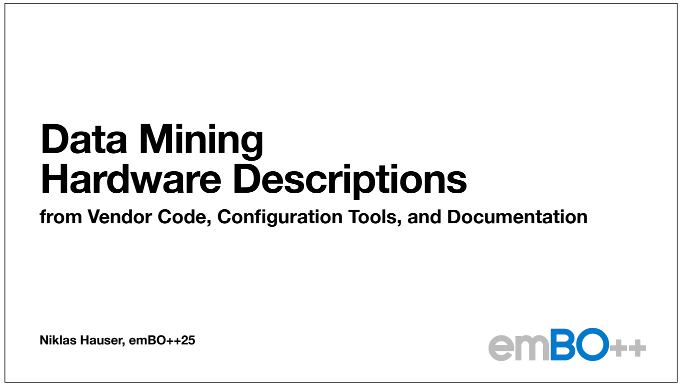
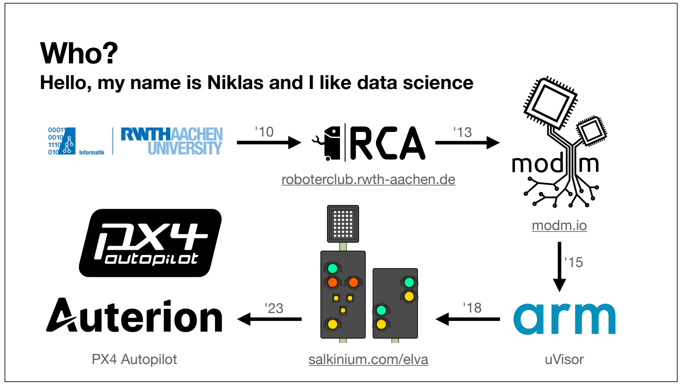
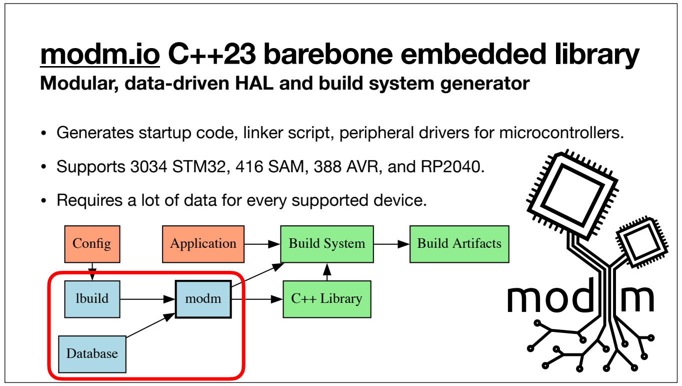


Thank you for the introduction.
Thanks to emBO for the opportunity to talk here.



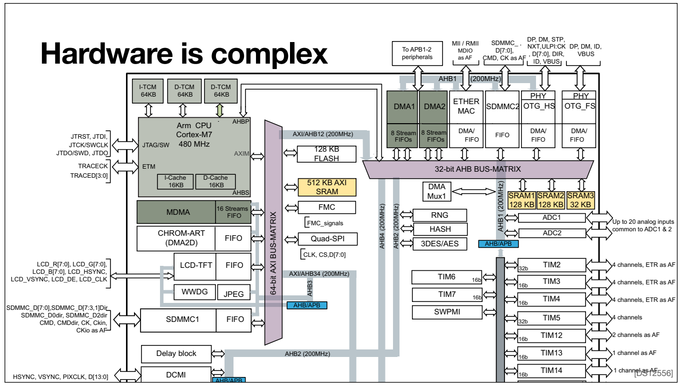
My name is Niklas.
- I started studying Computer Science some time ago.
- I began building autonomous robots in 2010.
- We created a C++ library which is known as modm.io, a C++23 library generator that supports several thousand Cortex-M devices.
- I then started at ARM working on Cortex-M sandboxing, before returning to the university to study for my masters degree.
- There, I worked on a digital modular signalling system for railways.
- I'm currently working at Auterion on the open-source PX4 Autopilot.



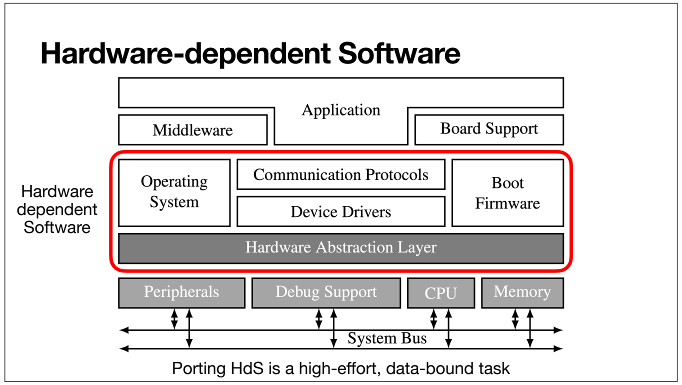
modm is a C++23 embedded library generator.
- The core of modm is a code generator written in Python called lbuild.
- It queries a database of device data and formats the results into C++23 code.
- The HAL is highly modular and configurable and it allows a very small maintainer team to support thousands of microcontrollers.

###

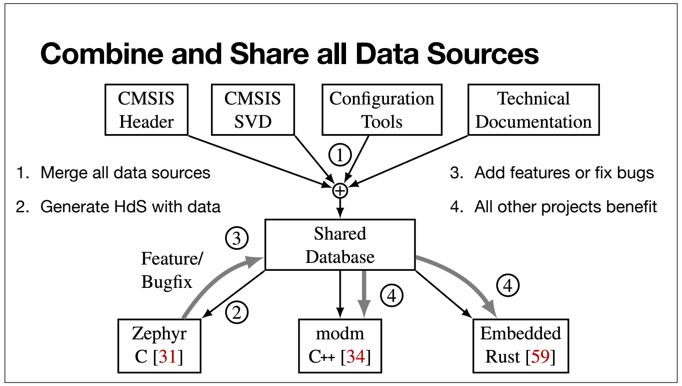
- Today we'll talk about the database part of this construct.



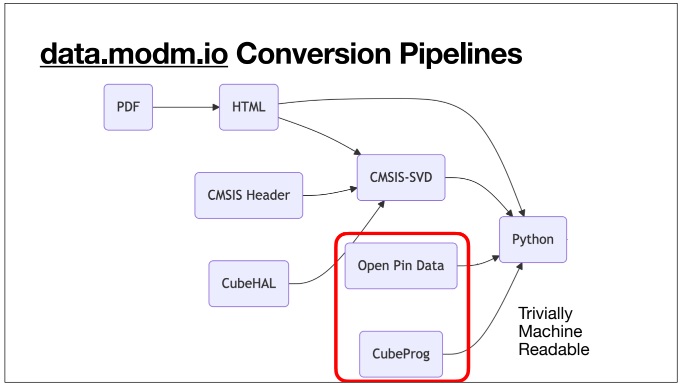
Microcontroller hardware is quite complex nowadays.
Here is a STM32H7 with its many internal busses.
You can see distributed memories in yellow, you can see lots of peripherals, and many DMA engines.
Everything also needs to be externally connected via the pins.
That's a lot of hardware to abstract.



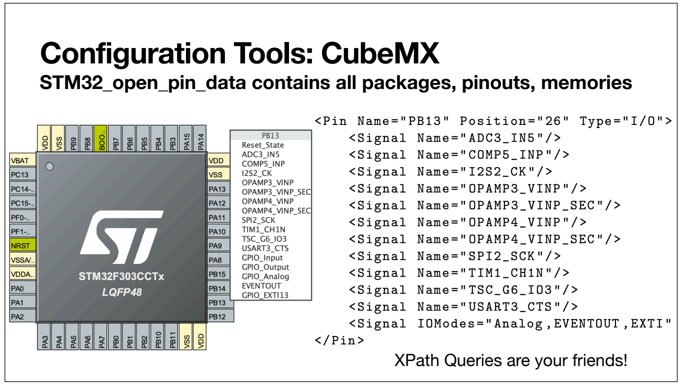
The HAL is actually part of the hardware-dependent software and there's a lot of it.
###
It's also operating systems, external sensors, communication protocols, and bootloaders.
So it's a fairly large topic, not just about microcontrollers itself.



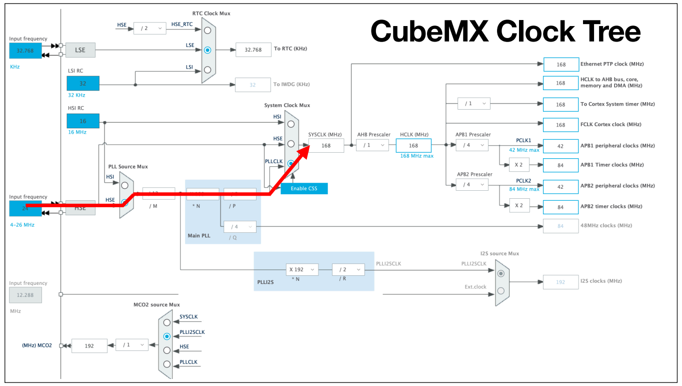
So the idea is to parse every data source I can find and merge it into a single database.
Then I can share this among all my embedded friends: Zephyr, modm and embassy.
And then I would benefit from any of their improvements to the database as well.



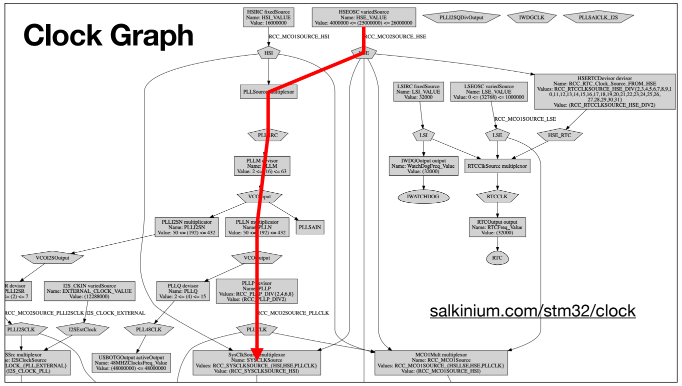
And I decided to make this an open-source project on GitHub.
It's split into individual pipelines, where each data source is converted eventually into Python.
###
Let's first focus on the trivially machine-readable data sources.



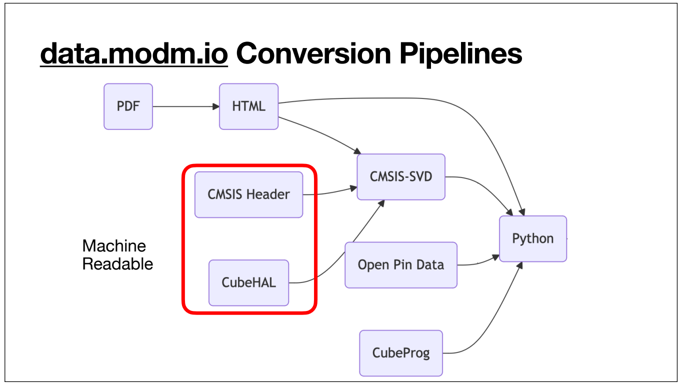
The most well known is the CubeMX GUI application, which allows you to configure the pin functions of the STM32.
This is actually backed by a XML database that STMicro actually publishes on GitHub with a BSD licence.
It contains the entire catalog of STM32 ever made, their package, their pinout, and all alternate functions.
It's undocumented but you can get very far with simple XPath queries.
Many people already use this, including Zephyr, embassy and KiCad to generate HALs and footprints!



However, the CubeMX database also contains a fully annotated graph of the entire STM32 clock tree.
###
A typical configuration is to have an external clock source fed into the PLL, which then increases the clock frequency and feeds it into the system clock, from which most peripherals are powered.



We can also render this clock graph as graphviz graph, and you can see that it contains all frequency limitations that are used to solve the problems of the clock tree in CubeMX.
Here we can follow the same configuration: external clock source gets fed into the PLL and comes out into the system clock.
But now there is a lot more detail visible.
You can also see that this is not really a tree, it's really a graph.
You can find more of these rendered clock graphs on my homepage.



So that was the easy part, let's now focus on more difficult data sources: source code.

Parsing CMSIS Header Files

and converting them back to CMSIS-SVD

1. Parse peripheral structs to reconstruct register order and offset.
2. Resolve numeric values of macros to reconstruct bit field order and offset.
3. Connect TypeDef instantiation (=peripheral) with macros via name matching.

```
#define PERIPH_BASE          0x40000000UL
#define AHB1PERIPH_BASE      (PERIPH_BASE + 0x00020000UL)
#define CRC_BASE             (AHB1PERIPH_BASE + 0x3000UL)
#define CRC                  ((CRC_TypeDef *) CRC_BASE)

/******************** Bit definition for CRC_CR register
#define CRC_CR_RESET_Pos     (0U)
#define CRC_CR_RESET_Msk     (0x1UL << CRC_CR_RESET_Pos)
#define CRC_CR_RESET         CRC_CR_RESET_Msk
```

```
typedef struct
{
  __IO uint32_t DR;
  __IO uint8_t  IDR;
  uint8_t       RESERVED0;
  uint16_t      RESERVED1;
  __IO uint32_t CR;
} CRC_TypeDef;
```

We can convert the CMSIS header files back into a register map:
We know the order and width of the registers from the typedef struct.
We know the order and width of the bit fields from the macros.
And we know the peripheral instance and address from the typedef cast.

This does not give us enumerations of any bit fields unfortunately, since they are simply not in the header files.

Parsing CubeHAL Header Files

CMSIS files are missing Bit Field Enumerations

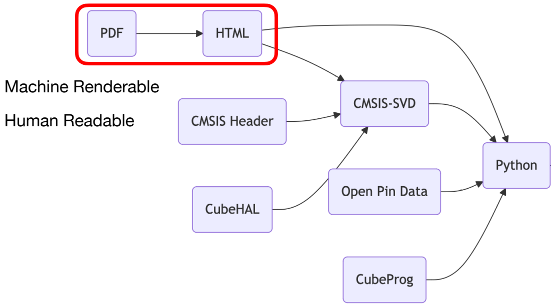
- Neither the STM32 CMSIS-SVD nor CMSIS Header define Bit Field Enumerations.
- We need to parse the Low-Level CubeHAL header files to reconstruct.

```
/******  Bit definition for RCC_CFGR register  ******/
#define RCC_CFGR_SW_Pos      (0U)
#define RCC_CFGR_SW_Msk      (0x3UL << RCC_CFGR_SW_Pos)
#define RCC_CFGR_SW          RCC_CFGR_SW_Msk
#define RCC_CFGR_SW_0        (0x1UL << RCC_CFGR_SW_Pos)
#define RCC_CFGR_SW_1        (0x2UL << RCC_CFGR_SW_Pos)

#define LL_RCC_SYS_CLKSOURCE_HSI    0x00000000U
#define LL_RCC_SYS_CLKSOURCE_HSE    RCC_CFGR_SW_0
#define LL_RCC_SYS_CLKSOURCE_PLL    RCC_CFGR_SW_1
#if defined(RCC_PLLR_SYSCLK_SUPPORT)
#define LL_RCC_SYS_CLKSOURCE_PLLR   (RCC_CFGR_SW_1|RCC_CFGR_SW_0)
#endif
```

For some of the bit field enumerations we need to parse the CubeHAL low-level header files.
Same procedure, we interpret the macros.
###
We can do a reverse lookup to see which macros use the register bit field definitions and then work backwards from that.
Annoying, but doable.
But does not give every bit field enumeration possible.

data.modm.io Conversion Pipelines

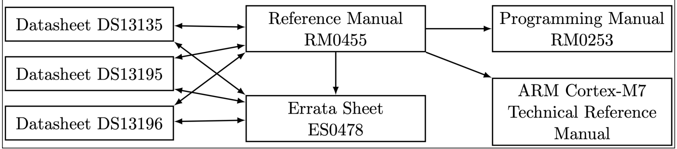


Now for the really hard stuff: parsing PDF datasheets.
PDFs are machine-renderable, but not machine-readable.
There's a lot of research out there on information extraction from PDFs, mostly relating to financial statements.

STMicro PDF Documentation

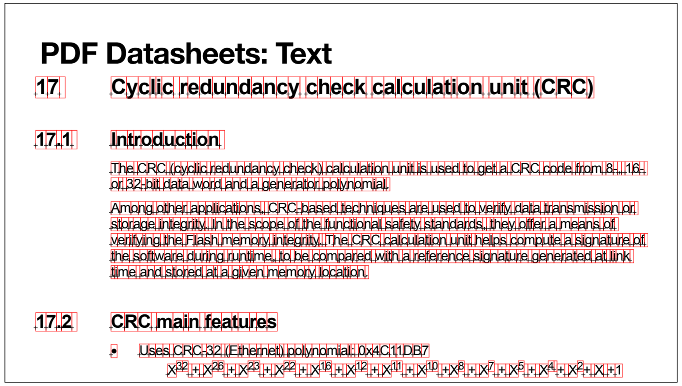
You can look, but you cannot parse

- STMicro publishes >2600 PDFs for documentation: ~15GB on disk.
- You must consult multiple PDFs with thousands of pages: STM32H7A3/B0/B3.
- How hard could it possibly be to make all these PDFs machine-readable?

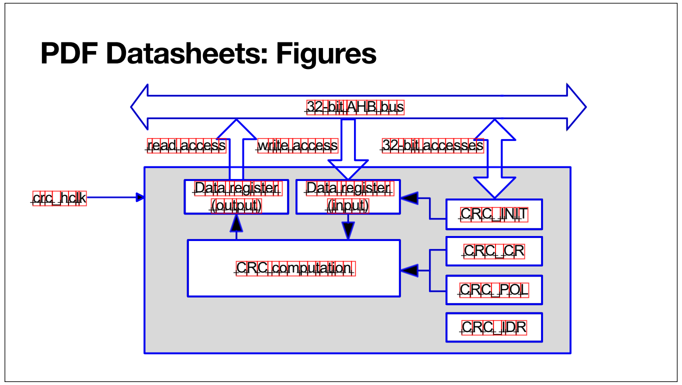


STMicro publishes a lot of PDFs: We are only looking at active components, microcontrollers, sensors, memories.
And there are over 2600 PDFs available: ~15GB.

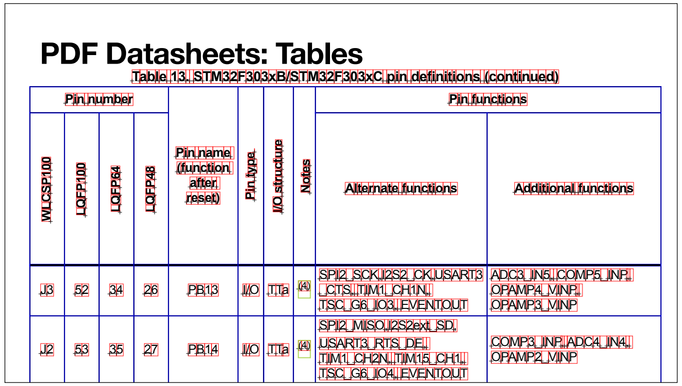
For one microcontroller, a lot of PDFs apply, here the STM32H7 family has 7 PDFs involved.
Nobody reads them all.



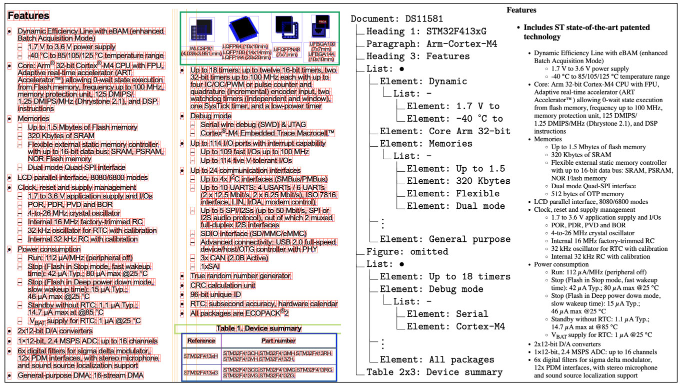
How can we access PDF data?
For text its relatively simple: each glyph is individually positioned on the page. There's no semantics for headings or lists or superscript. It's all just individually positions characters.
THERE IS NO NEED TO OCR PDFs!



Figures are a mix of vector graphics and text. There's no special indication that this is a figure, it must be detected.



A special case of a figure is a table, where the table cells are drawn in vector graphics and the text is placed inside that.
That's why if you just attempt to copy the text of the table into an editor, you usually get garbage.
Note the rotated text in the header, in which order is that copied? It's up to the PDF reader how to copy this text.



Here you can see the first page of a datasheet. We detect the double column layout manually, then convert each side. We need to simplify the problems, so first we

- Convert all 2D information into an abstract syntax tree.
- Then modify that AST to detect the hierarchy of the document and then normalize page breaks.
- Then format it as HTML.

If this sounds like a compiler, it's basically a PDF frontend, then a number of AST passes, then a HTML backend. And this actually works really well.

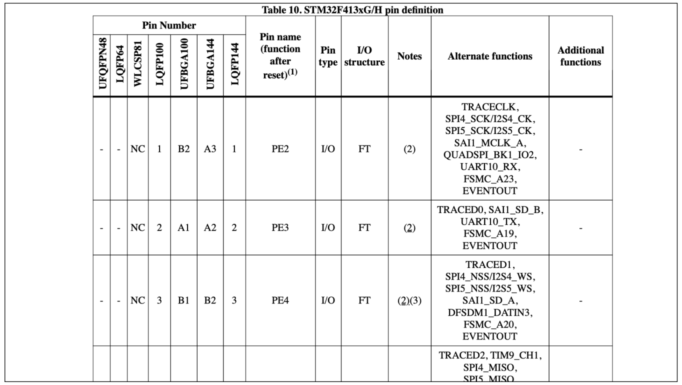


Table 10. STM32F413xG/H pin definition

Pin Number							Pin name (function after reset)(1)	Pin type	I/O structure	Notes	Alternate functions	Additional functions
UFQFPN48	LQFP64	WLCSP81	LQFP100	UFBGA100	UFBGA144	LQFP144						
-	-	NC	1	B2	A3	1	PE2	I/O	FT	(2)	TRACECLK, SPI4_SCK/I2S4_CK, SPI5_SCK/I2S5_CK, SAI1_MCLK_A, QUADSPI_BK1_IO2, UART10_RX, FSMC_A23, EVENTOUT	-
-	-	NC	2	A1	A2	2	PE3	I/O	FT	(2)	TRACED0, SAI1_SD_B, UART10_TX, FSMC_A19, EVENTOUT	-
-	-	NC	3	B1	B2	3	PE4	I/O	FT	(2)(3)	TRACED1, SPI4_NSS/I2S4_WS, SPI5_NSS/I2S5_WS, SAI1_SD_A, DFSDM1_DATIN3, FSMC_A20, EVENTOUT	-
											TRACED2, TIM9_CH1, SPI4_MISO, SPI5_MISO	

This is the result, for example the pin definition table in the datasheet.
This is a pure HTML table with minimal CSS to look similiar to the PDF.
All of the data is converted as is including line breaks.

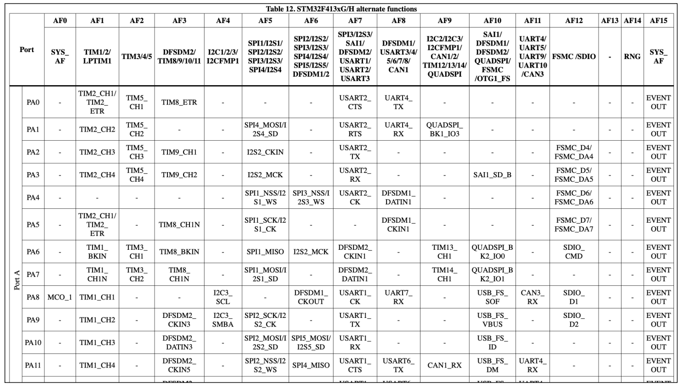


Table 12. STM32F413xG/H alternate functions

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SYS_AF	TIM1/2/LPTIM1	TIM3/4/5	DFSDM2/TIM8/9/10/11	I2C1/2/3/I2CFMP1	SPI1/I2S1/SPI2/I2S2/SPI3/I2S3/SPI4/I2S4	SPI2/I2S2/SPI3/I2S3/SPI4/I2S4/SPI5/I2S5/DFSDM1/2	SPI3/I2S3/SAI1/DFSDM2/USART1/USART2/USART3	DFSDM1/USART3/4/5/6/7/8/CAN1	I2C2/I2C3/I2CFMP1/CAN1/2/TIM12/13/14/QUADSPI	SAI1/DFSDM1/DFSDM2/QUADSPI/FSMC/OTG1_FS	UART4/UART5/UART9/UART10/CAN3	FSMC /SDIO	-	RNG	SYS_AF
Port A	PA0	-	TIM2_CH1/TIM2_ETR	TIM5_CH1	TIM8_ETR	-	-	-	USART2_CTS	UART4_TX	-	-	-	-	-	-	EVENTOUT
Port A	PA1	-	TIM2_CH2	TIM5_CH2	-	-	SPI4_MOSI/I2S4_SD	-	USART2_RTS	UART4_RX	QUADSPI_BK1_IO3	-	-	-	-	-	EVENTOUT
Port A	PA2	-	TIM2_CH3	TIM5_CH3	TIM9_CH1	-	I2S2_CKIN	-	USART2_TX	-	-	-	-	FSMC_D4/FSMC_DA4	-	-	EVENTOUT
Port A	PA3	-	TIM2_CH4	TIM5_CH4	TIM9_CH2	-	I2S2_MCK	-	USART2_RX	-	-	SAI1_SD_B	-	FSMC_D5/FSMC_DA5	-	-	EVENTOUT
Port A	PA4	-	-	-	-	-	SPI1_NSS/I2S1_WS	SPI3_NSS/I2S3_WS	USART2_CK	DFSDM1_DATIN1	-	-	-	FSMC_D6/FSMC_DA6	-	-	EVENTOUT
Port A	PA5	-	TIM2_CH1/TIM2_ETR	-	TIM8_CH1N	-	SPI1_SCK/I2S1_CK	-	-	DFSDM1_CKIN1	-	-	-	FSMC_D7/FSMC_DA7	-	-	EVENTOUT
Port A	PA6	-	TIM1_BKIN	TIM3_CH1	TIM8_BKIN	-	SPI1_MISO	I2S2_MCK	DFSDM2_CKIN1	-	TIM13_CH1	QUADSPI_BK2_IO0	-	SDIO_CMD	-	-	EVENTOUT
Port A	PA7	-	TIM1_CH1N	TIM3_CH2	TIM8_CH1N	-	SPI1_MOSI/I2S1_SD	-	DFSDM2_DATIN1	-	TIM14_CH1	QUADSPI_BK2_IO1	-	-	-	-	EVENTOUT
Port A	PA8	MCO_1	TIM1_CH1	-	-	I2C3_SCL	-	DFSDM1_CKOUT	USART1_CK	UART7_RX	-	USB_FS_SOF	CAN3_RX	SDIO_D1	-	-	EVENTOUT
Port A	PA9	-	TIM1_CH2	-	DFSDM2_CKIN3	I2C3_SMBA	SPI2_SCK/I2S2_CK	-	USART1_TX	-	-	USB_FS_VBUS	-	SDIO_D2	-	-	EVENTOUT
Port A	PA10	-	TIM1_CH3	-	DFSDM2_DATIN3	-	SPI2_MOSI/I2S2_SD	SPI5_MOSI/I2S5_SD	USART1_RX	-	-	USB_FS_ID	-	-	-	-	EVENTOUT
Port A	PA11	-	TIM1_CH4	-	DFSDM2_CKIN5	-	SPI2_NSS/I2S2_WS	SPI4_MISO	USART1_CTS	USART6_TX	CAN1_RX	USB_FS_DM	UART4_RX	-	-	-	EVENTOUT

Here is the alternate function table.
This is normally broken up across many pages, in the HTML its just one long table.

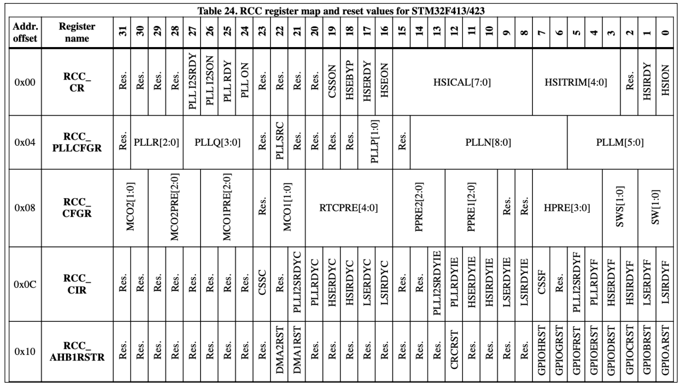
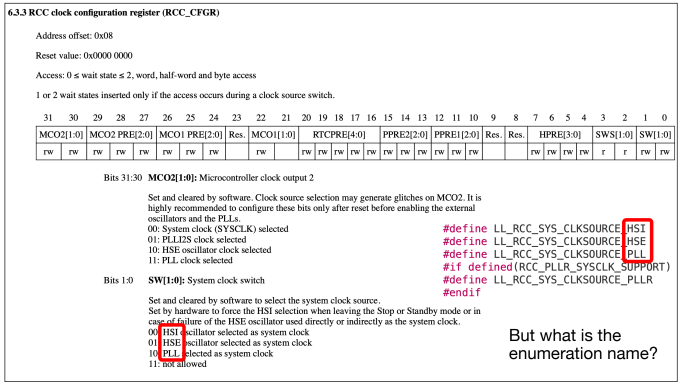


Table 24. RCC register map and reset values for STM32F413/423

Addr. 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	**RCC_ CR**	Res.	Res.	Res.	Res.	PLL I2SRDY	PLL I2SON	PLL RDY	PLL ON	Res.	Res.	Res.	Res.	CSSON	HSEBYP	HSERDY	HSEON	HSICAL[7:0]								HSITRIM[4:0]					Res.	HSIRDY	HSION
0x04	**RCC_ PLLCFGR**	Res.	PLLR[2:0]			PLLQ[3:0]				Res.	PLLSRC	Res.	Res.	Res.	Res.	PLLP[1:0]		Res.	PLLN[8:0]									PLLM[5:0]					
0x08	**RCC_ CFGR**	MCO2[1:0]		MCO2PRE[2:0]			MCO1PRE[2:0]			Res.	MCO1[1:0]		RTCPRE[4:0]					PPRE2[2:0]			PPRE1[2:0]			Res.	Res.	HPRE[3:0]				SWS[1:0]		SW[1:0]	
0x0C	**RCC_ CIR**	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	CSSC	Res.	PLLI2SRDYC	PLLRDYC	HSERDYC	HSIRDYC	LSERDYC	LSIRDYC	Res.	Res.	PLLI2SRDYIE	PLLRDYIE	HSERDYIE	HSIRDYIE	LSERDYIE	LSIRDYIE	CSSF	Res.	PLLI2SRDYF	PLLRDYF	HSERDYF	HSIRDYF	LSERDYF	LSIRDYF
0x10	**RCC_ AHB1RSTR**	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	DMA2RST	DMA1RST	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	CRCRST	Res.	Res.	Res.	Res.	GPIOHRST	GPIOGRST	GPIOFRST	GPIOERST	GPIODRST	GPIOCRST	GPIOBRST	GPIOARST

We also find the register layout information again for each peripheral. Note that the text is rotated only by CSS, so the table data is still easily accessible in HTML.



6.3.3 RCC clock configuration register (RCC_CFGR)

Address offset: 0x08

Reset value: 0x0000 0000

Access: 0 ≤ wait state ≤ 2, word, half-word and byte access

1 or 2 wait states inserted only if the access occurs during a clock source switch.

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
MCO2[1:0]		MCO2 PRE[2:0]			MCO1 PRE[2:0]			Res.	MCO1[1:0]		RTCPRE[4:0]					PPRE2[2:0]			PPRE1[2:0]			Res.	Res.	HPRE[3:0]				SWS[1:0]		SW[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw			rw	rw	rw	rw	r	r	rw	rw

Bits 31:30 **MCO2[1:0]:** Microcontroller clock output 2

Set and cleared by software. Clock source selection may generate glitches on MCO2. It is highly recommended to configure these bits only after reset before enabling the external oscillators and the PLLs.
00: System clock (SYSCLK) selected
01: PLLI2S clock selected
10: HSE oscillator clock selected
11: PLL clock selected

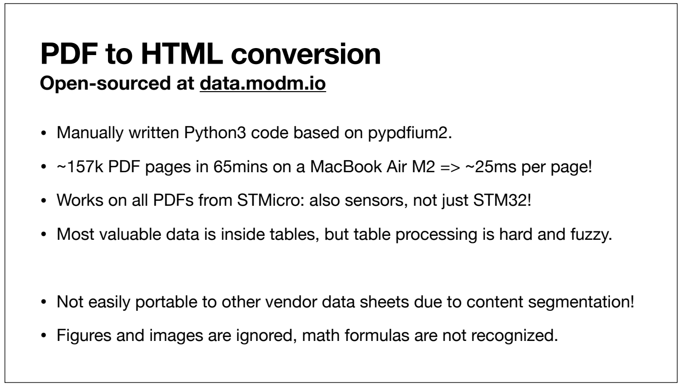
Bits 1:0 **SW[1:0]:** System clock switch

Set and cleared by software to select the system clock source.
Set by hardware to force the HSI selection when leaving the Stop or Standby mode or in case of failure of the HSE oscillator used directly or indirectly as the system clock.
00: HSI oscillator selected as system clock
01: HSE oscillator selected as system clock
10: PLL selected as system clock
11: not allowed

```
#define LL_RCC_SYS_CLKSOURCE_HSI
#define LL_RCC_SYS_CLKSOURCE_HSE
#define LL_RCC_SYS_CLKSOURCE_PLL
#if defined(RCC_PLLR_SYSCLK_SUPPORT)
#define LL_RCC_SYS_CLKSOURCE_PLLR
#endif
```

But what is the enumeration name?

And I can even convert the invisible table of the bit field and their enumerations description as a HTML table.
###
And indeed this is accurate, the PLLR does not exist for this device, so the guard in the CubeHAL header is actually correct.
###
Unfortunately we have the enumeration value and description, but not a name. That would need to be generated from the description and that not always easy to do automatically.



I'm very happy with this pipeline. It's written in Python3 using native bindings for pdfium (PDF renderer in Chrome). It's entirely deterministic, so the translated HTML is byte reproducible. It's also very fast with 25ms per page. All STMicro PDFs are supported, including sensors.
Some compromises: it's not easily portable to other vendors, since the format recognition is hardcoded. I'm only interested in tables and text, so figures are completely ignore (should be converted to SVG) and math formulas are turned into garbage.

PDF Formatting Mistakes

- The PDF sometimes have formatting mistakes: tables with missing cell borders.
- Apply git patch to HTML result: works, but fragile.

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

Even though it's deterministic and reproducible, some formatting mistakes are not easy to fix.
A classic is missing border cells in tables.
We could try to infer cells from whitespace analysis between text, but it's fairly unreliable for such issues.
I just apply a git patch to the HTML, which works because the HTML is so reproducible.

Interpreting Datasheet Tables

Substitution hell to fix typos in PDFs

```
package = package.replace("UFBGA/TFBGA64", "UFBGA64/TFBGA64")
package = package.replace("LPQF", "LQFP").replace("TSSPOP", "TSSOP")
package = package.replace("UFBG100", "UFBGA100").replace("UBGA", "UFBGA")
package = package.replace("UQFN", "UFQFPN").replace("WLCSP20L", "WLCSP20")
package = package.replace("UFQFPN48E", "UFQFPN48+E").replace("UFQFN", "UFQFPN")
package = package.replace("LQFP48 SMPS<br>UFQFPN48 SMPS", "LQFP48/UFQFPN48+SMPS")
package = package.replace("LQFP<br>64", "LQFP64").replace("LQFP<br>48", "LQFP48")
package = package.replace("UFQFPN<br>32", "UFQFPN32").replace("UFQFP<br>N48", "UFQFPN48")
package = package.replace("WLCSP<br>25", "WLCSP25")
```

But, still interpreting the HTML tables was actually way more annoying that converting the PDF.
You can to clean the data, because of many typos and random line breaks.
Here I'm using text substitution.

Interpreting Datasheet Tables

Substitution hell to fix typos in PDFs, now with more RegEx

```
patterns = {
    r" +": "", r".*?\(([A-Z]+|DMA2D)\).*?": r"\1",
    r"Reserved|Port|Power|Registers|Reset|\(.*?\)|_REG": "",
    r"(\d)/I2S\d": r"\1", r"/I2S|CANMessageRAM|Cortex-M4|I2S\dext|^GPV$": "",
    r"Ethernet": "ETH", r"Flash": "FLASH", r"(?i).*ETHERNET.*": "ETH",
    r"(?i)Firewall": "FW", r"HDMI-|": "", "SPDIF-RX": "SPDIFRX",
    r"SPI2S2": "SPI2", "Tamper": "TAMP", "TT-FDCAN": "FDCAN",
    r"USBOTG([FH])S": r"USB_OTG_\1S", "LCD-TFT": "LTDC", "DSIHOST": "DSI",
    "TIMER": "TIM", r"^VREF$": "VREFBUF", "DelayBlock": "DLYB",
    "I/O": "M", "I/O": "", "DAC1/2": "DAC12",
    r"[a-z]": ""
}
```

And then I decided to use Regex to fix many patterns in the register definitions.

Interpreting Datasheet Tables

RegEx hell to fix typos for bit field reconstruction

```
off_replace = {r" +": "", "0x000x00": "0x00", "to": "-", "x": "*", r"\(\d+\)": ""}
dom_replace = {r"Register +size": "Bit position"}
reg_replace = {
    r" +|\.+": "", r"\(COM(\d)\)": r"_COM\1",
    r"^[Rr]es$||0x[\da-fA-FXx]+|\(.*?\)|-": "",
    r"(?i)reserved|resetvalue.*": "", "enabled": "_EN", "disabled": "_DIS",
    r"(?i)Outputcomparemode": "_Output", "(?i)Inputcapturemode": "_Input", "mode": "",
    r"^TG_FS_": "OTG_FS_", "toRTC": "RTC", "SPI2S_": "SPI_",
    r"andTIM\d+_.*": "", r"x=[\d,]+": ""}
fld_replace = {
    r"\] +\d+(th|rd|nd|st)": "]", r" +|\.+|\[.*?\]|\[?\d+:\d+\]?|\(.*?\)|-|^[\dXx]+$|%|__|:0\]": "",
    r"Dataregister|Independentdataregister": "DATA",
    r"Framefilterreg0.*": "FRAME_FILTER_REG",
    r"[Rr]es(erved)?|[Rr]egular|x_x(bits)?|NotAvailable|RefertoSection\d+:Comparator": "",
    r"Sampletimebits|Injectedchannelsequence|channelsequence|conversioninregularsequencebits": "",
    r"conversioninsequencebits|conversionininjectedsequencebits|or|first|second|third|fourth": ""}
bit_replace = {r".*:": ""}
glo_replace = {r"[Rr]eserved": ""}
```

And this then got a little out of hand for the bit field enumerations.
I do not recommend using regex for this, there needs to be a better way.

Evaluation of Data Sources

Actual Science! OMG

Extracted 4 datasets with increasing complexity for ~2700 STM32 devices:

1. Interrupt vector table: PDF vs CMSIS Header
2. Package and pinout: PDF vs CubeMX Database
3. Pin functions: PDF vs CubeMX Database
4. MMIO register map and descriptions: PDF vs SVD vs Header

Compare PDF against machine-readable sources: Headers, SVD, CubeMX

Ok, but enough regexing around. Let's do some actual science!
We want to find out how accurate our data import pipelines actually are.
So we're going to compare the machine-readable data against the PDF data.
We evaluated in detail four data sets for this.
We fixed obvious spelling mistakes, but only as long as the fix is unambigious.

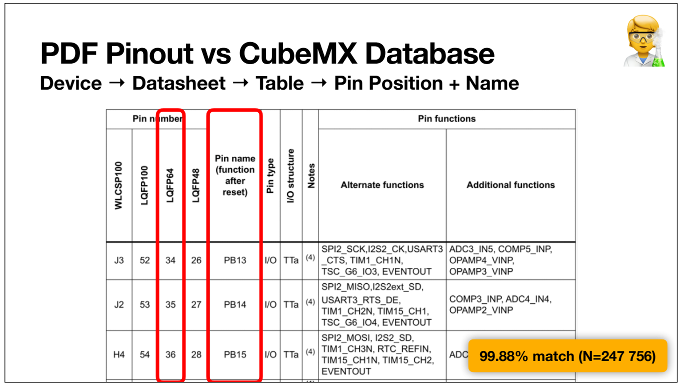
PDF Interrupt Table vs CMSIS Header

Device → Reference Manual → Table → Position + Name

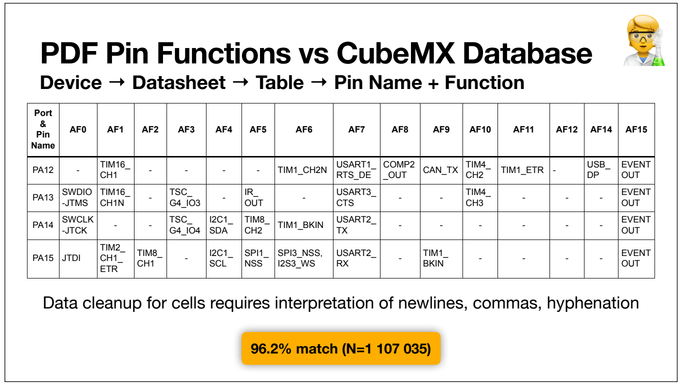
Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000 0000
-	-3	Fixed	Reset	Reset	0x0000 0004
-	-2	Fixed	NMI	Non maskable interrupt. The RCC clock security system (CSS) and the RAM parity check are linked to the NMI vector.	0x0000 0008
-	-1	Fixed	HardFault	All classes 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)	[illegible]
2	9	Settable	RTC	RTC interrupts (combined EXTI lines 17, [illegible]	[illegible]

98.8% match (N=190 109)

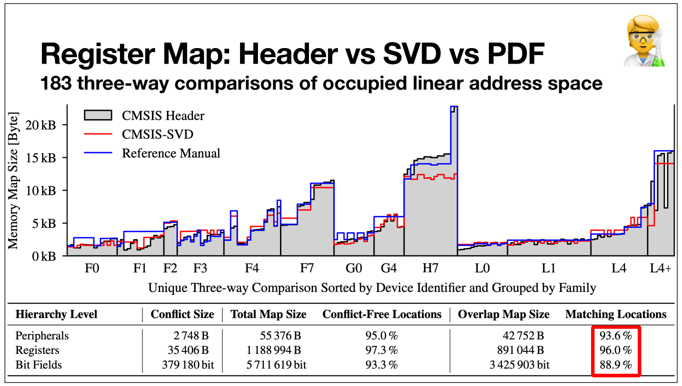
This is fairly easy: it's the interrupt vector table for STM32 microcontrollers.
Quite good.



The package pinout was extremely accurate. This is just the pin position and name on the package.



The signals are a bit more interesting, this is our first 2D data structure.
We looked at over a million signals in our dataset, didn't find any issues with our PDF-to-HTML pipeline, but many issues in the CubeMX database, as well as formatting issues in the PDF.
Still very accurate.

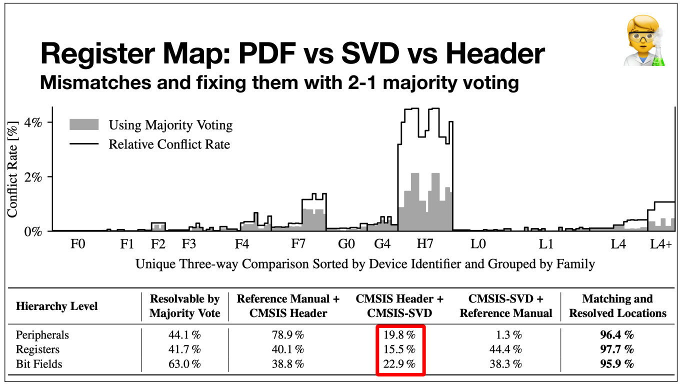


And finally we compared the register maps reconstructed form the CMSIS Header, vs the CMSIS-SVD vs the PDF.
And this was the most interesting part, because it shows that STMicro has three slightly different datasets for their hardware.

As a proxy for completeness we can look at the size of the register map. How many bytes are occupied by the registers.
You can see that the register map reconstructed from the reference manual is very accurate.

BUT the device resolution is not great:

- the CMSIS headers create 183 register maps,
- The CMSIS-SVD files only 100 register maps, and
- The PDFs only 53 register maps.

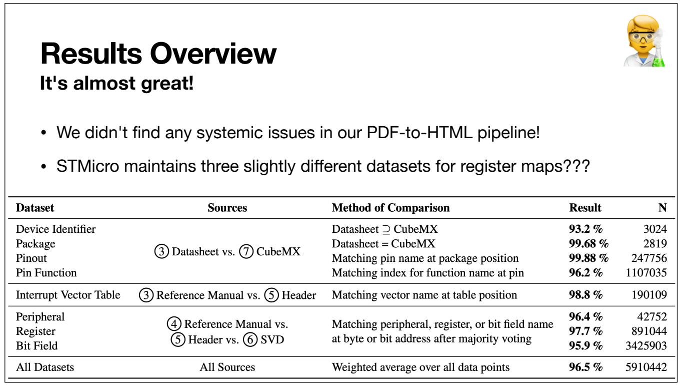


Hierarchy Level	Resolvable by Majority Vote	Reference Manual + CMSIS Header	CMSIS Header + CMSIS-SVD	CMSIS-SVD + Reference Manual	Matching and Resolved Locations
Peripherals	44.1 %	78.9 %	19.8 %	1.3 %	**96.4 %**
Registers	41.7 %	40.1 %	15.5 %	44.4 %	**97.7 %**
Bit Fields	63.0 %	38.8 %	22.9 %	38.3 %	**95.9 %**

Here is the conflict rate in more detail. We can see that the complex families like F7, H7 and L4 have the most conflicts overall.
Since we have three differing data sources, we can do majority voting and see how many differing registers we can fix.

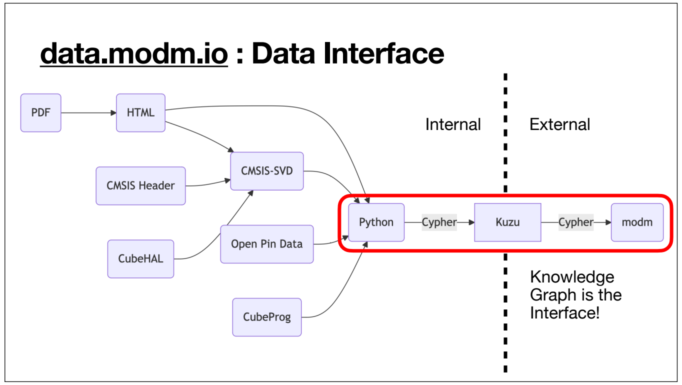
It works well for simple families, and improves the matching data quite a bit, but we can also see that the combination of CMSIS header and CMSIS-SVD is the least successful in majority voting.

This is very weird since the CMSIS header files are supposed to be generated from the CMSIS SVD files.

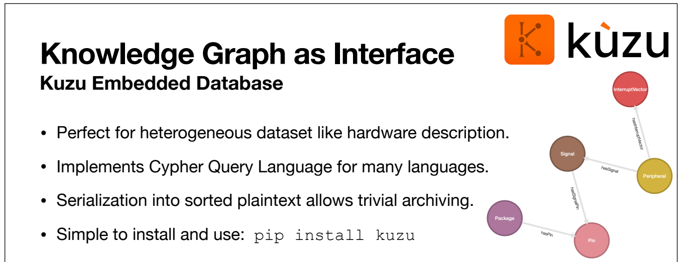


Dataset	Sources	Method of Comparison	Result	N
Device Identifier	3 Datasheet vs. 7 CubeMX	Datasheet ⊇ CubeMX	**93.2 %**	3024
Package		Datasheet = CubeMX	**99.68 %**	2819
Pinout		Matching pin name at package position	**99.88 %**	247756
Pin Function		Matching index for function name at pin	**96.2 %**	1107035
Interrupt Vector Table	3 Reference Manual vs. 5 Header	Matching vector name at table position	**98.8 %**	190109
Peripheral	4 Reference Manual vs. 5 Header vs. 6 SVD	Matching peripheral, register, or bit field name at byte or bit address after majority voting	**96.4 %**	42752
Register			**97.7 %**	891044
Bit Field			**95.9 %**	3425903
All Datasets	All Sources	Weighted average over all data points	**96.5 %**	5910442

Overall, the machine-readable data is very accurate with 96.5% match at 5.9 million data points.
As a result, I would not use the PDF or the CMSIS-SVD files as primary data sources unless necessary.
Extract as much as possible from the CubeMX database and CMSIS headers instead.



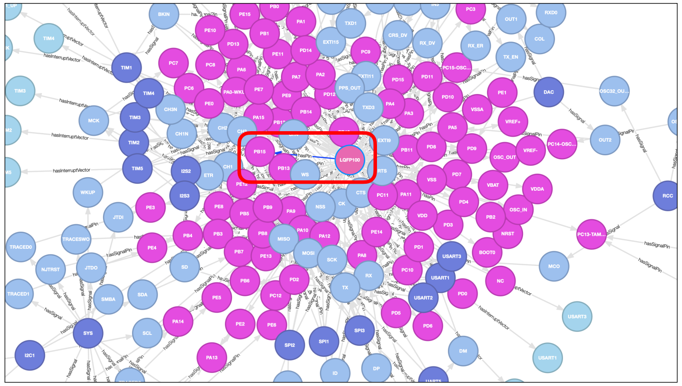
So the question is how to we make this data accessible?
We have a highly heterogeneous dataset, which includes clock graphs, so why not use a graph database?
The graph database then also acts as the interface to the external world.



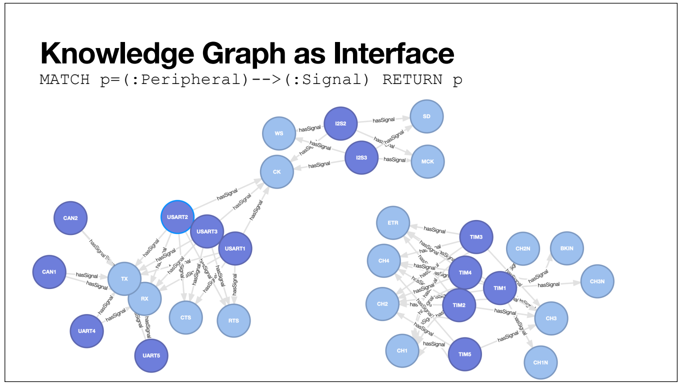

```
kuzu> MATCH (:Package)-[po:hasPin]->(pi:Pin)-[af:hasAlternateFunction]->(s:Signal)<--(pe:Peripheral)
      RETURN pi.name, po.position, pe.name, s.name, af.index;
```

pi.name STRING	po.position STRING	pe.name STRING	s.name STRING	af.index UINT8
PA0/WKUP	N3	ETH	CRS	11
PH2	K4	ETH	CRS	11
PA0/WKUP	N3	TIM1	CH1	1

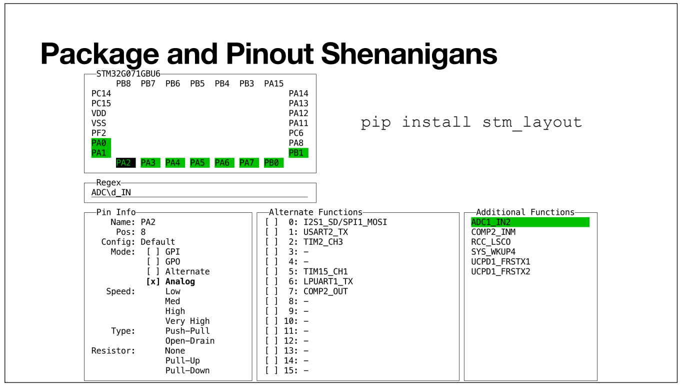
I chose Kuzu, which is a small and fast embedded graph database that implements the Cypher language.
It's easy to install and use and comes with many language bindings: C/C++, Rust, Python, Web Assembly.
You can see a part of the schema on the right and a cypher query at the bottom showing alternate functions.
The query returns a table which you can then use to generate code.



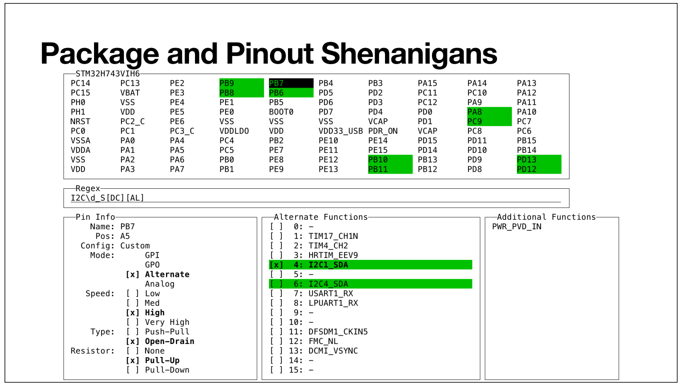
There a browser based explorer tool including graph visualization.
Here you can see the package node, surrounded by all the pins and their corresponding signals.
This is a bit chaotic



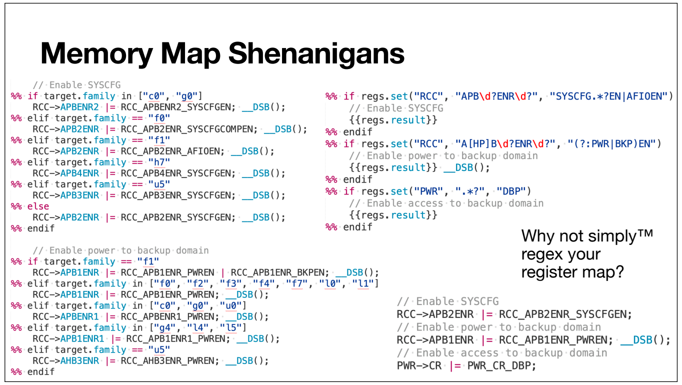
We can also query only a part of the graph, like specific relations.
Here we query all relations between peripherals and signals.



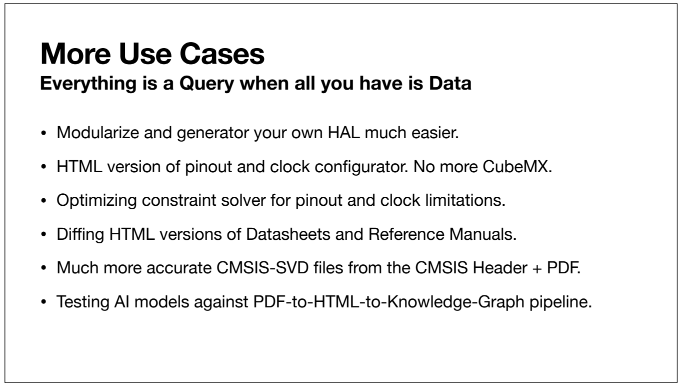
But there are also many other things you can do with this code.
For example REGEX your alternate functions!
This is a smol TUI tool based on the CubeMX database, here showing all the pins that have an ADC input signal.



This also works for BGA pins, here searching for all pins with I2C data and clock signals.
This is very useful to quickly find alternate functions, since the CubeMX gui is not great for searching like this.



I apologize to your eyeballs. This is code from modm.
modm uses Python Jinja templates to generate startup code, where we need to enable the clock to the system config and power peripherals.
And it's very annoying since the bit is in different registers depending on the family.
###
So instead, why not just regex the register?
###
This abomination actually works really well…



There are many more use cases that I didn't go into.
A nice one would be to create a simpler CubeMX application as a HTML page, something that can configure and generate code for other HALs.
You can apply a SAT solver to the database of course to solve design constraints and help with parts selection.
You can now diff PDFs via the HTML version.
And you can generate much more accurate SVD files than the official ones.
If you think your AI model can do better, I've basically built you a benchmark.
BEWARE.

Conclusion
and Future Work

- STMicro publishes several machine-readable data sources on GitHub!
- Parsing machine-readable data sources is easy and very accurate.
- Parsing PDF/HTML is difficult due to typos and formatting mistakes.
- modm-data: PDF2HTML works well, rest is "academic" code quality.

- Knowledge Graphs are a good database for heterogeneous data sets.
- Documentation and discoverability of Knowledge Graph Ontology is difficult.

There's a lot of machine-readable data on GitHub, it needs to be put in a good database. Knowledge graphs are still pretty niche.
Parsing PDFs is hard because of humans, rather than accessing the PDF. Some fuzzy matching required.
The PDF2HTML pipeline works really well in modm-data, the rest needs to be rewritten.

Data Mining Hardware Descriptions
Questions?

Niklas Hauser likes data science.

Homepage: salkinium.com

Fediverse: @salkinium@chaos.social

Code: github.com/salkinium

Thesis: salkinium.com/master.pdf

Paper: salkinium.com/hp23.pdf (peer-reviewed!)

GitHub: github.com/modm-io/modm-data

data.modm.io

If you want to know more details, including citations, check out master thesis and my peer-reviewed paper.
All the code is public and somewhat documented.
Do you have Questions?