionally, the entire table may be

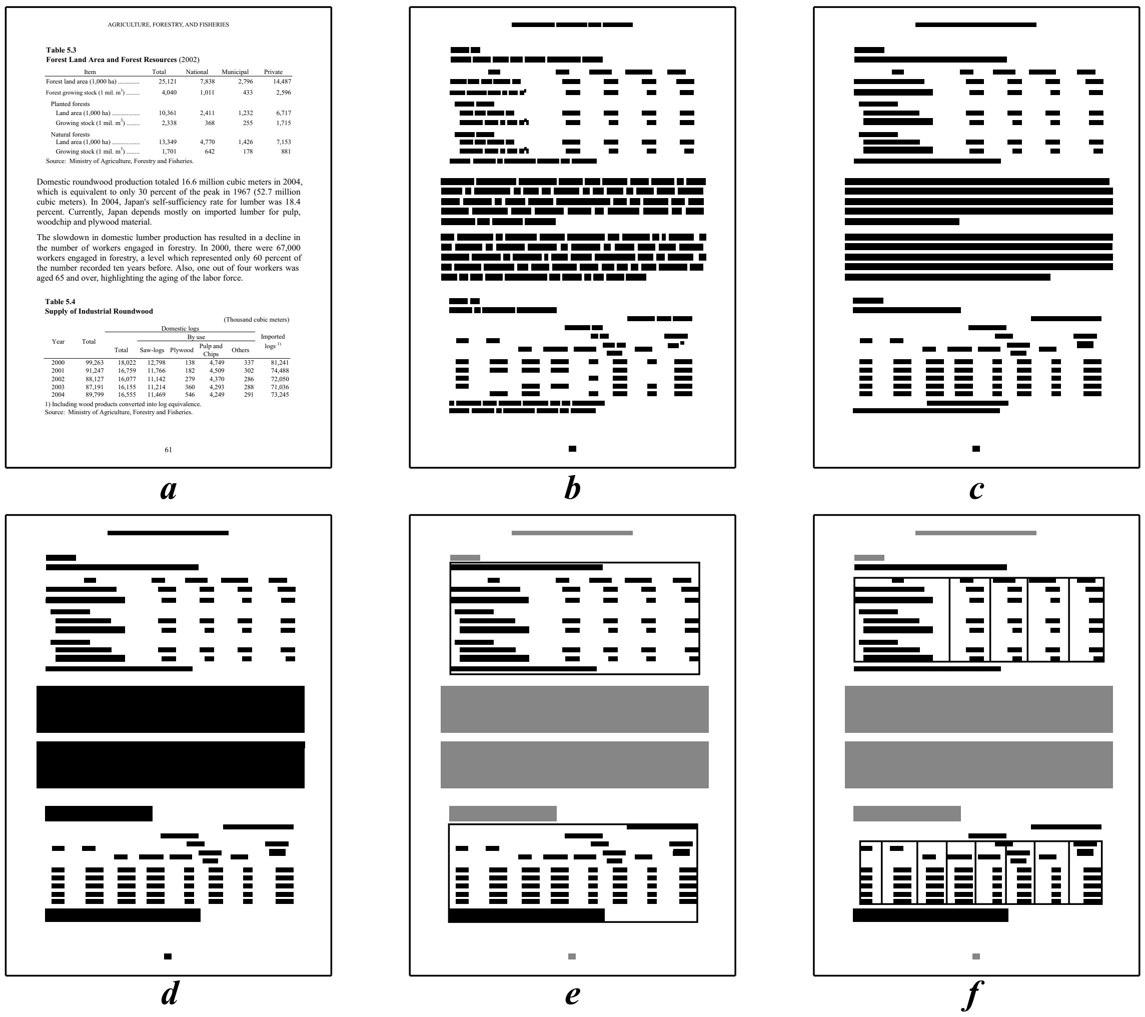

AGRICULTURE, FORESTRY, AND FISHERIES

Table 5.3
Forest Land Area and Forest Resources (2002)

Item	Total	National	Municipal	Private
Forest land area (1,000 ha)	25,121	7,838	2,796	14,487
Forest growing stock (1 mil. m^3)	4,040	1,011	433	2,596
Planted forests				
Land area (1,000 ha)	10,361	2,411	1,232	6,717
Growing stock (1 mil. m^3)	2,338	368	255	1,715
Natural forests				
Land area (1,000 ha)	13,349	4,770	1,426	7,153
Growing stock (1 mil. m^3)	1,701	642	178	881

Source: Ministry of Agriculture, Forestry and Fisheries.

Domestic roundwood production totaled 16.6 million cubic meters in 2004, which is equivalent to only 30 percent of the peak in 1967 (52.7 million cubic meters). In 2004, Japan's self-sufficiency rate for lumber was 18.4 percent. Currently, Japan depends mostly on imported lumber for pulp, woodchip and plywood material.

The slowdown in domestic lumber production has resulted in a decline in the number of workers engaged in forestry. In 2000, there were 67,000 workers engaged in forestry, a level which represented only 60 percent of the number recorded ten years before. Also, one out of four workers was aged 65 and over, highlighting the aging of the labor force.

Table 5.4
Supply of Industrial Roundwood

(Thousand cubic meters)

Year	Total	Domestic logs						Imported logs [1)]
		Total	By use					
			Saw-logs	Plywood	Pulp and Chips	Others		
2000	99,263	18,022	12,798	138	4,749	337		81,241
2001	91,247	16,759	11,766	182	4,509	302		74,488
2002	88,127	16,077	11,142	279	4,370	286		72,050
2003	87,191	16,155	11,214	360	4,293	288		71,036
2004	89,799	16,555	11,469	546	4,249	291		73,245

1) Including wood products converted into log equivalence.
Source: Ministry of Agriculture, Forestry and Fisheries.

61

Figure 3.1 TabbyPDF [SAM+18] detects tables by the regularity of the whitespace surrounding clusters of text: (a) input page, (b) individual text characters coalesced into words, (c) text words coalesced into lines, (d) text lines coalesced into paragraphs, (e) table search areas detected between table breaks in gray, and (f) tables with columns detected.

split up into multiple `<table>` tags for the purpose of pagination or styling [ETL05]. Therefore, when scraping complex websites for tabular data, the use of heuristics is still required to locate a symbolic table [CTT00, LKM01, LPL04, ETL05], thus reducing the advantage of the unambiguous table layout encoding.

For vector tables in PDFs to be detected, the whole input needs to be segregated into areas of text, figures, and tables. A common approach is coalescing the bounding boxes of text and graphics into larger clusters [RCVF03, CF04, SAM+18] and then deciding what type these clusters are. Figure 3.1 visualizes a heuristic algorithm using only the whitespace between clusters of text to detect tables with, without, or only with partial borders [SAM+18]. While this approach is more generic, the accuracy can suffer when tables with borders do not provide whitespace padding between the cell content and its borders [RCVF03, CF04, RPS16, SAM+18, RPS+18]. A more accurate, but less generic approach uses the properties of the graphic clusters to decide between a table and a figure, as exemplified in Figure 3.2 [RCVF03].

In practice, figures and tables are often composed of a mix of text with implicit whitespace borders and different font properties, alignment and and graphics with

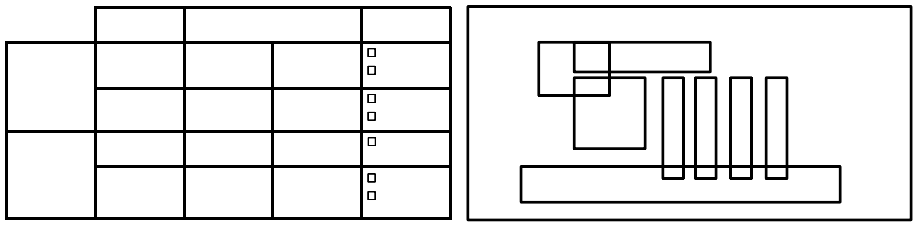


Figure 3.2 Differences in graphic line composition makes distinguishing between tables (left) and figures (right) easier, since the table graphics overlap each other in regular patterns, whereas the figure graphics are irregular [RCVF03].

various line types and widths, and therefore a combination of both approaches must be used to reliably categorize the input [RCVF03, CF04, CD16]. A particularly robust method is to first locate the table and figure captions, usually starting with "Table" or "Figure", via text search and then use specialized algorithms to find the corresponding graphics and text cluster in the immediate neighborhood [CD16]. Extending this idea to all text areas allows for detecting bullet point or numbered lists as one dimensional tables [LKM01, RPS16, CHCG15]. A completely different approach is to render the PDF input to images and then use machine learning to classify the areas visually, although, limited to inputs sufficiently similar to the training set [RMB+21].

Once a table has been detected and located, some methods choose to convert its contents directly into attribute-value pairs, with the indexing based on labeled domains [Wan96], simple table relation [Hur00], or customized relations [CTT00, LKM01, LPL04]. However, most approaches output a more detailed format, such as a subset of HTML that uses `<table>` only for tables and not for layout [SAM+18], or various specialized formats of XML or comma-separated values (CSVs) that preserve cell formatting information [LPL04, RCVF03, CF04, RPS16, SAM+18, RPS+18]. Figures are either ignored or converted into SVGs [CF04]. These intermediate representations are then passed to a separate step that interprets the table content.

3.1.2 Table Understanding

Extracting information from tables involves aligning the table structure and content with a schema that fits into a larger ontology, which describes the semantics of the data (cf. Section 2.4) [EHLN06, Ras17]. For generic input, the table structure is usually not known in advance, thus the schema needs to be derived either from the table itself, informed by an external ontology, or manually defined [EHLN06].

To derive a schema from the table itself, its structure together with styling, position, graphics, and font information is used to separate the header cells from the data cells to generate an indexing relation that fits into the ontology [Ras17, AS13]. However, deriving the schema only works well with tables that are relational in nature, especially if they contain header cells that already have a strong connection with the destination ontology [AS13].

A more robust, but complex technique uses an externally provided ontology to guide the understanding process. This method involves matching a set of user-defined [EAS13] or heuristically obtained [ETL05] mini-ontologies onto the table structure and then incrementally merging them into a larger ontology [TELN03, EAS13]. Alternatively, an already existing, external ontology [CHCG15] or one text-mined from the surrounding text [PA18, ZMH+21] can generate a feasible table schema mapping. Given good training data, machine learning can recognize tables with similar schemas but different formatting with 80–85% accuracy [PLW19]. The resulting ontologies and data can be modeled using knowledge graphs, which can then be further evolved to improve accuracy using various internal reasoning methods in combination with additional external data sources [Jah21, HBC+21].

To get a better understanding of how the research presented in this section is applied in practice, we introduce existing information extraction pipelines in the area of embedded software in the next section.

3.2 Hardware Description Data Pipelines

Extracting information from generic documents is a widespread use case for commercial and open-source tools, typically by applying OCR to scanned or photographed documents and heuristic algorithms on the obtained text [KLU15]. However, due to the format ambiguities inherent in generic documents, user input is usually required to help guide table detection and understanding [KLU15]. For example, Tabula [pdf12] is a popular, open-source Java application that can extract tables into Excel format with a GUI to solicit human user input. Khurso et al. [KLU15] compiled a number of methods and tools for table extraction in their survey. However, in this section, we are interested in tools that specifically extract information from technical documents related to embedded software and hardware.

Instabuild [pdf13a] is a commercial tool using OCR to extract a device pinout description from a screenshot of a datasheet and then convert it into a symbol and footprint for electronic design automation (EDA) tools, but requiring human supervision similar to Tabula. In contrast, uConfig [pdf17] extracts such device pinouts automatically using a carefully crafted parser that interprets the text bounding boxes inside the relevant figures. However, uConfig relies on hardcoded heuristics for pinout figure detection and understanding and thus only succeeds for PDF technical documentation from vendors with sufficiently similar pinout layouts. Finally, Datasheet2SVD [pdf20] uses Tabula to extract the memory map from reference manuals, for which the vendor did not publish a CMSIS-SVD file (cf. Section 2.3.2). However, Datasheet2SVD is limited to work for only two Renesas PDFs documents. None of these projects give any kind of evaluation metric for their accuracy or device coverage, and most have had little to no development activity in recent years [pdf17, pdf20].

There also exist projects that extract information from machine readable source such as CMSIS-SVD, CMSIS-Header, and the STM32CubeMX database [stm08] (cf. Section 2.3.3). modm-devices [modm16] is a Python pipeline that accumulates data on device pinouts, general purpose input/output (GPIO) signal connections, peripheral type and counts, and memory sizes for STM32, SAM, NRF, and AVR microcontrollers. This data is then used to inform the C++ HAL and toolchain generation in the modm project [modm09]. The embassy-rs data pipeline [stm21] does almost the same for generating the embassy-rs Rust HAL [rust20], but is limited to STM32 only.

Tool or Project	**Data Source**	**Output**	**Data Scope**	**Interaction**
Tabula	Any PDF	Excel, CSV	Any table	Supervised
Instabuild	Screenshot of datasheet PDF	EDA symbol	Pinout tables	Supervised
uConfig	Datasheet PDF	EDA symbol	Pinout figures	Scripted
Datasheet2SVD	Datasheet PDF	CMSIS-SVD	Register map	Scripted
modm-devices	CMSIS-Header, STM32CubeMX	Custom XML with Python API	Peripherals, pinouts and pin functions, memories	Scripted with manual patches
embassy-rs	CMSIS-SVD, STM32CubeMX	Custom JSON	Peripherals, pinouts and pin functions, register map	Scripted with manual patches

Table 3.1 Comparison of tools and projects that extract hardware description data from PDF and machine-readable sources.

In summary, PDF-based tools are limited to extracting very specific data for a limited number of devices, while the most extensive datasets are only generated from the machine-readable sources STM32CubeMX, CMSIS-Header and CMSIS-SVD. A comparison summary of all these tools and projects is given in Table 3.1. To get an overview of the use cases that can consume the hardware description data generated by the pipelines, we present related work in the area of embedded software in the next section.

3.3 Generating Hardware-dependent Software

We introduced the layered stack of hardware-dependent software (HdS) based on the work by Ecker et al. [EMD09] in Section 2.3. While their work extends beyond the scope of this thesis by also covering toolchain setups and verification concepts, it is kept abstract without resulting in a working HAL [EMD09]. In contrast, Beningo [BBA17] applies all the principles of HdS design described by Ecker et al. in a very practical guide by developing concrete HAL APIs in C for several peripherals with documentation and testing. Kormanyos [Kor18] provides a similar practical guide, however, with an additional emphasis on using the object-oriented and template C++ language features.

An example of a HAL project written in C is the Linux Zephyr RTOS [lin14], which supports a large number of platforms and specific devices and can supports customization via the KConfig and Linux DeviceTree [lin16] interface, which are also used by the Linux Kernel. The configuration and hardware description data is formatted as C pre-processor (CPP) definitions and is used by Zephyr as an implicit code generator that is built into the C/C++ toolchain and runs during compilation. A different approach is taken by the STM32CubeMX configuration tool, which instead generates its C HAL in a separate step before compilation. modm [modm09] is a C++20 library that also explicitly generates a HAL and hardware-dependent unit tests for many AVR, STM32, and SAM devices based on data from modm-devices and customization options. The Embedded Rust project [rust17a] generates parts of their HAL in the Rust language with a similar feature set and device coverage.

As mentioned in Section 2.3.4, languages other than C must provide their own bindings for the register map. For this purpose, a number of specialized code generators exist to convert CMSIS-SVD files [svd15b] into a specific representation (cf. Section 2.3.2). The unmodified SVD files of most vendors can be found on GitHub [svd15a] and, for STMicro only, in a special repository with manually written patches [stm17b] to improve their accuracy. These files are the input for these language-specific code generators: SVDConv [svd15d] for generic C, SVD2Rust [rust16] for Embedded Rust, SVD2Ada [ada15] for Embedded Ada, and SrcGen [svd15c] for generic Assembly, C or Clojure definitions.

Code generation is an essential tool for writing and maintaining embedded software, since the limited code space on most devices makes a runtime selection of HAL drivers and configuration options infeasible [EMD09] and instead moves these specializations into the toolchain or customized tools. Since we only discussed practical projects so far, we will now present related work in the area of model-driven software engineering (MDSE), starting with generating only specific parts of HdS before broadening the scope.

Holman et al. [HS15] generates a HAL out of manually written code templates as an open replacement for the STM32CubeMX tool, while Huning et al. [HOSP21] extends this idea by integrating the process into a vendor-independent MDSE GUI tool. The data for both approaches is manually provided using XML. Weiss et al. [WRSW21] source data describing the memory map to automate hardware in the loop (HiL) testing by forwarding peripheral access to a real device to validate hardware behavior directly rather than relying on mockups. However, most of this MDSE work is similar to and at least partially implemented by the practical code generation projects described earlier.

Looking beyond microcontrollers, I2CDevLib [i2c11] accumulates manually defined register maps for external devices such as sensors connected over inter-integrated circuit (I^{2}C) and serial peripheral interface (SPI), and provides basic C drivers for them. Cyanobyte [Fel20] continues this concept by generating device drivers against a number of HAL APIs from an abstract dataset. The main advantage of such a design is that projects with a custom HAL only need to provide a code template to gain access to all drivers [Fel20]. Yin et al. [YHZ$^+$11] builds on this concept by describing the algorithms required for the readout and conversion of sensor values as state machines to generate code from.

Expanding the scope again, Schirner et al. [SGD08] generate an entire HdS stack including tasks for a specific application by transforming a user-derived abstract system model using an architecture mapping that describes the hardware functionality in great detail. Acquaviva et al. [ABFV13] take an even more radical approach by using the electrical hardware representation, called register-transfer level (RTL), to extract state-machines and register maps that are transformed into C drivers. While both generators create highly functional HdS, they also require detailed inputs that significantly exceeds what is available from the technical documentation or any other publicly available source and is therefore not applicable for us.

In conclusion, we discussed related work in information extraction from PDF and HTML, especially table detection and understanding, and gave examples of existing such data pipelines in the area of embedded software, before finally describing code generation use cases that consume such data.

We note that even though data extraction from tables is a hard, but well understood problem, data pipelines in the embedded software space do not apply these lessons at scale and instead either focus only on extracting only specific data like pinout detection from documents (Instabuild, uConfig) or only extract data for specific devices (Datasheet2SVD). The most extensive pipeline projects (modm-devices, embassy-rs) eschew documents altogether and only use already machine-readable data (SVD, CMSIS Headers, STM32CubeMX database).

Projects using code generators are therefore limited to the scope of the easily accessible data, with a current focus on SVD files, or they are forced to manually build databases to enable their use case (I2CDevLib, Cyanobyte). However, several research ideas [SGD08, ABFV13, YHZ+11, WRSW21] use very extensive datasets for which a pipeline is missing as of now and therefore must derived the required data heuristically or via user input.

In the next chapter, we investigate how the lack of detailed data sources limits the process of porting and maintaining HdS and tooling, which we describe as a scenario as part of our problem statement.

4

Problem Statement

In the previous chapters, we provided an overview over how HdS interacts with the underlying hardware and gave examples of existing libraries that implement and abstract these mechanisms in several languages. However, the vast variety of different microcontroller hardware from many different vendors make the porting process of these libraries to new hardware a significant challenge due to the variety in functionality [EMD09, Kor18, BBA17]. For example, STMicro alone designed almost 3000 STM32 microcontrollers, each with different microprocessors, peripherals, memory sizes, and packages [modm16, modm09], requiring the use of GUI tools such as STM32CubeMX [stm08] to provide an overview of the configuration options of the HdS stack.

In this chapter, we describe the typical process of porting HdS to a new development board, containing a microcontroller and several external devices in Section 4.1 to understand how much effort the port requires and what kind of data we need to access in the technical documentation. From this description, we derived a set of challenges in Section 4.2. Then, in Section 4.3, we discuss how well the related work described in Chapter 3 meets these challenges, before stating a concise problem statement of the parts missing in the related work in Section 4.4. By tackling the problem statement, we achieve several individual contributions that we list in Section 4.5.

4.1 Porting Hardware-dependent Software

Our scenario focuses on porting a HdS stack to a new microcontroller connected to several external devices on the same development board. This process involves reading the technical documentation of the involved hardware and converting this information into code and configurations. In Figure 4.1, we listed the the individual steps involved in porting a HdS stack. We start our scenario at the lowest HdS layer and work upwards (cf. Figure 2.7), describing the data required for porting the

① boot firmware, ② HAL, and ③ device drivers for a ④ development board. We illustrate this porting process with a STM32 microcontroller, however, the steps are similar for most embedded devices. We conclude this section with a discussion of other data uses such as ⑤ configuration tools, build systems, ⑥ testing, and part evaluation.

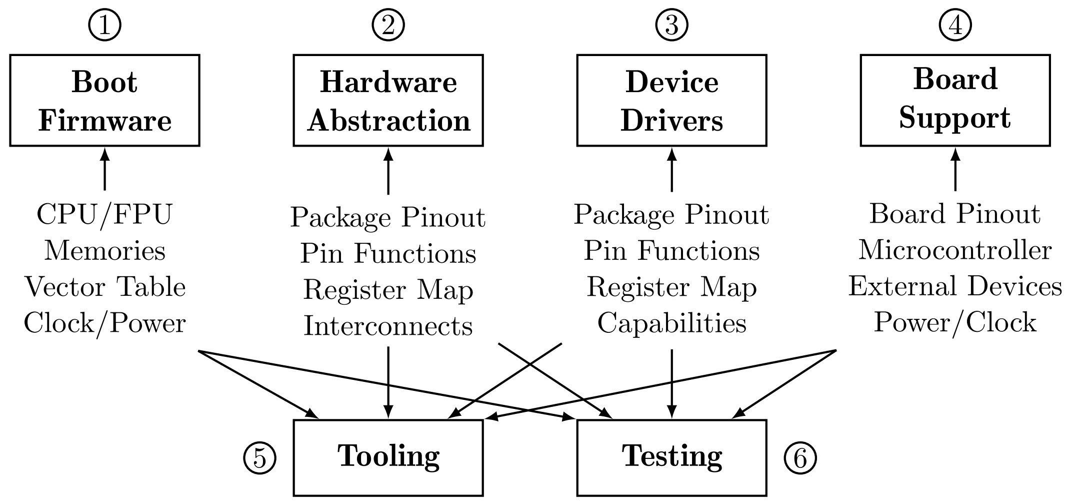


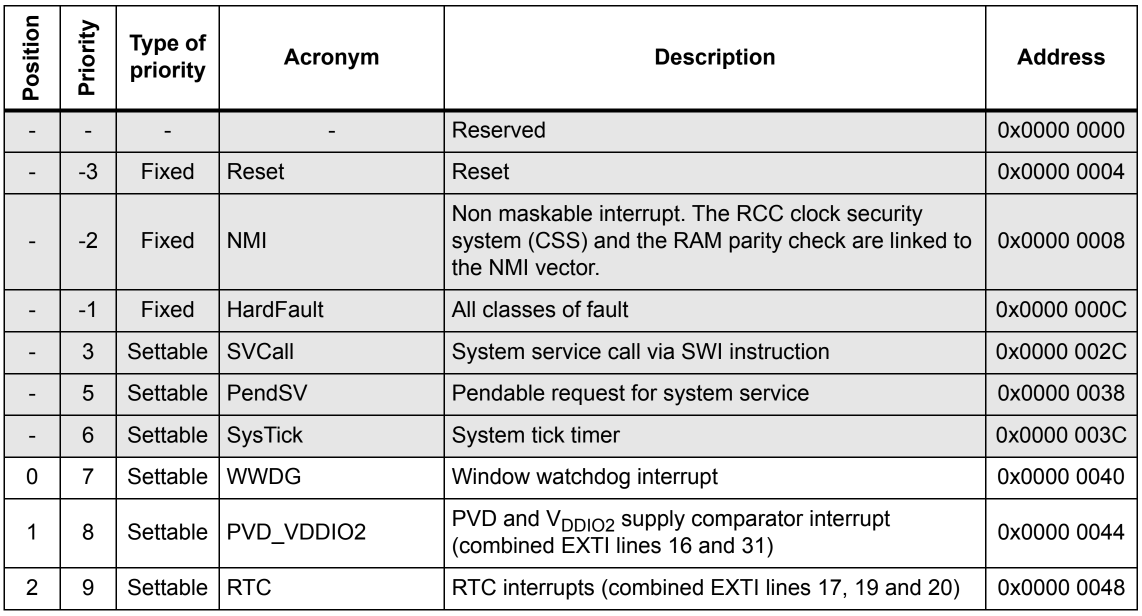
Figure 4.1 Porting process overview. Steps ① through ④ implement HdS stack layers and therefore depend on each other, while ⑤ tooling and ⑥ testing software is required at any point of the process. The individual steps depend on data that is manually transcribed from the technical documentation as part of the development effort.

4.1.1 Boot Firmware

The STM32 microcontroller series uses the ARM Cortex-M microprocessor, which boots on reset by jumping to the `Reset` function pointer located at the hardcoded address `0x0000 0004` and executing the instructions there [PM0253]. This address is part of the interrupt vector table defined in the reference manual as shown in Figure 4.2 and is unique for each device [RM0432]. A linkerscript tells the linker how to place the compiler-emitted instruction and data sections (including the interrupt vector table) into the correct hardware memories as defined in the reference manual [RM0432, PM0253].

Once the hardware jumps to the `Reset` handler, the boot process continues only in software, which configures the hardware further depending on the needs of the application [PM0253]. Typically the reset handler enables and configures the floating point unit (FPU), internal caches, bus peripherals to external memories, and copying data sections from read-only memory (ROM) to random-access memory (RAM). An optional second boot phase initializes the language runtime environment by setting up the heap, configuring exception handling, and calling static constructors [BBA17, EMD09, Kor18].

At this point, the device is ready to execute software, but only in the boot configuration, running at a low clock frequency and with no peripherals initialized [RM0432]. To achieve a high clock frequency, we have to configure the phase-locked loop (PLL) to multiply the input clock source, and then set up the clock and power graph to



Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000 0000
-	-3	Fixed	Reset	Reset	0x0000 0004
-	-2	Fixed	NMI	Non maskable interrupt. The RCC clock security system (CSS) and the RAM parity check are linked to the NMI vector.	0x0000 0008
-	-1	Fixed	HardFault	All classes of fault	0x0000 000C
-	3	Settable	SVCall	System service call via SWI instruction	0x0000 002C
-	5	Settable	PendSV	Pendable request for system service	0x0000 0038
-	6	Settable	SysTick	System tick timer	0x0000 003C
0	7	Settable	WWDG	Window watchdog interrupt	0x0000 0040
1	8	Settable	PVD_VDDIO2	PVD and V_{DDIO2} supply comparator interrupt (combined EXTI lines 16 and 31)	0x0000 0044
2	9	Settable	RTC	RTC interrupts (combined EXTI lines 17, 19 and 20)	0x0000 0048

Figure 4.2 The first 16 entries in gray of this interrupt vector table excerpt are reserved for signal handlers of the ARM Cortex-M0 microprocessor [PM0253], while the remainder is defined by the vendor and usually varies greatly between devices [RM0091].

distribute the clock to all required peripherals [RM0390]. Understanding where a peripheral is clocked from requires combining the overview renders in Figure 4.3 with the boundary address tables discussed in Section 2.3.1 [RM0390]. Once the device is configured, the boot process jumps to the `main` function to delegate control to the application. Table 4.2 summarizes the data we need to extract from the documentation for this step and the estimated effort to do so. However, in the next section, we will see that the porting of the HAL is already more complicated and requires much more data than this step.

Functionality	Data Description	Extraction Effort
CPU/FPU	Type, features, precision, and extensions	Low
ROM/RAM/caches	Type, locations, sizes, and power requirements	Medium
Interrupt vectors	Position and name of interrupt vectors	Medium
Power management	Supply type, power states, and clock gates	Medium
Clock graph	Clock distribution connections	High

Table 4.1 Summary of the data needed to port the boot firmware to a new STM32 device and the estimated effort required to extract it from the technical documentation.

4.1.2 Hardware Abstraction

Once we have configured the clock system, we write a GPIO driver that allows us to connect the microcontroller to external signals via its pin alternate functions. Both the pinout and the list of alternate functions is unique per device and described in the datasheet as a long table (cf. Figure 2.4), whose format has already been discussed in Section 2.2.

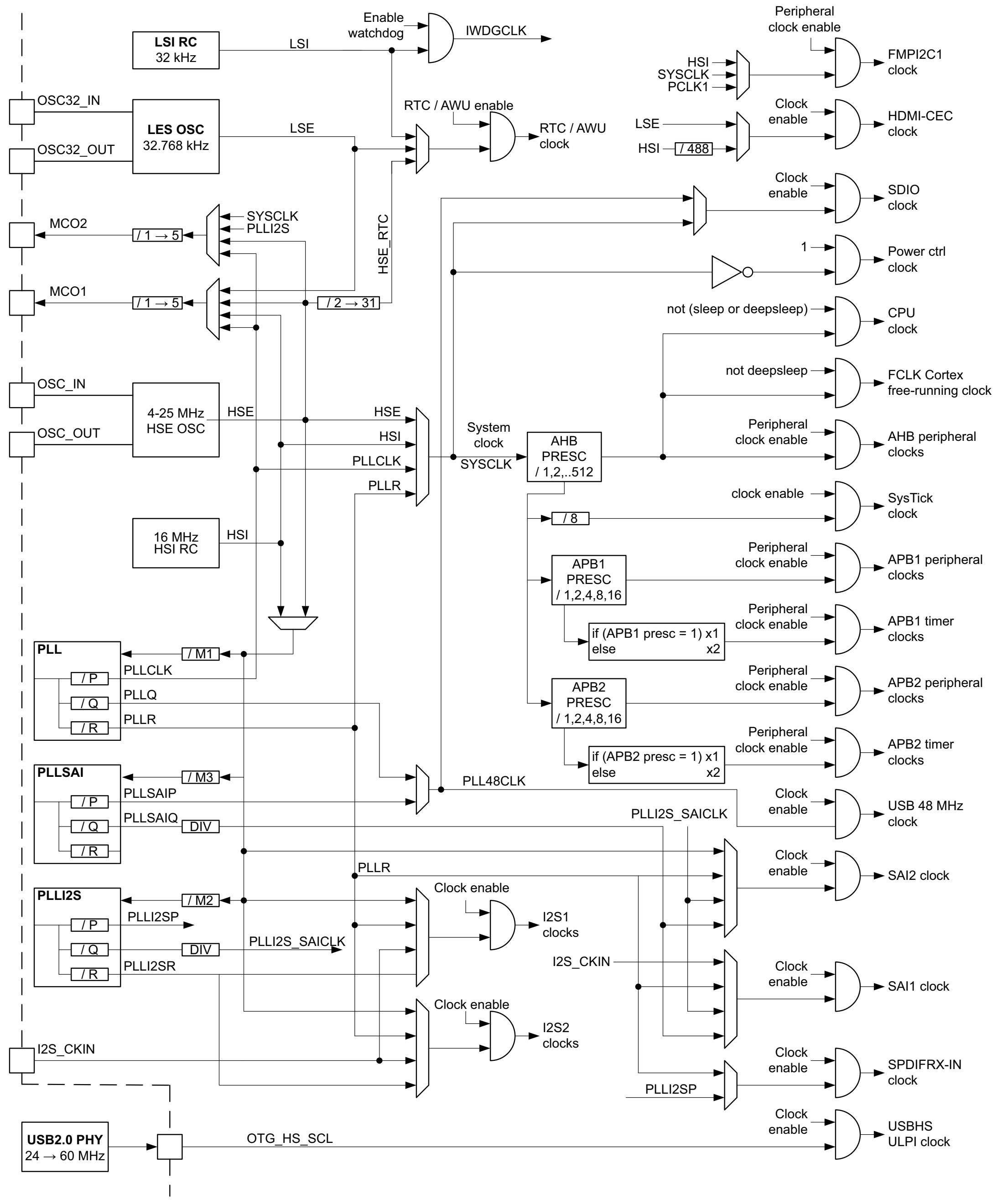


Figure 4.3 In this clock graph, the external clock signals enter on the device from the left and feed into multiple phase-locked loops (PLLs) that generate different frequencies. An internal network then distributed the clock to the peripherals on the right [RM0390]. The complexity of this figure and its contained graph data is much higher than for the interrupt vector table in Figure 4.2, making it a challenge to convert into code.

Next, we write HAL drivers for common special-purpose peripherals such as universal asynchronous receiver/transmitter (UART), SPI, I^2C, analog-to-digital converter (ADC), and digital-to-analog converter (DAC). For each peripheral, we consult the reference manual for the description of its functionality and register map as discussed in Section 2.3.1. If our HAL already has drivers for these peripheral types, we can check if they are compatible with our device's register map, preventing us from writing duplicate drivers [BBA17, Kor18]. For example, a simpler version of the CRC peripheral we discussed in length in Section 2.3.1 exists with a fixed polynomial and no data reversal options. This simpler CRC register maps subset is missing the `POL`, `POLYSIZE`, `REV_OUT`, and `REV_IN` bits in the register map from Figure 2.10, but functions identically otherwise [RM0432]. We can therefore determine peripheral compatibility by opening the reference manuals of every device our HAL already supports and manually comparing their CRC register maps.

More complicated drivers abstract the combination of several peripheral functions into a cohesive API [EMD09]. For example, instead of polling for new data, the UART driver configures a hardware interrupt to be triggered on data reception, which requires knowing which vector table position to insert the interrupt handler [BBA17, Kor18]. A common further driver improvement is to configure the DMA peripheral to transfer the received UART data to a memory buffer without any central processing unit (CPU) intervention at all [BBA17, Kor18]. For STMicro, we need to find the correct stream/channel combination for the peripheral event in the DMA trigger table from Figure 2.5, where the `UARTx_RX` events are all located in channel 4.

Table 4.2 provides an overview of the data required to port a HAL to a new device to show how much of the work is spent on finding, copying, and formatting data into code. More complex peripherals require accessing even more documentation for proper configuration. However, these peripherals often implement an external communication standard, such as CAN, Ethernet or universal serial bus (USB), whose entire descriptions are usually only referenced, but not reproduced in the documentation [BBA17, EMD09, Kor18]. Additionally, peripherals are usually not accessed on their own, but wrapped in a device driver, whose porting process we describe next.

Functionality	Data Description	Extraction Effort
Package pinout	Position and name of package pins	Medium
Pin functions	Index and name of pin functions	High
Peripherals	Name, type, instances, and features	Medium
MMIO register map	Description of each registers and bit field	High
DMA triggers	Peripheral events that trigger DMA transfer	Medium

Table 4.2 Summary of the data needed to port a HAL to a new STM32 device.

4.1.3 Device Drivers

External devices can range from simple analog temperature sensors connected via the ADC to complex communication modules connected via high-speed digital interfaces like UART or SPI, essentially acting similar to a peripheral connected via an external bus [BBA17]. The data we need to look up in the device datasheet is summarized in Table 4.3 and is largely comparable to the data required for the HAL. The software driver builds atop this HAL to abstract the communication and configuration of the device and represent its hardware features as a software API to the application (cf. Figure 2.7) [EMD09]. However, some of the device configuration is not determined by the driver, but by the way the hardware is connected on the development board and this information needs to be provided externally to the software as we describe next.

Functionality	Data Description	Extraction Effort
Capabilities	Input/output data ranges and features	Medium
Power and voltages	Electrical operating conditions	Low
Package pinout	Position and name of pins	Medium
Communication	Bus protocol and its configuration	Medium
MMIO Register map	Description of each registers and bit field	High

Table 4.3 Summary of the data and effort required for writing a new device driver.

4.1.4 Board Support Package

To assemble a complete hardware product, the microcontroller is connected to active devices and passive components via a printed circuit board (PCB), which routes the power and signals between all components [Kul17]. The PCB layout can also directly influence device configuration in software, for example, setting the I^2C address by pulling several device pins to a high or low voltage level [EMD09]. The hardware configuration of board components is represented in the HdS stack as the board support package (BSP) and includes the HAL and all device drivers in a configured state as specified by the PCB design. Figure 4.4 shows a block diagram of a STMicro IoT evaluation board chosen for its extensive suite of on-board devices connected to the internal peripherals. Figure 4.5 renders the pinout of the external connectors that can be used to integrate the board into a larger embedding system. The hardware connections and configurations of all components are described in the corresponding user manuals or datasheets. Table 4.4 lists the data we need to extract from the technical documentation for writing a new device driver.

Functionality	Data Description	Extraction Effort
Microcontroller	Part number and documentation	Low
External devices	Part numbers and documentation	Medium
Signal connections	Bus protocols, peripherals and pins	High
Power supply	Electrical operating conditions	Medium
Hardware configuration	Input Sources, Distribution	Medium

Table 4.4 Summary of the data required for a board support package.

Above the BSP exist the middleware and application layers of the HdS stack, which vendors provide separate documentation on. However, the data contained in these documents is context dependent and must be interpreted by a domain expert for a specific use case. We therefore describe next how software tooling can benefit from all the data we presented so far.

4.1.5 Configuration Tools and Build Systems

Tools that visualize the options for configuring the HAL, device drivers, and BSP are very useful for the discovery and understanding of hardware and software features. An example of a configuration tool is the STM32CubeMX application [stm08] which contains a large database of data as we described in Section 2.3.3. The build system

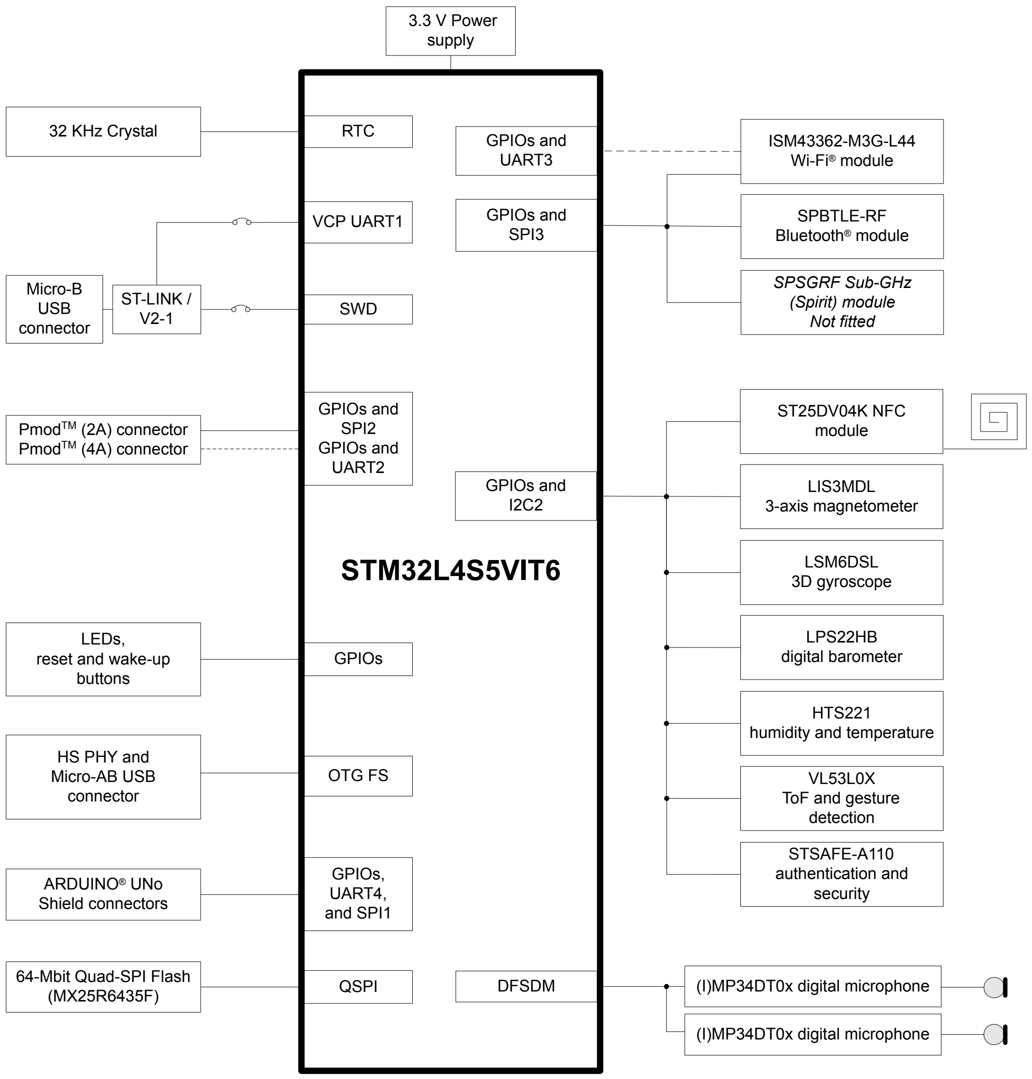

3.3 V Power supply
32 KHz Crystal
RTC
VCP UART1
Micro-B USB connector
ST-LINK / V2-1
SWD
Pmod™ (2A) connector
Pmod™ (4A) connector
GPIOs and SPI2 GPIOs and UART2
STM32L4S5VIT6
LEDs, reset and wake-up buttons
GPIOs
HS PHY and Micro-AB USB connector
OTG FS
ARDUINO® UNo Shield connectors
GPIOs, UART4, and SPI1
64-Mbit Quad-SPI Flash (MX25R6435F)
QSPI
GPIOs and UART3
GPIOs and SPI3
GPIOs and I2C2
DFSDM
ISM43362-M3G-L44 Wi-Fi® module
SPBTLE-RF Bluetooth® module
SPSGRF Sub-GHz (Spirit) module Not fitted
ST25DV04K NFC module
LIS3MDL 3-axis magnetometer
LSM6DSL 3D gyroscope
LPS22HB digital barometer
HTS221 humidity and temperature
VL53L0X ToF and gesture detection
STSAFE-A110 authentication and security
(I)MP34DT0x digital microphone
(I)MP34DT0x digital microphone

Figure 4.4 This hardware block diagram of a STM32L4 development board shows the on-board connections of four wireless communication modules, six sensors, two external security and memory devices, and several extensible wired interfaces [UM2708]. The BSP should provide an API for the application to configure all these interfaces and devices.

also needs to be configured according to the devices used in the project [EMD09]. In particular, the compiler must be informed of the architecture instruction set and language options to enable, which can differ depending on the CPU and FPU type, optional extensions, and layout of the ROM, RAM and caches [EMD09]. The debugger has to be told which debug hardware types and interfaces to enable [EMD09]. In addition, we also need to configure testing and simulation tools to validate the HdS implementation, which we describe next.

4.1.6 Testing and Simulation

Testing embedded software differs from more traditional software since the embedding system needs to be part of the test, which can make it more challenging to test the HdS layers independently [BBA17]. Therefore, reducing hardware exposure is key and can be archived by generating minimal test cases based on the hardware

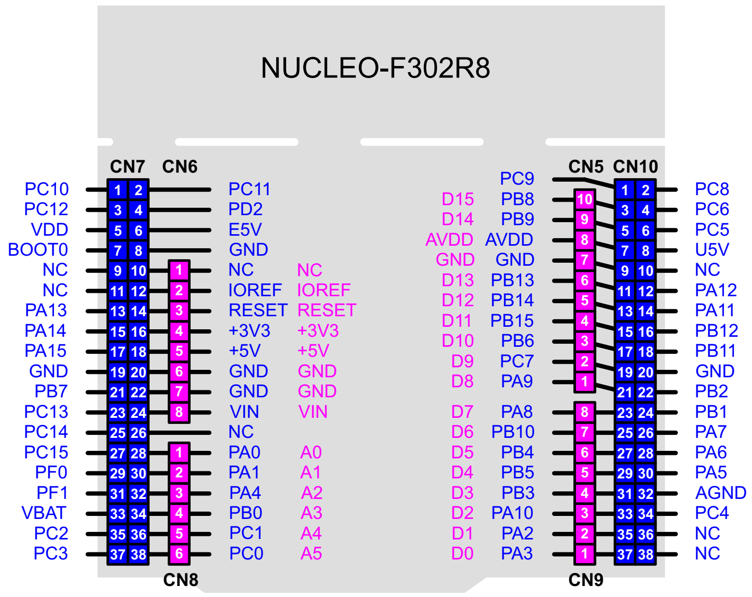

NUCLEO-F302R8
CN7
CN6
CN5
CN10
CN8
CN9
PC10 PC12 VDD BOOT0 NC NC PA13 PA14 PA15 GND PB7 PC13 PC14 PC15 PF0 PF1 VBAT PC2 PC3
PC11 PD2 E5V GND NC IOREF RESET +3V3 +5V GND GND VIN NC PA0 PA1 PA4 PB0 PC1 PC0
NC IOREF RESET +3V3 +5V GND GND VIN A0 A1 A2 A3 A4 A5
D15 D14 AVDD GND D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
PC9 PB8 PB9 AVDD GND PB13 PB14 PB15 PB6 PC7 PA9 PA8 PB10 PB4 PB5 PB3 PA10 PA2 PA3
PC8 PC6 PC5 U5V NC PA12 PA11 PB12 PB11 GND PB2 PB1 PA7 PA6 PA5 AGND PC4 NC NC

Figure 4.5 The external pinout for the Nucleo-64 development board renders multiple connectors and rows of pin names in an implicit tabular structure over the outline of the PCB [UM1724]. The embedded system interfaces with the embedding system through these connections [EMD09].

capabilities described in metadata [WRSW21]. However, due to the large amount of devices and their documentation, assembling this metadata can demand a lot of manual effort. A more flexible strategy is to simulate the hardware instead, which can make testing more reliable and reproducible [BBA17]. For example, the Renode tool simulates both the instruction set as well as a scripted peripheral implementation based on the MMIO register description [ant17]. However, if a more accurate peripheral behavior is required, we need to implement it ourselves by again consulting the technical documentation [ant17].

The steps we discussed so far apply only for porting new devices to the same HdS stack. Depending on the use case however, we may also want to port several HALs onto the same device for reasons we describe next.

4.1.7 Specialized Hardware Abstraction Layers

Some microcontrollers have such specialized hardware peripherals, which significantly impairs abstracting their functionality as an abstract HAL interface [EMD09]. Vendors advertise such peripherals as a unique selling points that help implement specific use cases, but require a high level of domain expertise to study the documentation and apply the vendor solution both on the hardware and software side of the problem [EMD09]. These specializations make it difficult to provide a truly universal HAL, since embedded hardware is highly fragmented [EMD09] and thus implementing multiple optimized HALs can be justified for use cases that we briefly describe next. However, the porting process and data required for each is comparable to the steps we discussed previously.

A high-performance brushless **motor controller** calls for time-synchronized analog readings for current control that are then transformed into specific waveforms via generation capabilities from the timer peripherals [MKM97]. The feature sets of the timer and analog peripherals can vary greatly between STMicro devices [modm16].

Audio processing on microcontrollers demands strict timing guarantees, which can be achieved by using DMA to transfer data from an input peripheral to memory, perform the processing operations using optimized digital signal processing (DSP) algorithms, before outputting the new audio stream on another peripheral also via a DMA transfer [MPSD20]. The CPU type and DMA trigger map is specific to the respective device (cf. Figure 2.5).

Wireless networking requires extensive knowledge of both the protocol and the hardware implementation, particularly if security and energy-efficiency is required [AN21, QRAS$^+$18]. Advanced features can realistically only be implemented directly on top of custom hardware peripherals and are therefore not portable by design [AN21, QRAS$^+$18].

Low-power embedded applications need to know the static power consumption of each enabled peripheral as well as the dynamic response of (re-)configuring the system at runtime [HBPT15, QRAS$^+$18]. Power-efficient scheduling abstractions need to work with arbitrary hardware limitations and thus can yield wildly different HAL and operating system (OS) concepts optimized for different microcontroller architectures [HBPT15].

Rendering 2D graphics for **user interfaces** can benefit greatly from hardware accelerators, which on microcontrollers are usually DMA-based blitting engines [Bod17]. However, the specific implementation of the accelerator can heavily influence the feature set of the graphics library [Bod17].

For each of these examples, the effort needed to port the specialized HAL to a new device increases even though the hardware remains the same across all HALs. In the worst case, the developers will look up the exact same data from the technical documentation for each HAL implementation, resulting in a lot of duplicated effort.

The scenario we described in this section shows the many important roles technical documentation plays in the porting process and beyond. For the purpose of implementing a new HdS stack or porting it over to a new device, developers need to extract a lot of specialized data from many different documents in multiple formats and transform it into code, tools, and configuration data. This process is complicated by the wide range of data the documentation provides, which intersects the expert domains of software and electrical engineering. To qualify this data and extraction effort in more detail, we introduce the challenges our design must address next.

4.2 Challenges

Our design needs to tackle several challenges that we derive from our scenario. Most challenges are concerned about the accuracy and resolution of the extracted data.

C1: Coverage The technical documentation bundles multiple similar devices into a single PDF to reduce duplication and provide context-dependent annotations in the form of table structure and footnotes to describe which data applies to what device. Our design must be able to understand these annotations and de-multiplex the contained data into a non-shared representation that covers each individual device.

C2: Fidelity The data derived from the documentation should contain enough detail for each hardware feature so that all relevant HdS implementation decisions can be derived sorely from the dataset and not have to rely on manually added information or heuristics.

C3: Correctness Our design must not introduce systemic errors during the data extraction process. Incorrect data needs to be detectable either via internal consistency checks or by comparison to external data sources.

C4: Clarity The extracted data must be unambiguously encoded so that querying different parts yields consistent results. If multiple sources with conflicting data exist, a deterministic strategy must be defined to merge them and produce a high-quality version.

C5: Maintainability Our design must be implementable with reasonable effort, where accessing the technical documentation does not require significantly higher cost compared to machine-readable sources. Wrong information in the documents that cannot be automatically corrected, needs to be automatically patchable also with reasonable effort. In particular, user intervention cannot be required due to the large volume of data.

C6: Extensibility Our design should allow for multiple input sources: technical documentation, source code, and proprietary databases to produce the most complete dataset possible. While this thesis focuses only on STMicro, our design should also allow for other vendors.

C7: Discoverability Our design should output the extracted data in a standardized format that aids with introspection and discovery of its content via third-party tools.

C8: Accessibility The dataset needs to have a simple API that presents a domain-specific view of the data on top of the storage format. This API helps embedded software engineers, who are not familiar with data science concepts, access the data.

In the following, we discuss how well the related work, presented in Chapter 3, can fulfill these challenges and what their deficiencies are.

4.3 Existing Work

In Chapter 3 we described the related work on the topics of information extraction, data pipelines, and embedded software. In this section, we assess how well these approaches address the challenges we formulated in the previous section.

4.3.1 Information Extraction

A recurring theme in the work we presented on information extraction is the focus on universal inputs, where the specific content domain and the document formatting and structure is unknown during the detection and extraction phase, therefore, the majority of the approaches are forced to use general heuristics to guide the process [CTT00, LKM01, LPL04, ETL05, RCVF03, CF04, RPS16, SAM+18, RPS+18]. When focusing on the detection of PDF tables, the common solution is to ignore vector graphics and detect and reconstruct the table structure only from the whitespace between the text of the cells [RCVF03, CF04, RPS16, SAM+18, RPS+18]. As a result, the accuracy of detection and conversion varies wildly per document and technique, which can make accessing the table content more difficult [RPS+18].

However, the format variation of technical documentation is restricted and its structural building blocks are much more specific than a generic PDF document. For example, the datasheets, reference manuals, errata sheets, and user manuals from STMicro all share the same style of formatting (cf. Section 2.1), so we can make simplifying assumptions for our conversion process. In particular, the detection of tables and figures can be guided by their caption as proposed by Clark et al. [CD16] and the tables cell partitioning can be guided by the vector graphics as implemented by Ramel et al. [RCVF03]. Similarly, text can be classified into headings, paragraphs, lines, lists, and sub-/superscripts if we know the font sizes and line spacings beforehand, rather than relying on heuristics [LKM01, RPS16, CHCG15].

Once the content has been detected, we need to comprehend it. In this regard, most work focuses on tables since they already contain an implicit structure to analyze. Extracting the data schema from the tabular structure works well with strongly relational tables. However, the tables in technical documentation are domain-specific and often introduce terminology and formatting that is bound to their context. Consequently, this approach is infeasible in practice. [Ras17, AS13]. Aligning the table data with an external schema requires an existing ontology [CHCG15], however, we could not find one for the embedded software domain. Moreover, text mining a substitute ontology from the document text [PA18, ZMH+21] seems challenging for such a technical domain and may require significant manual intervention. Since we do not want to parse all tables in the document, but extract a specific set of data for a specific purpose, we would instead manually match a mini-ontology onto a table structure [EAS13] and merge this data into a larger knowledge graph [TELN03, EAS13].

None of these works have applied table processing to technical documentation specifically, however, a few promising conversion tools exists, most notably the TEXUS processing pipeline [RPS+18] shown in Figure 4.6. While TEXUS is by far the closest practical solution for table processing technical documentation, it does not make use of captions or vector graphics to detect and understand tables and relies only on whitespace analysis of text and since it is a generic tool, does not provide an embedded software ontology either.

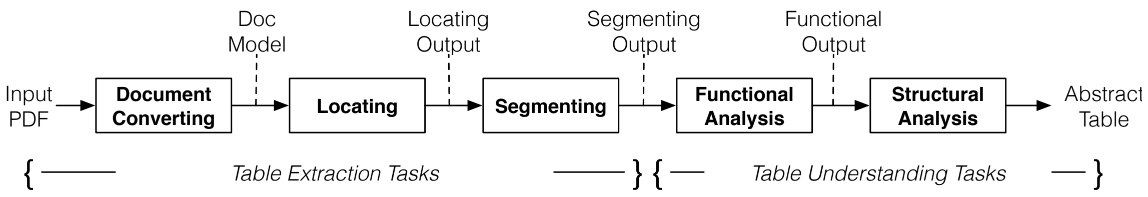


Figure 4.6 The end-to-end table processing pipeline from TEXUS [RPS+18].

4.3.2 Data Pipelines

Beyond generic information extraction tools exist several commercial and open-source projects that specialize in the embedded software domain. Instabuild [pdf13a] deploys computer vision to extract a pinout table from a datasheet screenshot, but requires user interaction (*↯C5: Maintainability*). uConfig [pdf17] extracts device pinouts by matching text positions inside the pinout figure diagram of the PDF directly without table processing and therefore cannot extract any other data (*↯C2: Fidelity*). Datasheet2SVD [pdf20] extracts the memory map from reference manuals based only on text, however, is limited to only two specific documents (*↯C1: Coverage*). No project gives any evaluation metric for their data correctness (*↯C3: Correctness*) or allows merging multiple sources (*↯C6: Extensibility*).

The two projects modm-devices [modm16] and embassy-rs-data [stm21] both extract data from already machine-readable datasets, specifically the STM32CubeMX database [stm08], CMSIS-Headers and CMSIS-SVD. However, these pipelines do not source PDF technical documents and are therefore limited to the data that the vendors provide in their tools and whose undocumented format can be reverse-engineered properly (*↯C2: Fidelity*). Both tools further store their data in custom formats (*↯C7: Discoverability*) and do not share any manual data fixes (*↯C5: Maintainability*). For STMicro in particular, the official CMSIS-SVD files are missing a lot of register descriptions [stm17a] and a crowd-sourced effort to patch them is not progressing fast enough [stm17b] (*↯C3: Correctness*).

In summary, the existing data pipelines that source PDF technical documents only do so for a small subset of the contained data, while pipelines that have high device coverage and data fidelity rely on vendor-provided machine-readable datasets. Thus, we identify a research gap in terms of a data pipeline that fulfills our outlined challenges (cf. Section 4.2).

4.3.3 Embedded Software

Code generators are widely used in academia and practice to convert data into code. Both the Linux Zephyr RTOS [lin14] and Embedded Rust [rust17a] use the language pre-processor to configure their HAL *during* compilation, while the modm libary [modm09] and the STM32CubeMX tool [stm08] generate their HALs using a template engine *before* compilation. All of these projects use their own datasets in their own format, either manually assembled of unknown quality or extracted via a data pipeline described previously. This diverse tooling landscape complicates data exchange, particularly if the templates themselves contain implicit data in the form of switching logic [HS15, HOSP21]. The exception are language-bindings generators that source the standardized and widely available CMSIS-SVD files, whose accuracy and completeness is, however, controlled only by the vendor.

Looking beyond HAL generation, I2CDevLib [i2c11] and Cyanobyte [Fel20] convert register maps and metadata into a basic device driver, but are limited to a manually assembled dataset due to the lack of machine-readable sources. Similarly, many projects in MDSE promise even more code generation [SGD08, ABFV13]. However, they require very detailed datasets that are simply not publicly available.

In conclusion, the information extraction approaches are focused on generic inputs, and cannot provide the domain-specific data found in technical documentation with the necessary accuracy. The existing specialized data pipelines rely on machine-readable data, which limits the extracted data to what the vendor provides, often substituting required but missing or incorrect data with manual transcriptions and patches. And finally, projects using code generators for their HAL or device drivers are not sharing their efforts due to incompatible data sources, formats and pipelines.

A solution could be a data pipeline that combines multiple input sources, including by table processing the technical documentation, to create the most complete dataset possible in an automated, unsupervised process. As illustrated in Figure 4.7, the database can be shared among multiple projects, so that improvements made by one project can benefit all, which could significantly reduce development effort. However, the difficulty of a solution exists primarily in accessing all data sources, especially technical documentation, with a reasonable effort. We present a concise problem statement that formulates what challenges such a design would need to solve in the next section.

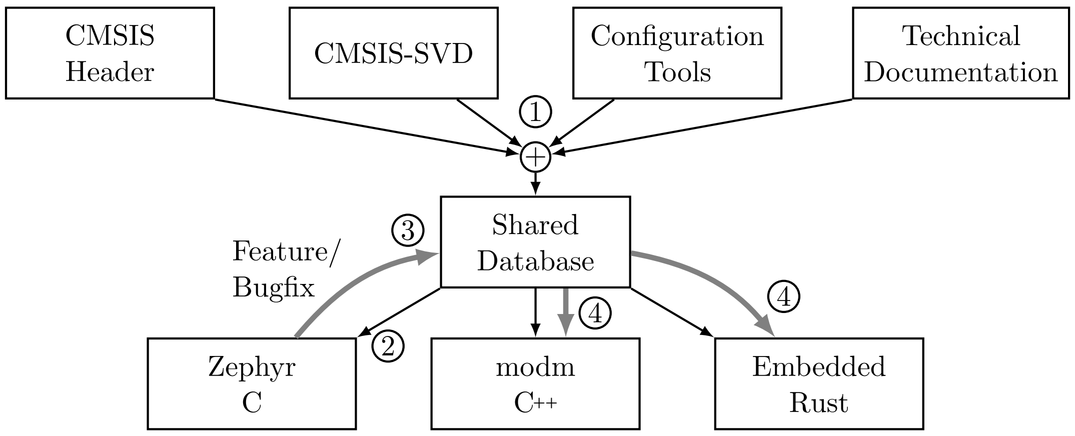


Figure 4.7 A data pipeline project ① combining multiple input sources into a shared database can help reduce overall development effort. ② A single project using the database can ③ add features or repair issues so that ④ all other projects benefit from the improvements.

4.4 Problem Statement

We described many use cases in our scenario that all require a large amount of data, most of which only exists in the technical documentation. While we found previous work that extracts information from generic documents and many projects that feed manually assembled data into template engines to code generate their HAL, we are not aware of any solution that connects both. Thus, we identify a research gap in terms of a data pipeline that fulfills the challenges outlined in Section 4.2. We split this problem into four parts: accessing technical documentation, processing its content, encoding the extracted information, and evaluating the quality of this data.

Since most technical documentation is available only as PDF, we first need to make their content accessible in a structured form, which is a complicated task due to the flat, print-oriented nature of the format. This process involves reverse-engineering the vendor-specific formatting style to associate individual page elements with a part of the structure and then converting that into a more suitable format. We need this conversion to be fast enough to work through tens of thousands of PDF pages and to yield reproducible results, so that we can make fine-grained, iterative improvements to our pipeline and get timely feedback about the performance of our tuning. In addition, the results need to be accurate enough so that the next steps have precise enough information that represents the content of the original document faithfully.

We then need to process the now accessible technical documentation to find and extract the information relevant to our use cases. However, the technical documentation is not written to be consumed as a detailed database, but presents an abstract and summarized view of the devices hardware optimized for human comprehension. We therefore need to understand in what form the data we require is available and what its encoding is. We already introduced the building blocks of technical documentation and table processing as an access paradigm. However, there is a large amount of content to consider with different types of information that may require additional context to interpret the content correctly, provided either externally by a domain expert or derived from the document itself. Therefore, decoding and combining multiple tables and texts may be required to generate data that is both accurate and detailed enough to substitute and augment machine-readable data sources.

Once we have the data for our use cases, we want to assemble it into a common representation that encodes it unambiguously and provides access to it for code generation tasks. These requirements calls for a format that can store and operate on large amounts of data and still be discoverable and easy to use later. Our design must also incorporate data provided externally through machine-readable sources to detect and repair conflicts between the difference sources to create the best possible dataset available.

After the design of a pipeline that achieves these requirements, we have to evaluate the corresponding implementation. For this, we extract data from the technical documentation that already exists in machine-readable form, so that we can prove the ability of our design to create such detailed datasets and compare its accuracy against it directly. Finally, suitable implementations should fulfill all the challenges we formulated in Section 4.2. With this problem statement in mind, we outline our contributions in the following section.

4.5 Contributions

With our thesis, we achieve several contributions:

Our pipeline design and implementation provides detailed access to technical documentation PDF content, via low-level primitives, as a high-level abstract syntax tree, and as HTML. In addition, we provide custom parsers for machine-readable data such as CMSIS header and SVD files and proprietary configuration tool databases. Even though, in this thesis, we specialize our implementation for data sources from STMicro, the pipeline design is flexible enough to be adapted for other data sources.

We use table processing and text mining paradigms to extract and convert data from the technical documentation in a deterministic process that yields completely reproducible results. We evaluate the extracted data from the technical documentation against machine-readable sources as well as check its internal consistency to establish a method to merge multiple sources and arbitrate conflicts based on qualitative metrics. We also provide a detailed analysis of the quality, trustworthiness and completeness of each data source, that can inform and guide future extraction work. The extracted data is unambiguously encoded as a knowledge graph using a custom ontology that describes the embedded hardware.

Our design is implemented as a pure Python package that handles all aspects of the conversion process unsupervised. Our implementation is highly modular so that parts of it can easily be reused for future projects. The source code is open-sourced and maintained as part of the modm project [modm22].

With the challenges and requirements of our solution formulated, we describe the design of our data pipeline in detail in the next chapter.

5 Design

The goal of this thesis is to assemble a large, detailed and high-quality dataset of microcontroller hardware descriptions to support the use cases described in Section 4.1. In this chapter, we introduce our design of a modular data processor to convert and merge technical documentation, source code, and configuration tool databases into a single knowledge graph that can then be accessed via simple or specialized APIs. We start by giving an overview of the design in Section 5.1, before detailing the individual transformation steps in Section 5.2. We conclude this chapter with a description of the access methods to the resulting knowledge graph in Section 5.3.

5.1 Modular Data Processor Overview

In this section, we give an overview of the design that transforms and merges multiple data sources into a shared representation annotated with domain-specific semantics. Specifically, the input sources are technical documents in PDF and HTML format and machine-readable data found in CMSIS and proprietary configuration tools and the output is a large knowledge graph containing a data model that is optimized for embedded software code generation.

The large variety in input and output formats and the many conversion steps between them contributes to the difficulty of our processor design. To manage this complexity, we split up the entire processor into six specialized data pipelines as described in Figure 5.1. Each pipeline converts only one data format into another and buffers the results in the respective archive, so that the individual conversion steps can be performed independently of each other. This modular design also allows for manual or automatic inspection of intermediary data between the stages to assess its quality and tune the conversion process iteratively without rerunning the entire pipeline. Consequently, the data processor can be composed of only those pipelines for which data sources are available or additional pipelines not covered by in this section, making our design easier to implement and universally scalable.

The end result is a large number of small knowledge graphs that are evolved into a single knowledge graph containing the entire dataset [TELN03, EAS13]. This final knowledge graph can be accessed via a predefined API for simpler use in existing code generation tools, converted into a standardized format such as CMSIS-SVD, or used directly via a graph query language such as SPARQL (cf. Figure 2.14). In the next section, we describe the tasks of each data pipeline.

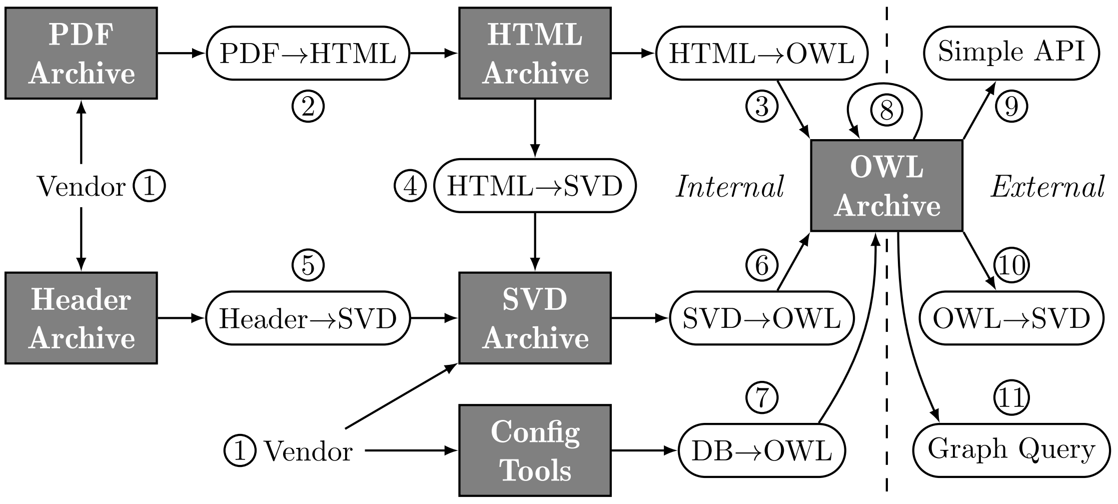


Figure 5.1 Data processor overview showing the internal pipelines on the left of the dotted line and the external access methods on the right. First, ① all data sources provided by the hardware vendor are imported. Then, ② the PDF technical documentation is converted to HTML and ③ the relevant tables contained within extracted into a knowledge graph. Additionally, the SVD memory maps are ④ extracted from the HTML, ⑤ CMSIS header files, and vendor-provided SVD files to be ⑥ merged into an optimal representation and stored as a knowledge graph. Finally, ⑦ the proprietary database (DB) contained in configuration tools is also converted into a knowledge graph. Then, ⑧ the separate knowledge graphs are evolved into one canonical knowledge graph by a merging strategy that corrects or at least minimizes data conflicts. This final knowledge graph can then be ⑨ accessed externally via a Python API, ⑩ converted into specialized formats such as SVD, or ⑪ accessed directly via a knowledge graph query language.

5.2 Data Processing Pipelines

In this section, we describe the design of each pipeline in detail, what their input and output formats are, which data they convert, and in what quality. We first describe what vendor data is imported (①), then we focus on the more complex pipelines that convert technical documentation (②, ③, ④), before we detail the pipelines using machine-readable inputs (⑤, ⑥, ⑦). We end this section with a description of the knowledge graph evolution (⑧). The access methods (⑨, ⑩, ⑪) are described in the next section.

5.2.1 Importing Vendor Data

Input: Raw input data from vendor website or repositories.
Output: Only the input formats relevant for our processor.

Our design uses four categories of input formats that are published by the vendors themselves. Since our processor design is modular, it can still operate *only* on technical documentation if other sources are not available. This is of particular importance when machine-readable sources are unknown.

The **technical documentation** is published on the vendor website as PDFs and can easily be scraped, since vendors do not typically authenticate access for public device documentation [Kul17]. The technical documents are also available from most electronics distributor websites [Kul17], thus are highly proliferated.

The **CMSIS header files** are usually bundled with example source code and published on the vendor website as an archive or source code repository. Since the header files become part of the application code, they are typically published under a very permissive license to allow their incorporation into proprietary applications. An archive of header files therefore likely already exists as part of a HAL projects foundations.

The **CMSIS-SVD files** are usually bundled with debug tools and published on the vendor website or GitHub also with a permissive license. Similarly to the CMSIS header files, an archive containing all vendor SVD files likely already exists.

The **configuration tools** are usually publicly available as well, however, finding and extracting the underlying proprietary database can require reverse-engineering the application, which may be prohibited by the license agreement. The extracted database should therefore be kept private.

These categories cover the typical data sources for ARM Cortex-M vendors which adhere to the standardized CMSIS formats. Additional vendor-specific data formats can be converted into their a small knowledge graph via their own pipeline, for example, parsing the source code of a C-based HAL to extract data not encoded anywhere else. Now that we located all vendor-provided data, we can start converting it using the pipelines we describe next.

5.2.2 Converting PDF to HTML

Input: Technical documentation as PDF.
Output: Technical documentation as HTML.

In this step we reverse-engineering the formatting style of the PDF to assign the equivalent HTML semantics to characters, vector graphics, and images. The content in the resulting HTML is then much easier to access for the next pipelines [CTT00, LKM01, LPL04, ETL05]. Inspired by the extraction tasks of TEXUS (cf. Figure 4.6) [RPS$^+$18], we first abstract the PDF contents into an internal document model, before locating table, figure and image areas using their caption [CD16] and vector graphics shape [RCVF03], with the remaining areas containing only characters. We then convert each content area separately into an abstract syntax tree (AST) to cluster the PDF objects into a hierarchy first [RCVF03, CF04, SAM$^+$18]:

(i) Characters are first linked into horizontal and vertical lines based on their position and rotation [CF04], then grouped into headings, paragraphs, lists, and annotations based on their line spacing, indentation, and font properties. Font rendering and metadata information such as bold, italic, and linked characters are also preserved.

(ii) Tables are segmented into a grid structure where cells can span multiple rows and columns, either using the vector graphics [RCVF03] or using whitespace analysis [RPS+18]. This grid structure is then converted into a table model, where the header and data cells are identified through vector graphic and font information.

(iii) Figures and images are converted into objects storing the caption and the contained vector graphics, bitmaps, and characters as verbatim data. Unlike tables, no processing of figures or images is possible at this stage, since we do not know how to interpret the vector graphics or pixel data semantically.

Since PDF is a paginated format, these steps must be performed for each page separately resulting in many small ASTs. These small ASTs now describe the logical content hierarchy together with vital formatting metadata, such as text indentation spacing to indicate lexical scope, that further contextualize the individual object semantics. We then unpaginate the content by merging these small ASTs into one large AST by using the text indentation information and the shape of the tree to align the page beginning with the previous page end. The large AST is then modified by a number of passes to align it further to the HTML content model, for example, merging paginated tables back into one table, adding list beginning and end markers, and coalescing individual characters rendering properties into groups. Serializing the AST into a simple subset of HTML is performed by recursively walking the tree and converting each abstract node into the HTML equivalent.

Due to the lack of reference data, the quality of the results must be assessed manually, however, this requires no special tooling, since HTML is easily inspected by a plain text editor and rendered by any web browser. Compared to previous work using heuristics (cf. Section 3.1), this pipeline is tuned manually and creates reproducible HTML output, which allows to iteratively improve the accuracy of the conversion over time.

5.2.3 Converting HTML to OWL

Input: Technical documentation as HTML.
Output: Knowledge graphs modeled in OWL.

Since technical documentation regularly consists out of hundreds to thousands to pages (cf. Section 2.1), we provide hierarchical access based on the chapter name, heading, and table caption to speed up searching for texts and tables of interesting in the entire HTML document. Once we discovered our content, a lightweight wrapper allows for simple text mining of paragraphs and lists and gives access to table content via the abstract table model described in Section 2.2.

Depending on the information we want to extract, we need to combine and clean up data from multiple tables and texts and decide how to encode the result in a knowledge graph. Especially important is the creation of an unambiguous encoding of the extracted information, which can be difficult for our domain of embedded software, since there exists many entities that share the same name, but differ in relations. For example, different packages can have the same pin name (modeled as an entity), but the pin itself is connected to different signals (modeled as relations to signal entities). Thus, when adding both pins to the same knowledge graph, the signal relations are collapsed and now point to both pin's signals, creating an ambiguity. Therefore, we create one knowledge graph per data source and defer merging to a later stage to preserve as much context as possible and prevent accidental ambiguous encodings.

We want to exemplify the generic description of the pipeline's design with the STMicro technical documentation to give a better understanding of the process. The datasets we describe here are chosen for their duplication in machine-readable sources to the benefit of our evaluation. However, technical documentation usually contains many more datasets that can be extracted using similar methods to those presented in these examples:

Device identifiers from datasheets: The order information chapter near the end of the datasheet contains a legend of STM32 order codes, that describes the meaning of each entry and their possible values. We use this to create a list of all possible identifier combinations via the n-fold cartesian product. However, this list overlaps with other datasheet device lists, creating an ambiguity, therefore we filter it with the partial identifiers from the device order table found in the introduction. This filtering finally creates a true n-to-1 mapping of device identifiers to a specific datasheet.

Device identifiers from reference manuals: Since reference manuals apply to a larger group of devices than datasheets, they do not contain order information and we cannot recreate an exact list of device identifiers. Instead we find the first mention of `STM32.*?` in the introduction, which acts as a filter for an externally provided list of device identifiers that apply for this document. We defer the resolution of the filter into a true n-to-1 mapping to the knowledge graph evolution stage (⑧), where we have access to the device identifier lists from the datasheets and configuration tools.

Interrupt vector table from reference manuals: The nested vector interrupt controller (NVIC) chapter contains a table with a map of entry position to vector names (cf. Figure 4.2). If multiple tables are found, the caption of the table contains a device filter that cannot be resolved locally, since the reference manual does not contain a list of device identifiers. Therefore, we encode all vector tables with the device filter and defer their resolution to a later stage.

Device pinouts and pin signals from datasheets: The pin description chapter contains a table that lists the pin number per package, pin name, type, structure, and signal functions (cf. Figure 2.4). For the pinout, we map each package via a pin number relation to the pin name in a process we described in Section 2.2. In some tables, the pin numbers column contain additional device filters that must be resolved to unambiguously map the device identifier to the package. However, for the pin signals, we also need to consider a number of separate alternate function tables in the same chapter, which explicitly list the alternate function *index* for the signal, rather than the sum of all signals. We combine these alternate function tables with the additional functions column and map them to each pin.

To avoid inter-document data conflicts, we create a separate knowledge graph encoding the extracted data for *each* document, whose quality can be more easily manually verified due to the smaller scope. The knowledge graphs are modeled using OWL semantics, however, in this pipeline we only use it to store data in the graph without adding any advanced features like meta-modeling or rule definitions (cf. Section 2.4). We delegate the meta-modeling and merging of these small knowledge graphs to stage ⑧, which can evaluate all graphs at the same time to formulate the best merging strategy.

5.2.4 Converting HTML to SVD

Input: Technical documentation as HTML.
Output: MMIO register descriptions encoded as SVD.

The reference manual contains the description of the MMIO register map as a series of table and text patterns repeated for each peripheral (cf. Section 2.3.1). As a starting point, we find the peripheral boundary address table (cf. Figure 2.9) in the RCC chapter to build the address space mapping of the on-device peripheral instances.

Then, for each peripheral chapter in the document, we perform the same four steps:

(1) Find and parse the register map summary table (cf. Figure 2.10).

(2) For each register name in the summary table, find the corresponding section in the chapter (cf. Figure 2.11) and parse the register bit table as well as the bit descriptions underneath to get the bit positions, bit names, and value ranges.

(3) We now have redundant information for register and bit names, positions, and values, which we use to fix inconsistencies in the data by weighted voting to create the most accurate register map version.

(4) We associate the completed register description with the instance name and address in the peripheral boundary address table.

This pipeline formats the register map into a SVD file rather than a knowledge graph, since SVD is standardized and there are a number of libraries and applications that can operate on it directly [svd15d, rust16, ada15, svd15c] (cf. Section 3.3). The SVD files from this pipeline are then merged with the vendor-provided SVD files and the ones derived from the CMSIS headers separately in step ⑥ and converted into a knowledge graph.

5.2.5 Converting Header Files to SVD

Input: CMSIS header files for C-language bindings.
Output: MMIO register descriptions encoded as SVD.

CMSIS header files are code generated C-language bindings that adhere to a strict naming schema as defined by the generator tool (cf. Section 2.3.2). We illustrate the four step process of parsing this file to recreate a register map with the CRC peripheral definitions from Listing 2.5:

(1) The peripheral base address is available in the CPP statement `#define CRC ((CRC_TypeDef*)0x40023000)` and equivalent to the entry in Figure 2.9.

(2) The peripheral register map from Figure 2.10 is generated as the C-language struct `CRC_TypeDef`, with the reserved register at offset `0xC` explicitly stated.

(3) The peripheral register bits from Figure 2.11 are formatted as CPP statements `#define CRC_CR_RESET (0x1 << 0)`.

(4) The peripheral register bit fields are part of the SVD standard, but *not* represented in the STM32 CMSIS header files. For example, the bit field values for `REV_IN[1:0]` from Figure 2.11 are only available as individual bits instead of numeric configuration values with names. However, if other vendors define these bit fields as CPP statements, we can extract them here.

We store the extracted definitions in a tree data structure that is easy to serialize into an SVD file. The header files also contain a number of other CPP macros that describe additional device properties, which are added to the SVD file as well.

5.2.6 Converting SVD to OWL

Input: Multiple MMIO register descriptions encoded as SVD.
Output: Merged MMIO register descriptions modeled in OWL.

At this step, three register descriptions of varying resolutions and qualities exist: (a) extracted from technical documentation, (b) converted from the CMSIS header files, and (c) provided by the vendor. We merge these three representations into one by filling in missing and eliminating wrong information via majority voting.

However, as we will discuss in depth in Chapter 7, some data conflicts are decided by only two sources if the third one is missing the data point altogether. In that case, we manually specify an arbitration strategy by categorically trusting one source more than the other or by patching one source before merging. Since creating patches requires a lot of manual effort, we can use it only as a last resort to repair the wrong data patterns with the highest occurrence, which limits the effectiveness of this approach.

This process results in a version with the most accurate and complete data overall and the highest resolution and fidelity per device. At this point we convert the register description into a knowledge graph.

5.2.7 Converting Tooling Data to OWL

Input: Tool database in various machine-readable formats.
Output: Knowledge graphs modeled in OWL.

Configurations tools often contain easily accessible and internal databases with detailed information on the vendor's devices. For STMicro, the STM32CubeMX configuration tool described in Section 2.3.3 contains an XML database, from which we extract the (a) device identifier list, and for each device the (b) packages and pinouts, (c) pins and signals, and (d) interrupt vector table.

The same caveats about choosing an unambiguous knowledge graph modeling discussed in Section 5.2.3 applies to this data too, therefore we reuse the class structure and encode each device in a separate knowledge graph.

5.2.8 Evolving OWL

Input: Multiple knowledge graphs modeled in OWL.
Output: A single knowledge graph modeled in OWL.

The previous steps have now created various knowledge graphs, which this step transforms into more accurate and consistent knowledge graphs. At this point, we must finally resolve the issues with ambiguous encoding that we deferred from the previous steps, by carefully deciding which entities are equal and can be merged and which ones need to be placed in a namespace to preserve their information. With this in mind, we perform the following transformations:

(i) Filtering the very large list of all possible device identifiers from the datasheets with the list of devices from STM32CubeMX that STMicro actually produces: We can then use this list to resolve the device filters for the reference manuals, resulting in a complete n-to-1 mapping of device identifier to datasheet and reference manual.

(ii) Merging data generated from the datasheets and reference manuals with data from STM32CubeMX to create the most accurate version of the package pinouts, pin functions, and the interrupt vector table: This step compares all sources and chooses the best data combination based on an manually written selection algorithm in combination with data patches and statistical metrics.

(iii) Deduplicating data and determining its inter-compatibility: For example, collapsing all peripheral register descriptions into a minimal compatible set is particularly helpful for deciding how many different HAL drivers at what level of specialization need to be written to cover all devices we want to support, as described in Section 4.1.

(iv) Inferring new relations between entities added from different sources: For example, in Figure 2.13, we added an $\xrightarrow{\text{enable}}$ relation between register bits and pin signals, so that the code generator can use this information directly (cf. Section 2.4). By adding this data at this stage where we have access to all data, we can detect emerging patterns and discover new edge cases that may be invisible when focusing on a smaller group of devices.

To make the transformations reproducible, we do not modify existing knowledge graphs in-place, but create a single new graph and copy the transformed data over. This knowledge graph contains the most compact representation of the data, with all compatible entities deduplicated and all non-compatible entities namespaced with their respective scope, usually the device identifier. This extensive graph is then considered to be the final and definitive data source for the external access methods described in the next section.

5.3 Accessing OWL

The knowledge graph resulting from the pipeline contains the sum of all device data, resulting in a large graph with multiple namespaces to prevent ambiguity. Embedded software developers may not be familiar with semantic web concepts and implementations and may find accessing the knowledge graph directly too complicated or tedious when handling the scenarios describing in Section 4.1.

We therefore provide ⑨ a software wrapper API that presents a simpler data encoding optimized for the code generation use cases presented in our scenario. For example, given a specific device identifier, the interrupt vector table can be returned as a map of integer to string, which makes it easier to convert to a generated array in the firmware. Similarly, given a device identifier, ⑩ an SVD file is produced from the register description in the graph, which can then be passed to existing tools such as debuggers or language binding generators. Embedded developers with previous experiences of knowledge graphs may use ⑪ a graph query language such as SPARQL to modify, extract, and format the data they are interested in for their specific HdS scenario.

For advanced use cases, we expect developers to convert the relevant subset of the graph data into an intermediary data format that contains only the information relevant to the task in an encoding that is optimal for it. If issues with the graph structure or data quality are discovered that require the graph model to be changed, thus breaking backwards compatibility, the indirect access via a software wrapper gives an opportunity to transparently pass the fixes along or to choose an upgrade path compatible with the use case.

In this chapter, we broke down the complex process of converting and merging multiple data sources into self-contained pipelines that each perform a manageable number of tasks. This modular design can be tailored to operate only the specific data sources a vendor publishes and can be extended to include new data sources and new hardware devices in the future. Our intermediary archives give many opportunities to peek into the data between the pipelines with just a text editor and a web browser to verify and tune the effectiveness of each conversion step individually. The resulting knowledge graph contains the most accurate and highest fidelity dataset by merging the best of all data sources together. Finally, we described how we provide a simplified API access for common use cases. The following chapter presents a look at the actual implementation of this design.

6

Implementation

After we introduced the modular design of our pipeline, we elaborate the concrete implementation in this chapter, starting with a detailed explanation of how each data format is accessed and converted in Section 6.1. We conclude this chapter with a description of the resulting knowledge graph ontology and how to access the data stored within in Section 6.2.

6.1 Data Processing Pipelines

In this section, we present how we access the individual data formats and convert them in the same order as specified in Figure 5.1. Even though our design uses separate pipelines for each conversion, we specifically placed all code into a single Python 3.9 project with one submodules for each format and each conversion to maximize code reuse. We further split each submodule into a vendor-agnostic part for shared algorithms and data structures and a vendor-dependent part in anticipation of future vendor specializations. To speed up the conversion process, we use multi-processing as a substitute for multi-threading, which works works well for our use case of converting many individual files. All code is published as an open-source project on GitHub [modm22].

6.1.1 Importing Vendor Data

The four main data sources described in Section 5.2.1 are all publicly accessible on the vendors website as downloadable archives or on GitHub as source repositories:

(i) The **technical documentation** is published on the STMicro homepage [stm07], spread across multiple subpages with their metadata collected in a number of JavaScript object notation (JSON) files. A Python script collects all metadata and compares it to the local archive to download all documents, specifically reference manuals, datasheets, user manuals, technical notes, data briefs, application notes, errata sheet, and programming manuals as of June 2022 (cf. Section 2.1). The documentation copyright prohibits republication without written consent, therefore we store the archive in a private Git repository.

(ii) The **STM32CubeMX database** is published by STMicro on GitHub [stm20] with the stated intention of being used as a data source for HdS code generation. We import v6.5.0 of the database.

(iii) The **CMSIS header files** for STM32 are published by STMicro on GitHub [stm19] and further aggregated in a single repository by the modm project [modm17]. We import the latest revision of each header file as of June 2022.

(iv) The **CMSIS-SVD files** for STM32 and other ARM Cortex-M vendors are aggregated by open-source contributors from various sources into a single GitHub repository [svd15a]. We import the latest revision of each SVD file as of June 2022.

The machine-readable input sources are all available as Git repositories, since they are already used by a number of projects and tools (cf. Section 3.3), while the technical documentation can be downloaded from the STMicro homepage without access limitations. Therefore, all necessary data for our processor can be acquired using only an internet connection on commodity hardware with moderate data storage requirements.

6.1.2 Accessing PDF

To access the low-level data structures of the PDF documents described in Section 2.1, we built a small content abstraction model on top of the pypdfium2 v1.9.1 Python bindings [pdf21] to the C++ PDFium library [pdf13b]. ¬The PDF object stream exposes a rich hierarchical content model [pdf08] that we have only partially abstracted, in particular, we did not implement any content modification methods, since we only read the PDFs. We give a brief overview of the important classes in the API:

The **Document** class contains a metadata catalog of standardized keys, such as author, producer, and creator, which can be used to identify the vendor and the visual style of the PDF document [pdf08]. The table of content yields the names and page numbers of all chapter and section headings without needing to parse the document first, which is useful for splitting a document into sections for parallel processing [pdf08].

The **Page** class uses the dimension and rotation information to scale the content via affine transformation into a normalized view, so that our conversion code can ignore page and content orientations [pdf08]. The page content is placed in a tree data structure that provides an efficient spatial access, since the conversion process often queries small areas of the page, for example, when converting table cell content.

The **Character** class binds a Unicode point to a specific font to render a single glyph at a position and angle with font size, fill, and stroke colors [pdf08]. To understand what area is used by a glyph, bounding boxes are provided: the loose bounding box is the maximum space any glyph in the font can use, while the tight bounding box is the actual space it takes [pdf08]. Similar glyph rendering effects can also be achieved using combinations of different parameter settings [pdf08]. For example, to render differently sized glyphs, one font can be scaled up and down, or a different font can be used for every scale to the same effect. The choice is up to the PDF creator program, which can be identified from the metadata.

The **Path** class holds vector graphics defined by a list of points that can be jumped to or rendered as a line or bezier curve with a width, fill, and stroke colors, based on the Postscript language [pdf08]. The point positions are normalized by resolving all affine transformation matrices in the rendering instructions.

In addition, there exists an Image class to render bitmaps and a Link class to model inter-document and web references. With this abstraction in mind, we present our PDF to HTML conversion algorithm next.

6.1.3 Converting PDF to HTML

This pipeline is the most complex part of our design. Therefore, we break down the pipeline into many small steps that each perform a specialized operation. Since we need to reconstruct the content of the entire PDF document without any formatting or structural hints, we make a few simplifying assumptions that reduce the practical implementation effort without any effect on the conversion accuracy:

(i) The formatting style remains identical throughout the entire document, which allows for detecting the style per document metadata, rather than page content.

(ii) Tables and figures are marked with captions and the separation between tables, figures and text is clearly defined. Table cells in particular are encapsulated on all sides by a border rendered in vector graphics. We can therefore limit the use of heuristic whitespace analysis to detect tables and reconstruct their structure.

(iii) Characters are placed only horizontally and vertically rotated 90°, and not at arbitrary angles, and their Unicode point equals their glyph. This assumption simplifies text line detection and avoids the use of OCR.

For STMicro documents, which are the primary focus of our implementation, we detect two rendering styles via the document metadata creator tag, which we call black/white and blue/gray. The differences between these styles lie in their color scheme and table rendering style, as well as several magic numbers such as line and paragraph spacing, heading sizes, sub-/superscript offsets, and content areas inside a page. These style variations are passed as arguments to the conversion algorithm, which operates on the PDF abstraction model described in Section 6.1.2. In the following, we list the individual steps of this algorithm:

(i) **Detect page layout**: Some datasheets have a two-column layout on the first pages, which we can detect by probing for characters in specific parts of the page. The subsequent steps operate on each detected content area separately.

(ii) **Detect text lines**: We convert individual characters into horizontal and vertical lines depending on rotation by coalescing their origin positions. Next, we merge the super- and subscript character into their corresponding lines by checking for bounding box overlap and comparing font sizes.

(iii) **Detect graphic areas**: All paths and images are clustered into the smallest possible areas based on overlapping bounding boxes. The figures in the blue/gray style documents lack a frame box, but have a minimum guard distance to the next element, so we also consider characters close to graphics in our clustering algorithm.

(iv) **Partition page content**: We find tables and figures via their bold caption and assign it to the correct graphic cluster from the previous step. The remaining, uncaptioned graphic clusters are categorized by analyzing the path patterns: table paths are rectangular paths that overlap in specific ways, while figures contain bezier curves and intersect each other [RCVF03]. To detect register bit tables (cf. Figure 2.11), we check for a row of numbers above the cluster bounding box.

(v) **Detect table structure**: We convert the table structure into an abstract model using the vector graphics: based on the horizontal and vertical lines we create a cell grid [RCVF03] and then merge cells together depending on their graphic borders. Additional cell properties such as fill or stroke colors are currently ignored. For the numbers on top of the register bit tables, we add a virtual row and columns based on the whitespace between the character clusters.

(vi) **Detect text structure**: The content areas not detected as tables or figures now only contain text. We categorize each text line into headings, paragraphs, lines, lists, note/caution/warning annotations, and register bit descriptions. The category type is decided by regex matching the text line beginning and since each category can span multiple lines, we use the AST (cf. Section 5.2.2) to store the context necessary to detect when a category begins and ends and to arbitrate ambiguous formatting as best as possible. Tables and figures are also added to the AST in this step at the right place, with each table cell getting their own small AST to detect the text structure.

(vii) **Merge area ASTs**: We create one large AST by inserting the page area AST so that the local scope continues correctly. This unpagination requires finding the node that matches the current category and indentation. For example, an area starting with a list entry must be matched to the correct list or sublist scope at the end of the previous area.

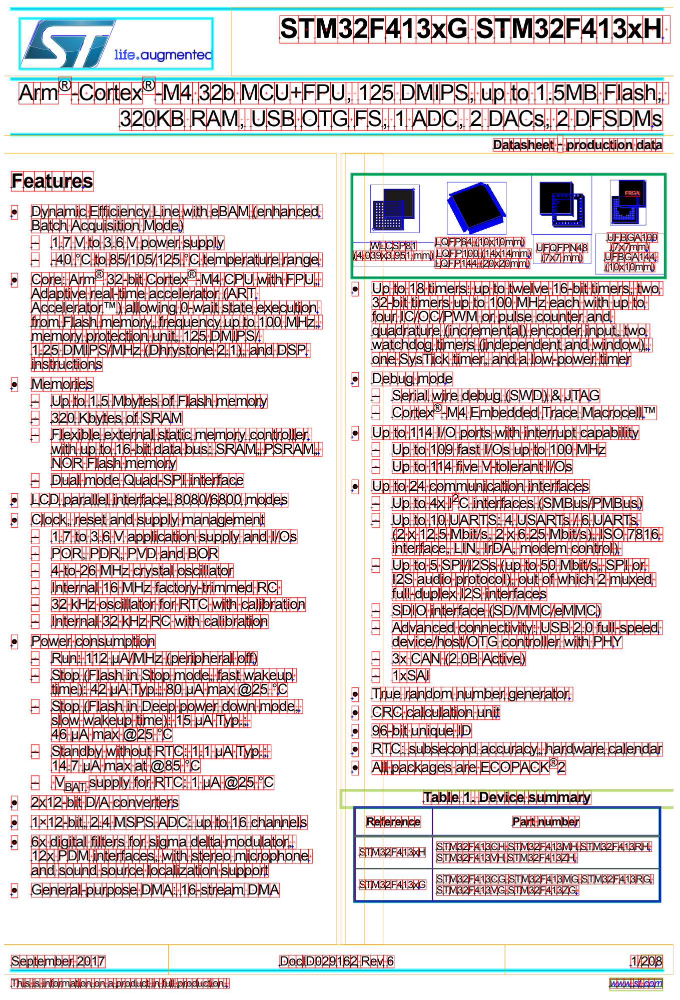
(viii) **Rewrite AST**: To further align the AST with the HTML content model, we run several small algorithms over the tree structure to rewrite it. We list the most important operations here:

   (a) We unflatten sequential list elements into a hierarchical group structure as is expected by the HTML list tags.

   (b) We convert the bit register descriptions into virtual tables. The descriptions were not detected as a table, since they have no borders, so they were treated as text, until we can rewrite this here.

(c) We merge tables spanning multiple pages into one table, by aligning their grids and concatenating the cells. We also merge two 16-bit wide register bit tables into one 32-bit wide table to provide easier access later on.

(ix) **Serialize AST into HTML**: The AST is now very similar to HTML, so we can walk the tree recursively and convert each node into its HTML equivalent. Only at this last step do we work with individual characters by merging their rendering properties into a compact range and converting it to bold, italic, super- and subscript, and newline HTML tags.



STM32F413xG STM32F413xH

Arm®-Cortex®-M4 32b MCU+FPU, 125 DMIPS, up to 1.5MB Flash, 320KB RAM, USB OTG FS, 1 ADC, 2 DACs, 2 DFSDMs

Datasheet - production data

Features

- Dynamic Efficiency Line with eBAM (enhanced Batch Acquisition Mode)
  - 1.7 V to 3.6 V power supply
  - -40 °C to 85/105/125 °C temperature range
- Core: Arm® 32-bit Cortex®-M4 CPU with FPU, Adaptive real-time accelerator (ART Accelerator™) allowing 0-wait state execution from Flash memory, frequency up to 100 MHz, memory protection unit, 125 DMIPS/ 1.25 DMIPS/MHz (Dhrystone 2.1), and DSP instructions
- Memories
  - Up to 1.5 Mbytes of Flash memory
  - 320 Kbytes of SRAM
  - Flexible external static memory controller with up to 16-bit data bus: SRAM, PSRAM, NOR Flash memory
  - Dual mode Quad-SPI interface
- LCD parallel interface, 8080/6800 modes
- Clock, reset and supply management
  - 1.7 to 3.6 V application supply and I/Os
  - POR, PDR, PVD and BOR
  - 4-to-26 MHz crystal oscillator
  - Internal 16 MHz factory-trimmed RC
  - 32 kHz oscillator for RTC with calibration
  - Internal 32 kHz RC with calibration
- Power consumption
  - Run: 112 µA/MHz (peripheral off)
  - Stop (Flash in Stop mode, fast wakeup time): 42 µA Typ.; 80 µA max @25 °C
  - Stop (Flash in Deep power down mode, slow wakeup time): 15 µA Typ.; 46 µA max @25 °C
  - Standby without RTC: 1.1 µA Typ.; 14.7 µA max at @85 °C
  - V_{BAT} supply for RTC: 1 µA @25 °C
- 2x12-bit D/A converters
- 1×12-bit, 2.4 MSPS ADC: up to 16 channels
- 6x digital filters for sigma delta modulator, 12x PDM interfaces, with stereo microphone and sound source localization support
- General-purpose DMA: 16-stream DMA

- Up to 18 timers: up to twelve 16-bit timers, two 32-bit timers up to 100 MHz each with up to four IC/OC/PWM or pulse counter and quadrature (incremental) encoder input, two watchdog timers (independent and window), one SysTick timer, and a low-power timer
- Debug mode
  - Serial wire debug (SWD) & JTAG
  - Cortex®-M4 Embedded Trace Macrocell™
- Up to 114 I/O ports with interrupt capability
  - Up to 109 fast I/Os up to 100 MHz
  - Up to 114 five V-tolerant I/Os
- Up to 24 communication interfaces
  - Up to 4x I^2C interfaces (SMBus/PMBus)
  - Up to 10 UARTS: 4 USARTs / 6 UARTs (2 x 12.5 Mbit/s, 2 x 6.25 Mbit/s), ISO 7816 interface, LIN, IrDA, modem control)
  - Up to 5 SPI/I2Ss (up to 50 Mbit/s, SPI or I2S audio protocol), out of which 2 muxed full-duplex I2S interfaces
  - SDIO interface (SD/MMC/eMMC)
  - Advanced connectivity: USB 2.0 full-speed device/host/OTG controller with PHY
  - 3x CAN (2.0B Active)
  - 1xSAI
- True random number generator
- CRC calculation unit
- 96-bit unique ID
- RTC: subsecond accuracy, hardware calendar
- All packages are ECOPACK®2

Table 1. Device summary

Reference	Part number
STM32F413xH	STM32F413CH STM32F413MH STM32F413RH STM32F413VH STM32F413ZH
STM32F413xG	STM32F413CG STM32F413MG STM32F413RG STM32F413VG STM32F413ZG

September 2017 DocID029162 Rev 6 1/208

This is information on a product in full production. www.st.com

```
Document: DS11581
 └Heading 1: STM32F413xG
 └Paragraph: Arm-Cortex-M4
 └Heading 3: Features
 └List: •
   └Element: Dynamic
     └List: -
       └Element: 1.7 V to
       └Element: -40 °C to
   └Element: Core Arm 32-bit
   └Element: Memories
     └List: -
       └Element: Up to 1.5
       └Element: 320 Kbytes
       └Element: Flexible
       └Element: Dual mode
   └:
   └Element: General purpose
 └Figure: omitted
 └List: •
   └Element: Up to 18 timers
   └Element: Debug mode
     └List: -
       └Element: Serial
       └Element: Cortex-M4
   └:
   └Element: All packages
 └Table 2x3: Device summary
```

Figure 6.1 The debug view of the first page of the STM32F413 datasheet [DS11581] as well as a simplified excerpt of the resulting AST. This document shows a multi-column layout with multi-line nested lists, a figure without caption at the top right and a captioned table at the bottom. Note the insertion of the omitted figure in the *middle* of the list, which is the correct outcome of merging the two column ASTs, despite the strange look.

To speed up conversion and to limit the resulting HTML file sizes, we use the table of content to split up each document into chapters and convert them in parallel. The resulting HTML files are then stored in a folder named after the document. A minimal style sheet renders the HTML file in a web browser in a similar way as the original PDF document for a manual visual comparison. With this step, we conclude the conversion of PDF technical documentation into HTML.

Conversion Challenges

During the implementation, we found several issues in the STMicro documents that we had to find workarounds for that limit the accuracy of the conversion. We expect documents from other vendors to present similar challenges to the implementation:

- Some documents have empty or corrupted metadata, which prevents the detection of vendor and rendering style. We provide a manual mapping from document name to vendor and style as a substitute in step (i).
- None of the documents use the size parameter to scale their characters, instead including a separate font for each size and setting the size parameter to 1. As a workaround, we use the height of the loose bounding box as a proxy for the font size in step (ii).
- Rotated characters have an empty loose bounding box, which breaks our font size workaround. We can use the tight bounding box as a fallback, however, its height depends on the rendered glyph, making this an unreliable solution when merging large super- and subscript glyphs into text lines with smaller glyphs. We therefore added an cache that matches the tight bounding box and unicode value to non-rotated characters and uses their loose bounding box as a substitute in step (ii). Since the cache only contains characters previously seen, this workaround of a workaround can still result in the occasional wrongly merged line.
- Tables borders are rendered as filled rectangles in the black/white style, but as stroked lines in the blue/gray style. We therefore convert the black/white rectangles to lines before passing it to the table partitioning algorithm in step (v).
- In some documents, tables are missing cell borders, which confuses our table partitioning algorithm in step (v), leading to wrongly merged cells. Due to the difficulty in discovering and fixing these issues automatically, we instead opted to manually write patches that are applied to the resulting HTML files automatically by our pipeline, therefore maintaining reproducible results.

At the end of this step, we have a folder of files that only use a subset of the HTML syntax in a predictable way. In particular, headings use normalized sizes (`<h1>`, ..., `<h6>`), lists use the correct indexing scheme (ordered `<ol>` vs. unordered `<ul>`), and the `<table>` tag is used *only* for tables and not for layout, to avoid issues in the detection of symbolic tables (cf. Section 3.1.1). This normalized HTML is now simpler to access than the PDF and its use of syntax is independent of vendor formatting style, which makes accessing the HTML for table processing significantly easier, as we describe next.

6.1.4 Accessing HTML

We use the HTML parser built into Python to receive a stream of opening and closing tags and all text inbetween. Due to the simplified HTML syntax, we can directly convert the stream into a thin abstraction of Figure, Table, and Text Python class.

The Text class, which is further specialized into Heading, List, and TableCell, provides methods for basic text mining via regex matching and substitutions, while the Table class presents three main access methods to the tabular structure to facilitate table processing (cf. Section 2.2):

(i) **Column-row access**: The most basic access simply provides the cell at position (x, y) without regard for header structure and therefore may return the same merged cell for multiple positions. This raw grid access may be necessary if the table lacks a header or if the header structure and cell partitioning complicates proper conversion into a data model.

(ii) **Boxhead-row access**: We generate the labeled domains for the boxhead and then convert each row into a dictionary of domain-cell pairs returned as a list. This access method is useful when the stub cannot be unambiguously identified, for example, if every table column contains cells with duplicate or empty values that would create an ambiguous value-row mapping otherwise.

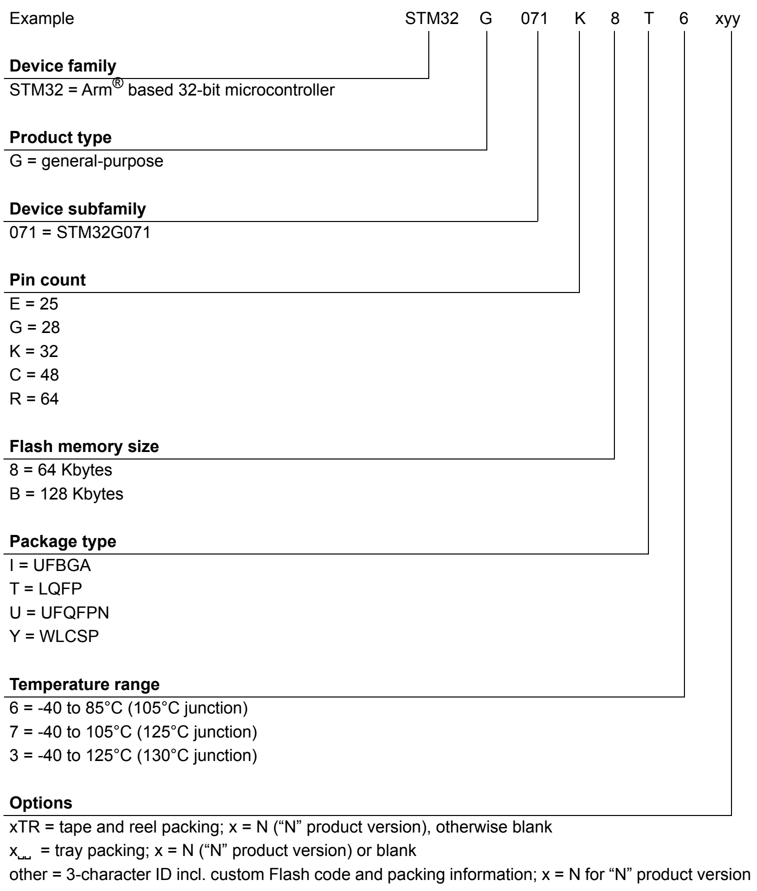
(iii) **Boxhead-stub access**: We dedicate one or multiple columns as the stub and can now generate the partial function δ that provides attribute-cell pairs as intended by Wang (cf. Figure 2.6 and Listing 2.1).

The formatting HTML tags in the text are all preserved until there is enough context to decide how to interpret them via a substitution function of the Text class. In particular, table cells can format lists as comma-, slash-, or newline-separated values, therefore the code guiding the table processing must decide what meaning `<br/>` and other characters have locally, based on the data getting extracted. Additional cleanup or filtering functions on the output are delegated to the calling code.

6.1.5 Converting HTML to OWL

To simplify the data extraction process, we wrap the HTML documents into Python classes based on their type: datasheet or reference manual. We then write several algorithms that run on each document to extract the data using the HTML access methods, clean and filter it, and convert it into OWL format. This approach works especially well if combined with assertion checks to validate the assumption that all documents of the same type have a similar structure. Running this shared code on all documents will then quickly show where and why these assumptions are wrong, allowing fast and simple iteration. In this section, we will detail a number of data processing steps that achieve the design described in Section 5.2.3:

We extract the **device identifiers** from the order information displayed in Figure 6.2. The contained graphics are interpreted as a table with a broken cell structure, therefore we repair the table with a specialized AST pass in the previous pipeline. We then use the column-row access method and HTML filtering methods to create a key-value map by finding lines with a `=` sign for values and without for keys. The concatenated n-fold cartesian product of this mapping then yields all possible identifier combinations, which are filtered through the device summary table seen at the bottom right of Figure 6.1. Since the order information naming schema varies slightly across all datasheets, we use our own modified schema.


Example
STM32 G 071 K 8 T 6 xyy
Device family
STM32 = Arm® based 32-bit microcontroller
Product type
G = general-purpose
Device subfamily
071 = STM32G071
Pin count
E = 25
G = 28
K = 32
C = 48
R = 64
Flash memory size
8 = 64 Kbytes
B = 128 Kbytes
Package type
I = UFBGA
T = LQFP
U = UFQFPN
Y = WLCSP
Temperature range
6 = -40 to 85°C (105°C junction)
7 = -40 to 105°C (125°C junction)
3 = -40 to 125°C (130°C junction)
Options
xTR = tape and reel packing; x = N ("N" product version), otherwise blank
x␣␣ = tray packing; x = N ("N" product version) or blank
other = 3-character ID incl. custom Flash code and packing information; x = N for "N" product version

```
Platform:    STM32
Family:      G0
Name:        71
Pin:         K
Size:        8
Package:     T
Temperature: 6
Variant:     N
```

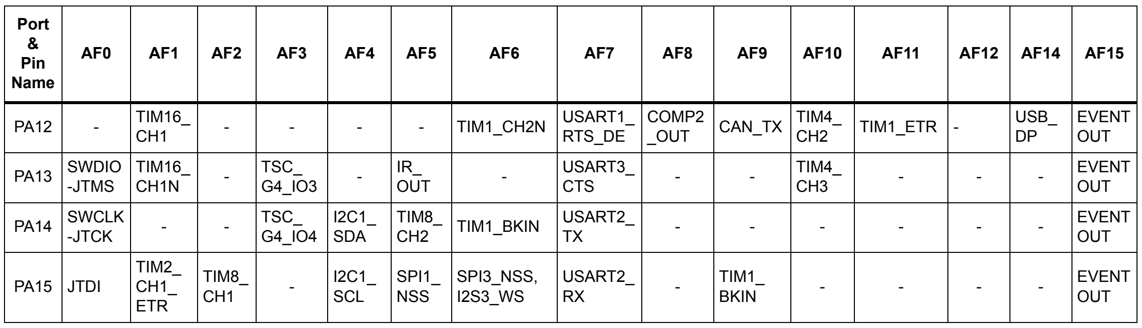
Figure 6.2 The order information on the left details all possible combinations of identifier values [DS12232]. Note the ambiguous `Options` key, with multiple = signs and overlapping meanings, which needs to be treated specially. We normalize the order information into the naming schema on the right.

We use the boxhead-stub access to extract the **interrupt vector table** with only minimal normalization effort (cf. Figure 4.2). However, some reference manuals contain multiple interrupt vector tables with device filters in the captions, which we attach to the data to be specialized for the respective device group in the knowledge graph evolution stage.

We extract the **package and pinout** data with the boxhead-row access method and repair and normalize both the package as well as the pin name. The package names are de-facto standardized and the pin names follow a manually detectable pattern, therefore we can use a regex pattern to detect and fix all mistakes.

The **pin signals** are located in the alternate function table seen in Figure 6.3 and accessed via the boxhead-stub method. However, to limit the table width each cell's signal name list has newlines inserted in it, which can lead to issues when a newline is used as the list delimiter instead of a comma or slash. In these rare cases, we use a manually supplied list of regex substitution patterns to repair wrongly separated signal names.

After normalization, all data is converted into OWL via the owlready2 v0.37 Python library [Lam17, owl17] using the ontology described in Section 6.2. We found that the extraction of tabular data from the STMicro technical documentation worked very well in practice since the table captions and header structures were very similar



Port & Pin Name	AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF14	AF15
PA12	-	TIM16_ CH1	-	-	-	-	TIM1_CH2N	USART1_ RTS_DE	COMP2 _OUT	CAN_TX	TIM4_ CH2	TIM1_ETR	-	USB_ DP	EVENT OUT
PA13	SWDIO -JTMS	TIM16_ CH1N	-	TSC_ G4_IO3	-	IR_ OUT	-	USART3_ CTS	-	-	TIM4_ CH3	-	-	-	EVENT OUT
PA14	SWCLK -JTCK	-	-	TSC_ G4_IO4	I2C1_ SDA	TIM8_ CH2	TIM1_BKIN	USART2_ TX	-	-	-	-	-	-	EVENT OUT
PA15	JTDI	TIM2_ CH1_ ETR	TIM8_ CH1	-	I2C1_ SCL	SPI1_ NSS	SPI3_NSS, I2S3_WS	USART2_ RX	-	TIM1_ BKIN	-	-	-	-	EVENT OUT

Figure 6.3 An excerpt of an alternate function table [DS9118] which shows the forced use of newlines inside signal names to limit the table width. Note the inconsistent use of - vs. , as the list delimiter in the `AF0` and `AF6` columns.

across all documents. However, data stored in textual descriptions does not share these properties, as we will discuss in the step.

6.1.6 Converting HTML to SVD

The MMIO register descriptions are compiled in the reference manual and their extraction is implemented as several algorithms in the document Python wrapper, as described in Section 6.1.5. From the register boundary address table (cf. Figure 2.9) we derive the peripheral instance names, their bus and address ranges, and most importantly, a reference to the chapter that describes their register layout.

In these referenced chapters, we then combine multiple tables to derive the correct peripheral register layout. At times, this approach involves parsing multiple layouts depending on a device filter as exemplified in Figure 6.4, or discovering additional instance-specific boundary address offsets shown in Figure 6.5 that need to be propagated back into the boundary address table. Some register summary tables show a condensed view of register arrays either via formulas in the offset column, or via horizontal or vertical dots in the cells themselves, which we expand into a normalized view. Missing table cell borders are a common source of errors at this stage, particularly between the reset values and the bit name, as visible in the second table in Figure 6.4. We cannot reliably detect and remove the spurious zeros and ones inserted into the bit name, therefore we manually patch the HTML to insert the missing separation.

We then find the correct subsection of the chapter that describes the individual registers in more details (cf. Figure 2.11) by searching the subsection heading for the register name, which worked well for simple peripherals where an exact match was easily found. However, for more complex peripherals, particularly ones with many similar registers, we saw occasional discrepancies between how the register is named in the register summary table and the subsection, usually to de-duplicate the description of multiple registers into one subsection. In these cases, we first tried to guess the de-duplication pattern with a reverse regex matching, where a lower-case `x` in the register name usually denotes a wildcard, without particularly useful results, since subsections did not always contain the `x` either. We then measured the edit distance between the register name and all subsection names to try and find the closest match, however, since register names are usually short and somewhat similar

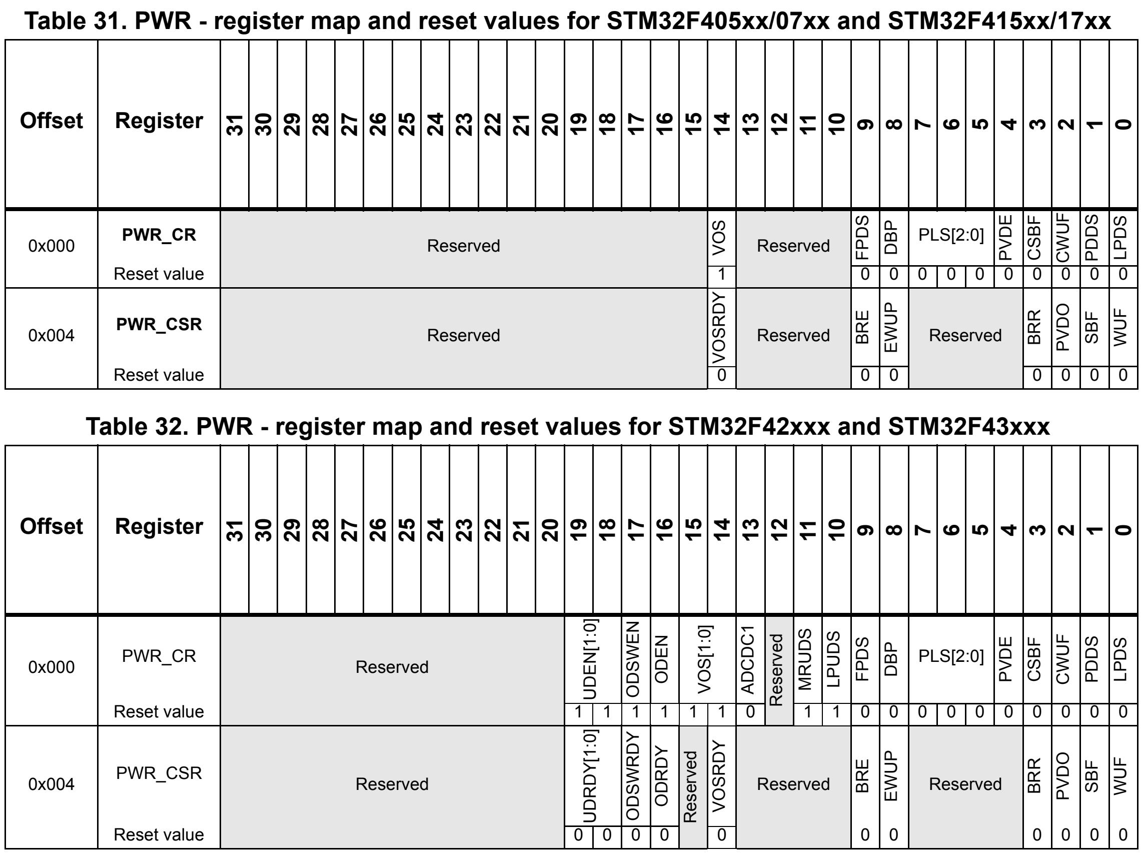


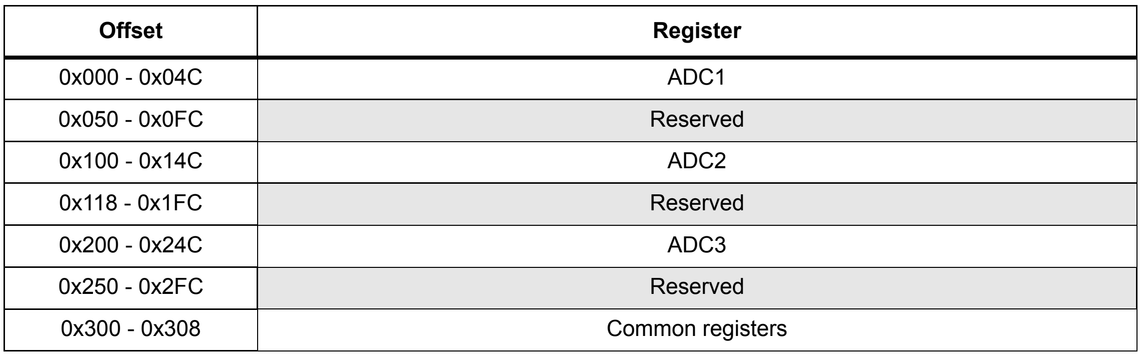
Table 31. PWR - register map and reset values for STM32F405xx/07xx and STM32F415xx/17xx

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x000	PWR_CR	Reserved																	VOS	Reserved				FPDS	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS
	Reset value																		1					0	0	0	0	0	0	0	0	0	0
0x004	PWR_CSR	Reserved																	VOSRDY	Reserved				BRE	EWUP	Reserved				BRR	PVDO	SBF	WUF
	Reset value																		0					0	0					0	0	0	0

Table 32. PWR - register map and reset values for STM32F42xxx and STM32F43xxx

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x000	PWR_CR	Reserved												UDEN[1:0]		ODSWEN	ODEN	VOS[1:0]		ADCDC1	Reserved	MRUDS	LPUDS	FPDS	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS
	Reset value													1	1	1	1	1	1	0		1	1	0	0	0	0	0	0	0	0	0	0
0x004	PWR_CSR	Reserved												UDRDY[1:0]		ODSWRDY	ODRDY	Reserved	VOSRDY	Reserved				BRE	EWUP	Reserved				BRR	PVDO	SBF	WUF
	Reset value													0	0	0	0		0					0	0					0	0	0	0

Figure 6.4 These power (PWR) peripheral register summary tables are similar enough to be listed in the same reference manual chapter, but different enough that they required separate tables with device filters in their captions. Note the missing cell table border between the reset values and the bit names in the bottom right of the second table, which causes the cells to be merged and corrupting the bit name [RM0090].

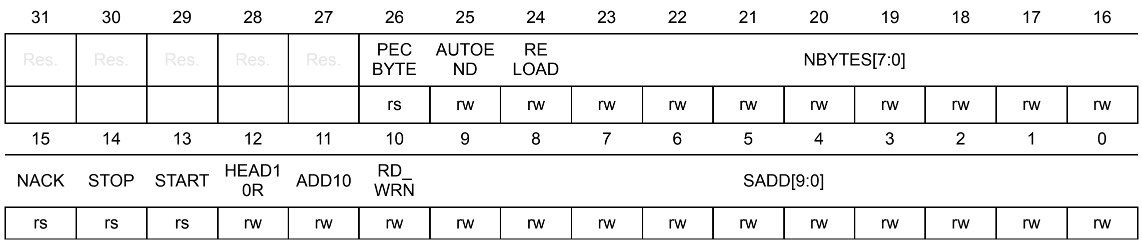


Offset	Register
0x000 - 0x04C	ADC1
0x050 - 0x0FC	Reserved
0x100 - 0x14C	ADC2
0x118 - 0x1FC	Reserved
0x200 - 0x24C	ADC3
0x250 - 0x2FC	Reserved
0x300 - 0x308	Common registers

Figure 6.5 The ADC1, ADC2, ADC3 peripheral and common registers all share the same boundary base address and only in the peripheral chapter are these additional offsets specified. Our parsing code must account for such special cases and keep track of offsets at multiple levels of our tree data structure [RM0033].

already, this technique often resulted in multiple matches of the same distance, which then required manual intervention.

Additionally, once the mapping was established, we found that some registers can have multiple roles and therefore also multiple descriptions within the same sub-section, which confused our algorithm that had to look for certain keywords in the text and recognize tables without captions by structural patterns. Some register bit table description were also missing cell borders as shown in Figure 6.6 or exhibiting other formatting mistakes, which required even more HTML patches.



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	PEC BYTE	AUTOE ND	RE LOAD	NBYTES[7:0]							
					rs	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NACK	STOP	START	HEAD1 0R	ADD10	RD_ WRN	SADD[9:0]									
rs	rs	rs	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 6.6 This I^{2}C register description table with missing cell borders does not survive the conversion into HTML intact and needs to be patched manually [RM0431].

On their own, each problem was easily addressable by adding an edge case for the specific chapter in specific document or by crafting an HTML patch, however, the sheer mass of register descriptions in each reference manual turned this manual correction process into an unreasonable effort, even when the relative error rate was low. The quality improvement we could have achieved by comparing multiple representations of the same register simply did not measure up against the additional complexity of the extraction process. Thus, we abandoned our data merging approach and only use the register layout extracted from the summary table going forward.

We convert all information into a lightweight tree data structure with custom Python classes as nodes for Device $\ni$ Peripheral $\ni$ Register $\ni$ BitField as defined in the SVD format [svd15b] that also contain additional metadata such as device or instance filters. To link the peripheral instances to the register descriptions including offset correction, we run several small tree algorithms on this structure that results in a normalized tree without instance filters. As a final step, we resolve the device filters by manually partitioning the device list into subsets and filtering the tree nodes, resulting in multiple register description trees per reference manual. The tree structures then trivially serialize into a XML format that fits the SVD standard.

Even though we could not completely implement the redundant extraction in our design for this step (cf. Section 5.2.4), the data from register summary table is complete enough to be compared with the STMicro CMSIS header and SVD files for our evaluation in Chapter 7. We also could not extract the textual register and bit field *descriptions* as well as all enumerated bit field values and their corresponding descriptions (cf. Figure 2.11). However, the CMSIS header and SVD files from STMicro also do not contain all of this information either, therefore the impact on our processor as a whole is minimal. For other vendors, these data sources may be structured differently, so that a higher level of detail can be more easily extracted. We continue our implementation with the parsing of the CMSIS headers.

6.1.7 Converting Header Files to SVD

We access the header file data via the CppHeaderParser v2.7.4 Python library [cpp10], which parses the C and CPP definitions in the header and gives us a list of types for peripheral register structure and macro names. To resolve the CPP macros into numeric values, we generate a C++ source file that prints the value of every macro into a JSON file, effectively outsourcing the macro resolution to the compiler toolchain. We then match the macro names to the peripheral and register names of the C type definitions elements to build our memory map. The same tree data structure and Python classes from the previous section are reused to serialize to SVD.

The only errors we encounter in this pipeline for the data provided by STMicro are the occasional syntax errors introduced by spurious characters in the macro definitions, such as missing closing `)` brackets or duplicate `0x` prefixes, which are patchable. Since CPP macros are only lazily evaluated when they are encountered by the CPP in software, catching all syntax issues requires systematically using *all* register bits in a firmware, which is unlikely for typical projects. Our implementation accounts for these issues by manually patching the headers which only requires a reasonable effort, since the syntax error patterns are highly regular.

6.1.8 Converting SVD to OWL

We merge the three SVD file sources from the vendor, CMSIS header files, and technical documentation by traversing each tree starting from the root and copying all non-conflicting nodes to a new tree structure. A node is conflicting if the memory address of the register in byte or bit field in bits does not contain an identical name. Since not all memory locations exist in all three sources, we employ three arbitration strategies to merge conflicts with two or three sources: (a) majority voting only if $^2/_3$ of sources agree on the memory location name, (b) explicit manual patching, and as a fallback (c) statistical trustworthiness of each sources. From our evaluation of these data conflicts in Section 7.4.6, we derive the trustworthiness as CMSIS header files > technical documentation > CMSIS-SVD files. The resulting tree data structure is trivially convertible into OWL using the ontology described in Section 6.2.

6.1.9 Converting Tooling Data to OWL

To access the internal database of the STM32CubeMX tool, we parse the XML files with the lxml Python library [xml05] that provides an XML path language (XPath) query interface. Even though no schema is provided, the data structure is easy to reverse-engineer and does not differ between devices, which makes the use of XPath selectors very easy. We can convert most data directly into OWL without any significant normalization steps or patches, making this dataset the easiest to work with *by far*, even though the amount of accessed data is still quite large.

6.1.10 Evolving OWL

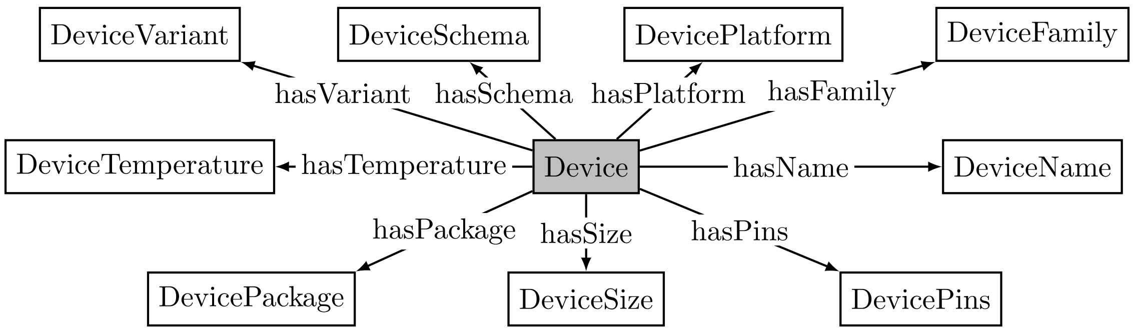
We create a single knowledge graph by comparing of each dataset, as described in the next chapter, and then choosing to use either only the best source, perhaps with manual patches applied, or to merge a specific subset by majority voting if possible. In practice, we prefer to use machine-readable sources over the technical documentation, since accessing them is significantly easier to maintain while also providing a much higher device resolution. Therefore, we use the technical documentation only to generate patches using our evaluation code, specifically to correct the package pinouts and the pin functions of the CubeMX dataset, while the MMIO register description have already been merged in the previous step. The resulting single knowledge graph now contains the device identifiers, interrupt vector table, packages, pins, signals, and register map for thousands of STM32 devices, which we can access using the methods described in the next section.

6.2 Accessing OWL

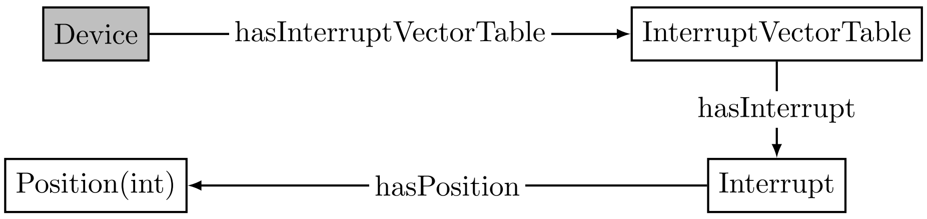
The large knowledge graph generated at the end of our pipeline design contains all data in one namespace per device to prevent ambiguous encoding as discussed in Section 5.2.3. Figure 6.7 visualizes our own custom ontology that encodes the extracted data. Since we do not use any advanced OWL features such as metamodeling, our ontology is much simpler compared to larger ones (cf. Section 2.4). We intend for this ontology to be modified and extended as necessary for future work and therefore should be treated as a starting point, rather than a definitive schema.

To provide a better user experience, we implement a thin Python wrapper that receives a device identifier string and then filters the knowledge graph so that only data for this device is contained, yielding a much smaller and simpler graph. The user can then query the graph directly via the SPARQL query language or use one of the wrapper methods that return a Python data structure. In particular, we provide functions for the device identifiers, interrupt vector table, package-pin, and pin-signal mappings, while the MMIO register map is accessible via our Python tree structure described in Section 6.1.6 that can be serialized into a CMSIS-SVD file for use with existing tools. This step concludes the pipeline implementation of this thesis.

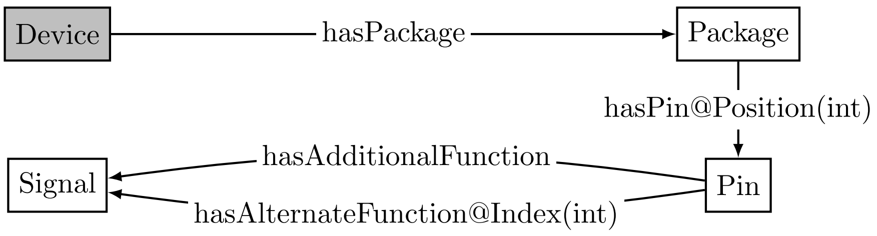
In the implementation, we continued the philosophy of the design by breaking down each pipeline into many small tasks to keep the complexity low and to allow for simple addition of new data sources in future. We can easily adapt and amend our modular implementation should existing data sources change or a different representation of the extracted data be required. We selected libraries specifically so that we could implement the entire pipeline in Python only, which allows us to share and reuse a lot of classes and algorithms natively across multiple pipelines. Apart from the PDF to HTML conversion, which has a high algorithmic complexity, the main difficulty in the remaining pipelines is in the normalization of entity naming and the unambiguous encoding of the results. However, due to our use of multiple data sources we were able to detect and compensate a lot of issues simply by comparison, which is a significant improvement over a manual process. In the next chapter, we evaluate the performance of our pipeline and discuss the quality of the data sources and the resulting knowledge graph in detail.



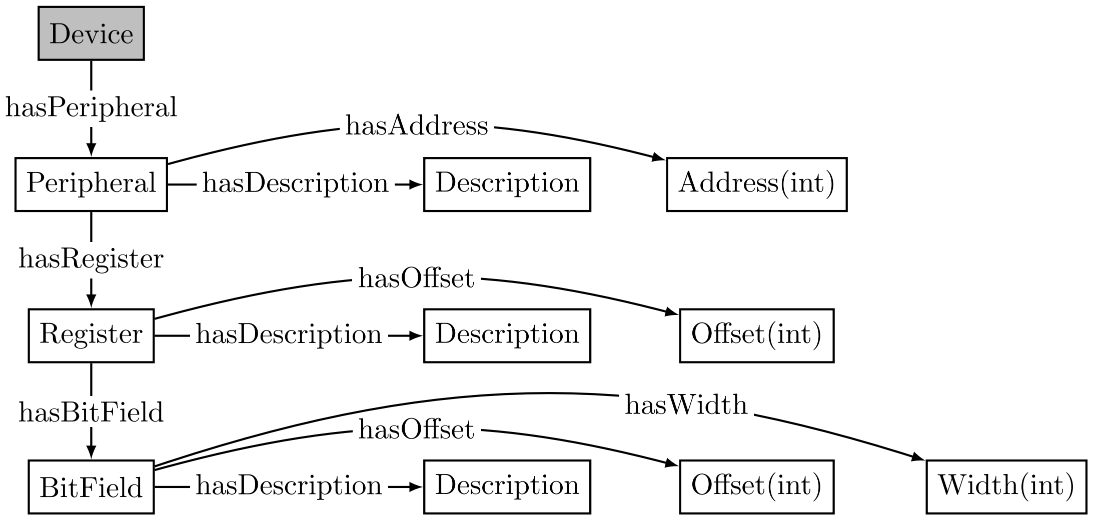
(a) The **device identifier** is modeled as one node with the full device name and the individual keys as per naming schema that we derived in Figure 6.2.



(b) The **interrupt vector table** is a simple one-to-one mapping of interrupt to position.



(c) The **package, pinout, and pin functions** are encoded using relations with integer attributes to unambiguously map the pin to the package position and an alternate pin function to its index. Some packages duplicate pins at multiple positions, usually for the power supply. Similarly, an alternate function can be connected to a pin at more than one index.



(d) The **register descriptions** are strictly hierarchical and are inspired by the SVD standard, so that the conversion between the formats is simple to implement.

Figure 6.7 Our simple ontology uses the Device (in gray) as the primary node from which all other nodes can be reached. All entities for one device are placed into their own namespace.

7

Evaluation

In this chapter, we evaluate the implementation effort of our pipeline and establish the quality of the extracted datasets by comparing them against each other. To this end, in Section 7.1, we describe how the pipelines are executed and which input data is converted into intermediary artifacts and knowledge graphs, before estimating the pipeline performance in Section 7.2 and the implementation effort of each conversion step in Section 7.3. Subsequently, in Section 7.4, we compare the data we extracted from the technical documentation with their machine-readable counterpart to derive the quality of our implementation and the accuracy of the documentation. We conclude this chapter with a discussion of our findings in Section 7.5.

7.1 Evaluation Setup

In this section, we introduce the amount and sizes of the input sources in Section 7.1.1 and the generated artifacts in Section 7.1.2. The pipeline and all support and evaluation code is written in Python and therefore require no special hardware or software setup. We execute all code and measurements on a 2015 MacBook Pro with a quad-core Intel Core i7 processor running at 2.2 GHz and 16 GB of RAM.

7.1.1 Input Sources

The input sources can be downloaded from the internet as described in Section 6.1.1. For each input source, we list the number of files and their total size on disk to understand the storage requirement. However, the data is compressed during network transfer, so we also add the zip archive size to understand how much data needs to be downloaded, while the compression ratios hint at duplication in the data sources:

(i) The **technical documentation** has been scraped from the STMicro homepage by an automated task every day since 16th February 2022, resulting in a total of 1056 PDFs in the archive, taking up about 5.1 GB on disk and 1.8 GB compressed (ratio 2.8).

(ii) The **STM32CubeMX database** expands into 1316 individual XML files with a size of 340 MB on disk and 64 MB compressed (ratio 5.3).

(iii) The **CMSIS header files** for STM32 consist of 230 individual files with a size of 290 MB on disk and 27 MB compressed (ratio 10.7).

(iv) The **CMSIS-SVD files** for STM32 contain 99 files with a size of 200 MB on disk and 10 MB compressed (ratio 20).

The compression ratios are lower for sources describing the hardware more generically (PDF and STM32CubeMX) and higher for sources that only encode a specific type of information (header and SVD files), which is explained by the massive duplication of information in the register descriptions. While the technical documentation is about 18 times larger than all compressed machine-readable sources combined, we also archive *all* PDF documents from the STMicro homepage including previous revisions of the same document, even if our current implementation only processes tables in datasheets and reference manuals for STM32. However, we also do not access all machine-readable information either, therefore these numbers should be considered as only the maximum size of the input sources.

7.1.2 Conversion Artifacts

The individual pipelines create several intermediate formats until they are converted into OWL format. Our implementation converts a subset of the sources available in the archive, therefore we also provide the number and size of the files for comparison.

(i) The **PDF to HTML** pipeline converts the latest revision of 576 PDFs with 125463 pages in total. The archive includes 52 reference manuals (79379 pages, 63% of total), 276 datasheets (37805 pages, 30%), 154 errata sheets (4429 pages, 3%), and 94 user manuals (3850 pages, 3%). About 40% of the STMicro datasheets in our archive are not about STM32 microcontrollers, but also include sensors, memory, and communication devices. The HTML files result in a combined size of 580 MB on disk and 185 MB compressed.

(ii) The **HTML to OWL** conversion also only converts the latest revision of the technical documents, resulting in one knowledge graph for each 162 datasheets and 44 reference manuals. They take up 83 MB on disk and about 13 MB compressed.

(iii) The **STM32CubeMX to OWL** pipeline creates one knowledge graph for each of the 2899 STM32 devices, resulting in 104 MB on disk and 12 MB compressed.

(iv) The **CMSIS header files** are converted into 183 SVD files, while the **reference manuals** create 56 SVD files. Combined, they take up 390 MB on disk and 43 MB compressed.

(v) The **evolved knowledge graph** has a size of 613 MB on disk, which reduces to 92 MB compressed.

In each pipeline step, we filter out only the required information from the input sources, which reduces the size of the final knowledge graph considerably, with the MMIO register descriptions taking up about 2/3 of the space. In our design, a developer only needs to download the final knowledge graph, which considerably reduces the download size requirements. While the converted HTML retains the original PDF copyright prohibiting republication, the final knowledge graph is a derivative work and can therefore be distributed freely. With these input sources and output artifacts in mind, we describe the performance of the conversion in the next section.

7.2 Pipeline Performance

We measure the performance by the average time it takes for each pipeline to convert all input data into artifacts on the same computer over 10 runs. The PDF to HTML conversion takes a little over two hours to complete (about 60 ms per page), with the longest time spend on converting the reference manuals. The remaining tasks run *much* shorter, only between 3–7 minutes each, for a total of 20 minutes. This significantly shorter runtime shows the benefit of converting the PDF to an intermediary HTML format first, since the algorithmic complexity and the amount of data to process would significantly slow down all pipelines depending on the PDF content. However, each pipeline can also operate on just a small chunk of the input, for example, just one PDF page or a single HTML chapter, which reduces the development cycle even further to just a few seconds.

To utilize all processing power, we perform the conversion of each input file in parallel, which saturates the processor fully, while using about 6 GB of RAM or about 40% of the available memory on our machine. The limited memory consumption points to a compute-bound behavior, meaning that the pipeline can be sped up by spreading the tasks over more and faster processors. We have tested this hypothesis by splitting the PDF to HTML into 20 jobs running in parallel on the free tier of GitHub Actions [git18], reducing the total conversion time to 25–30 minutes. However, the costly PDF to HTML pipeline only needs on to be run on all documents when the implementation is tuned to improve the accuracy of the conversion. In that case, a developer would first convert a few PDF pages to test their changes in seconds, then advance to convert the whole document in minutes, before running the pipeline on all documents in hours. While the long conversion runtime is not ideal, this gradual progression at least makes the impact more manageable in practice.

The PDF to HTML pipeline also needs to convert new documents whenever they are published by STMicro. Over a period of 142 days, we counted 212 new documents in our archive with a total of 7942 new pages, of which only 56 were datasheets and reference manuals with a total of 3304 pages. At about 60 ms conversion time per page, new STMicro publications could add up to 20 minutes of total runtime *each year*, of which 8.5 minutes are due to datasheets and reference manuals. Since the subsequent pipelines run much faster and only operate on the latest revision of each document, their runtime increase only marginally. However, if there are content changes in these new revisions, the subsequent pipeline implementations may need to be adapted at an unknown, but likely reasonable, cost. In addition to the runtime performance, we discuss the cost of processor development in the next section.

7.3 Implementation Effort

Our problem statement defined several challenges in Section 4.2, including *C5: Maintainability*, which stated that accessing the technical documentation must be implementable with "reasonable effort" compared to machine-readable sources. We approximate this metric by two measurements: lines of code and time spent on each implementation.

Since all code in our processor and support code is written in Python 3 with a similar coding style, we measure the effort of implementation by proxy of the lines of code (LoC) as measured with the pygount v1.4 library [py16] and summarized in Table 7.1. The largest task is the PDF to HTML conversion, which additionally applies 29 manually created patches to fix wrong PDF formatting in 2409 lines of HTML. Although not part of the pipeline, we wrote an additional 963 LoC only for comparing data sources against each each other.

While the technical documentation pipelines consists of only about twice the LoC as the machine-readable pipelines, they took about 3.5 times as long to implement. In total, we spent about four months implementing the processor, which was mostly spent on the PDF to HTML pipeline at about 9 weeks due to its algorithmic complexity and the amount of fine-tuning and patching required (cf. Section 6.1.3). Meanwhile, parsing the configuration tool database (cf. Section 6.1.9) only took about one week to implement, so the implementation time is very unevenly distributed.

Combining both LoC and time spent implementing, we estimate that accessing the technical documentation took us about three times more effort overall compared to accessing machine-readable sources. However, since most of the pipelines can be reused when adding new sources or vendors, the effort is likely less for future work. We therefore claim that accessing technical documentation can be implemented with reasonable effort. After reviewing the performance and implementation effort of our design, we now assess the extracted data quality.

Task or Pipeline	**Lines of Code**
Downloading technical documentation	105
PDF to HTML conversion	+ 3104
HTML to OWL conversion	+ 1024
Accessing technical documentation	= 4530
Downloading machine-readable sources	124
STM32CubeMX to OWL conversion	+ 789
Accessing CMSIS-SVD files	+ 817
CMSIS header to SVD conversion	+ 481
Accessing machine-readable sources	= 2211

Table 7.1 This code size comparison of each task in our pipeline shows that our implementation requires about twice the LoC for accessing the technical documentation than the machine-readable sources. However, the technical documentation pipelines required about 3.5 as much time to implement.

7.4 Quality of Extracted Data

So far, we have demonstrated that we can access tabular data from the technical documentation with reasonable effort. However, we also need to investigate if the extracted data has a level of quality suitable to fulfill the challenges outlined in our problem statement (cf. Section 4.2). Since the technical documentation is written by humans for humans, we expect the content to contain spelling mistakes, copy/paste errors, and ambiguous naming. To understand the scope of these issues, we evaluate the quality of the data extracted from the technical documentation in comparison to existing machine-readable sources.

In particular, we want to compare the data derived from the technical documentation with data from machine-readable sources. We have to resort to a relative comparison, since we do not have access to ground truth data, such as a detailed system engineering or manufacturing data, and we cannot reverse-engineer such data from the hardware for thousands of devices. Furthermore, as we will describe in detail in this section, different data sources can use different names to refer to the same entities and relations, and aggregate data into unequally large groups of devices, so that a naive one-to-one comparison is not always possible. As all our input sources are published by the vendor themselves, we also cannot claim some data to be more authoritative than others to arbitrate grouping and naming conflicts to find a canonical data representation.

As a consequence, we can only evaluate the completeness of data if we can find an individual device mapping from one source to another. Otherwise we can only check for conflicts in the union of both sources, thereby loosing the ability to distinguish between correctly and incorrectly added or removed data points. For example, while we can assign each device identifier a unique package and pinout for both the datasheets and the STM32CubeMX database, the number of MMIO register descriptions we can extract from the reference manuals and CMSIS header files differs by a factor of over three (56 vs. 183).

We additionally need to normalize the names of entities into a common representation to make a comparison technically possible. In particular, we cannot use a statistical metric such as editing distance between two strings, as especially the technical documentation uses naming schemas with differing prefix or abbreviation patterns depending on the document and data encoded. We already discovered this issue when trying to match register names with their textual descriptions in Section 6.1.6. We therefore first run a set of manually assembled regex substitutions on all sources to rename diverging data points and then require an exact string match to establish equality. To keep the comparison fair, the substitutions do not take the context of the data point into consideration, thus only transform the data representation and not its content. For example, the names `ADC_CH1` and `ADC_Channel1` refer to the same entity and we can convert one to the other naming scheme without loosing or distorting information.

Finally, we only compare datasets that are encoded in the same way in the PDF documents and were extracted by the code paths. This restriction prevents introducing any systemic issues in our comparisons that could result from using more beneficial formatting to encode data, for example, using a less ambiguous header structure in tables. In practice, we only compare device data that uses only the same hardware implementation and is therefore directly comparable.

We start our evaluation with the PDF to HTML conversion in Section 7.4.1, which lays the foundation for the following dataset comparisons: device identifiers in Section 7.4.2, interrupt vector tables in Section 7.4.3, packages and pinouts in Section 7.4.4, pins and signals in Section 7.4.5, and register descriptions in Section 7.4.6.

7.4.1 PDF to HTML Conversion

We fine-tuned the accuracy of this pipeline through iterative manual comparison between the PDF and resulting HTML to discover formatting issues and then adapt the code to address them. To avoid regressions, we assembled a minimal set of PDF pages with challenging formatting, whose generated HTML is then checked against a known-good version. In addition, the pipeline output is completely reproducible, so we can inspect what effects our code changes introduce by comparing the result against the archived HTML. We stopped tuning our implementation when the formatting of the resulting HTML was good enough to not cause any further issues in the table processing code in the following pipelines. Nevertheless, we could not fix formatting issues present in the PDF itself with a better algorithm as that would have required also understanding the content of the document. In these cases, we patched the resulting HTML manually, almost exclusively adding missing table cell borders that caused unrelated cells to be merged.

Our implementation also has several known limitations that we intentionally did not address to limit coding effort, since the subsequent pipelines do not access this content anyways. We discuss several use cases that may require these limitations to be fixed in Section 8.2.

Figures, inter-document references, and links are converted into placeholders, with figures showing only the caption and a rectangle containing the text `(omitted)` to help debug issues with figure detection, while references and links are rendered only as underlined text without functionality. The lack of figures and links makes exploring the document content less convenient for *discovering* new data to extract, however, *extracting* data from figures is not trivial [CD16] and outside the scope of this thesis.

Figures inside table cells are not detected correctly and cause the cell layout to be corrupted resulting in misshapen table structures. We found that such tables are usually used to convey concepts rather than data, therefore our subsequent pipelines do not access them anyways.

Mathematical formulas are often rendered using a mix of graphics and glyphs. This pattern is not correctly detected and the pipeline may convert it into an omitted figure, a small table, or text with mixed sub- and superscript. However, these formatting artifacts do not corrupt their immediate surroundings and are therefore easy to identify visually when reading a document.

Code blocks are often rendered into their own frame box, which is then detected as a figure and omitted completely. If the code block is detected as text, the indentation is not preserved, making the result hard to read.

Background graphics are either detected as figures and omitted, which then also removed the text placed on top of them, or detected as extra cell borders when a table is rendered on top. We only encountered this problem in the fifth revision of the STM32G441 datasheet [DS12960], which renders a rotated gray `DRAFT` text in the page background using vector graphics. Since the document is clearly not finalized, we simply ignore this issue until a non-draft revision is released.

While the major source of quality issues is the use of vector graphics in contexts we do not access in later pipelines anyways, if we wanted to use the HTML as a replacement format for PDF, these formatting errors are detrimental to the understanding of the document content. However, for our purposes of extracting structured tabular data, these limitations come with a good trade-off in implementation effort and do not have an impact on data quality, as we describe in the next sections.

7.4.2 Device Identifiers

Before we can compare any datasets, we first need to understand which devices it belongs to. This mapping needs to be non-overlapping so that we can have an unambiguous relation from device identifier to dataset for comparison.

The STM32CubeMX database includes a single XML file with a list of 2974 STM32 devices, which we assume is the most accurate inventory of devices STMicro designed. However, we discovered several devices for which there exists no datasheet, reference manual, or mention on the STMicro homepage: STM32G471, STM32L485, STM32L041C4, and STM32G071x6. The STM32G441 only has a draft datasheet without a corresponding reference manual. We believe these devices are yet to be announced as new products, thus we removed these devices from the list, resulting in 2899 devices.

For each datasheet, we produce the list of identifiers via n-fold cartesian product (cf. Section 6.1.5), which generates a total of 12934 STM32 identifiers, over 4 times the STM32CubeMX amount. These generated identifiers map onto each datasheet and reference manual without any overlaps or gaps.

However, the STM32CubeMX list of device identifiers is not a true subset of the datasheet list of identifiers, with 203 missing devices. We therefore make use of the naming schema we introduced in Figure 6.2 to investigate if there is a pattern to the missing devices. We start with only the family and name keys and incrementally add more keys with the results listed in Table 7.2.

We notice that the identifier set matches well until the package key is added, when the datasheet identifier list explodes with 2536 additional devices. Our implementation does not respect that the pin key, describing the number of pins on a device, interlocks with the package key, and therefore not all combinations can be valid. Since the STM32CubeMX identifier list is still a true subset at this point, this oversupply is not a relevant issue apart from the large number of non-existent devices.

However, when we add the temperature key, the missing devices begin to manifest with the full STM32CubeMX identifier list containing 203 devices that cannot be mapped to a datasheet. The temperature key denotes the maximum junction temperature for safe device operation, which is a physical property of the manufacturing

Naming Schema Key Combinations	CubeMX		Datasheet	
Family+Name	159		159	
Family+Name+Pin	618		623	+5
Family+Name+Pin+Size	1170		1180	+10
Family+Name+Pin+Size+Package	1681		4217	+2536
Family+Name+Pin+Size+Package+Variant	1864		6004	+4140
Family+Name+Pin+Size+Package+Temperature	2651	−178	8936	+6463
Family+Name+Pin+Size+Package+Temperature+Variant	2899	−203	12934	+10238

Table 7.2 Comparing sets of identifiers with an increasing amount of naming schema keys reveals an implicit interlocking of the pin and package keys that is not considered by our pipeline and an issue with missing temperature values in the datasheets.

process [Kul17]. We confirmed manually that the relevant datasheets are simply missing these temperature combinations and are not lost in our pipeline implementation. Since we could not find any mention of junction temperature in the text or tables we access in other pipelines, we do not patch the datasheets and instead proceed to only use a maximum of $2899 - 203 = 2696$ (93%) devices for comparing dataset from datasheets. The reference manuals do not have this limitation as they apply for a much broader range of devices, thus we can use all 2899 devices.

7.4.3 Interrupt Vector Table

The interrupt vector table is extracted from the reference manual (cf. Figure 4.2) and compared with the vector tables of the CMSIS header files. We can only check for naming conflicts at the same position, but not for completeness, since the reference manual contains the maximum population of the vector table, but the header files remove the vectors for peripherals not available on the device. We also ignore datasheets for multi-core devices that use a different table layout, leaving 2626 device to compare.

Our pipeline was able to find and assign the correct table for all devices and after normalizing vector names, we were able to match 175158 out of 177270 (98.8%) compared vector positions. Of the mismatched positions, 1107 (0.6%) were missing completely, which may still be correct due to the identifier grouping differences, while 1005 (0.6%) had incompatible names. The mismatched names were almost always closely related, but differed only on the peripheral instance number, for example, `TIM7` vs. `TIM2` and `TIM1_CC` vs. `TIM_CC`. When we looked up these failures in the reference manuals, we could confirm that these discrepancies were all part of the document and not introduced by our pipeline. While we could use this information to patch the HTML, the effort required to correct over one thousand table entries across all reference manuals is unreasonable, especially when we already have an alternative data source in the CMSIS header files. We continue our evaluation with another simple data structure that maps a position to a name.

7.4.4 Package and Pinout

The package and pinout are extracted from a shared table in the datasheet (cf. Figure 2.4) which contains a package name, the pin positions and its associated pin name. We were able to create a one-to-one mapping from device identifier to package for both the datasheet and STM32CubeMX database, therefore we also check for data completeness in this comparison.

We compared 2696 devices with a total of 237399 pins from the STM32CubeMX database against the data derived from the datasheets by first finding the correct package, which was successful for 2690 devices, and then matching both the name and the position of the pin on the package. During manual inspection, we noticed a pattern of missing thermal and exposed pads in the datasheet, which were only mentioned in the table footnotes or pinout figures, but not explicitly listed in the pinout tables. Since these pads are always connected to GND, we need no extra context to add these pads for the `UFQFPN32`, `UFQFPN48`, `UFBGA169`, `UFBGA176`, and `TFBGA240` packages.

With these fixes, we matched 237105 (99.88%) matched pin positions and names, with 2635 devices (97.7%) matching the whole package correctly. We are left with $2690-2635 = 55$ devices that share 294 issues where pins were either missing, added, or unequal in their name and/or position. We investigated each issue manually and classified them into 13 mistakes in the datasheet as listed in Table 7.3 and 9 issues with the STM32CubeMX database as detailed in Table 7.4. The largest source of errors is the confusion of packages in devices with an optional switched mode power supply (SMPS) feature, which is identified by the variant key and only differs slightly, followed by missing entries or typos in datasheet tables, with plain wrong data being very rare. In no cases did we find bugs in our pipeline implementation or evaluation code, with the packages for $2696 - 2690 = 6$ devices simply missing from the datasheet.

Both this dataset as well as the interrupt vector table are one-dimensional data structures mapping one position to a name, which makes the extraction and comparison easy to implement. We evaluate a more complex two-dimensional data structure next, where two indexes map to one entry.

7.4.5 Pin Functions

In this evaluation step, we compare the pin function tables in the datasheets (cf. Figure 6.3) with the STM32CubeMX database. We must exclude the STM32F1 device family due to a different hardware implementation of pin functions, leaving us with 2558 devices with a total of 1039429 pin functions. However, since the pin function tables in the datasheet contain the union of functions for all devices described, we can only check for conflicts in a signal name against the alternate and additional function index. After the normalization step, we are find 997191 (95.94%) matching pin-function pairs, with the remaining $1039429 - 997191 = 42238$ pairs either missing, added, or wrong in one or the other source.

Figure 7.1 plots the number of pin function conflicts per device as a histogram, showing most devices having only a few issues, with the distribution exponentially

Affected Devices	−Missing, +Added, =Renamed Positions	Cause of Issue in Datasheet Table and Figures
STM32G431CBYx	−A4, +A43	Typo for position A4, figure is correct.
STM32L412TBY6P	−E5	Typo for position F5, figure is correct.
STM32H745XxHx, STM32H747XxHx, STM32H755XxHx, STM32H757XxHx	+VDD	VDD name is placed into the position column instead of name column.
STM32H742XxHx	−F2	Missing pin position, figure shows F2=VSS.
STM32H747ZIY6	−A13	Missing pin position, figure shows A13=NC.
STM32H750XBH6	−G2, −F1	Missing pin position, figure shows G2=NC, F1=NC.
STM32H757ZIY6	−A13	Missing pin position, figure shows A13=NC.
STM32L071VxIx, STM32L072VxIx	−E3	Missing pin position, figure shows E3=VSS.
STM32L151QCH6, STM32L152QCH6, STM32L162QCH6	−K1	Missing pin position, figure shows K1=OPAMP3_VINM.
STM32L053CxUx, STM32L063CxUx	Pins 2...7 renamed	Position cells are shifted down by 1 row.
STM32L062C8U6	−46	Missing position row, figure shows 46=PB9.
STM32L412CBxxP	22=(PB11, VDD), 45=(PB8, PB9), 46=(PB9, VDD)	Missing both package column and figures for the SMPS package variant. Our pipeline instead uses the closest non-variant match.
STM32L562QEI6P	B4=(PG15, VDD12), M11=(PG11, VDD12)	Missing both package column and figures for the SMPS package variant. Our pipeline instead uses the closest non-variant match.

Table 7.3 These pin position and name mismatches are all attributed to mistakes in the datasheet: missing entries, typos in cells, and formatting issues. Our pipeline could not find two packages for devices with a switched mode power supply (SMPS) feature, and instead used the closest non-variant match.

decreasing. However, there are several outliers between 80–120 conflicts per device, which are exclusively STM32H7, STM32L1, and STM32L4 devices. To investigate this distribution further, we calculate the absolute and relative conflict rate per device family as listed in Table 7.5. The most common absolute conflicts are indeed in these three families, however, the STM32L1 family has a relative conflict rate of over one fifth, while all other families have between 1.2% and 8.2%, pointing to a systemic issue in the data.

Since this amount of conflicts was too much to manually inspect, we instead investigated the most prominent patterns that emerged, in particular, the conflict of an analog or special hardware function ("additional function") with a digital signal multiplexer ("alternate function"). We verified that the hardware implementation of the GPIO module is identical across all compared devices, therefore these conflicts are easy to detect, since an analog signal cannot be routed through the digital multiplexer and neither the other way around. When we applied these assumptions to our data, we found that particularly the STM32L1 family suffers from systemic

Affected Devices	–Missing, +Added, =Renamed Positions	Cause of Issue in STM32CubeMX Database
STM32F038E6Y6	E2=(PB1, NPOR)	Wrong entry, datasheet table and figure both show E2=NPOR.
STM32F048TxY6	D2=(NPOR, PB1), F2=(PB1, NPOR)	Wrong entry, datasheet table and figure both show D2=PB1 and F2=NPOR.
STM32L452REYxP	29 renamed pins	Uses non-variant instead of SMPS package.
STM32L476QxIxP, STM32L496QxIxS, STM32L4P5QxIxS, STM32L4R5QxIxP	C6=(PG14, VDD12), L11=(PB11, VDD12)	Uses non-variant instead of SMPS package.
STM32L476QxIxP, STM32L496QxIxS, STM32L4P5QxIxS, STM32L4R5QxIxP	C6=(PG14, VDD12), L11=(PB11, VDD12)	Uses non-variant instead of SMPS package.
STM32L4R5AII6P	C6=(PG15, VDD12), M10=(PH11, VDD12)	Uses non-variant instead of SMPS package.
STM32L552QEI6	B4=(V15SMPS, PG15), M10=(VLXSMPS, PG13), M11=(V15SMPS, PG11), M9=(VDDSMPS, PG14)	Uses SMPS instead of non-variant package.
STM32L552VET6	Pins 20. . . 51, 98, 99 renamed	Uses SMPS instead of non-variant package.
STM32L552ZETx	Pins 31. . . 73, 126. . . 143 renamed	Uses SMPS instead of non-variant package.

Table 7.4 The STM32CubeMX database is often using the wrong package for devices with an optional switched mode power supply (SMPS) feature as indicated by the variant key in the identifier. As shown in the first two rows, only three other pins were simply wrong, with the rest of the data matching the datasheet.

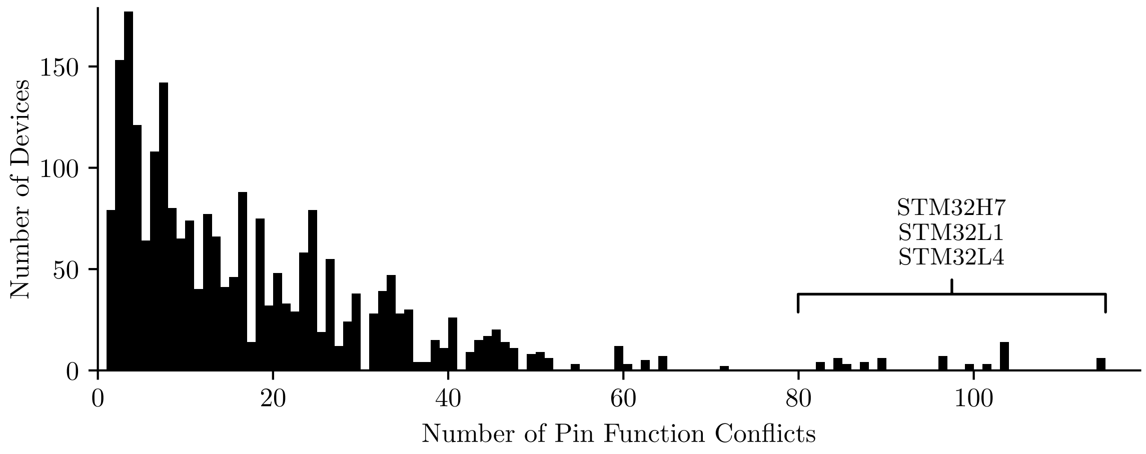


Figure 7.1 This histogram shows the number of pin function conflicts on the x-axis per device on the y-axis. Most devices only have a few conflicts, with their number per device dropping exponentially until they pick up again at around 80 conflicts. The 80–120 range contains only STM32H7, STM32L1, and STM32L4 devices pointing to systemic issues in their pin function data.

Family	Number of Functions	Number of Conflicts	Absolute Rate of Conflicts	Relative Rate of Conflicts
STM32H7	215442	10574	25.0%	4.9%
STM32L1	30128	6539	15.5%	**21.7%**
STM32L4	174070	4224	10.0%	2.4%
STM32G0	66525	4052	9.6%	6.1%
STM32F4	130518	3796	9.0%	2.9%
STM32F7	114590	3130	7.4%	2.7%
STM32F0	29487	2428	5.7%	8.2%
STM32G4	85415	1539	3.6%	1.8%
STM32F2	22151	1411	3.3%	6.4%
STM32L0	61202	1220	2.9%	2.0%
STM32F3	43386	1024	2.4%	2.4%
STM32L5	19778	858	2.0%	4.3%
STM32WB	8487	677	1.6%	8.0%
STM32WL	5888	384	0.9%	6.5%
STM32U5	32362	382	0.9%	1.2%

Table 7.5 The conflict rates of pin functions sorted by absolute rate. The STM32H7 devices have the largest amount of pin functions, and therefore also the largest absolute share of conflicts, while their relative rate of about 5% is comparable to other families. Meanwhile, over one fifth of the pin functions of the STM32L1 family conflict, pointing to a systemic issue in the data.

conflicts of only a few analog functions, which were mapped wrong across the whole device range, explaining why the family is such an outlier. A selection of prominent patterns including their explanations is presented in Table 7.6. Even more patterns emerge from this conflict data when we continue to check basic assumptions about them, however, the validation procedure is the same as before.

Even though we were not able to check for completeness of the pin function data due to differing device set sizes, just analyzing conflicts between the datasets is enough to point out areas for further manual investigation. By validating basic assumptions of the data, such as the difference between an analog and digital signal multiplexer, we can cluster these conflicts into a patterns that point to the faulty data very quickly. This step concludes the comparisons with the STM32CubeMX database, in the next section we will evaluate two new data sources.

7.4.6 Register Descriptions

To evaluate the register description, we compare three sources: the CMSIS-SVD files, the CMSIS header files, and the reference manuals. For a fair comparison, we must exclude the STM32L5 and STM32U5 families built around the ARMv8-M architecture, which aliases the register map into two partitions [arm16] and that changes both the documentation and header structure. In addition, despite our best efforts, we could not find a CMSIS-SVD file for every device, particularly not for devices released more recently such as the STM32WB and STM32WL families. Therefore, we are left with 2583 (89.1%) devices for which all three sources exist.

Occurances	Functions	Conflict	Description and Cause of Issues
654 23	`COMPx_INP` `COMPx_INM`	A≠14	Comparator input is analog, STM32CubeMX database is wrong for the entire STM32L15x family.
446	`TIMx_ETR`	1≠A	Digital signal where the STM32CubeMX database is wrong for the entire STM32L1 family.
140 140	`USB_DM` `USB_DP`	A≠0	Analog signals, STM32CubeMX database is wrong for the entire STM32L1 family.
514 319	`SYS_WKUP` `SYS_TAMP`	A≠0	Special digital input signal hardwired into PA0 pin to wake up from deep sleep and tamper detection. STM32CubeMX database is wrong for the entire STM32L1 family.
497 497	`RCC_OSC_IN` `RCC_OSC_OUT`	A≠0	Special analog signal that must be configured by reset and clock control peripheral. This is a datasheet issue on some STM32F2/F4 devices and a STM32CubeMX database issue on STM32L1 family.
296 105 93 69	`UCPDx_FRSTX`	6≠A 4≠A 0≠A 1≠A	Digital signal that is wrong in the STM32CubeMX database for the entire STM32G0 family.

Table 7.6 A selection of the most interesting and common patterns of pin function conflicts. All of these should have been easy to catch even without a comparison with other sources, by simply validating how digital vs. analog signals are multiplexed.

Since these sources have limited device resolution (cf. Section 7.1.2), we perform 183 unique three-way comparisons by checking for name conflicts, but not completeness.

Each register description is a tree structure made of peripherals ∋ registers ∋ bit fields (cf. Section 2.3.1), therefore we perform the same conflict check at each level. First, we convert each level into a flat memory map by expanding the peripheral, register, or bit field from the tuple [address, width] into a range of bytes or bits with the corresponding name attached. This flattening step is important to remove the peripheral hierarchy, as all three sources have slight differences in how they group registers into peripherals. For example, the reference manual specifies one contiguous register file for the DMA peripheral, while the CMSIS header files separates each DMA stream into its own mini-peripheral (cf. Figure 2.5). The total size of each memory map and the amount each source contributes and overlaps is listed in Table 7.7. On average, all three sources contribute about the same amount of memory locations to the map for peripherals and registers, however, the CMSIS header files contain fewer bit fields than the other two sources. This pattern also reflects in the percentage of overlapping memory locations with two or three sources, which is the lowest for bit fields. If we were to merge these three sources naively by selecting only overlapping memory locations with matching names, the resulting memory map would be quite incomplete.

Hierarchy Level	Total Memory Map Size	Reference Manual	CMSIS Header	CMSIS SVD	Overlap	Matching
Peripherals	55 704 B	77.8%	78.5%	86.5%	77.0%	62.7%
Registers	1 189 630 B	79.3%	73.8%	71.1%	74.9%	47.6%
Bit Fields	5 598 867 bit	74.8%	49.2%	72.6%	61.1%	30.7%

Table 7.7 The flat memory map locations are contributed similarly by all three sources, except for the bit fields, where the CMSIS header files are missing a significant amount of data. We call memory locations with more than one source overlapping. If the names of overlapping locations are identical after normalization, we call them matching. The overlap of memory locations gets progressively worse per hierarchy level, with matching locations yielding unacceptably incomplete results.

To better understand these numbers, we plot the size of the memory map per three-way comparison, starting with the peripherals in Figure 7.2. Here we observe that all three sources mostly agree, with the CMSIS-SVD files sometimes providing more peripherals than the other sources. However, as mentioned before, a fair comparison of this data is not possible, since the definition of a peripheral is not the same across all three sources.

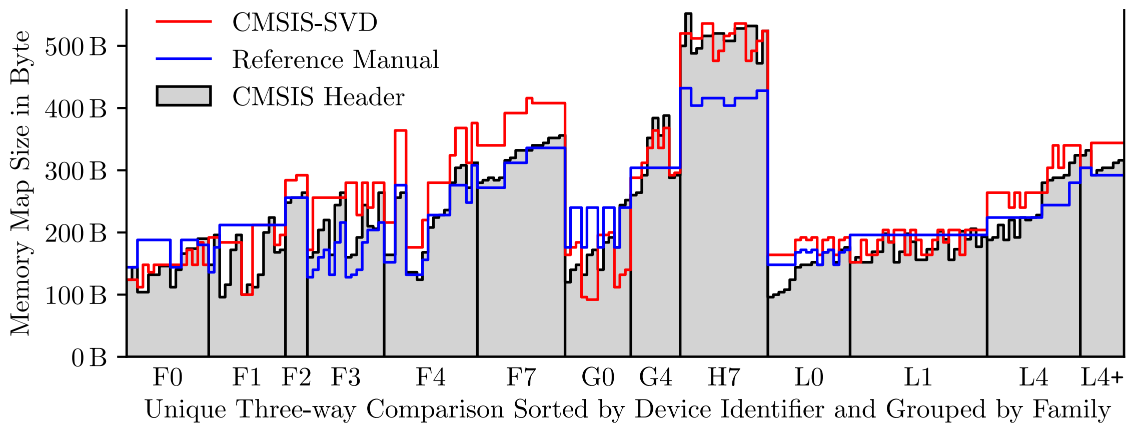


Figure 7.2 For this graph, we arrange the peripheral memory map sizes for each comparison by device family and alphabetical order. This arrangement simultaneously sorts the families roughly by size, as STMicro uses larger numbers in their device identifiers to indicate more capabilities. We can see the different device resolutions of each data source, for example, all STM32L1 devices derive the same register description from only one reference manual.

Therefore, we continue with the register level in Figure 7.3, which breaks down the peripheral grouping and allows for a more neutral comparison. The figure visualizes how closely, even with a limited device resolution, our pipeline can reconstruct the register map from the reference manual for the STM32F3, STM32F4, and STM32F7 devices. The reason for the large discrepancies in the STM32H7 devices is due to the CMSIS header files defining registers related to dual-core management, which are not classified as peripherals in the reference manual or simply omitted in the CMSIS-SVD files. The peak in the STM32H7 devices is caused by a large array of registers in a graphics accelerator peripheral, which is correctly interpreted by the CMSIS header and reference manual pipelines, but not the CMSIS-SVD one. Investigating further, we found that while the SVD standard allows for register arrays, STMicro does not always use it correctly, and thus, these arrays sometimes contain fewer and only one register.

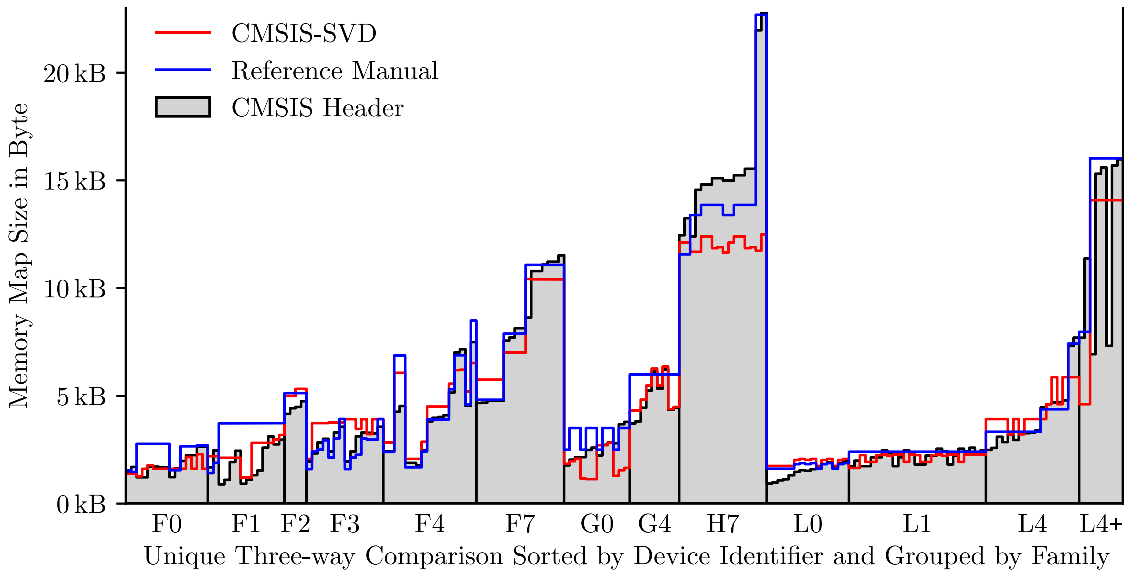


Figure 7.3 The largest register memory map contains over 23 kB of registers and is placed right next to the smallest with a mere 1810 B. The two peaks in the STM32H7 headers and the two dips in the STM32L4+ headers are caused by the inclusion and omission of the graphics accelerator peripheral GFXMMU, whose register file includes an 8 KiB lookup table. Notice how closely the reference manual matches the CMSIS header for the STM32F3 family, while the SVD files are better suited for STM32G4 devices.

The bit field memory map sizes in Figure 7.4 show a lack of data from the CMSIS header files compared to the reference manuals and SVD files. On inspection, we noticed that the CMSIS header files do not contain bit field definitions for registers that contain only a single integer value, since a 32-bit or 16-bit value can be natively constructed using the C `uint32_t` or `uint16_t` respectively. A reasonably easy fix could be to substitute the missing bit fields with their register name and width, however, not all such bit fields span the whole register and the C language bindings do not give a hint of their true width. Such a reconstruction would also inadvertently leak the register naming conflicts into the bit field level. We therefore did not attempt this reconstruction and instead continued the evaluation only with the bit fields explicitly mentioned in the original sources.

While we cannot give a definitive measure of the completeness of the memory maps, these figures demonstrate the effectiveness of our pipelines. The register descriptions reconstructed from the reference manual and the CMSIS header files contain about the same amount of distinct memory locations as the CMSIS-SVD files provided by STMicro. In the special cases of interpreting register arrays and integer-only bit field descriptions, our reference manual pipeline performs consistently better than the SVD or CMSIS header files respectively. However, these findings are based only on the quantity of distinct memory locations and we also want to classify its quality.

We continue our evaluation by finding all memory locations whose names do not match after normalization for all three levels resulting in Table 7.8. Compared to the whole memory map size, the amount of conflict-free locations is fairly high at above 97% for registers and still 93% for bit fields. However, if we take into account

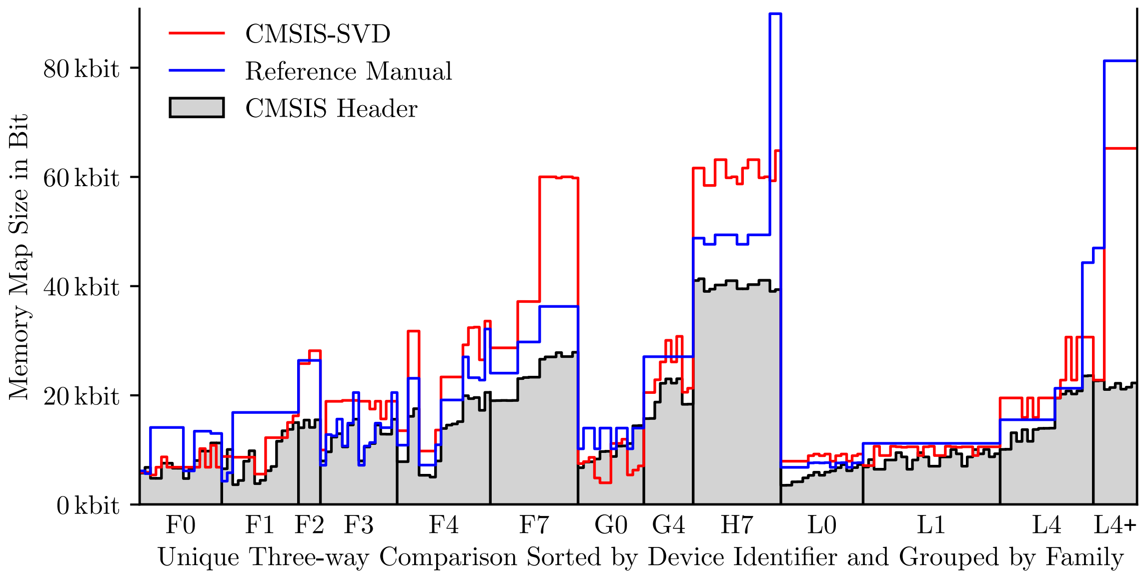


Figure 7.4 The bit field memory maps extracted from the CMSIS header files are omitting descriptions for registers with only a single large bit field. The effect can be multiplied by arrays as seen with the STM32L4+ devices. In contrast, the SVD files tend to define all possible bit fields, even if they do not exist on the specific device.

the map overlap from Table 7.7, the registers drop by just 1 percentage point, while the bit fields loose over 4 points. Not only do the bit fields have a lower overlap, their rate of conflict is 5 times higher than for the registers.

Hierarchy Level	Conflict Size	Total Map Size	Conflict-Free Locations	Overlap Map Size	Matching Locations
Peripherals	2 780 B	55 704 B	95.0%	42 744 B	93.5%
Registers	32 694 B	1 189 630 B	97.3%	890 548 B	96.3%
Bit Fields	645 416 bit	5 598 867 bit	93.3%	3 419 745 bit	89.0%

Table 7.8 The number of conflicts per level and their percentage of conflict-free locations relative to the total size of the memory map or just the locations where two or more sources overlap. The low overlap of just 61% (cf. Table 7.7) lowers the bit field numbers even more .

Since we have three sources, we can try to improve these results by applying majority voting, which requires three sources per location with two agreeing sources overruling one other. These requirements are fulfilled by about 45–64% of overlapping memory locations as shown in Table 7.9. We can also see significant differences per level in the source combinations that agreed most during the voting process, particularly on register and bit field level, where the combinations using the reference manuals agree most often, demonstrating the accuracy and usefulness of our pipeline. This simple voting mechanism is enough to significantly improve the percentage of conflict-free overlapping memory, even bringing bit fields back up to 96%. However, the non-overlapping memory is still 25% for registers and 39% for bit fields, therefore we cannot extrapolate these numbers to the rest of the map.

When we plot the relative conflict rate of registers per device family in Figure 7.5, we discover that both the conflict distribution as well as the majority voting opportunities are not equally distributed among the memory maps. The largest amount of

Hierarchy Level	Resolvable by Majority Vote	Reference Manual + Header	CMSIS Header + CMSIS-SVD	Reference Manual + CMSIS-SVD	Matching + Resolved Locations
Peripherals	45.9%	75.4%	23.3%	1.3%	96.5%
Registers	44.8%	54.8%	15.4%	29.8%	98.0%
Bit Fields	63.5%	39.0%	22.8%	38.2%	96.0%

Table 7.9 Conflicts can be resolved by majority vote, only if two source agree over one other. The reference manual and header files agree the most, however, at bit field level, this pattern becomes less clear. With the voting mechanism, we can increase the accuracy of the memory map, but only for overlapping locations.

conflicts is attributed to the STM32F7, STM32H7, and STM32L4+ families, which are complex microcontrollers with large numbers of registers and bit fields, while the simpler devices have fewer conflicts to begin with and a higher share of majority voting. For the bit field conflict distribution shown in Figure 7.6, these patterns are more spread out and the simpler devices have even more opportunities to use majority voting.

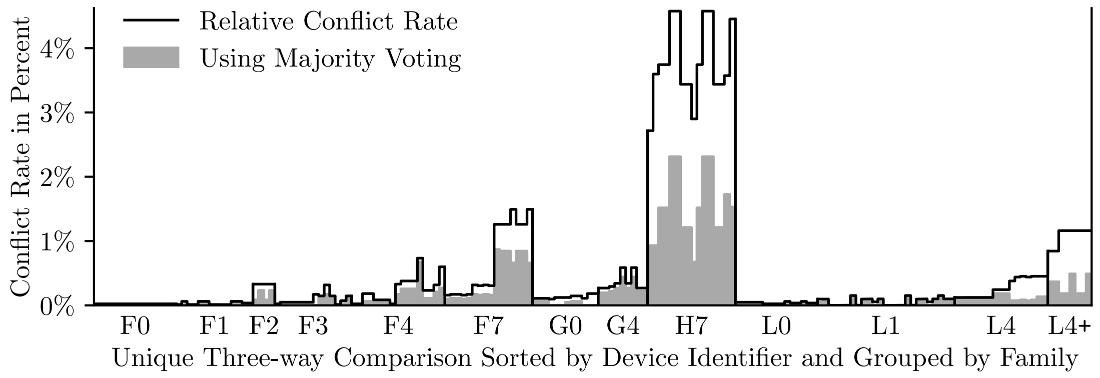


Figure 7.5 The distribution of register conflicts is not equally distributed, with simple devices having almost no conflicts, while complex devices in the STM32H7 family have a significant 4% conflict rate. The opportunity to use majority voting also decreases with complex devices.

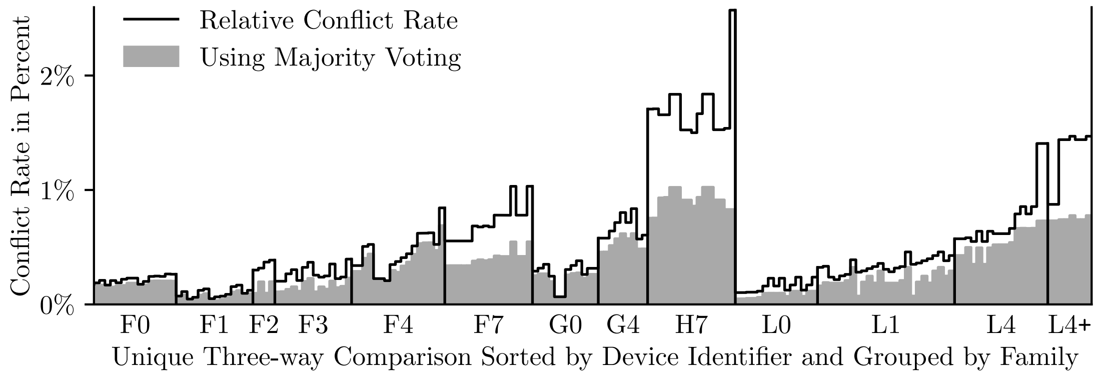


Figure 7.6 The distribution of bit field conflicts is spread wider than for the registers, however, it compensates with an even higher use of majority voting across all devices.

To investigate the outliers in the STM32F7, STM32H7, and STM32L4+ devices, we aggregate the register conflicts into the top four associated peripherals as compiled in Table 7.10, which confirms that the STM32H7 devices alone are responsible for over half of the conflicts. We noticed the peripherals with the most register conflicts overall are the digital filter for sigma delta modulators (DFSDM), USB, DMA, and high-resolution timer (HRTIM) peripherals. Comparing the three sources manually, we found a lot of register locations to have aliases with differing bit fields depending on the peripheral runtime configuration. While our reference manual pipeline extracts these location aliases faithfully, the CMSIS-SVD files do not always encode these aliases correctly, and the CMSIS headers compromise by using a single neutral name instead of a C union. Our linear memory map model does not model multiple names per location, instead keeping only the last name of register or bit field aliases, which can lead to artificial conflicts. A solution could be to merge the aliases into one neutral name before the comparison, provided we define a manual conversion list of alias combinations to single name, which requires significant effort, or simply use the CMSIS header name for these locations.

We also noticed slightly different names in the reference manuals than in the other sources, however, the bit field structures and their functionality appears to be compatible, according to the associated textual descriptions. We image these issues to be the results of the large complexity of the peripherals, having many registers with similar names and interlocking functionality. Therefore, we should always check for such patterns in the conflict set to find pathological issues that can only be properly resolved with manual intervention.

Family	Share of Conflicts	Top 4 Peripherals with Register Conflicts			
STM32F0	0.3%	100% DBGMCU			
STM32F1	0.5%	65% FSMC	13% ADC	13% USB	6% SDIO
STM32F2	1.8%	42% ETH	32% USB	14% FSMC	11% ADC
STM32F3	1.1%	41% ADC	32% HRTIM	20% EXTI	4% CEC
STM32F4	5.8%	20% USB	18% I2C	15% FSMC	10% QSPI
STM32F7	13.4%	48% DFSDM	19% USB	12% FSMC	8% DSI
STM32G0	2.0%	61% DMAMUX	12% SYSCFG	14% EXTI	6% COMP
STM32G4	4.1%	39% DMAMUX	24% HRTIM	10% ADC	7% UCPD
STM32H7	57.7%	19% DFSDM	15% DMA	10% HRTIM	8% ECC
STM32L0	0.9%	61% FLASH	26% COMP	13% SYSCFG	
STM32L1	1.3%	59% RI	26% FSMC	8% RTC	7% OPAMP
STM32L4	4.1%	63% DFSDM	18% RTC	10% FSMC	4% DAC
STM32L4+	7.0%	36% DFSDM	30% DSI	11% FSMC	8% DMA
Total	100%	23% DFSDM	11% USB	8% DMA	7% HRTIM

Table 7.10 The STM32H7 family is responsible for the majority of register conflicts. The peripherals with the most conflicts are all very complex, which probably contributes to the issue in general.

We conclude our evaluation with a summary of all the compared datasets in Table 7.11. Our implementation was able to match existing machine-readable data sources both in quantity and quality with high accuracy. We took care to eliminate systemic problems in our pipelines by validating the consistency of our results and finding justifications for outliers manually. We will discuss these findings in the context of the problem statement (cf. Section 4.4) in the next section.

Dataset	Sources	Method of Comparison	Result
Device Identifier	Datasheet vs. STM32CubeMX	Datasheet $\supseteq$ STM32CubeMX	**93.0%** (N=2899)
Interrupt Vector Table	CMSIS header vs. reference manual	Matching vector name at table position	**98.8%** (N=177270)
Package	Datasheet vs. STM32CubeMX	Datasheet = STM32CubeMX	**99.77%** (N=2696)
Pinout	Datasheet vs. STM32CubeMX	Matching pin name at package position	**99.88%** (N=237399)
Pin Function	Datasheet vs. STM32CubeMX	Matching function index for function name at pin	**95.9%** (N=1039429)
Peripheral	CMSIS header vs. CMSIS-SVD vs. Reference manual	Matching peripheral name at byte address after majority voting	**96.5%** (N=42744)
Register	CMSIS header vs. CMSIS-SVD vs. Reference manual	Matching register name at byte address after majority voting	**98.0%** (N=890548)
Bit Field	CMSIS header vs. CMSIS-SVD vs. Reference manual	Matching bit field name at bit address after majority voting	**96.0%** (N=3419745)
All Datasets	All Sources	Weighted average over all data points	**96.5%** (N=5812730)

Table 7.11 The summary of all data comparisons we performed for this evaluation. The overall quality of the extracted data is very high when compared to the machine-readable sources.

7.5 Discussion

In Chapter 4, we described the scenario of porting hardware-dependent software (HdS) to a new microcontroller with many steps that require access to data from technical documentation. However, existing work does not provide a way to extract this use case specific information from the PDF and thus we identified a research gap for a specialized data pipeline. We identified eight challenges that a suitable solution needs to address and formulated a concise problem statement in four parts: accessing technical documentation, processing its content, encoding the extracted information, and assessing its quality. In this section, we discuss how well our design and implementation solved the aspects raised in the problem statement and associated challenges.

Accessing Technical Documentation

We successfully accessed the technical documentation by converting each PDF into HTML using a custom parser with a vendor-specific implementation, rather than a generic heuristic approach to improve accuracy and reproducibility. Our implementation can operate on individual pages or chapters rather than the whole document, allowing for a rapid iteration cycle to help reverse-engineer the formatting style and find workarounds for tricky corner cases. By converting the chapters of large documents in parallel, we can convert 125 thousand pages in about 2 hours on consumer hardware (cf. Section 7.2). The accuracy of the resulting HTML is high enough to not cause any issues for the next pipelines, with patches required only to repair already existing formatting issues in the PDF (✓ *C5: Maintainability*). However, our implementation has several known limitations, in particular, the omission of figures, that make the output less convenient for human consumption, but are not relevant for our purposes (cf. Section 7.4.1). The implementation effort was about twice that of accessing only machine-readable sources and required more algorithmic complexity, however, we expect the modular design to make it less costly to add new documentation styles from more vendors in future (cf. Section 7.3).

Processing Technical Documentation

We implemented both a table processing interface via the Wang abstract table model (cf. Section 2.2) and a simple regex-based text mining interface to access the HTML content (cf. Section 6.1.4). Both methods were simple to implement and worked well in practice even for complex table structures and text fields (cf. Section 6.1.5), yielding very detailed datasets (✓ *C2: Fidelity*). We were even able to derive additional context from the table caption and surrounding text to selectively access information for a single device identifier (cf. Figure 6.4), which increased our device resolution for the register descriptions to 56 maps generated out of only 44 reference manuals (✓ *C1: Coverage*). However, we found that regex-based text mining is too limiting in practice, as footnotes and text present important information often using different keywords and phrases, making it difficult to write matching patterns for. We were also unable to find the correct textual register descriptions particularly for complex peripherals due to these limitations (cf. Section 6.1.6).

Encoding Extracted Information

We chose to encode the extracted data using a knowledge graph, for which we created a custom ontology for our hardware description data (cf. Section 6.2). The main difficulty was the normalization of the extracted data, which often had naming differences between sources. However, we were able to solve this using only regex substitution patterns. For sources that allowed majority voting, we were able to merge machine-readable source with the technical documentation to detect and repair a large portion of mistakes in the data (cf. Section 7.4.6) (✓ *C4: Clarity*). While the use of namespaces encoded all data unambiguously (✓ *C4: Clarity*), it led to a large knowledge graph that contained partial duplicates, simply because the devices are frequently very similar in hardware. However, using simple compression, we were able to reduce the final graph size to under 100 MB, which is significantly less than the combined input sources of almost 2 GB. The final knowledge graph is easily discoverable using third-party ontology editors such as Protégé [Mus15] (✓ *C7: Discoverability*) and can be accessed via a custom Python API for integration with code generation tools (✓ *C8: Accessibility*).

Validating Extracted Information

The results of our extensive evaluation consistently demonstrated the ability of our pipelines to extract highly accurate and complete data from the technical documentation when compared to machine readable sources (✓ *C3: Correctness*). On top, our comparisons were able to find issues in the machine readable source, particularly in the pinouts and pin functions that would have been very difficult to find manually. We can now use these identified patterns to much more effectively guide a manual patching effort to increase the quality of all sources (✓ *C5: Maintainability*). Even for the very large and complex register descriptions, we were able to reconstruct a good enough device resolution to match other sources and repair conflicts via majority voting. These results gives us high confidence in using our pipeline for extracting data without machine-readable counterparts and achieving similar accuracy, at least for STMicro technical documentation. However, the architecture of our pipelines and evaluation code is flexible enough to accommodate new data sources in future (✓ *C6: Extensibility*).

Practical Considerations

Our pipeline does not exist in a vacuum and there are practical aspects of our use cases that may require us to compromise our approach. In this thesis, we only extracted data that already exists in machine-readable form for the purpose of evaluating this data via a direct comparison, which yielded good results only *after* a normalization step. We noticed the data in the technical documentation to often be inconsistently named across devices, not different enough to be wrong or ambiguous especially given the context. However, a clear canonical name was sometimes difficult to identify. Unlike the machine-readable source, which are at some point either turned into code (STM32CubeMX database) or compiled into firmware (CMSIS-SVD and header), thus requiring a more consistent naming scheme to reduce implementation efforts, the technical documentation is consumed only by human engineers, who can compensate these differences with sufficient experience. This different means that data extracted from technical documentation needs to be carefully post-processed to turn it into a high-quality dataset that is consistent across a large number of devices. In addition, we can file bug reports for the STM32CubeMX database in the official GitHub repository [stm08], but we did not find a way to report issues in the technical documentation to STMicro. Thus, we recommend to use machine-readable sources as much as possible, especially after patching the issues we identified in our evaluation, and only use the technical documentation for data that does not exist in other formats.

In the case of merging the three register descriptions sources, we have another practical issue to consider: backward compatibility. Embedded projects that use the CMSIS header files to build their HdS cannot simply switch to new language bindings without a lot of refactoring, particularly if the data is improved again and again. A controlled upgrade solution must not remove or change existing data and instead only add new data under new names. We would therefore recommend to use the CMSIS header files as the primary source of register definitions and only fill up all missing memory locations from the technical documentation, since they corresponded to the header files on most collisions (cf. Table 7.9). The CMSIS-SVD files matched the fewest times and since we could not find an explanation as to why they are even different from the CMSIS header files, considering the C language bindings are supposed to be generated out of these SVD files, we would simply ignore this data source.

In conclusion, the data extracted from the STMicro technical documentation is very accurate compared to the STM32CubeMX database, CMSIS headers, and SVD files. Our modular design splits the problem into multiple fast pipelines that generate very detailed datasets for thousands of devices using table processing and simple text mining. The final dataset is encoded as a knowledge graph with a custom ontology to facilitate discovery through standardized tooling and access through a simplified Python API. While our design and implementation addresses all challenges listed in Section 4.2, we have identified several limitations in our approach that need to be resolved in future to expand on our design. We discuss this future work and provide a conclusion to the entire thesis in the next chapter.

8 Conclusion

In this chapter, we summarize the findings of this thesis and provide a concise conclusion based on our evaluation and discussion results in Section 8.1. Moreover, we outline potential future work that might extend and build upon our proposed design in Section 8.2.

8.1 Conclusion

In this thesis, we presented the design and implementation of a data processor to extract data from technical documentation describing embedded hardware, compare and merge it with other data, and format the result as a knowledge graph. While machine-readable data sources for describing microcontroller hardware are available in standardized CMSIS files and proprietary databases of configuration tools, their scope and fidelity is decided entirely by the vendor. Most data required for porting HdS is only available in the documentation in the form of PDFs, requiring a human developer to manually extract the data and convert it into code, rather than using model-driven software engineering tools. Particularly for HdS projects supporting many embedded devices, the porting effort is significant and cannot easily be shared with other projects implementing a different HAL or using an incompatible programming language. Existing work in information extraction via table processing uses heuristic approaches to parse PDFs often with user supervision to guide the process and a focus on generic documents, rather than embedded technical documentation.

We therefore identified a research gap for a data processor design that specializes in unsupervised information extraction from technical documentation for the purpose of providing data for code generation to help with HdS porting. We designed a conversion data processor that converts PDF to HTML as an intermediary format, and then uses table processing and simple text mining to convert tabular data into a embedded-specific knowledge graph ontology with a simple API. Our implementation of said data processor design for STMirco microcontrollers is modular, fast, and delivers very accurate results, while requiring only three times the implementation effort compared to only parsing machine-readable sources. The processor runtime was about 2.5 hours on our commodity hardware, which reduces to a few minutes after the first run due to caching all intermediary artifacts, thus allowing fast iteration for development in practice. We demonstrated the extraction of a variety of data with different complexity for several thousand STM32 microcontrollers: device identifier, interrupts vector table, package and pinout, pin functions, and MMIO register descriptions.

During our evaluation, we discovered the data in the technical documentation to be very similar in quality to the machine-readable sources, with about 96.5% matching and/or non-conflicting comparisons on average. Through a statistical analysis of the conflicting data, we were even able to identify several patterns of issues in the machine-readable data, which would have been very difficult to find otherwise, to guide a manual patching effort more effectively. We also successfully merged and corrected the register descriptions from the technical documentation, CMSIS header, and SVD files by resolving conflicts using majority voting.

In conclusion, our data processor presents a significant improvement over existing generic information extraction solutions when applied to technical documentation, due to a specialized PDF parser, unsupervised and fast operation, domain-specific data encoding, and highly accurate results. With over a thousand captioned tables in our HTML archive of STMicro technical documentation, there is significant potential for extracting data not available in a machine-readable format and therefore only published in the documentation. We expect many new use cases to be made possible by our work, some of which we describe next.

8.2 Future Work

While our implementation addresses all design challenges as discussed in Chapter 7, we have intentionally limited ourselves to use only STMicro sources to guarantee a fair comparison in our evaluation. As a result, the data our data processor extracted is not enough to fully satisfy our scenario from Section 4.1. Thus, in this section, we explore the future work needed to expand our design to its full potential.

Most data encoded in tables are easy to extract using existing methods, for example, the DMA event trigger table in Figure 2.5 has a similar structure as the pin function table in Figure 6.3, thus the same boxhead-stub access can be used. We can also apply existing extraction methods to other documents, for example, extracting the register descriptions from the datasheets describing externally connected devices such as sensors, communication modules, and memories (cf. Section 4.1.3). Other use cases require additional analysis on top of the extracted data. For example, we can write an algorithm to compare all register descriptions to discover peripheral compatibility as required during HAL porting described in Section 4.1.2.

We are confident that investing additional effort into the meta-modeling of our ontology to encode additional limitations with the goal of making the dataset easier to reason about, is beneficial as well. For example, we did not extract the STM32F1 pin functions, since these functions are remapped in groups, rather than individually, which imposes an interlocking constraint on the pin function groups. Modeling this information directly in OWL would make the knowledge graph more independent of wrapper code so that projects using our processor are not required to use Python for correctly accessing the stored data.

We could utilize the intermediary artifacts of our data processor (cf. Section 7.1.2) for new use cases, for example, just as the CMSIS-SVD files inform a debugger of the register map, we can inform an IDE of the register description by finding the right section in the HTML documentation and presenting this information as a tool tip. We also convert all revisions of the same PDF document to HTML and with some minor post-processing we could visualize the differences between two revisions to understand the changes in more depth. In a similar vain, we could develop our evaluation setup into a "unit test" harness to continuously validate the completeness and accuracy of the technical documentation against machine-readable sources for new revisions. Thus, these use cases all relate to making the extracted data more accessible to embedded developers to simplify and improve their workflow in the specific context of their project.

Finally, adding new data sources from other vendors would constitutes another direction for future work. Since we are parsing microcontroller documentation from STMicro, accessing the frequently referenced documentation on the Arm Cortex-M architecture (cf. Section 2.1) would also be useful. Additionally, several complex peripherals are contributed by other hardware vendors, for example, all STM32 USB peripherals are licensed from Synopsis [RM0432], therefore comparing other vendor's documentation to perhaps resolve the register naming consistency issues we discovered during our evaluation (cf. Section 7.4.6) may be interesting. To make adding new formatting styles to the PDF parser easier, it may be beneficial to apply more heuristics to tune the parser automatically [RMB+21], rather than implement everything manually as it is done now. A more generic neural net could be trained using the very accurate HTML output of our data processor in comparison to the PDF.

Apart from these generic improvements, there are two specific aspects we intentionally did not implement in our data processor: state-of-the-art text mining and processing figures.

Text Mining

In this thesis, we only implemented text mining through a simple regex matcher, which worked well for searching table captions for known patterns. However, it broke down quickly when trying to interpret more advanced text (cf. Section 6.1.6). Since technical documentation uses a lot of domain-specific jargon with many nouns and abbreviations only making sense in the context of the document, the use of traditional text mining approaches may be difficult. A custom semantic parser could be seeded by the data extracted from table processing first to make domain-specific text mining possible.

Apart from general access to the text, we identified the need for a "semantic hash function" that condenses a set of descriptions into an unambiguous set of names that faithfully represent these descriptions. The enumerated field values in the bit field descriptions (cf. Figure 2.11) have a numeric value and a description, but no short name, which is, however, required by the SVD standard to be able to assign the numeric value to a variable in the code generator (cf. Section 2.3.1).

However, in contrast to tables, whose headers gives the content a structure, text is much less constrained both in semantics as well as in the locality of similar data. The implementation effort required for a custom text mining setup specialized on the domain of embedded technical documentation could be quite significant. This effort has to be carefully weight against the data that could potentially be made accessible and if this comparison yields unfavorable results, the focus should be on extracting data from simpler content types. The only other content type in technical documentation is figures, which combine text with vector graphics to present information.

Processing Figures

Our data processor detects and then intentionally discards figures in the PDFs to reduce the implementation effort, since we believe that the technical documentation presents important information mostly as tables. However, with 1027 captioned tables and 998 captioned figures, the lack of figures is quite noticeable in the converted HTML and hindered our comprehension significantly compared to the PDF. To include the figures in the HTML, we would translate the Postscript-based graphics into SVGs, which has a compatible feature set [EHLN06] and remains much easier to access programmatically compared to images (cf. Section 3.1). We then add a Figure class with access methods to our Python API (cf. Section 6.1.4).

To understand what access methods are required, we look at three figures which our scenario pointed us to. Figure 4.5 renders the pinout of the Nucleo-144 development board as an implicit table without borders or headers, which could be reconstructed using whitespace analysis [RCVF03, CF04, CD16]. Figure 4.4 places all connected components into their own box, which can be interpreted as individual cells in a table with a non-regular 2D structure. These rendering concept would not be difficult to process, particularly if the interpretation is provided manually as part of the HTML to OWL data processor, similarly to how the table structure is configured manually. For complex renderings such as the clock graph in Figure 4.3, extracting text is not enough and we would need to recognize graphics shapes into modular blocks and then reconstruct the clock graph based on the interconnections.

Finally, there are many more opportunities for future work to build on our data processor to extract information from technical documentation in other fields than just for firmware development. For example, every datasheet contains a chapter on electrical characteristics of the devices, which is filled with very detailed tables on power consumption, voltage levels, and analog properties, which would be very interesting to electrical engineers. Similarly, access to mechanical drawings and dimensions of device footprints and packaging can be very useful for hardware engineers. Our work has laid the foundation for developing challenging new use cases that require automatically processing technical documentation and thus provides a vital contribution to improving the porting process of embedded software.

Bibliography

[ABFV13] Andrea Acquaviva, Nicola Bombieri, Franco Fummi, and Sara Vinco. Semi-automatic generation of device drivers for rapid embedded platform development. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(9):1293–1306, 2013.

[ada15] AdaCore: CMSIS-SVD to Ada Generator. `https://github.com/AdaCore/svd2ada`, 2015.

[AN21] Farzad Asgarian and Khalil Najafi. Bluesync: Time synchronization in bluetooth low energy with energy efficient calculations. *IEEE Internet of Things Journal*, 2021.

[ant17] Renode simulator. `https://renode.io`, 2017.

[arm15] CMSIS Project Page. `https://arm-software.github.io/CMSIS_5`, 2015.

[arm16] Armv8-M Architecture Reference Manual. `https://developer.arm.com/documentation/ddi0553/latest`, 2016.

[AS13] Marco D Adelfio and Hanan Samet. Schema extraction for tabular data on the web. *Proceedings of the VLDB Endowment*, 6(6):421–432, 2013.

[BBA17] Jacob Beningo, Jacob Beningo, and Anglin. *Reusable Firmware Development*. Springer, 2017.

[Bod17] Joel Bodenmann. *2D Hardware Acceleration*. PhD thesis, Haute Ecole d'Ingénierie, 2017.

[boot17] mcuboot: Secure boot for 32-bit Microcontrollers. `https://github.com/mcu-tools/mcuboot`, 2017.

[BXHP20] Bruce Belson, Wei Xiang, Jason Holdsworth, and Bronson Philippa. C++20 coroutines on microcontrollers – what we learned. *IEEE Embedded Systems Letters*, 13(1):9–12, 2020.

[CD16] Christopher Clark and Santosh Divvala. Pdffigures 2.0: Mining figures from research papers. In *2016 IEEE/ACM Joint Conference on Digital Libraries (JCDL)*, pages 143–152. IEEE, 2016.

[CF04] Hui Chao and Jian Fan. Layout and content extraction for pdf documents. In *International Workshop on Document Analysis Systems*, pages 213–224. Springer, 2004.

[CHCG15] Xu Chu, Yeye He, Kaushik Chakrabarti, and Kris Ganjam. Tegra: Table extraction by global record alignment. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1713–1728, 2015.

[cpp10] CppHeaderParser: Parse C++ header files in Python. `https://pypi.org/project/CppHeaderParser`, 2010.

[cpp17] A C++, compile-time, reactive RTOS for the Stack Resource Policy based Real-Time For the Masses kernel. `https://github.com/korken89/crect`, 2017.

[CTT00] Hsin-Hsi Chen, Shih-Chung Tsai, and Jin-He Tsai. Mining tables from large scale html texts. In *COLING 2000 Volume 1: The 18th International Conference on Computational Linguistics*, 2000.

[DS11581] Datasheet: STM32F413. `https://www.st.com/resource/en/datasheet/stm32f413zh.pdf`.

[DS12232] Datasheet: STM32G071. `https://www.st.com/resource/en/datasheet/stm32g071rb.pdf`.

[DS12556] Datasheet: STM32F750. `https://www.st.com/resource/en/datasheet/stm32h750ib.pdf`.

[DS12960] Datasheet: STM32G441. `https://www.st.com/resource/en/datasheet/stm32g441vb.pdf`.

[DS13139] Datasheet: STM32F7B3. `https://www.st.com/resource/en/datasheet/stm32h7b3ai.pdf`.

[DS13195] Datasheet: STM32F7A3. `https://www.st.com/resource/en/datasheet/stm32h7a3qi.pdf`.

[DS13196] Datasheet: STM32F7B0. `https://www.st.com/resource/en/datasheet/stm32h7b0ib.pdf`.

[DS9118] Datasheet: STM32F303. `https://www.st.com/resource/en/datasheet/stm32f303vc.pdf`.

[EAS13] Ivan Ermilov, Sören Auer, and Claus Stadler. User-driven semantic mapping of tabular data. In *Proceedings of the 9th International Conference on Semantic Systems*, pages 105–112, 2013.

[EHA$^+$13] Johan Eriksson, Fredrik Häggström, Simon Aittamaa, Andrey Kruglyak, and Per Lindgren. Real-time for the masses, step 1: Programming api and static priority srp kernel primitives. In *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 110–113. IEEE, 2013.

[EHLN06] David W Embley, Matthew Hurst, Daniel Lopresti, and George Nagy. Table-processing paradigms: a research survey. *International Journal of Document Analysis and Recognition (IJDAR)*, 8(2):66–86, 2006.

[EMD09] Wolfgang Ecker, Wolfgang Müller, and Rainer Dömer. Hardware-dependent software. In *Hardware-dependent Software*, pages 1–13. Springer, 2009.

[ES0478] Errata Sheet: STM32H7A3/7B3 and STM32H7B0. `https://www.st.com/resource/en/errata_sheet/es0478-stm32h7a3xig-stm32h7b0xb-and-stm32h7b3xi-device-errata-stmicroelectronics.pdf`.

[ETL05] David W Embley, Cui Tao, and Stephen W Liddle. Automating the extraction of data from html tables with unknown structure. *Data & Knowledge Engineering*, 54(1):3–28, 2005.

[Fel20] Nicholas Felker. Design of cyanobyte: An intermediate representation to standardize digital peripheral datasheets for automatic code generation. In *2020 IEEE Sensors Applications Symposium (SAS)*, pages 1–6. IEEE, 2020.

[GFK$^+$20] Gabriel Gaspar, Peter Fabo, Michal Kuba, Juraj Dudak, and Eduard Nemlaha. Micropython as a development platform for iot applications. In *Computer Science On-line Conference*, pages 388–394. Springer, 2020.

[git18] GitHub Actions: Continuous Integration/Delivery Service. `https://github.com/features/actions`, 2018.

[HBC$^+$21] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, et al. Knowledge graphs. *Synthesis Lectures on Data, Semantics, and Knowledge*, 12(2):1–257, 2021.

[HBPT15] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. Operating systems for low-end devices in the internet of things: a survey. *IEEE Internet of Things Journal*, 3(5):720–734, 2015.

[HOSP21] Lars Huning, Timo Osterkamp, Marco Schaarschmidt, and Elke Pulvermüller. Seamless integration of hardware interfaces in uml-based mdse tools. 2021.

[HS15] Krisztián Holman and Zoltán Szabó. Microcontroller based application prototyping using domain specific modeling. In *2015 IEEE 13th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, pages 199–202. IEEE, 2015.

[Hur00] Matthew Francis Hurst. *The Interpretation of Tables in Texts*. PhD thesis, University of Edinburgh, 2000.

[i2c11] Open-source device database for I^{2}C devices. `https://www.i2cdevlib.com`, 2011.

[Jah21] Rebecca Jahn. *Reasoning in Knowledge Graphs: Methods and Techniques*. PhD thesis, Wien, 2021.

[KLU15] Shah Khusro, Asima Latif, and Irfan Ullah. On methods and tools of table detection, extraction and annotation in pdf documents. *Journal of Information Science*, 41(1):41–57, 2015.

[Kor18] Christopher Kormanyos. *Real-Time C++: Efficient Object-Oriented and Template Microcontroller Programming*. Springer, 2018.

[Kul17] Nihal Kularatna. *Electronic Circuit Design: From Concept to Implementation*. CRC Press, 2017.

[Lam17] Jean-Baptiste Lamy. Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artificial intelligence in medicine*, 80:11–28, 2017.

[LFL$^+$16] Per Lindgren, Emil Fresk, Marcus Lindner, Andreas Lindner, David Pereira, and Luís Miguel Pinho. Abstract timers and their implementation onto the arm cortex-m family of mcus. *ACM SIGBED Review*, 13(1):48–53, 2016.

[LIJ$^+$15] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic web*, 6(2):167–195, 2015.

[lin14] Zephyr Embedded RTOS. `https://www.zephyrproject.org`, 2014.

[lin16] The DeviceTree Specification. `https://www.devicetree.org`, 2016.

[LKM01] Kristina Lerman, Craig Knoblock, and Steven Minton. Automatic data extraction from lists and tables in web sources. In *IJCAI-2001 Workshop on Adaptive Text Extraction and Mining*, volume 98. Citeseer, 2001.

[LLL$^+$15] Per Lindgren, Marcus Lindner, Andreas Lindner, David Pereira, and Luís Miguel Pinho. Rtfm-core: Language and implementation. In *2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*, pages 990–995. IEEE, 2015.

[LPL04] Shijun Li, Zhiyong Peng, and Mengchi Liu. Extraction and integration information in html tables. In *The Fourth International Conference onComputer and Information Technology, 2004. CIT'04.*, pages 315–320. IEEE, 2004.

[LWX$^+$21] Rujiao Long, Wen Wang, Nan Xue, Feiyu Gao, Zhibo Yang, Yongpan Wang, and Gui-Song Xia. Parsing table structures in the wild. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 944–952, 2021.

[MKM97] Aengus Murray, Paul Kettle, and Finbarr Moynihan. Advances in brushless motor control. In *Proceedings of the 1997 American Control Conference*, volume 6, pages 3985–3989. IEEE, 1997.

[modm09] modm: a barebone embedded library generator. `https://modm.io`, 2009.

[modm16] modm-devices: curated data for AVR and ARM Cortex-M devices. `https://github.com/modm-io/modm-devices`, 2016.

[modm17] Aggregated CMSIS header files for STM32. `https://github.com/modm-io/cmsis-header-stm32`, 2017.

[modm22] modm Data Repository. `https://github.com/modm-io/modm-data`, 2022.

[MPSD20] Tomasz Marciniak, Kacper Podbucki, Jakub Suder, and Adam Dąbrowski. Analysis of digital filtering with the use of stm32 family microcontrollers. In *Advanced, Contemporary Control*, pages 287–295. Springer, 2020.

[mpy14] MicroPython – Python for Microcontrollers. `https://micropython.org`, 2014.

[Mus15] Mark A. Musen. The protégé project: a look back and a look forward. *AI Matters*, 1(4):4–12, 2015.

[owl17] Owlready2: Ontology-oriented programming in Python. `https://owlready2.readthedocs.io`, 2017.

[PA18] Martha O Perez-Arriaga. Automated development of semantic data models using scientific publications. 2018.

[PCO19] André Pinho, Luis Couto, and José Oliveira. Towards rust for critical systems. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 19–24. IEEE, 2019.

[pdf08] PDF 32000-1:2008 Format Specification. `https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/PDF32000_2008.pdf`, 2008.

[pdf12] Tabula Project. `https://tabula.technology`, 2012.

[pdf13a] InstaBuild: Image-based Footprint and Pinout Parser. `https://www.snapeda.com/instabuild`, 2013.

[pdf13b] PDFium: PDF rendering library. `https://pdfium.googlesource.com/pdfium`, 2013.

[pdf17] uConfig: PDF-to-KiCAD Pinout Footprint Parser. `https://github.com/Robotips/uConfig`, 2017.

[pdf20] Datasheet PDF-to-SVD Parser. `https://github.com/brainstorm/datasheet2svd`, 2020.

[pdf21] pypdfium: Python bindings to PDFium. `https://github.com/pypdfium2-team/pypdfium2`, 2021.

[PL17] Sebastian Plamauer and Martin Langer. Evaluation of micropython as application layer programming language on cubesats. In *ARCS 2017; 30th International Conference on Architecture of Computing Systems*, pages 1–9. VDE, 2017.

[PLW19] Rasmus Berg Palm, Florian Laws, and Ole Winther. Attend, copy, parse end-to-end information extraction from documents. In *2019 International Conference on Document Analysis and Recognition (ICDAR)*, pages 329–336. IEEE, 2019.

[PM0253] Programming Manual: ARM Cortex-M7 processor. `https://www.st.com/resource/en/programming_manual/pm0253-stm32f7-series-and-stm32h7-series-cortexm7-processor-programming-manual-stmicroelectronics.pdf`.

[py16] pygount: count lines of code using pygments. `https://pypi.org/project/pygount`, 2016.

[QRAS+18] Qahhar Muhammad Qadir, Tarik A Rashid, Nawzad K Al-Salihi, Birzo Ismael, Alexander A Kist, and Zhongwei Zhang. Low power wide area networks: A survey of enabling technologies, applications and interoperability needs. *IEEE Access*, 6:77454–77473, 2018.

[Ras17] Roya Rastan. *Automatic Tabular Data Extraction and Understanding.* PhD thesis, University of New South Wales, Sydney, Australia, 2017.

[RCVF03] J-Y Ramel, Michel Crucianu, Nicole Vincent, and Claudie Faure. Detection, extraction and representation of tables. In *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.*, pages 374–378. IEEE, 2003.

[RM0033] Reference Manual: STM32F2x5/2x7. `https://www.st.com/resource/en/reference_manual/cd00225773-stm32f205xx-stm32f207xx-stm32f215xx-and-stm32f217xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf`.

[RM0090] Reference Manual: STM32F4x5/4x7/4x9. `https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf`.

[RM0091] Reference Manual: STM32F0x1/0x2/0x8. `https://www.st.com/resource/en/reference_manual/rm0091-stm32f0x1stm32f0x2stm32f0x8-advanced-armbased-32bit-mcus-stmicroelectronics.pdf`.

[RM0390] Reference Manual: STM32F446. `https://www.st.com/resource/en/reference_manual/rm0390-stm32f446xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf`.

[RM0431] Reference Manual: STM32L7x/73x. `https://www.st.com/resource/en/reference_manual/rm0431-stm32f72xxx-and-stm32f73xxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf`.

[RM0432] Reference Manual: STM32L4+. `https://www.st.com/resource/en/reference_manual/rm0432-stm32l4-series-advanced-armbased-32bit-mcus-stmicroelectronics.pdf`.

[RM0455] Reference Manual: STM32H7A3/7B3 and STM32H7B0. `https://www.st.com/resource/en/reference_manual/rm0455-stm32h7a37b3-and-stm32h7b0-value-line-advanced-armbased-32bit-mcus-stmicroelectronics.pdf`.

[RMB+21] Johannes Rausch, Octavio Martinez, Fabian Bissig, Ce Zhang, and Stefan Feuerriegel. Docparser: Hierarchical document structure parsing from renderings. In *35th AAAI Conference on Artificial Intelligence (AAAI-21)(virtual)*, 2021.

[RPS16] Roya Rastan, Hye-Young Paik, and John Shepherd. A pdf wrapper for table processing. In *Proceedings of the 2016 ACM Symposium on Document Engineering*, pages 115–118, 2016.

[RPS+18] Roya Rastan, Hye-Young Paik, John Shepherd, Seung Hwan Ryu, and Amin Beheshti. TEXUS: table extraction system for PDF documents. In *Australasian Database Conference*, pages 345–349. Springer, 2018.

[rtos03] FreeRTOS: Real-time operating system for microcontrollers. `https://freertos.org`, 2003.

[rust10] Rust on Embedded Devices. `https://www.rust-lang.org/what/embedded`, 2010.

[rust16] RustEmbedded: CMSIS-SVD to Rust Generator. `https://github.com/rust-embedded/svd2rust`, 2016.

[rust17a] Embedded Rust. `https://github.com/rust-embedded`, 2017.

[rust17b] RTIC: Real-Time Interrupt-driven Concurrency. `https://rtic.rs`, 2017.

[rust20] embassy-rs. `https://github.com/embassy-rs`, 2020.

[SAM+18] Alexey Shigarov, Andrey Altaev, Andrey Mikhailov, Viacheslav Paramonov, and Evgeniy Cherkashin. Tabbypdf: web-based system for pdf table extraction. In *International Conference on Information and Software Technologies*, pages 257–269. Springer, 2018.

[SGD08] Gunar Schirner, Andreas Gerstlauer, and Rainer Domer. Automatic generation of hardware dependent software for mpsocs from abstract system specifications. In *2008 Asia and South Pacific Design Automation Conference*, pages 271–276. IEEE, 2008.

[Sin12] Amit Singhal. Introducing the Knowledge Graph: things, not strings. `https://blog.google/products/search/introducing-knowledge-graph-things-not`, 2012.

[stm07] STM32 32-bit ARM Cortex-M microcontrollers. `https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html`, 2007.

[stm08] STM32CubeMX Initialization Code Generator. `https://www.st.com/en/development-tools/stm32cubemx.html`, 2008.

[stm17a] STM32 CMSIS-SVD Device Coverage. `https://stm32-rs.github.io/stm32-rs`, 2017.

[stm17b] STM32 CMSIS-SVD Patches. `https://github.com/stm32-rs/stm32-rs/tree/master/devices`, 2017.

[stm19] STM32Cube MCU Overall Offer. `https://github.com/STMicroelectronics/STM32Cube_MCU_Overall_Offer`, 2019.

[stm20] STM32 Open Pin Data. `https://github.com/STMicroelectronics/STM32_open_pin_data`, 2020.

[stm21] embassy-rs: stm32-data. `https://github.com/embassy-rs/stm32-data`, 2021.

[svd15a] CMSIS-SVD Collection Repository. `https://github.com/posborne/cmsis-svd`, 2015.

[svd15b] CMSIS System View Description Documentation. `https://arm-software.github.io/CMSIS_5/SVD/html/index.html`, 2015.

[svd15c] Extensible ARM CMSIS SVD spec based, multi-language source code generator. `https://github.com/postspectacular/cmsis-svd-srcgen`, 2015.

[svd15d] SVDConv: CMSIS-compliant device header file generator. `https://www.keil.com/pack/doc/cmsis/SVD/html/svd_SVDConv_pg.html`, 2015.

[TELN03] Yuri A Tijerino, David W Embley, Deryle W Lonsdale, and George Nagy. Ontology generation from tables. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003. WISE 2003.*, pages 242–249. IEEE, 2003.

[UM1724] User Manual: Nucleo-64 Development Boards. `https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf`.

[UM2708] User Manual: STM32L4+ IoT Discovery Kit. `https://www.st.com/resource/en/user_manual/um2708-discovery-kit-for-iot-node-multichannel-communication-with-stm32l4-series-stmicroelectronics.pdf`.

[Wan96] Xinxin Wang. Tabular abstraction, editing, and formatting. 1996.

[Wei16] Wang Wei. Survey of attacks and defenses on stack-based buffer overflow vulnerability. In *7th International Conference on Education, Management, Information and Computer Science (ICEMC 2017)*, pages 324–328. Atlantis Press, 2016.

[WRSW21] Kevin Weiss, Michel Rottleuthner, Thomas C Schmidt, and Matthias Wählisch. Philip on the hil: Automated multi-platform os testing with external reference devices. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–26, 2021.

[xml05] lxml: XML and HTML with Python. `https://lxml.de`, 2005.

[YHZ$^+$11] Song Yin, Li Li Huang, Hong Zhao, Yang Wang, and Ping Xia. Portability of wsn sensor driver using abstraction layer and fsm. In *Applied Mechanics and Materials*, volume 44, pages 461–465. Trans Tech Publ, 2011.

[ZB21] Koen Zandberg and Emmanuel Baccelli. Femto-containers: Devops on microcontrollers with lightweight virtualization & isolation for iot software modules. 2021.

[ZMH$^+$21] Gohar Zaman, Hairulnizam Mahdin, Khalid Hussain, Jemal Abawajy, Salama A Mostafa, et al. An ontological framework for information extraction from diverse scientific sources. *IEEE access*, 9:42111–42124, 2021.

[ZSJY20] Xu Zhong, Elaheh ShafieiBavani, and Antonio Jimeno Yepes. Image-based table recognition: data, model, and evaluation. In *European Conference on Computer Vision*, pages 564–580. Springer, 2020.

A

Appendix

A.1 List of Abbreviations

ADC	analog-to-digital converter
API	application programming interface
AST	abstract syntax tree
BIOS	basic I/O system
BSP	board support package
CAN	controller area network
CMSIS	common microcontroller software interface standard
CMSIS-SVD	CMSIS system view description
CPP	C pre-processor
CPU	central processing unit
CRC	cyclic redundancy check
CSV	comma-separated value
DAC	digital-to-analog converter
DFSDM	digital filter for sigma delta modulators
DMA	direct memory access
DSP	digital signal processing
EDA	electronic design automation
FPU	floating point unit
GPIO	general purpose input/output
GUI	graphical user interface

HAL hardware abstraction layer

HdS hardware-dependent software

HiL hardware in the loop

HRTIM high-resolution timer

HTML hypertext markup language

I^{2}C inter-integrated circuit

IDE integrated development environment

IoT internet of things

JSON JavaScript object notation

KG knowledge graph

LoC lines of code

MAC media access control

MDSE model-driven software engineering

MMIO memory-mapped input/output

NVIC nested vector interrupt controller

OCR optical character recognition

OS operating system

OWL web ontology language

PCB printed circuit board

PDF portable document format

PLL phase-locked loop

RAM random-access memory

RDF resource description framework

RDFS RDF schema

ROM read-only memory

RTFM real-time for the masses

RTL register-transfer level

RTOS real-time operating system

SMPS switched mode power supply

SPI serial peripheral interface

SPO subject-predicate-object

SRP stack resource policy

SVD system view description

SVG scalable vector graphics

SWRL semantic web rule language

UART	universal asynchronous receiver/transmitter
UEFI	unified extensible firmware interface
USART	universal synchronous/asynchronous receiver/transmitter
USB	universal serial bus
XML	extensible markup language
XPath	XML path language