# 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.

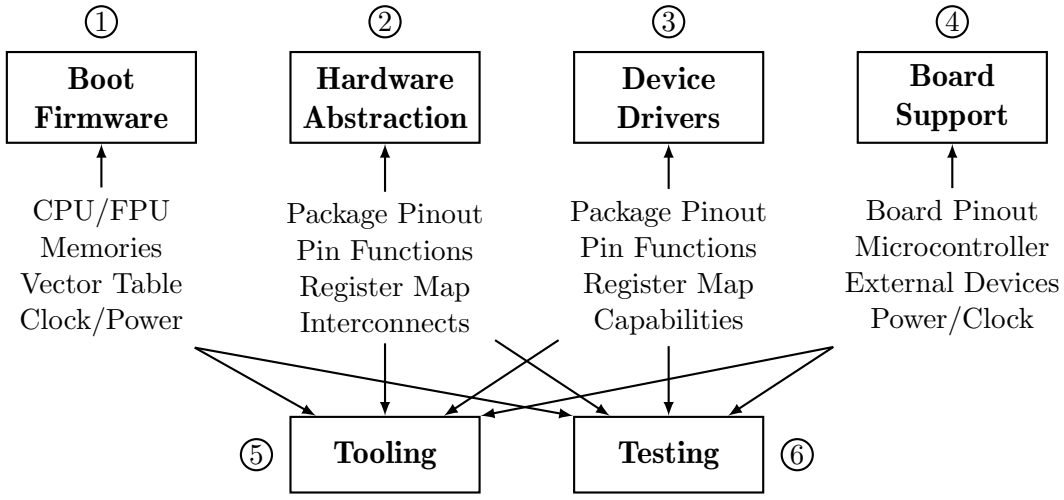


**Figure 4.1** Porting process overview. Steps ① through ④ implement HdS stack layers and therefore depend on each other, while ⑤ tooling and ⑥ testing software is required at any point of the process. The individual steps depend on data that is manually transcribed from the technical documentation as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

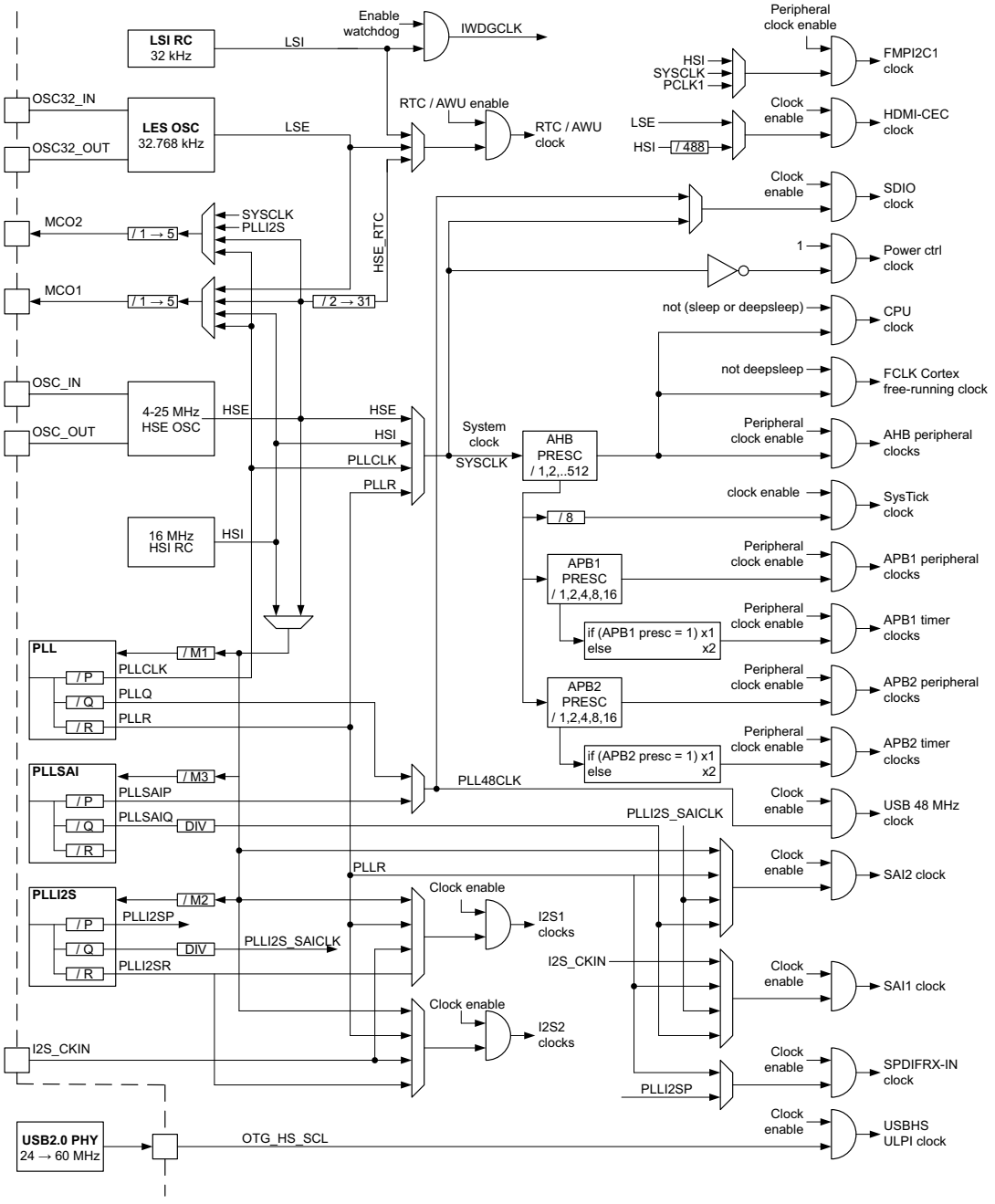


**Figure 4.3** In this clock graph, the external clock signals enter on the device from the left and feed into multiple phase-locked loops (PLLs) that generate different frequencies. An internal network then distributed the clock to the peripherals on the right [RM0390]. The complexity of this figure 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

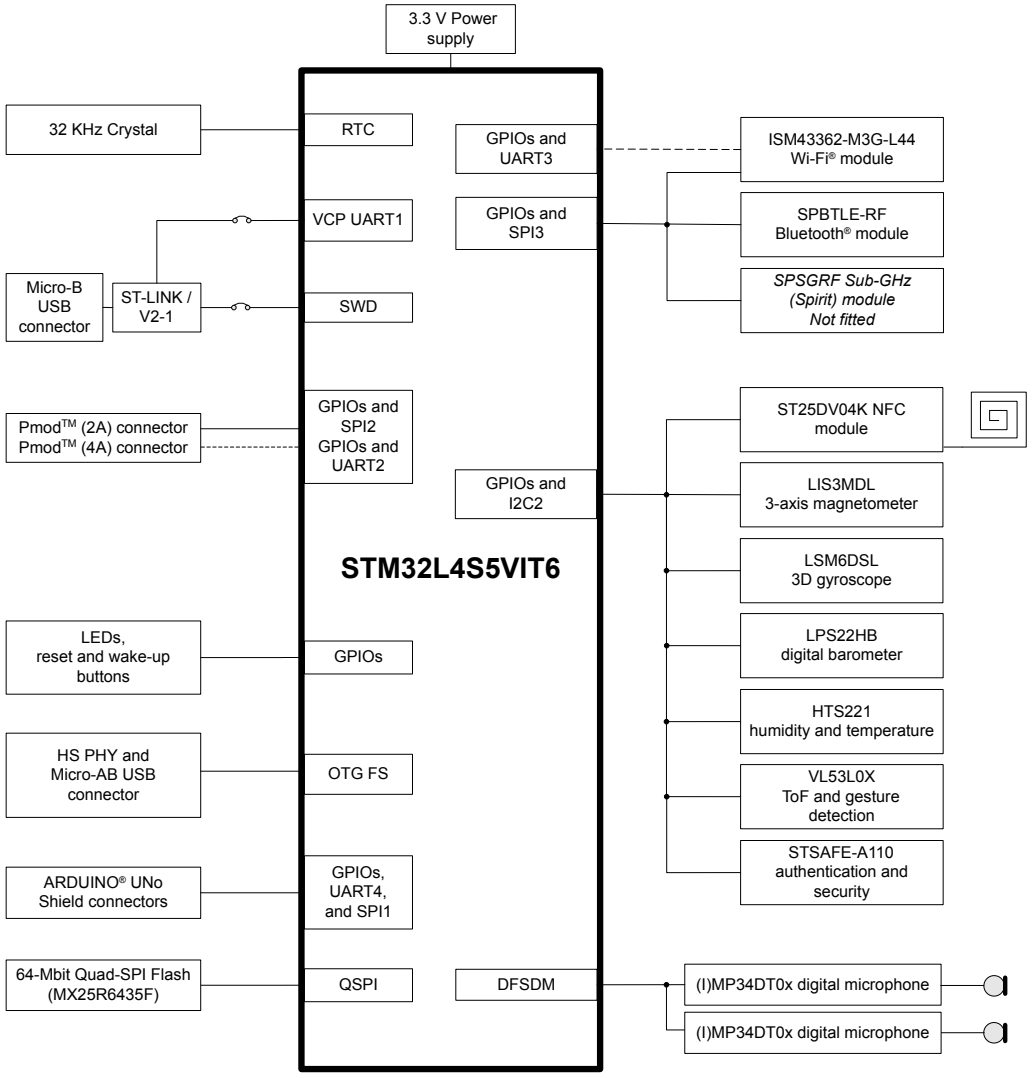


**Figure 4.4** This hardware block diagram of a STM32L4 development board shows the on-board connections of four wireless communication modules, six sensors, two external security and memory devices, and several extensible wired interfaces [UM2708]. The BSP should provide an API for the application to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

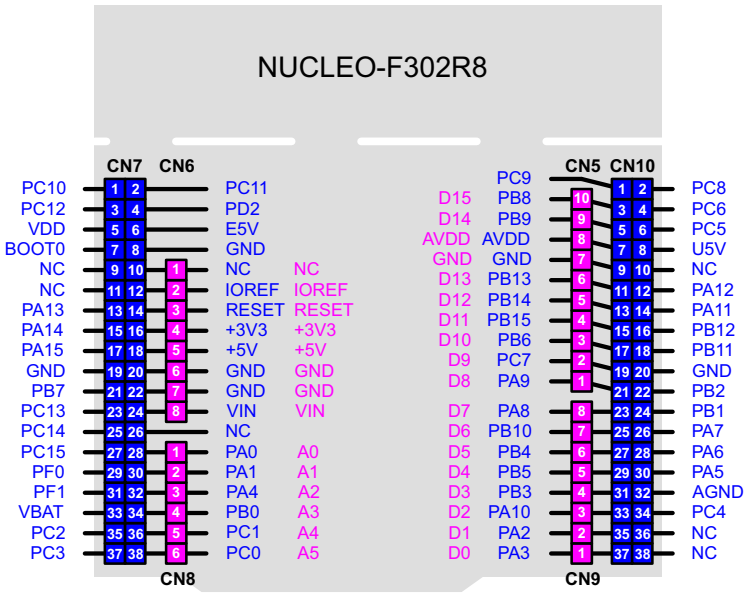


**Figure 4.5** The external pinout for the Nucleo-64 development board renders multiple connectors and rows of pin names in an implicit tabular structure over the outline of the PCB [UM1724]. The embedded system interfaces with the embedding system through these connections [EMD09].

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.

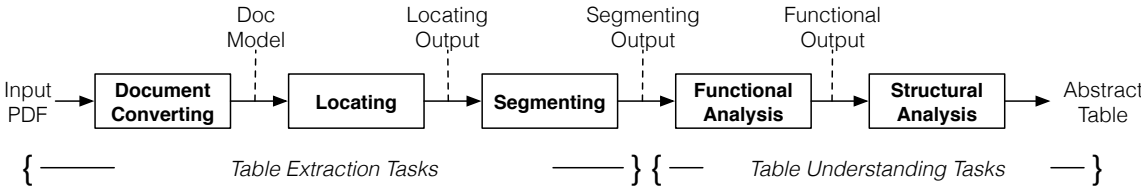


**Figure 4.6** The end-to-end table processing pipeline from TEXUS [RPS+18].

### 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.

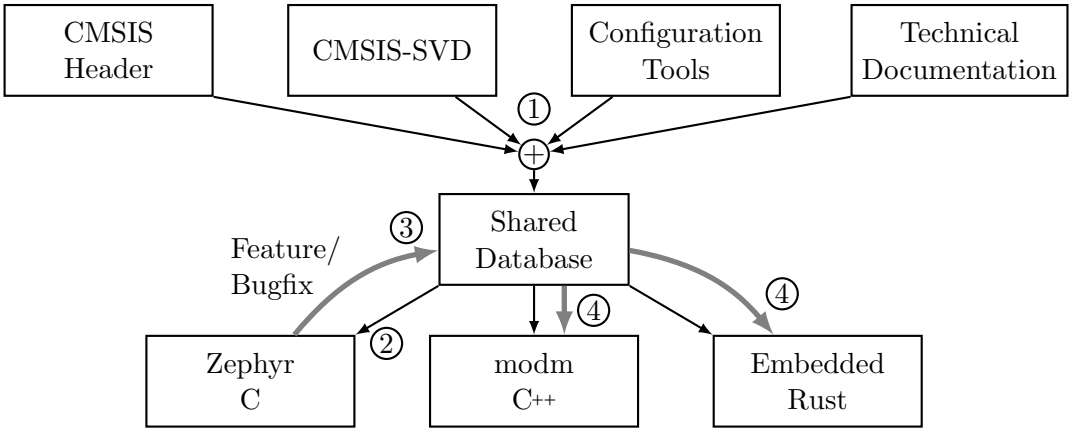


**Figure 4.7** A data pipeline project ① combining multiple input sources into a shared database can help reduce overall development effort. ② A single project using the database can ③ add features or repair issues so that ④ all other projects benefit from the improvements.

## 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.