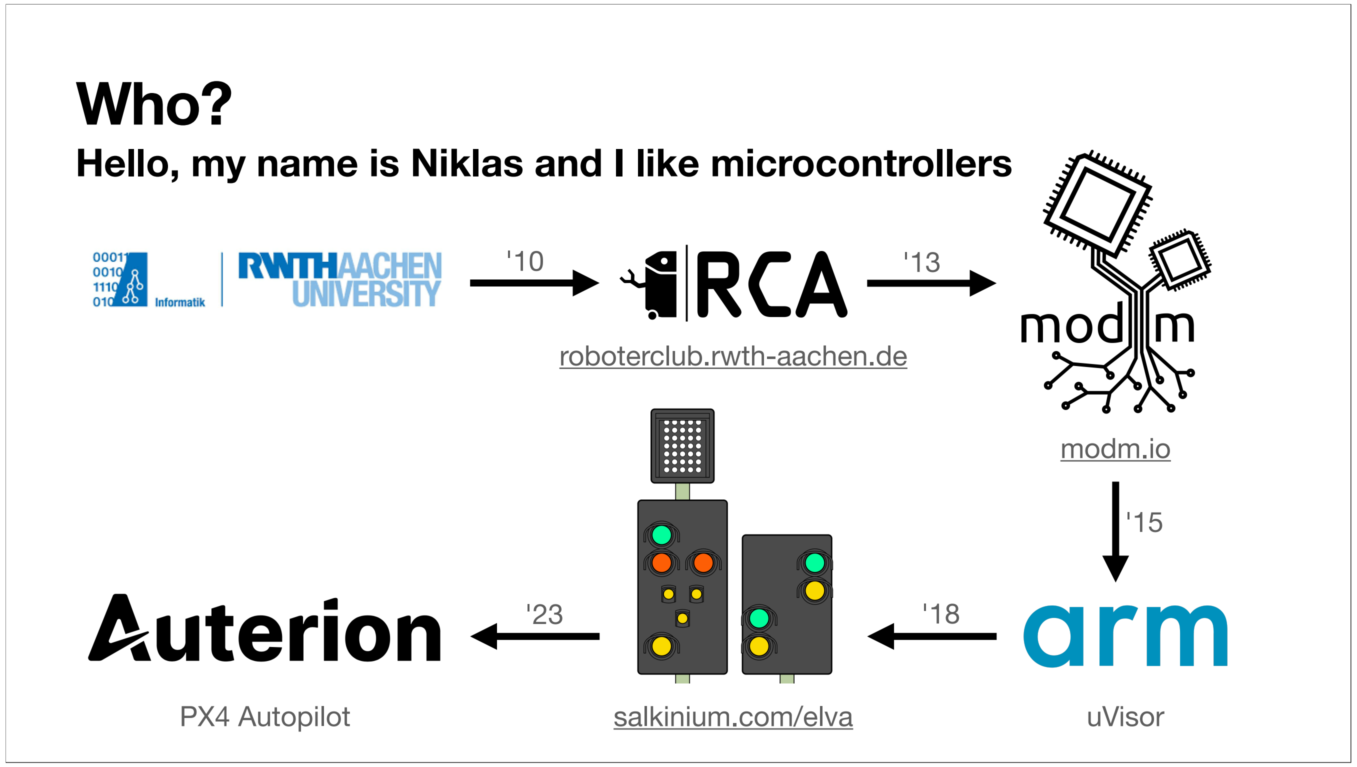
Debugging Microcontrollers

Debugging and Profiling ARM Cortex-M with GDB and Python.

Niklas Hauser, CCCamp23

Thank you for the introduction.
Thanks to Milliways for the opportunity to talk here.
And thanks to the Camp for the general awesomeness.



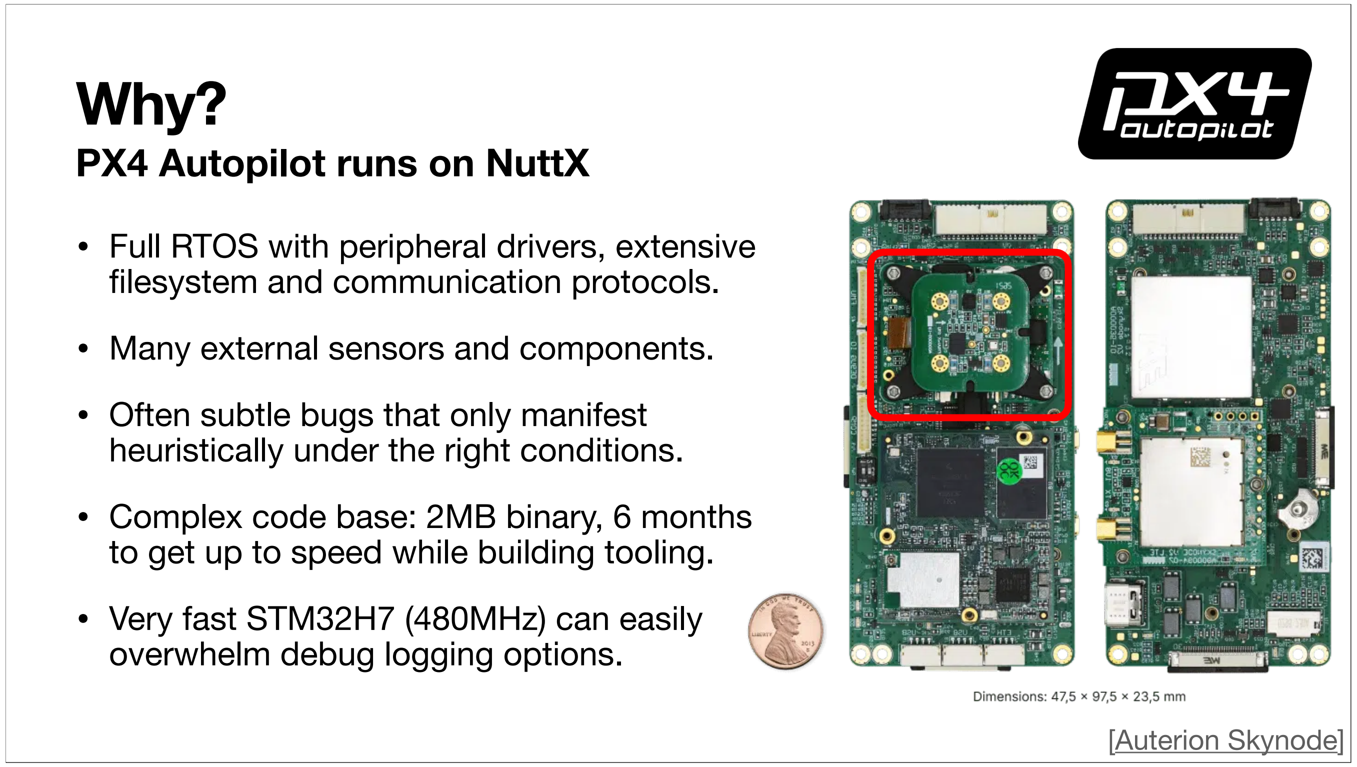
Who?

Hello, my name is Niklas and I like microcontrollers

The short introduction is: My name is Niklas and I like microcontrollers.
The longer introduction is:

- I started studying Computer Science at the RWTH Aachen University some time ago.
- By "studying" I mean, building autonomous robots at the Roboterclub Aachen e.V. starting in 2010.
- For this project, we built a C++ library which today is known as modm.io, a C++23 library generator that supports 3700+ Cortex-M devices, which I co-maintain.
- I then got bored and started at ARM working on ARMv8-M sandboxing, before I got bored of that and returned to the university to study for my masters degree.
- And by "studying" I mean, digitizing the railway signalling lab in the transportation engineering department and designing a 32nd-scale modular signalling system out of PCBs and 3D prints.
- I finished my masters degree and now work at Auterion debugging the open-source PX4 Autopilot for commercial drones.

In summary: My name is Niklas and I like microcontrollers.



Why?

PX4 Autopilot runs on NuttX

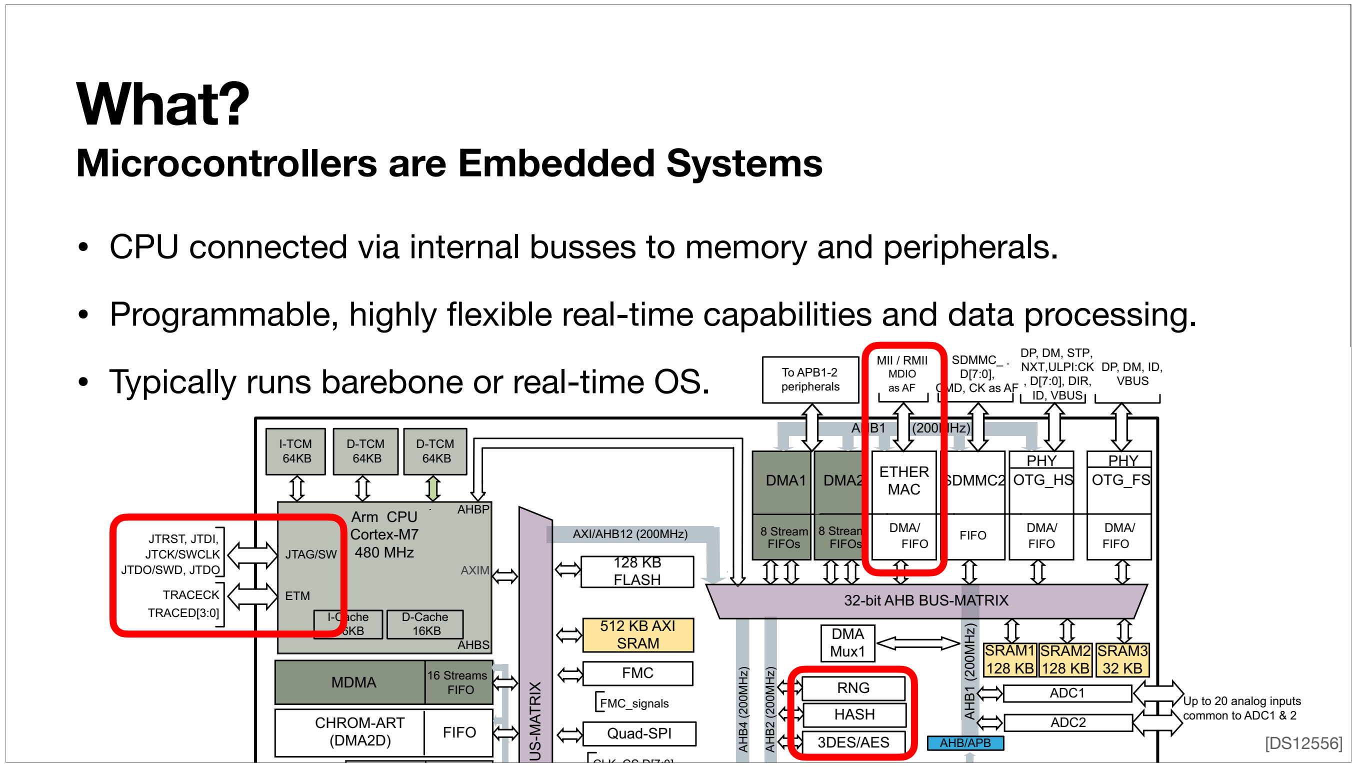
- Full RTOS with peripheral drivers, extensive filesystem and communication protocols.
- Many external sensors and components.
- Often subtle bugs that only manifest heuristically under the right conditions.
- Complex code base: 2MB binary, 6 months to get up to speed while building tooling.
- Very fast STM32H7 (480MHz) can easily overwhelm debug logging options.

Dimensions: 47,5 × 97,5 × 23,5 mm

[Auterion Skynode]

For my day job I debug the Skynode Autopilot that you can see here on the right.

- Contains a very fast STM32H7 in the top left with a lot of sensors.
- The rest of the board is for linux and wired and wireless connectivity.
- The Autopilot runs PX4, which is an open-source project.
- NuttX RTOS is complex and has subtle bugs.
- My job… is just… debug.
- Difficult because large code base and fast processor.
- I want to share how I debug and some tools I wrote to help me.



What?

Microcontrollers are Embedded Systems

- CPU connected via internal busses to memory and peripherals.
- Programmable, highly flexible real-time capabilities and data processing.
- Typically runs barebone or real-time OS.

[DS12556]

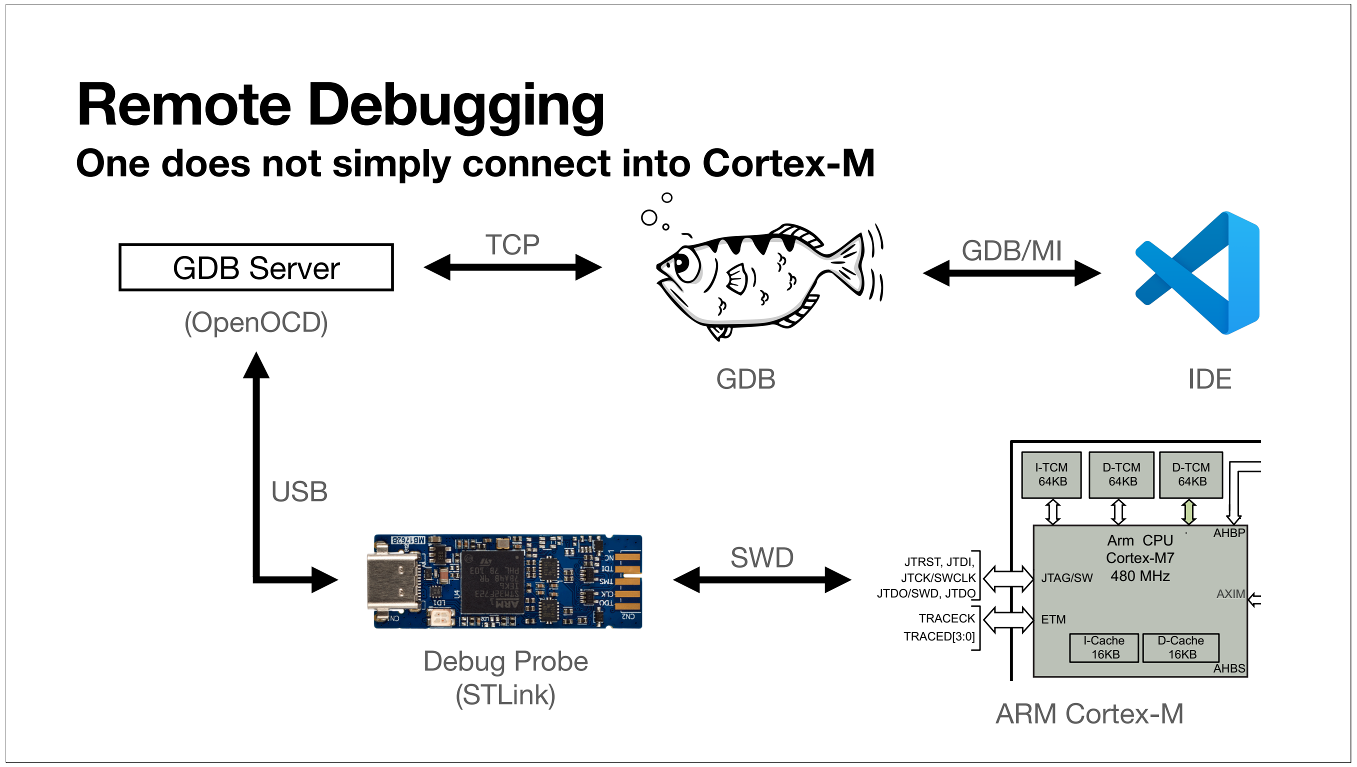
This talk is about microcontrollers, specifically with the ARM Cortex-M architecture.

Microcontrollers contain a microprocessor, here a Cortex-M7 in light green on the left, connected via a bus system to non-volatile memories, like Flash, and volatile memories like SRAM (yellow), as well as a number of special purpose peripherals.
These architectures can be quite complex, here you can see the CPU to several busses in gray, connected to several peripherals on different clock domains. You can also see Direct Memory Access (DMA) peripherals, that can move data around without CPU interaction.

Peripherals can be internal, like the Random Number Generator (RNG) down here,
or external, like the Ethernet Media Access Controller up there which connects over Media Intependent-Interface (MII) to an external PHY via the microcontroller pins.

Memories typically in the kilobytes, high-end devices can have a few megabytes, so microcontrollers cannot natively run Linux.
Allows for highly configurable, very low-latency responses to internal and external events via an RTOS.

On the left, the CPU itself also has some external signals for debugging, and in this talk I'll show you how to use them.



Remote Debugging

One does not simply connect into Cortex-M

If you want to debug a microcontroller here on the bottom right

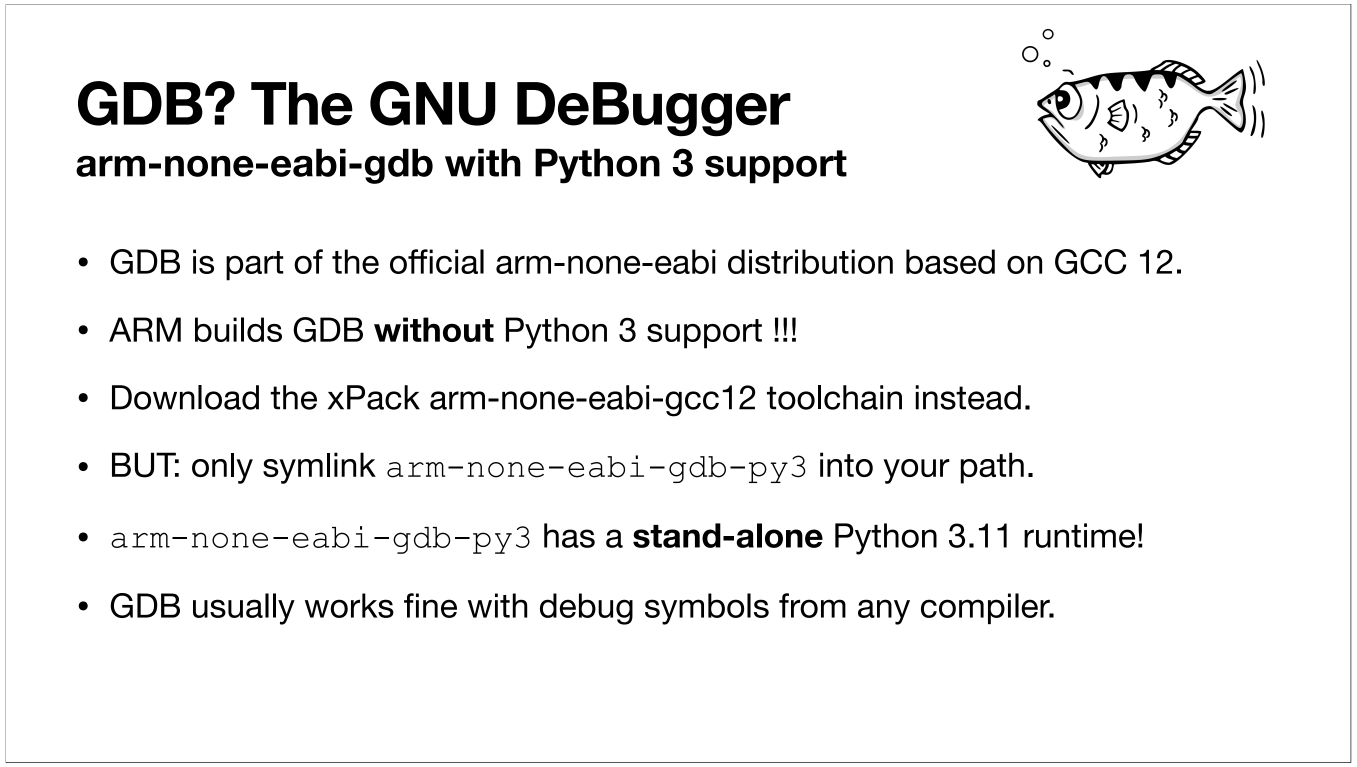
- You need to connect the Serial Wire Debug (SWD) signals to a hardware debug probe
- I can recommend the STLinkv3, since it costs only 11€ and has a built-in Serial

- The debug probe then communicates over USB to the driver software
- Typically this is OpenOCD (On-Chip Debug)
- Implements the GDB server protocol

- GDB connects to the GDB Server via TCP
- You can already debug now using the GDB command line

- Most IDEs wrap the debug functionality
- Communicate with GDB using the Machine Interface
- MI is an ASCII protocol for communicating with GDB as a User Interface

- This has a lot of latency, it's definitely not real-time.
- Particularly bad if you connect GDB over Wifi to Server. This will be a problem for later.

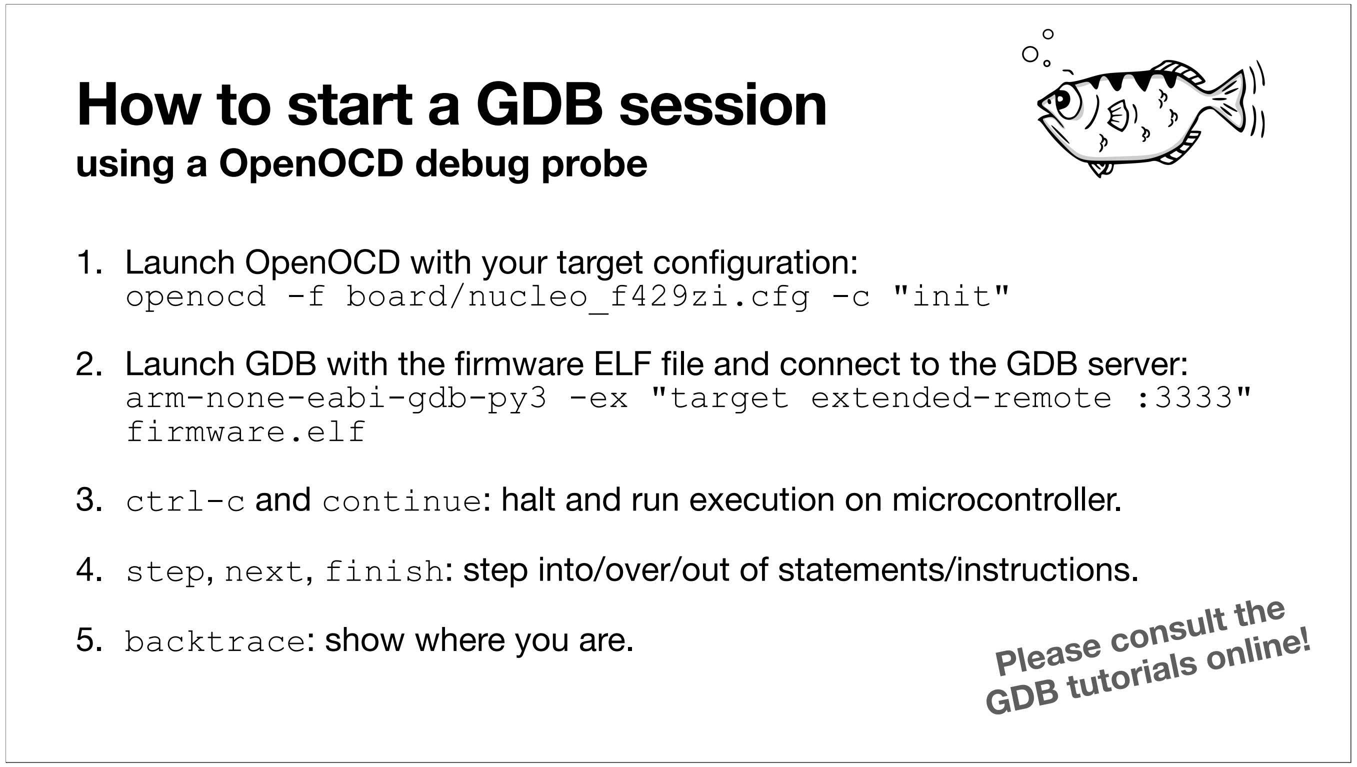


GDB? The GNU DeBugger

arm-none-eabi-gdb with Python 3 support

- GDB is part of the official arm-none-eabi distribution based on GCC 12.
- ARM builds GDB **without** Python 3 support !!!
- Download the xPack arm-none-eabi-gcc12 toolchain instead.
- BUT: only symlink `arm-none-eabi-gdb-py3` into your path.
- `arm-none-eabi-gdb-py3` has a **stand-alone** Python 3.11 runtime!
- GDB usually works fine with debug symbols from any compiler.

- GDB is the debugger that comes with your arm-none-eabi toolchain
- ARM provides an official and tested version, use that one for compilation.
- But the GDB is not compiled with Python3 support.
- So you need to install the xPack version, but only symlink the GDB, not the rest
- Alternatively use the GDB from your distribution at your own risk.



How to start a GDB session

using a OpenOCD debug probe

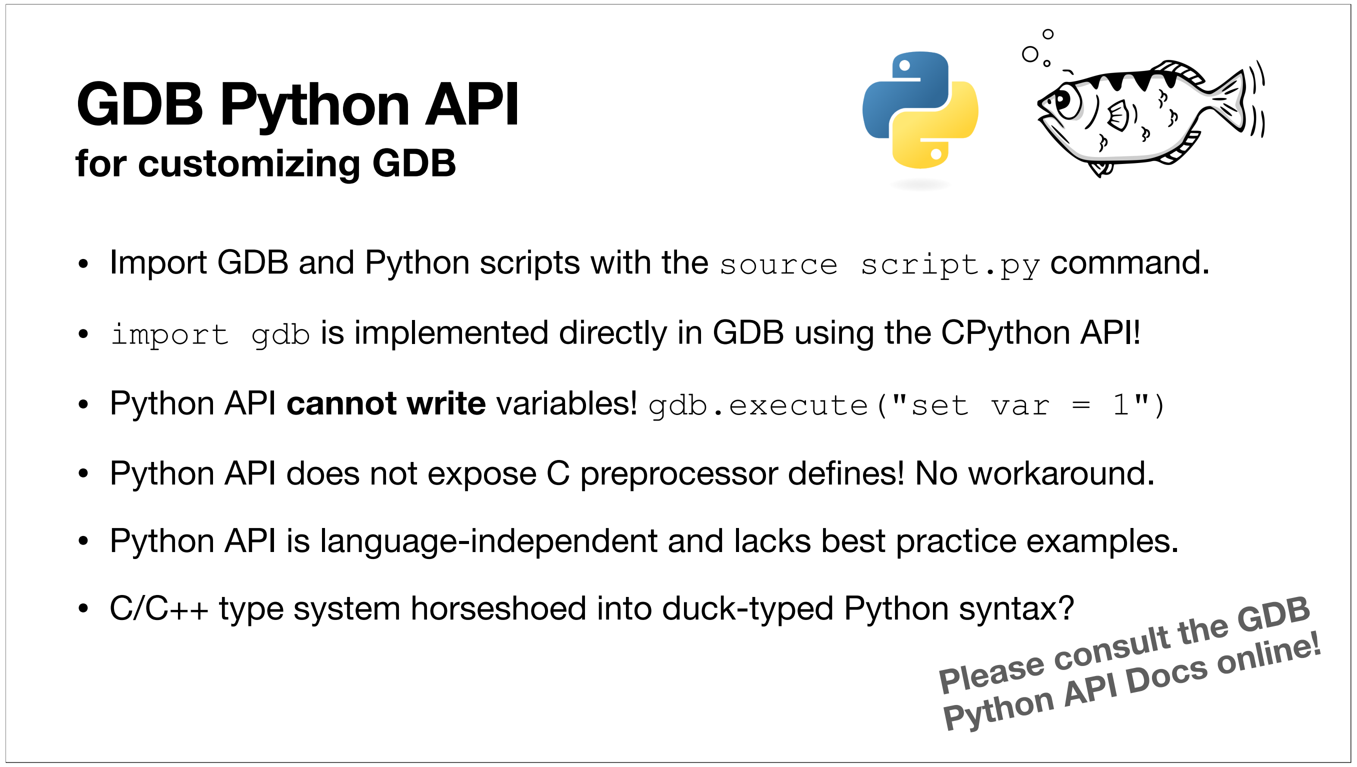
1. Launch OpenOCD with your target configuration:
   ```
   openocd -f board/nucleo_f429zi.cfg -c "init"
   ```
2. Launch GDB with the firmware ELF file and connect to the GDB server:
   ```
   arm-none-eabi-gdb-py3 -ex "target extended-remote :3333"
   firmware.elf
   ```
3. `ctrl-c` and `continue`: halt and run execution on microcontroller.
4. `step`, `next`, `finish`: step into/over/out of statements/instructions.
5. `backtrace`: show where you are.

Please consult the GDB tutorials online!

Let's go over the basics of how to launch a GDB debug session.

- Connect the debug probe to the microcontroller, make sure it has power
- Launch OpenOCD with the correct target configuration and issue the init command
- Launch GDB from another process with the ELF file that contains the debug symbols and connect locally
- Note that you can also pass an IP with the port if you need to debug over the local network.
- Super basic commands are: ctrl-c for interrupting execution.
- GDB has now HALTED the CPU, while you debug.
- Be careful what you debug, drones tend to fall out of the sky if the Avionics fail.
- You can single step through your code.
- And you can show where you are with the backtrace command.

There are many GDB tutorials online, please refer to them if you're a beginner.



GDB Python API

for customizing GDB

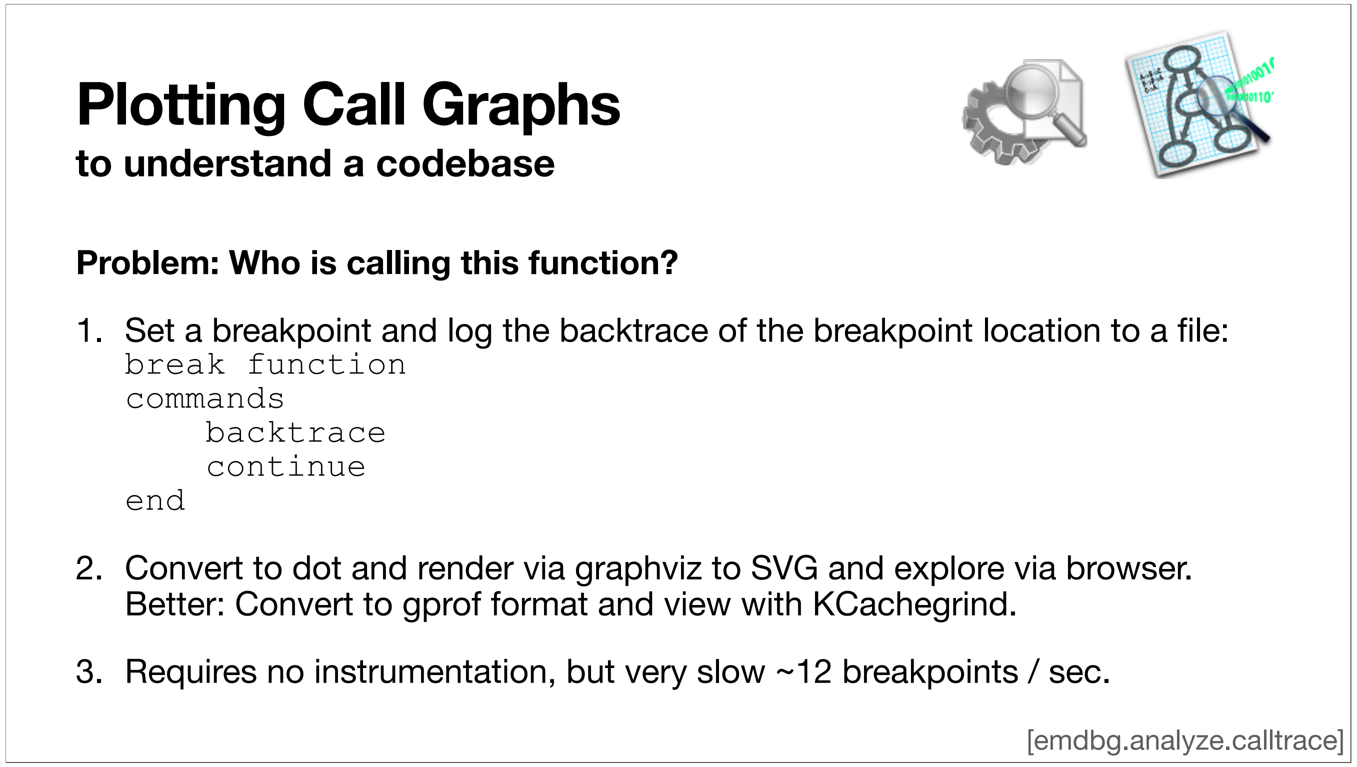
- Import GDB and Python scripts with the `source script.py` command.
- `import gdb` is implemented directly in GDB using the CPython API!
- Python API **cannot write** variables! `gdb.execute("set var = 1")`
- Python API does not expose C preprocessor defines! No workaround.
- Python API is language-independent and lacks best practice examples.
- C/C++ type system horseshoed into duck-typed Python syntax?

Please consult the GDB Python API Docs online!

GDB has it's own script language, but it's quite limited.
To extend GDB a Python API exists:

- You can source Python scripts inside GDB
- The gdb module only exists inside GDB you cannot call it from outside, it directly interfaces with the C API
- There are some limitations:
  - You cannot write variables, you must do this via the GDB scripting language
  - You cannot access all the debug information, some stuff is missing like C preprocessor macros. A big issue for knowing the NuttX configuration.
  - Very well documented at API level, but how to use it to do non-trivial things is fairly unclear. I had to read source code to figure out what the limitations are.
  - language independ API can be a bit wonky for C/C++ semantics

I'm not going to show you a lot of code, rather talk about concepts.



Plotting Call Graphs

to understand a codebase

Problem: Who is calling this function?

1. Set a breakpoint and log the backtrace of the breakpoint location to a file:
   ```
   break function
   commands
       backtrace
       continue
   end
   ```

2. Convert to dot and render via graphviz to SVG and explore via browser.
   Better: Convert to gprof format and view with KCachegrind.

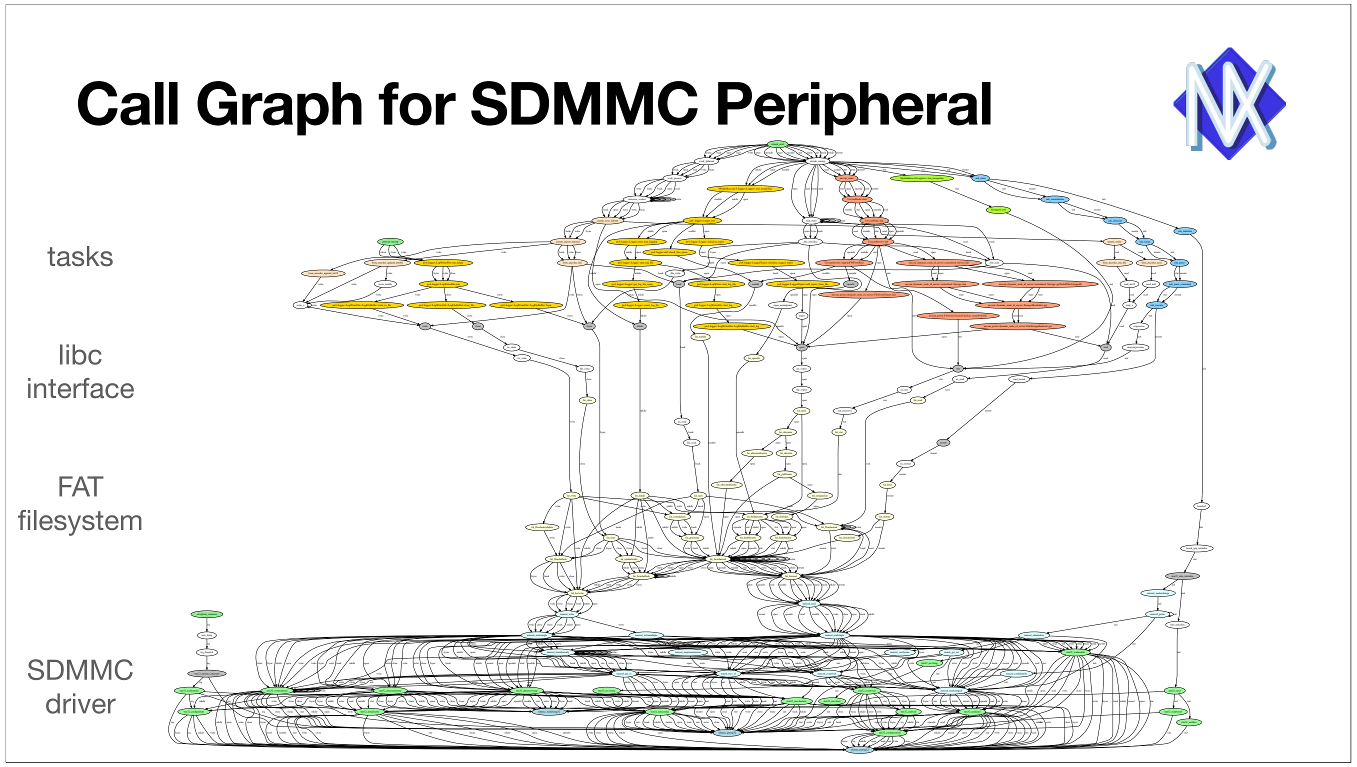
3. Requires no instrumentation, but very slow ~12 breakpoints / sec.

[emdbg.analyze.calltrace]

Let's start with something simple:

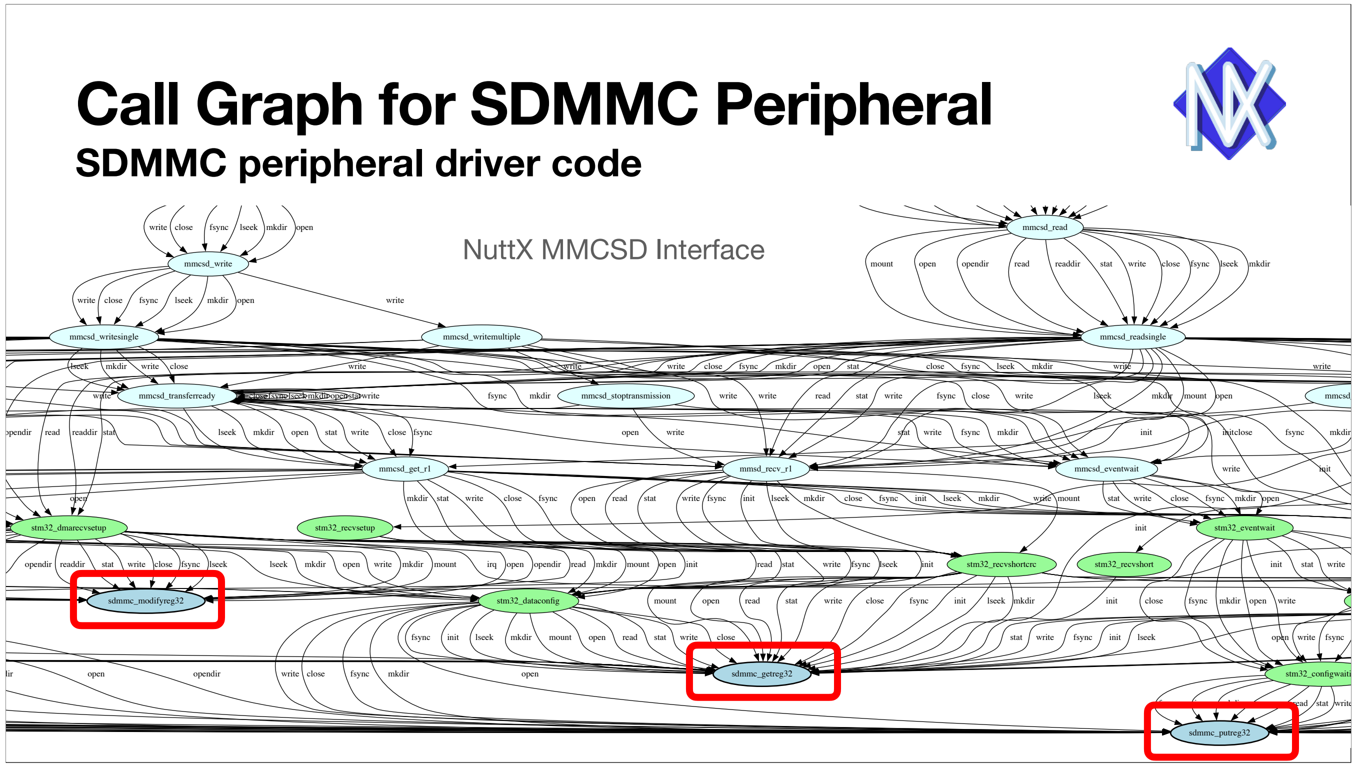
- PX4 Autopilot project contains several million lines of code
- Only started working with it about 6 months ago
- So I need to get an overview: who is calling this function?
- I can set a breakpoint on a function and attach a backtrace to this breakpoint then immediately continue.
- Log the GDB output and postprocess it with a Python script
- Convert to a graph either via graphviz for custom visualizations
- Or use standardized format and view with KCachegrind

This is very slow, because of the latency.



Call Graph for SDMMC Peripheral

- One source of issues is logging data to the SDCard inside the Autopilot.
- I wanted to know what functions were involved in accessing the SDCard peripheral.
- This is the resulting call graph.
- Split into the SDMMC peripheral at the bottom
- FAT filesystem in the middle
- the NuttX libc interface above that
- and the tasks.



Call Graph for SDMMC Peripheral

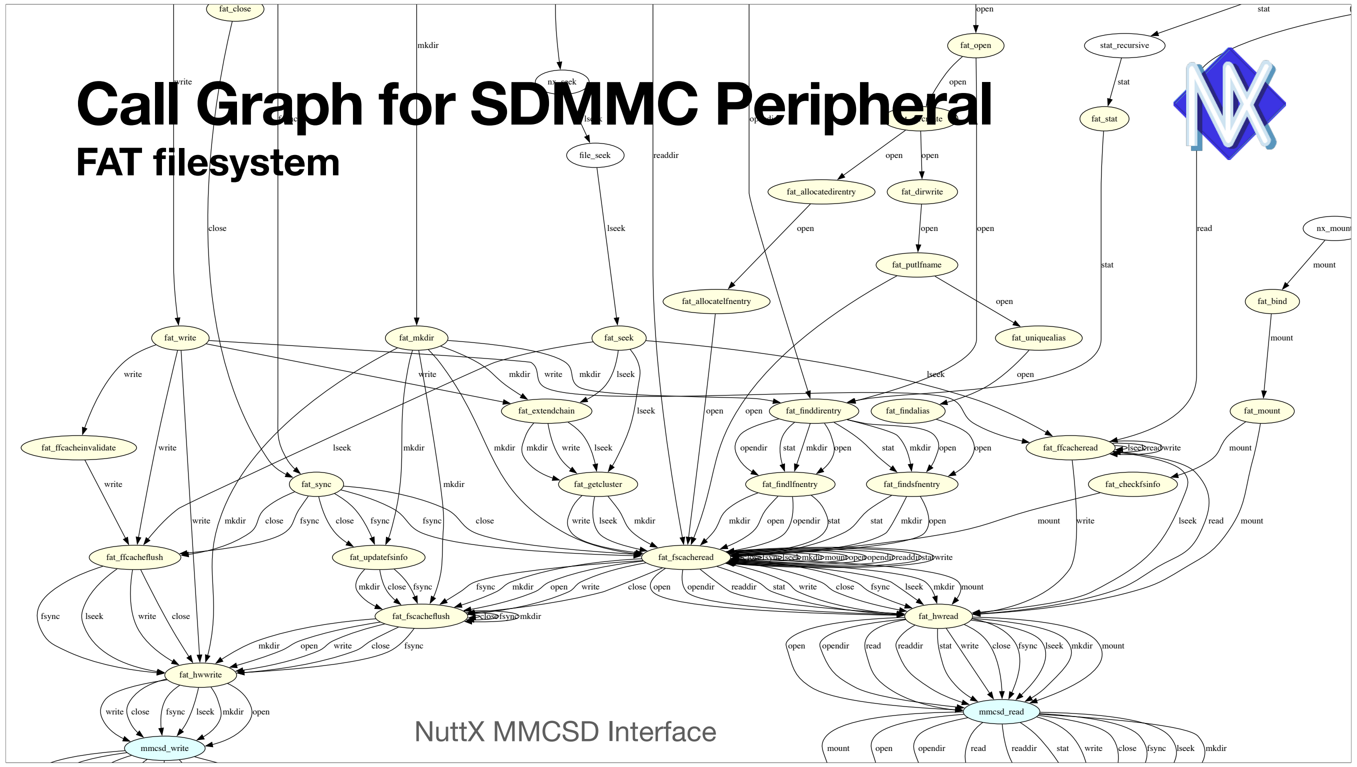
SDMMC peripheral driver code

I've put breakpoints on these three functions in the SDMMC peripheral.
The read, write and modify registers.

You can see that the entire driver only has two entry points:
Write and Read Data.

Call Graph for SDMMC Peripheral

FAT filesystem

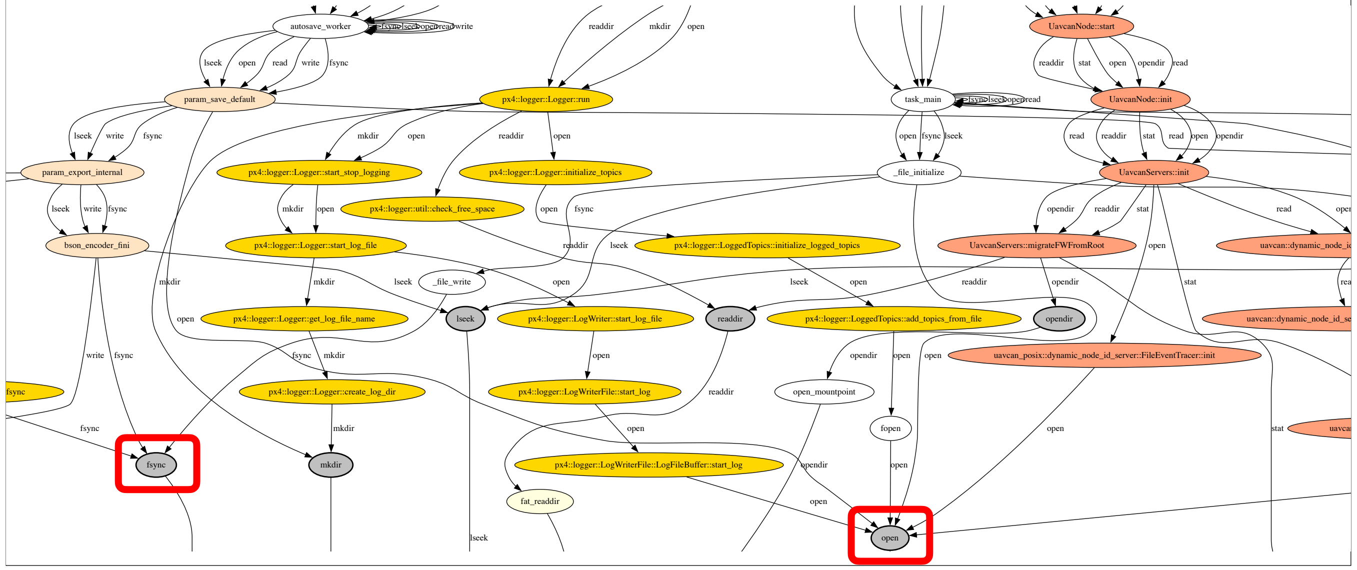


NuttX MMCSD Interface

Above that we can see the FAT filesystem implementation in yellow.
This is a lot of code

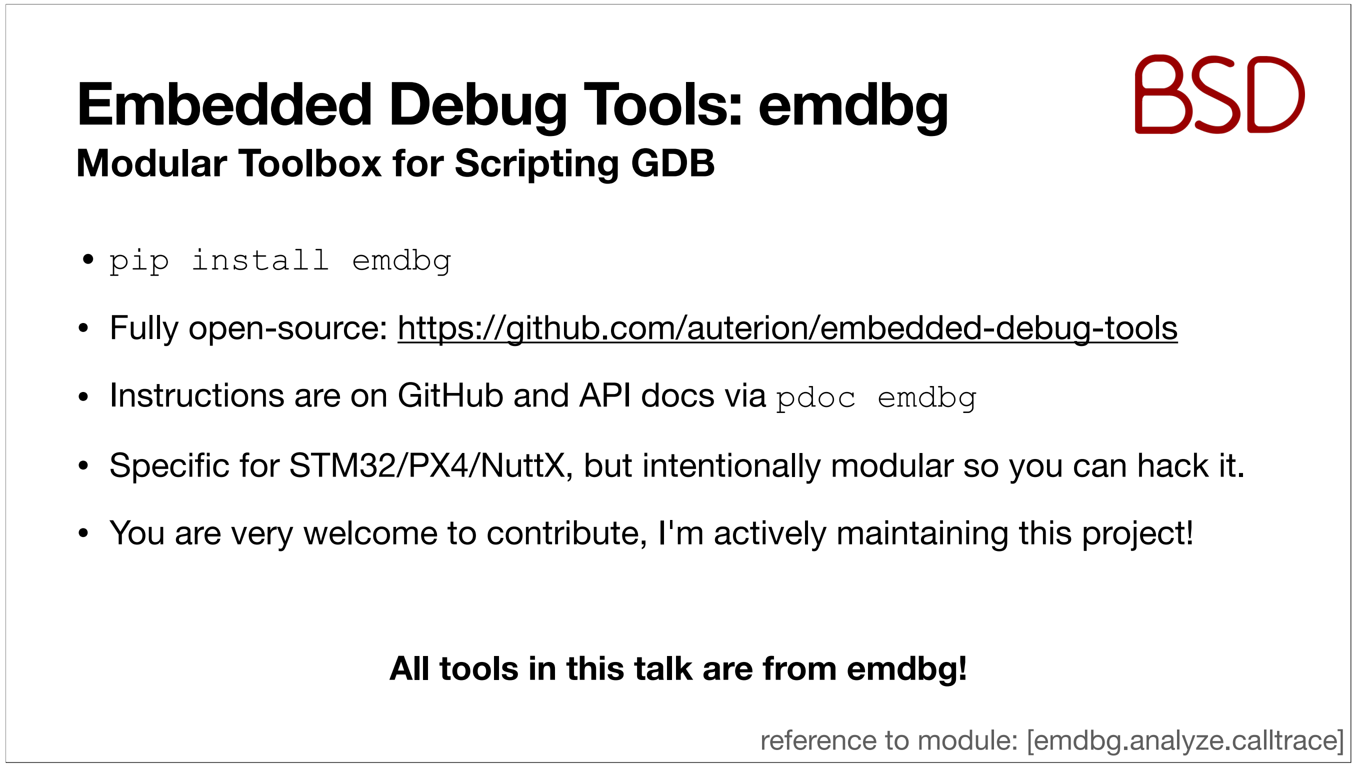
Call Graph for SDMMC Peripheral

libc and tasks


autosave_worker
param_save_default
param_export_internal
bson_encoder_fini
px4::logger::Logger::run
px4::logger::Logger::start_stop_logging
px4::logger::Logger::initialize_topics
px4::logger::util::check_free_space
px4::logger::Logger::start_log_file
px4::logger::LoggedTopics::initialize_logged_topics
_file_write
px4::logger::Logger::get_log_file_name
lseek
px4::logger::LogWriter::start_log_file
readdir
px4::logger::LoggedTopics::add_topics_from_file
opendir
px4::logger::Logger::create_log_dir
px4::logger::LogWriterFile::start_log
open_mountpoint
fopen
px4::logger::LogWriterFile::LogFileBuffer::start_log
fat_readdir
fsync
mkdir
open
task_main
_file_initialize
UavcanNode::start
UavcanNode::init
UavcanServers::init
UavcanServers::migrateFWFromRoot
uavcan_posix::dynamic_node_id_server::FileEventTracer::init
lseek
open
read
write
fsync
mkdir
readdir
opendir
stat

And finally above that we can see standard libc functions
For example: fsync on the bottom left and open on the bottom right
The yellow tasks is the actual logger.

The great thing about this is that you don't need to instrument your code.
You can do this on any firmware without modifcations.
Very helpful for getting to know a code base.



BSD

Embedded Debug Tools: emdbg

Modular Toolbox for Scripting GDB

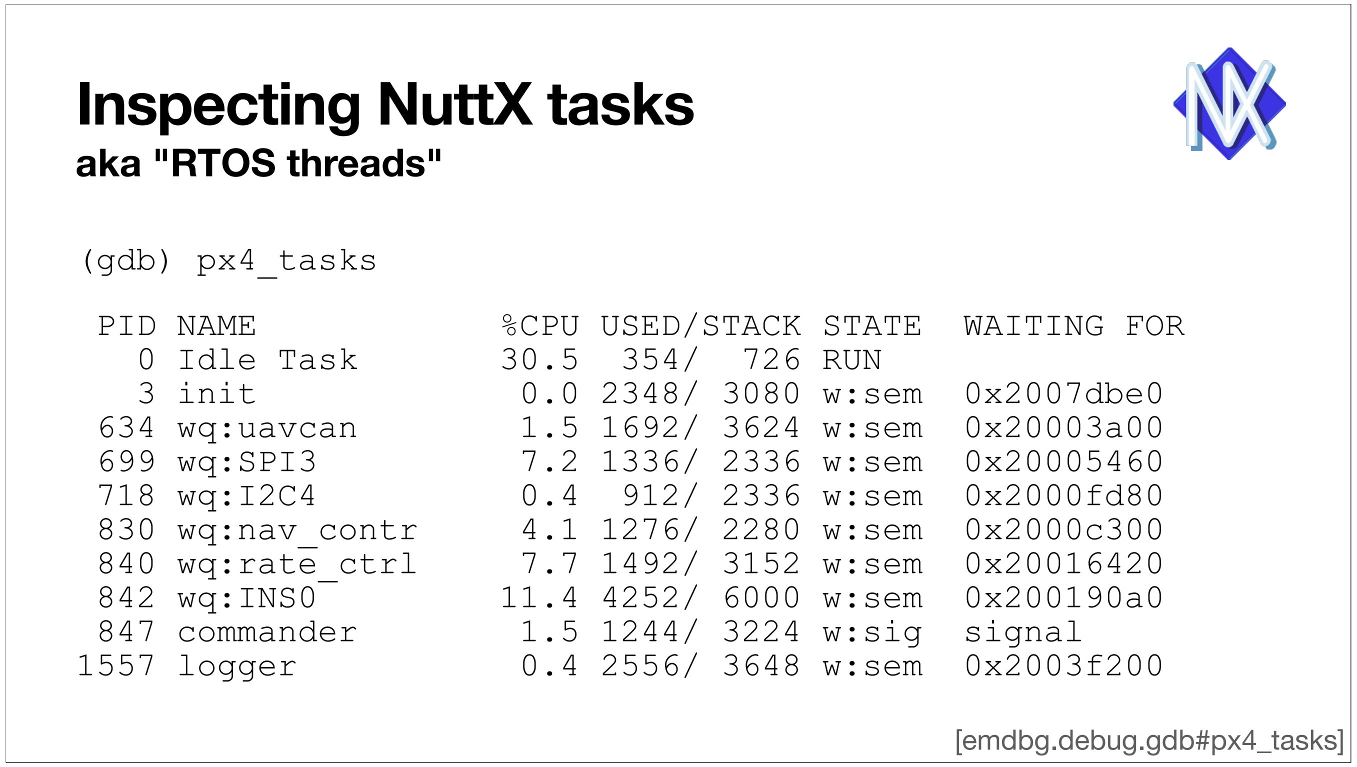
- `pip install emdbg`
- Fully open-source: https://github.com/auterion/embedded-debug-tools
- Instructions are on GitHub and API docs via `pdoc emdbg`
- Specific for STM32/PX4/NuttX, but intentionally modular so you can hack it.
- You are very welcome to contribute, I'm actively maintaining this project!

All tools in this talk are from emdbg!

reference to module: [emdbg.analyze.calltrace]

You can look at how this is implemented in detail in the open-source Embedded Debug Tools.
Python3 library, BSD licensed, fully open-source on GitHub with lots of documentation.
It's highly modular so you can reuse it, even though the higher level tools are STM32/PX4/NuttX specific
Actively maintained, feel free to contribute or just use it as a reference, you can also ask questions there.

The reference at the bottom right refers to the python module.
Ok, let's look at some more GDB tools.



Inspecting NuttX tasks

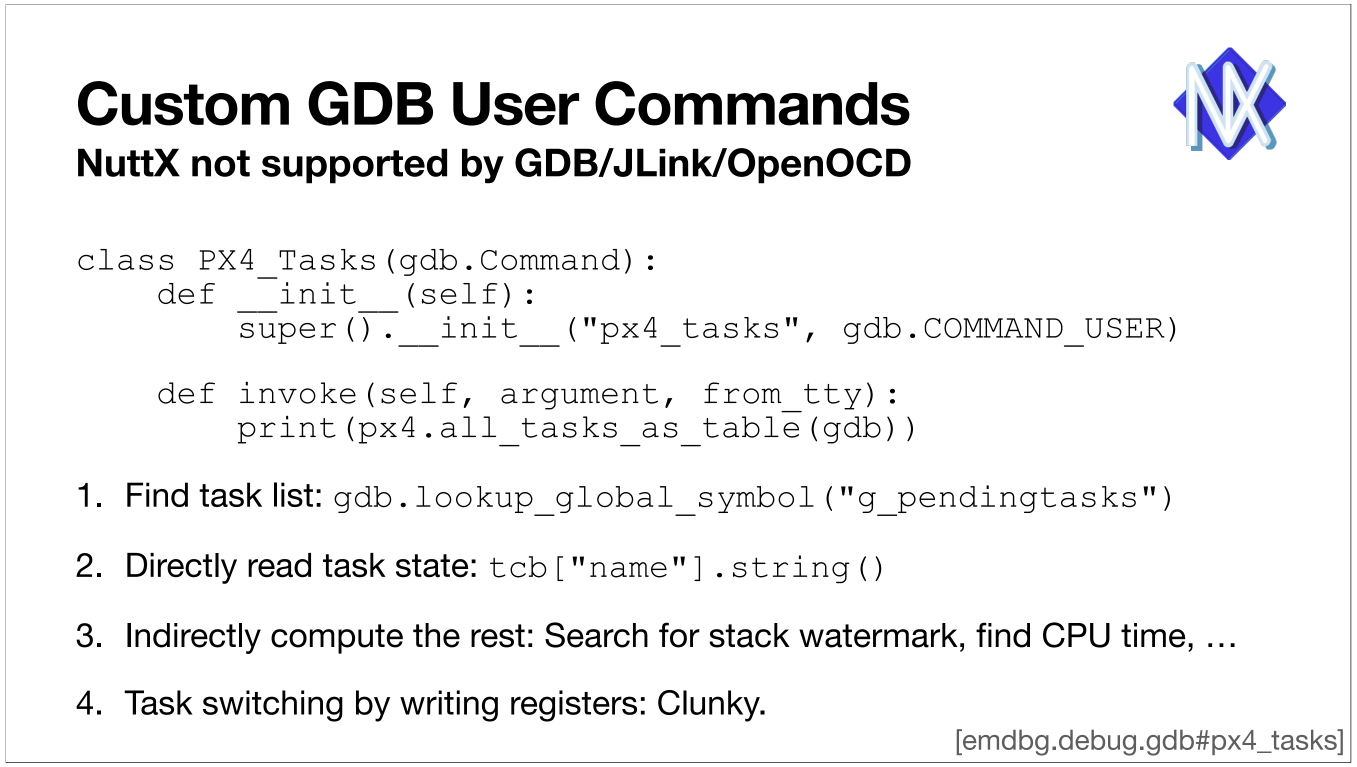
aka "RTOS threads"

```
(gdb) px4_tasks

 PID NAME             %CPU USED/STACK STATE  WAITING FOR
   0 Idle Task        30.5  354/  726 RUN
   3 init              0.0 2348/ 3080 w:sem  0x2007dbe0
 634 wq:uavcan         1.5 1692/ 3624 w:sem  0x20003a00
 699 wq:SPI3           7.2 1336/ 2336 w:sem  0x20005460
 718 wq:I2C4           0.4  912/ 2336 w:sem  0x2000fd80
 830 wq:nav_contr      4.1 1276/ 2280 w:sem  0x2000c300
 840 wq:rate_ctrl      7.7 1492/ 3152 w:sem  0x20016420
 842 wq:INS0          11.4 4252/ 6000 w:sem  0x200190a0
 847 commander         1.5 1244/ 3224 w:sig  signal
1557 logger            0.4 2556/ 3648 w:sem  0x2003f200
```

[emdbg.debug.gdb#px4_tasks]

NuttX is a RTOS with preemptive threads, PX4 uses a lot of threads.
Here I've created our first GDB Python tool: px4_tasks
it lists all the threads with their PID, name, CPU and stack usage, and most importantly what it's waiting for.

Semaphores are the bain of my existance, therefore being able to see their state is great for my sanity.



Custom GDB User Commands

NuttX not supported by GDB/JLink/OpenOCD

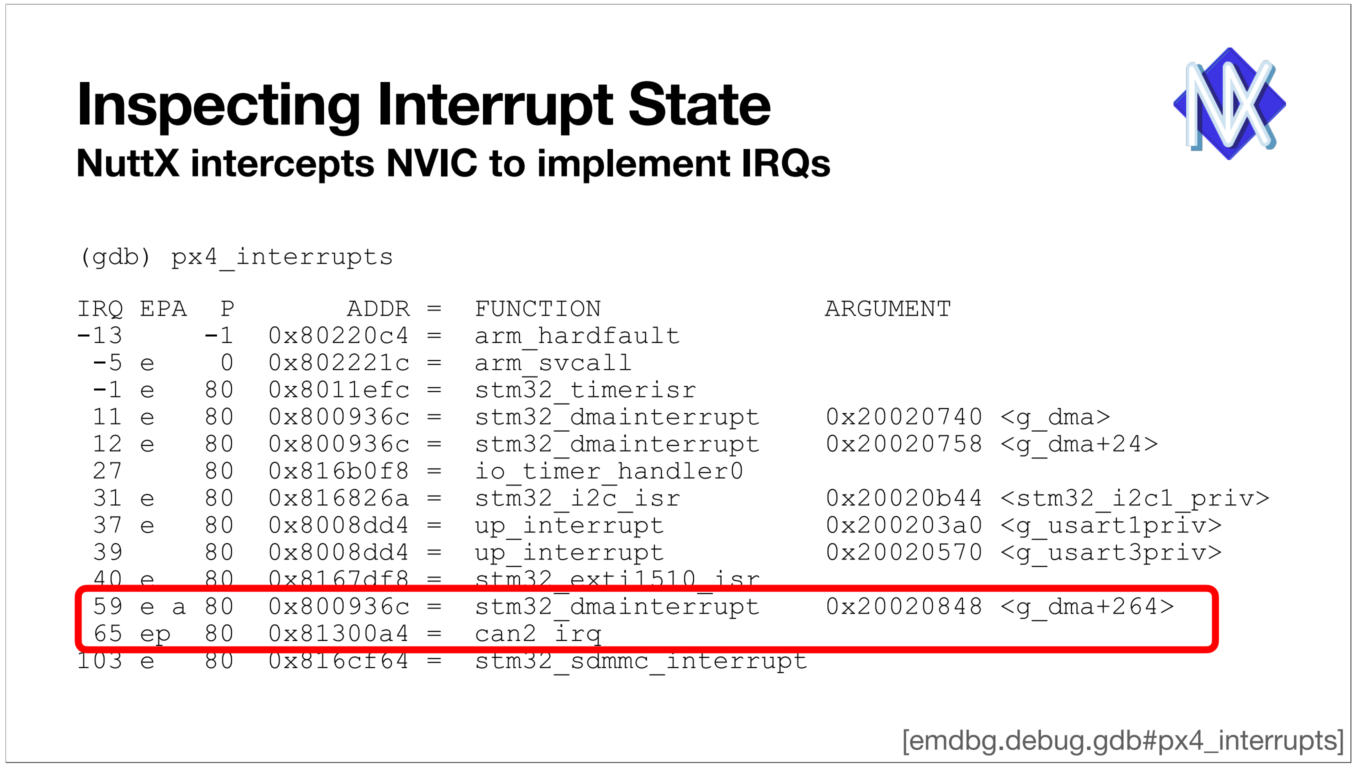
```
class PX4_Tasks(gdb.Command):
    def __init__(self):
        super().__init__("px4_tasks", gdb.COMMAND_USER)

    def invoke(self, argument, from_tty):
        print(px4.all_tasks_as_table(gdb))
```

1. Find task list: `gdb.lookup_global_symbol("g_pendingtasks")`
2. Directly read task state: `tcb["name"].string()`
3. Indirectly compute the rest: Search for stack watermark, find CPU time, …
4. Task switching by writing registers: Clunky.

[emdbg.debug.gdb#px4_tasks]

So how does this work?

This is implemented as a custom GDB user command.
- GDB can lookup symbols in various scopes.
- NuttX uses a so called ready list of tasks that are runnable.
- We find that in SRAM and then iterate over each task struct.
- There are simple attributes of the struct that we can directly access
- Others need some more code like a binary search to find the stack watermark, look at timers to figure out CPU time etc
- So, complexity can be a scaled up or down



Inspecting Interrupt State

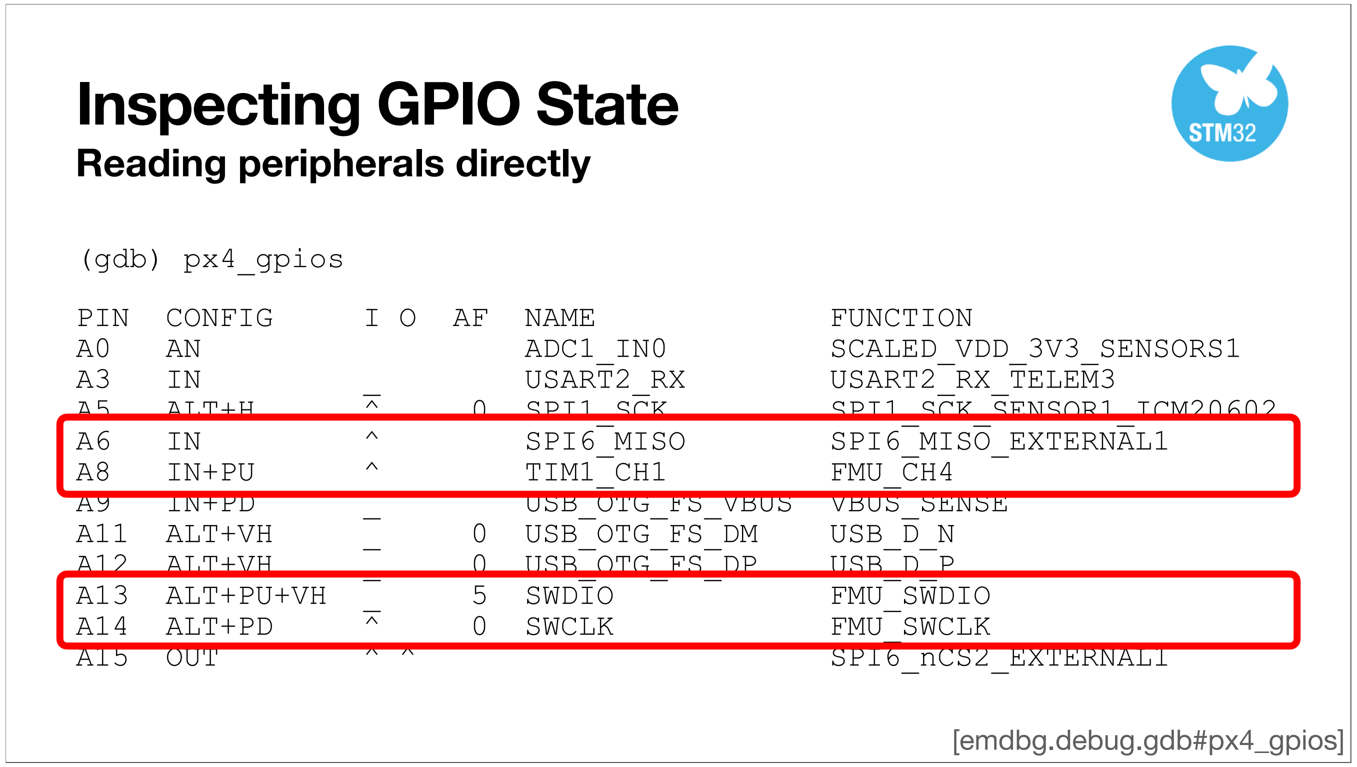
NuttX intercepts NVIC to implement IRQs

```
(gdb) px4_interrupts

IRQ EPA  P       ADDR =  FUNCTION               ARGUMENT
-13      -1  0x80220c4 =  arm_hardfault
 -5 e     0  0x802221c =  arm_svcall
 -1 e    80  0x8011efc =  stm32_timerisr
 11 e    80  0x800936c =  stm32_dmainterrupt     0x20020740 <g_dma>
 12 e    80  0x800936c =  stm32_dmainterrupt     0x20020758 <g_dma+24>
 27      80  0x816b0f8 =  io_timer_handler0
 31 e    80  0x816826a =  stm32_i2c_isr          0x20020b44 <stm32_i2c1_priv>
 37 e    80  0x8008dd4 =  up_interrupt           0x200203a0 <g_usart1priv>
 39      80  0x8008dd4 =  up_interrupt           0x20020570 <g_usart3priv>
 40 e    80  0x8167df8 =  stm32_exti1510_isr
 59 e a 80  0x800936c =  stm32_dmainterrupt     0x20020848 <g_dma+264>
 65 ep   80  0x81300a4 =  can2_irq
103 e    80  0x816cf64 =  stm32_sdmmc_interrupt
```

[emdbg.debug.gdb#px4_interrupts]

Another example, NuttX implements its own dynamic interrupt dispatcher in assembly.
Therefore every interrupt handler is the same function, not very useful for debugging.
This small tool finds the NuttX dispatch table and renders it.
You can also see the priority (all the same, NuttX doesn't support nested interrupts)
and whether the interrupt is enabled, pending, or active.

Here the IRQ 59 DMA is active, and IRQ 65 is pending, so it'll be next.
Super useful to figure out what functions to put a breakpoint on.



Inspecting GPIO State

Reading peripherals directly

```
(gdb) px4_gpios

PIN  CONFIG      I O  AF  NAME             FUNCTION
A0   AN                   ADC1_IN0         SCALED_VDD_3V3_SENSORS1
A3   IN          _        USART2_RX        USART2_RX_TELEM3
A5   ALT+H       ^      0 SPI1_SCK         SPI1_SCK_SENSOR1_ICM20602
A6   IN          ^        SPI6_MISO        SPI6_MISO_EXTERNAL1
A8   IN+PU       ^        TIM1_CH1         FMU_CH4
A9   IN+PD       _        USB_OTG_FS_VBUS  VBUS_SENSE
A11  ALT+VH      _      0 USB_OTG_FS_DM    USB_D_N
A12  ALT+VH      _      0 USB_OTG_FS_DP    USB_D_P
A13  ALT+PU+VH   _      5 SWDIO            FMU_SWDIO
A14  ALT+PD      ^      0 SWCLK            FMU_SWCLK
A15  OUT         ^ ^                       SPI6_nCS2_EXTERNAL1
```

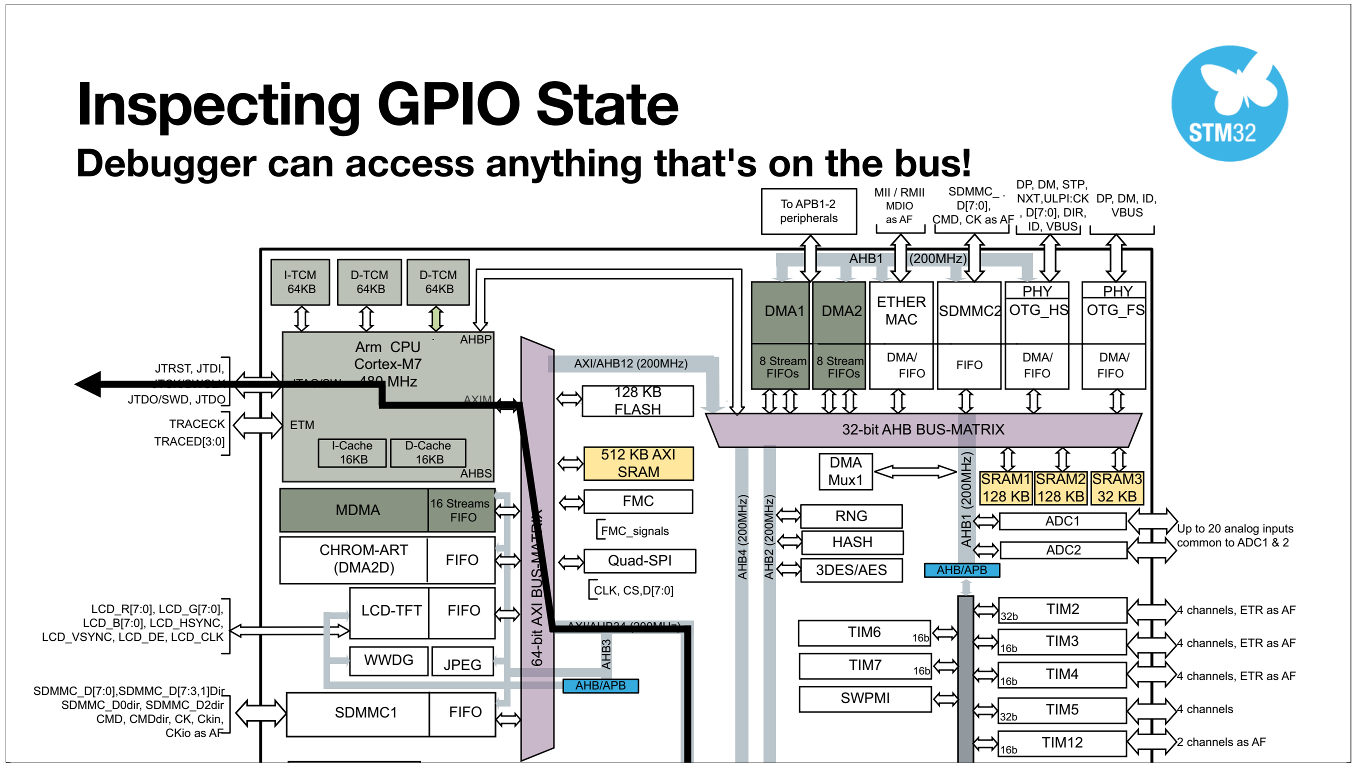
[emdbg.debug.gdb#px4_gpios]

Ok, enough NuttX, let's look at some STM32 specific tools.

I often need to know the state the microcontroller pins, so this tool shows them to me.
You can see the pin name, the Configuration, the input and output state, the alternate function, and signal names and function.

For example, the pin A6 and A8 are inputs with a pullup, both currently high.

Pin A13 and A14 are the SWD connection, we can see that their are interally connected via the alternate function.
The data connection is also configured for Very High datarate (VH).



Inspecting GPIO State

Debugger can access anything that's on the bus!

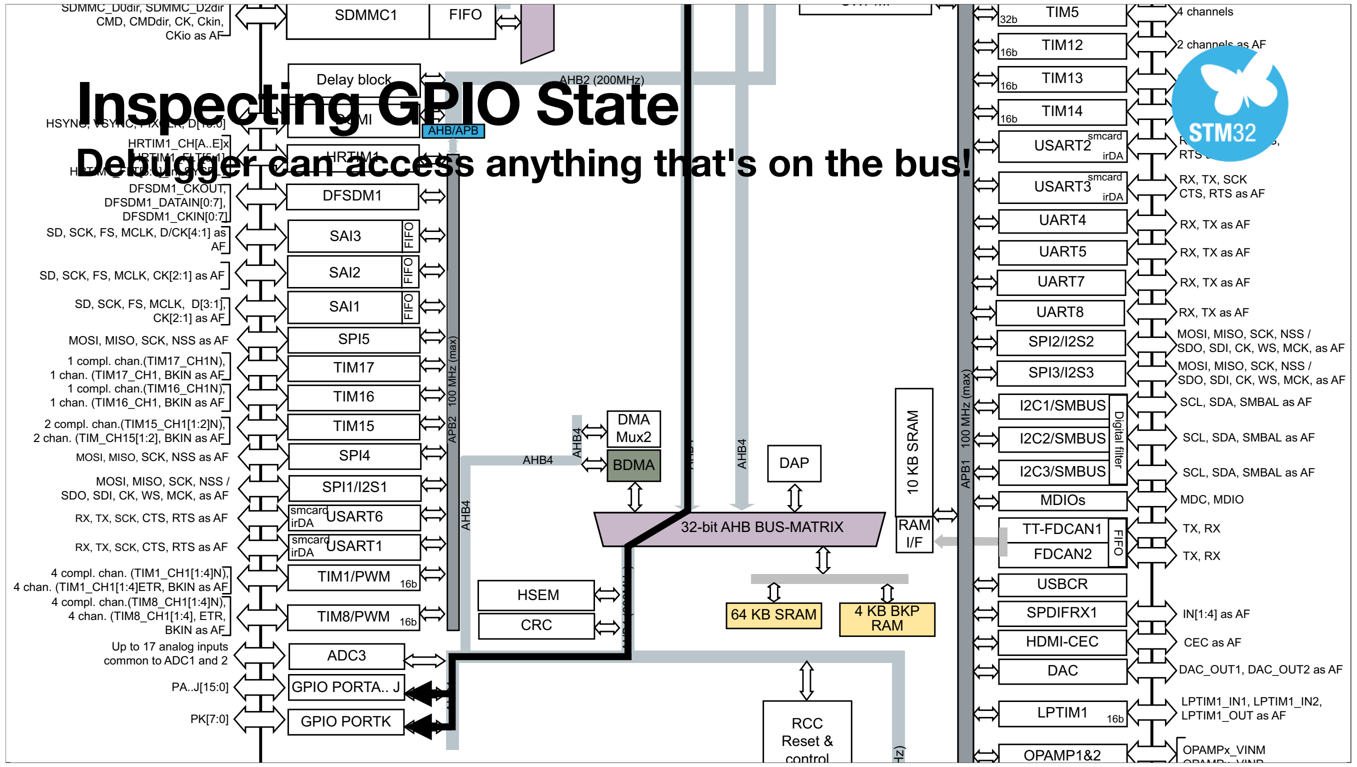
Well, how does this work?

The debugger can not only access the internal SRAM, but also every other bus:

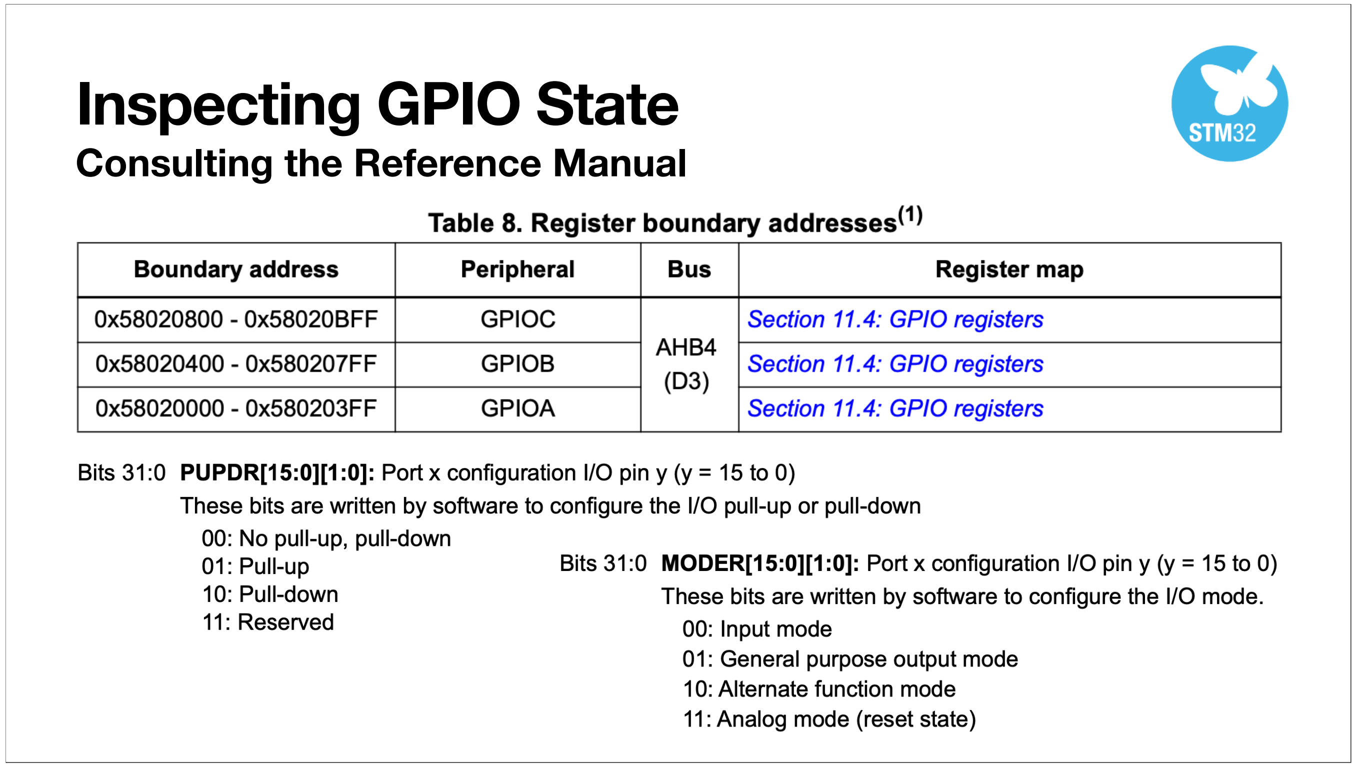
It first goes through the internal 64-bit bus matrix

Inspecting GPIO State

Debugger can access anything that's on the bus!



then goes through another 32-bit bus matrix before finally accessing the register file of the GPIO peripherals.



Inspecting GPIO State

Consulting the Reference Manual

Table 8. Register boundary addresses(1)

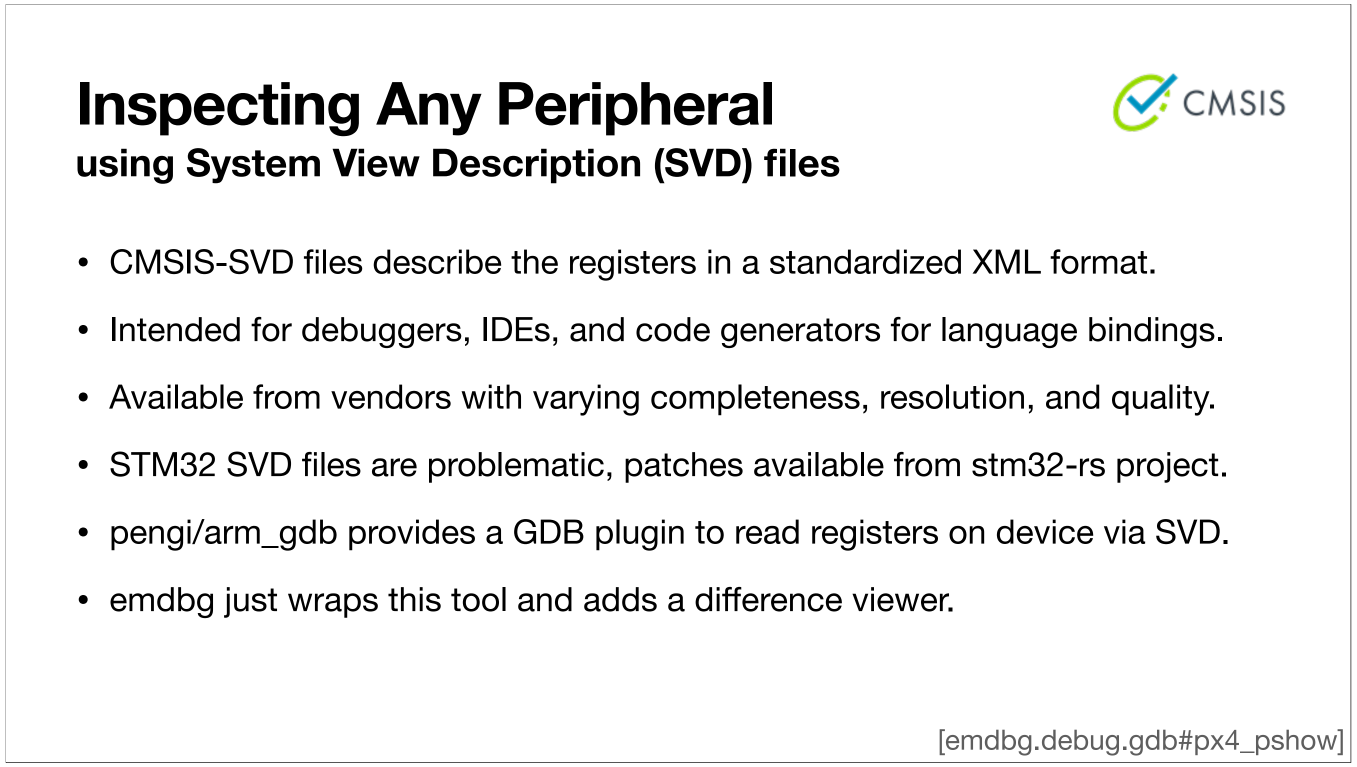
Boundary address	Peripheral	Bus	Register map
0x58020800 - 0x58020BFF	GPIOC	AHB4 (D3)	*Section 11.4: GPIO registers*
0x58020400 - 0x580207FF	GPIOB		*Section 11.4: GPIO registers*
0x58020000 - 0x580203FF	GPIOA		*Section 11.4: GPIO registers*

Bits 31:0 **PUPDR[15:0][1:0]:** Port x configuration I/O pin y (y = 15 to 0)
These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

Bits 31:0 **MODER[15:0][1:0]:** Port x configuration I/O pin y (y = 15 to 0)
These bits are written by software to configure the I/O mode.
00: Input mode
01: General purpose output mode
10: Alternate function mode
11: Analog mode (reset state)

For this to work I need to know the exact address of the GPIO peripheral, which I can find in the STM32H7 reference manual.
I also need to interpret the bits inside the register, which I can also find in there.
I then wrote a short Python script that iterates over each pin and converts the bit fields to the table you just saw.

ok, but I cannot write a custom parser for every peripheral, wouldn't it be nice if we had a machine-readable register description?



Inspecting Any Peripheral

using System View Description (SVD) files

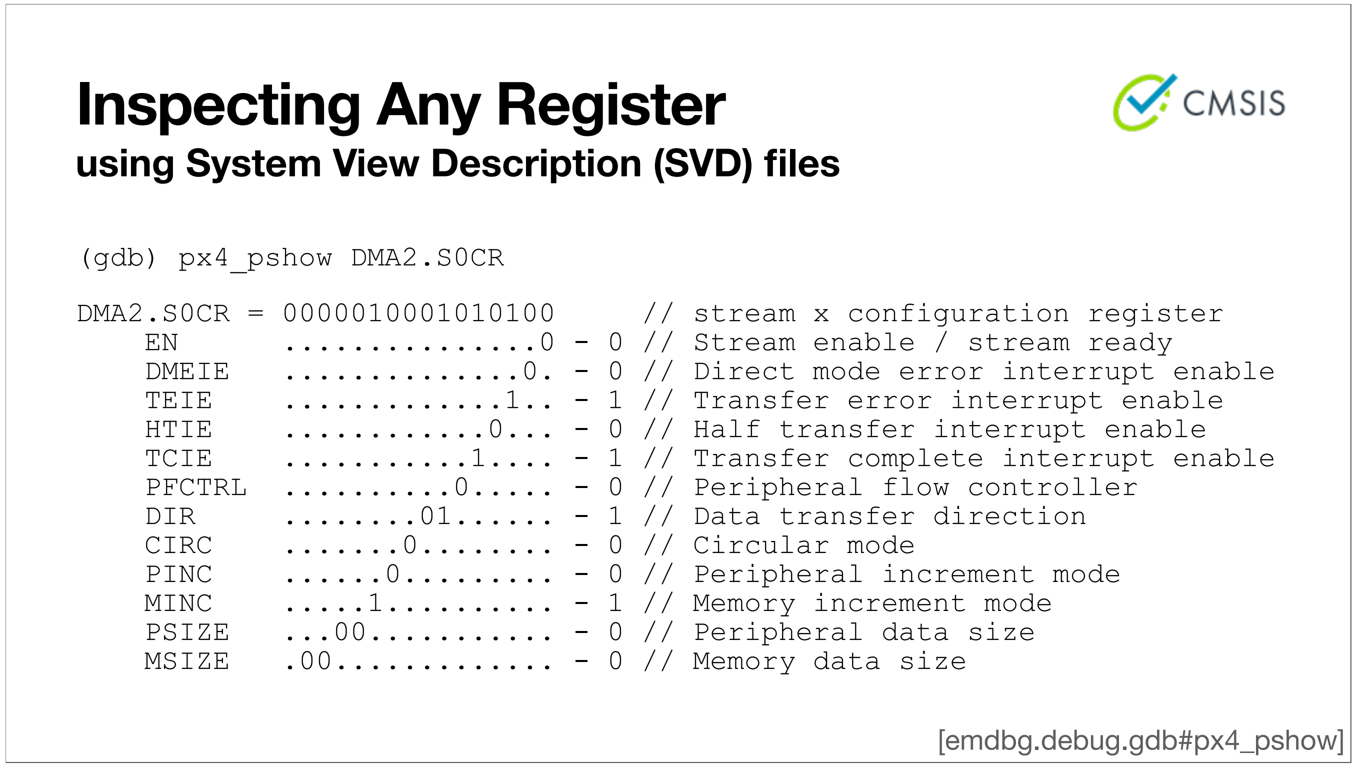
- CMSIS-SVD files describe the registers in a standardized XML format.
- Intended for debuggers, IDEs, and code generators for language bindings.
- Available from vendors with varying completeness, resolution, and quality.
- STM32 SVD files are problematic, patches available from stm32-rs project.
- pengi/arm_gdb provides a GDB plugin to read registers on device via SVD.
- emdbg just wraps this tool and adds a difference viewer.

[emdbg.debug.gdb#px4_pshow]

Well… the System View Description files are exactly that.
A standardized XML format that describes the register maps, so I don't have to copy it out of the PDFs.

The STM32 SVD files are a little broken, there are manual patches available.
There is a great GDB plugin from pengi that does the heavy lifting for you, it loads the SVD file and tells the debug probe to read the right memory and converts this into a structured form.

emdbg just wraps this tool for convenience.



Inspecting Any Register

using System View Description (SVD) files

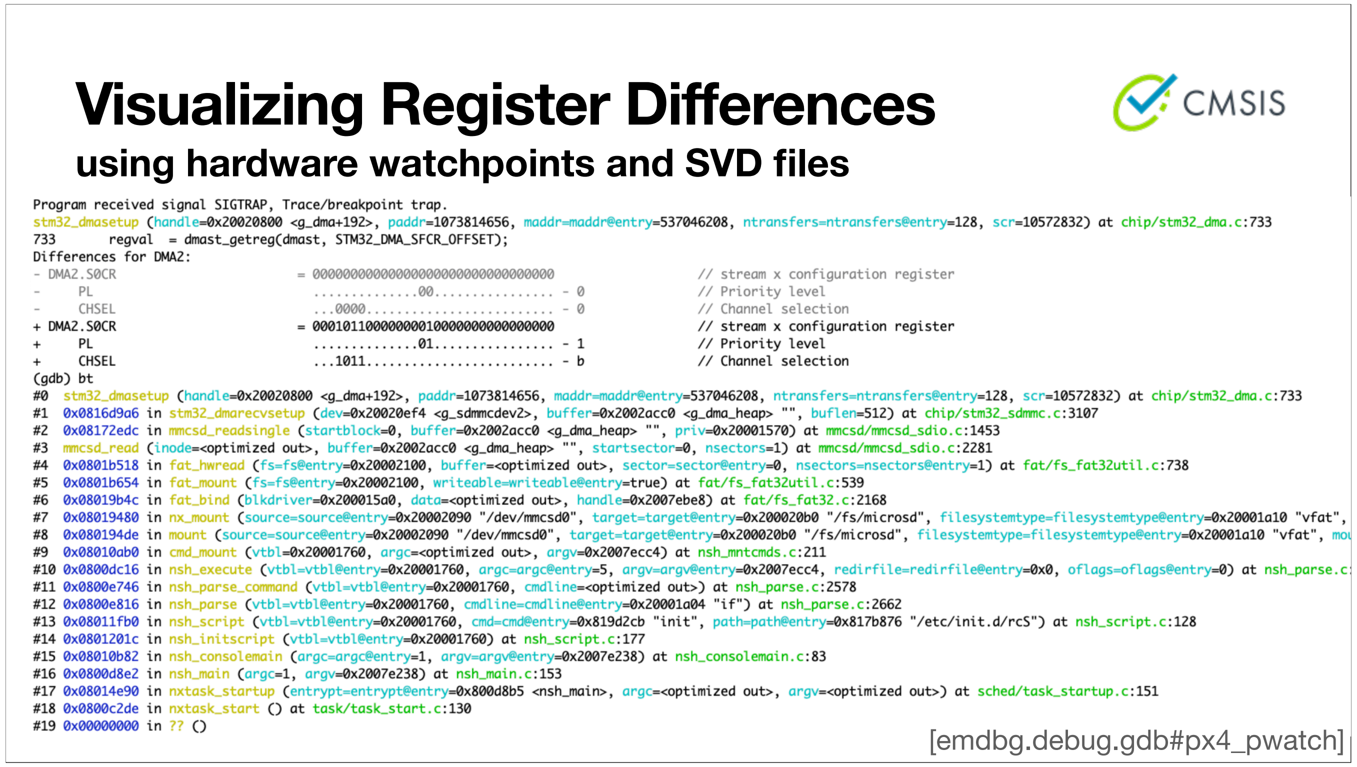
```
(gdb) px4_pshow DMA2.S0CR

DMA2.S0CR = 0000010001010100     // stream x configuration register
    EN      ...............0 - 0 // Stream enable / stream ready
    DMEIE   ..............0. - 0 // Direct mode error interrupt enable
    TEIE    .............1.. - 1 // Transfer error interrupt enable
    HTIE    ............0... - 0 // Half transfer interrupt enable
    TCIE    ...........1.... - 1 // Transfer complete interrupt enable
    PFCTRL  ..........0..... - 0 // Peripheral flow controller
    DIR     ........01...... - 1 // Data transfer direction
    CIRC    .......0........ - 0 // Circular mode
    PINC    ......0......... - 0 // Peripheral increment mode
    MINC    .....1.......... - 1 // Memory increment mode
    PSIZE   ...00........... - 0 // Peripheral data size
    MSIZE   .00............. - 0 // Memory data size
```

[emdbg.debug.gdb#px4_pshow]

This is what it looks like

here we're looking at the DMA2 stream 0 configuration register.
You can see the raw register value and then below all the bitfields with the name and description

This is very helpful for a quickly checking the configuration of a peripheral without manually unfiddling the bits.
You still need to check the reference manual for the larger picture.



Visualizing Register Differences

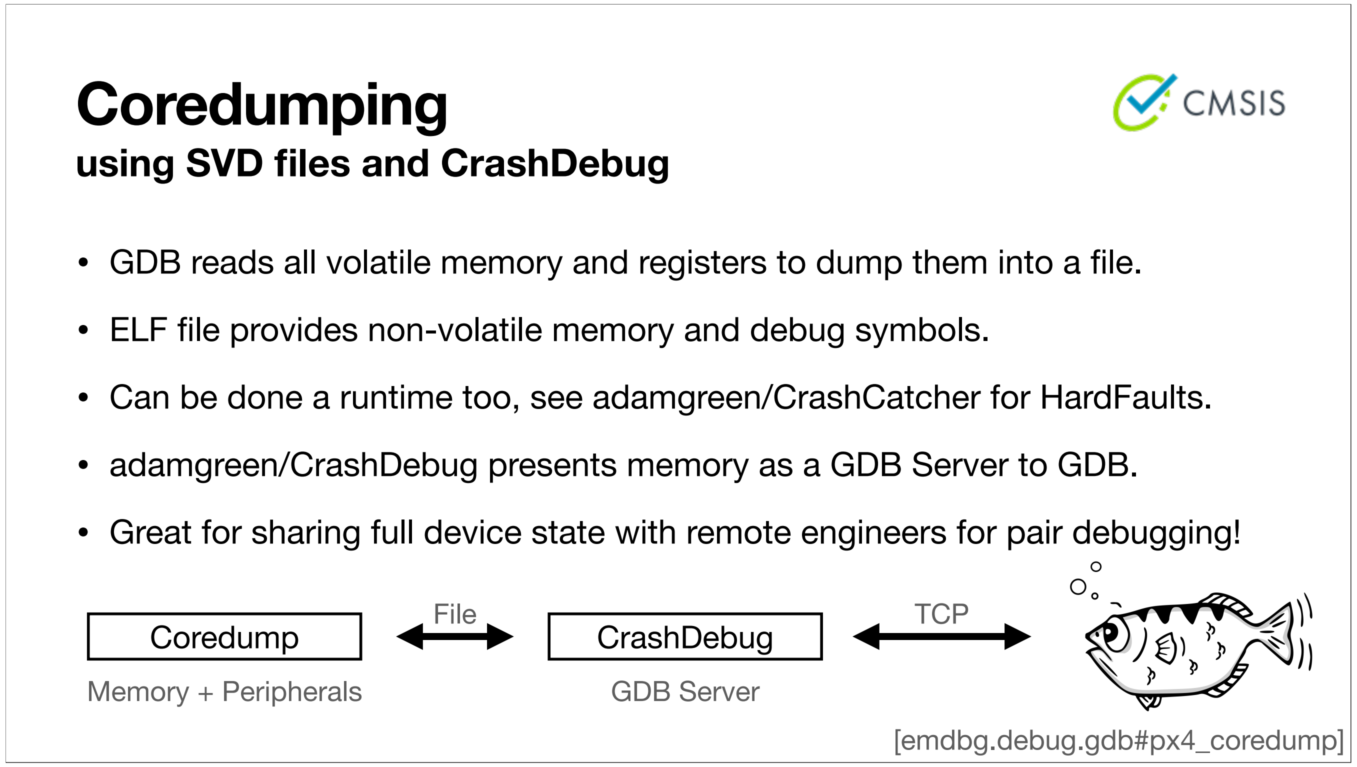
using hardware watchpoints and SVD files

CMSIS

```
Program received signal SIGTRAP, Trace/breakpoint trap.
stm32_dmasetup (handle=0x20020800 <g_dma+192>, paddr=1073814656, maddr=maddr@entry=537046208, ntransfers=ntransfers@entry=128, scr=10572832) at chip/stm32_dma.c:733
733       regval  = dmast_getreg(dmast, STM32_DMA_SFCR_OFFSET);
Differences for DMA2:
- DMA2.S0CR                         = 00000000000000000000000000000000                    // stream x configuration register
-     PL                              ..............00................ - 0                // Priority level
-     CHSEL                           ...0000......................... - 0                // Channel selection
+ DMA2.S0CR                         = 00010110000000010000000000000000                    // stream x configuration register
+     PL                              ..............01................ - 1                // Priority level
+     CHSEL                           ...1011......................... - b                // Channel selection
(gdb) bt
#0  stm32_dmasetup (handle=0x20020800 <g_dma+192>, paddr=1073814656, maddr=maddr@entry=537046208, ntransfers=ntransfers@entry=128, scr=10572832) at chip/stm32_dma.c:733
#1  0x0816d9a6 in stm32_dmarecvsetup (dev=0x20020ef4 <g_sdmmcdev2>, buffer=0x2002acc0 <g_dma_heap> "", buflen=512) at chip/stm32_sdmmc.c:3107
#2  0x08172edc in mmcsd_readsingle (startblock=0, buffer=0x2002acc0 <g_dma_heap> "", priv=0x20001570) at mmcsd/mmcsd_sdio.c:1453
#3  mmcsd_read (inode=<optimized out>, buffer=0x2002acc0 <g_dma_heap> "", startsector=0, nsectors=1) at mmcsd/mmcsd_sdio.c:2281
#4  0x0801b518 in fat_hwread (fs=fs@entry=0x20002100, buffer=<optimized out>, sector=sector@entry=0, nsectors=nsectors@entry=1) at fat/fs_fat32util.c:738
#5  0x0801b654 in fat_mount (fs=fs@entry=0x20002100, writeable=writeable@entry=true) at fat/fs_fat32util.c:539
#6  0x08019b4c in fat_bind (blkdriver=0x200015a0, data=<optimized out>, handle=0x2007ebe8) at fat/fs_fat32.c:2168
#7  0x08019480 in nx_mount (source=source@entry=0x20002090 "/dev/mmcsd0", target=target@entry=0x200020b0 "/fs/microsd", filesystemtype=filesystemtype@entry=0x20001a10 "vfat",
#8  0x080194de in mount (source=source@entry=0x20002090 "/dev/mmcsd0", target=target@entry=0x200020b0 "/fs/microsd", filesystemtype=filesystemtype@entry=0x20001a10 "vfat", mou
#9  0x08010ab0 in cmd_mount (vtbl=0x20001760, argc=<optimized out>, argv=0x2007ecc4) at nsh_mntcmds.c:211
#10 0x0800dc16 in nsh_execute (vtbl=vtbl@entry=0x20001760, argc=argc@entry=5, argv=argv@entry=0x2007ecc4, redirfile=redirfile@entry=0x0, oflags=oflags@entry=0) at nsh_parse.c
#11 0x0800e746 in nsh_parse_command (vtbl=vtbl@entry=0x20001760, cmdline=<optimized out>) at nsh_parse.c:2578
#12 0x0800e816 in nsh_parse (vtbl=vtbl@entry=0x20001760, cmdline=cmdline@entry=0x20001a04 "if") at nsh_parse.c:2662
#13 0x08011fb0 in nsh_script (vtbl=vtbl@entry=0x20001760, cmd=cmd@entry=0x819d2cb "init", path=path@entry=0x817b876 "/etc/init.d/rcS") at nsh_script.c:128
#14 0x0801201c in nsh_initscript (vtbl=vtbl@entry=0x20001760) at nsh_script.c:177
#15 0x08010b82 in nsh_consolemain (argc=argc@entry=1, argv=argv@entry=0x2007e238) at nsh_consolemain.c:83
#16 0x0800d8e2 in nsh_main (argc=1, argv=0x2007e238) at nsh_main.c:153
#17 0x08014e90 in nxtask_startup (entrypt=entrypt@entry=0x800d8b5 <nsh_main>, argc=<optimized out>, argv=<optimized out>) at sched/task_startup.c:151
#18 0x0800c2de in nxtask_start () at task/task_start.c:130
#19 0x00000000 in ?? ()
```

[emdbg.debug.gdb#px4_pwatch]

We can also use this in combination with hardware watchpoints.
The device itself watches a memory region for writes and then triggers a breakpoint.
For example here I can see what code actually writes to this Stream 0 register, and what the differences were.
You can also see the backtrace here, which shows that the SDCard driver from before is calling the DMA driver.



Coredumping

using SVD files and CrashDebug

CMSIS

- GDB reads all volatile memory and registers to dump them into a file.
- ELF file provides non-volatile memory and debug symbols.
- Can be done a runtime too, see adamgreen/CrashCatcher for HardFaults.
- adamgreen/CrashDebug presents memory as a GDB Server to GDB.
- Great for sharing full device state with remote engineers for pair debugging!

Coredump ← File → CrashDebug ← TCP →

Memory + Peripherals

GDB Server

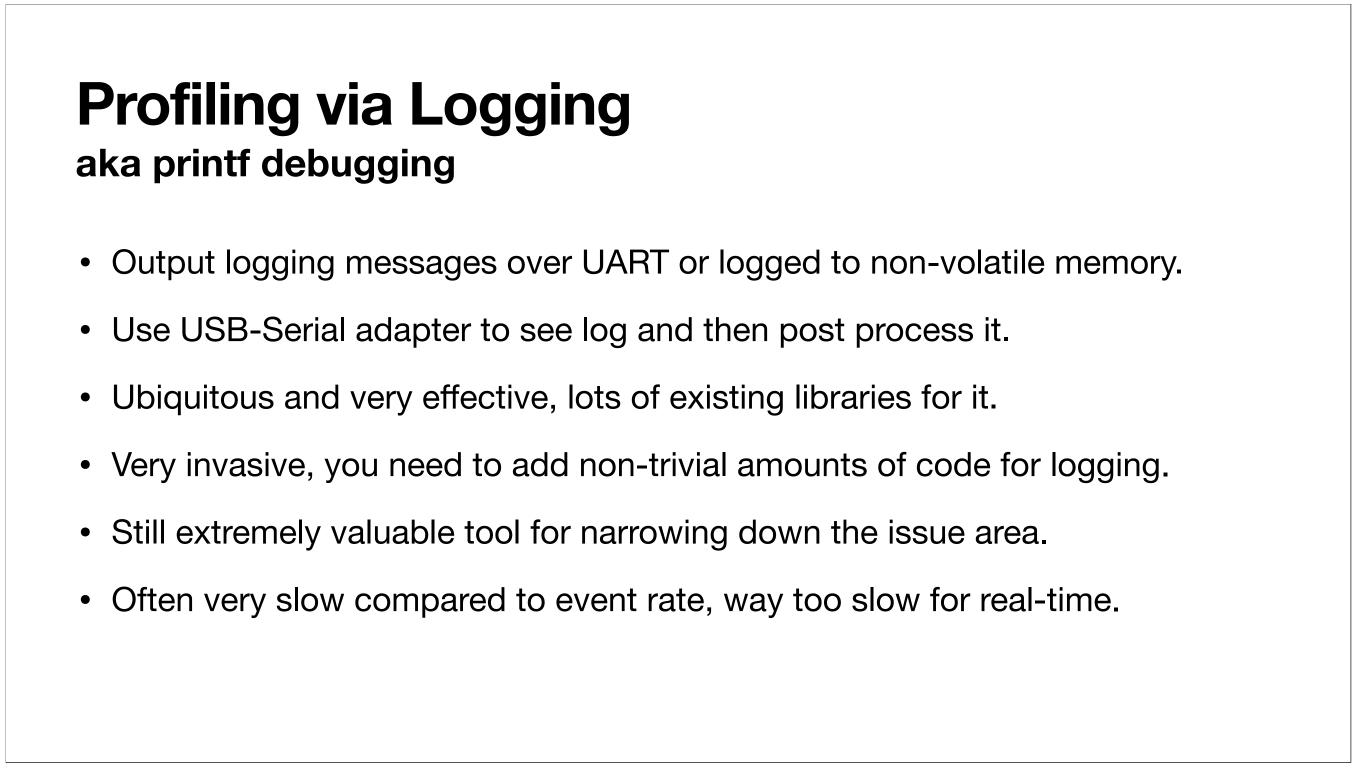
[emdbg.debug.gdb#px4_coredump]

Finally, the last GDB tool I'll show you is coredumping support.
Not natively implemented, so we need to do it ourselves:

It's relatively simple: read out all volatile memory like SRAM and peripherals and store them in a file.
The non-volatile memory comes from the ELF file.
The SVD files are very useful so we do not have to copy 4GBs of data. Only takes a few seconds.
Then write a GDB server that pretends to be connected to a device, but actually serves any data from the coredump file.
The CrashDebug utility from Adam Green does exactly that and it works really well. I can recommend it.

You can archive devices in their buggy state and share it with other engineers or try again later.

All of there tools are running, while the CPU is halted. What if you cannot halt the CPU?

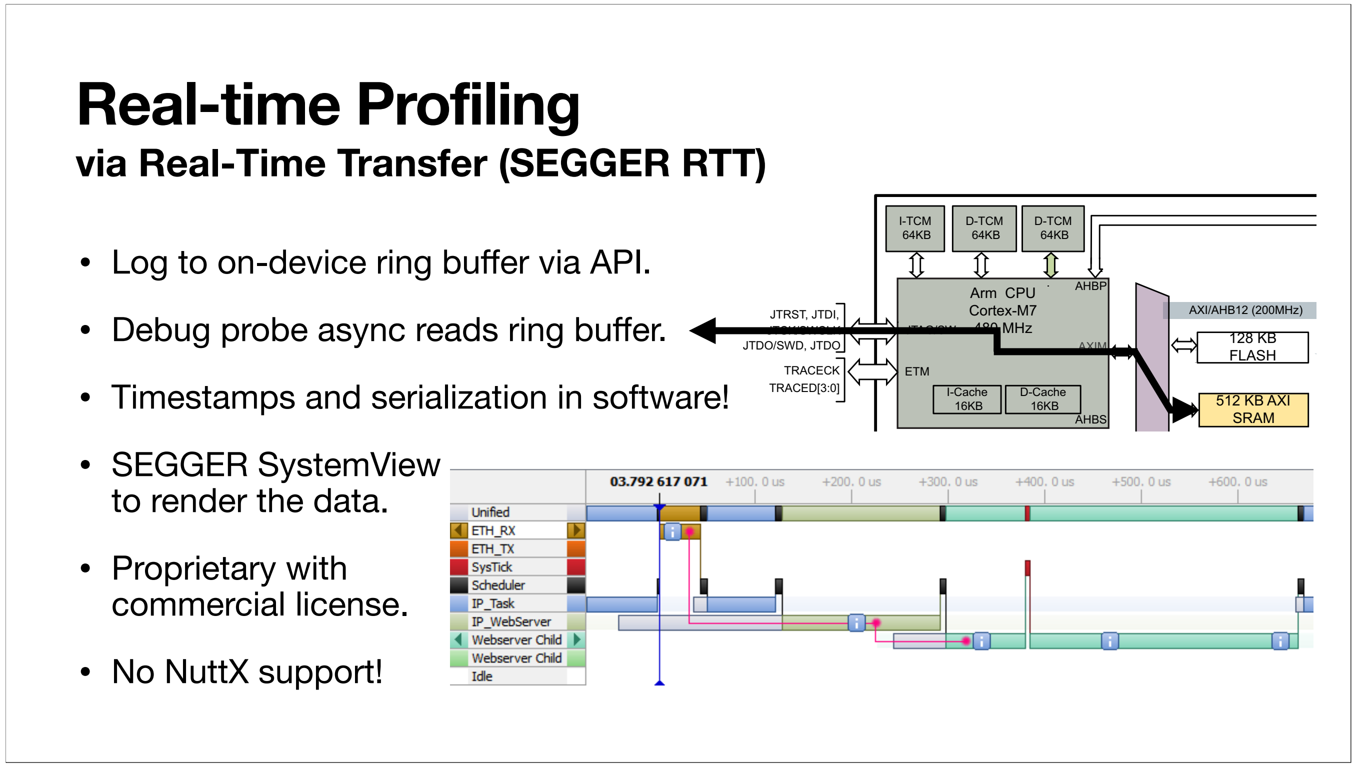


Profiling via Logging

aka printf debugging

- Output logging messages over UART or logged to non-volatile memory.
- Use USB-Serial adapter to see log and then post process it.
- Ubiquitous and very effective, lots of existing libraries for it.
- Very invasive, you need to add non-trivial amounts of code for logging.
- Still extremely valuable tool for narrowing down the issue area.
- Often very slow compared to event rate, way too slow for real-time.

You need to profile.
The simplest profiling method is logging, usually over Serial link.
It's very low-cost, very effective and everyone uses it.
And it is of course a necessary tool to get an idea of what went wrong.
But it's waaaaay to slow for our processor (480MHz).
A lot of events.



Real-time Profiling

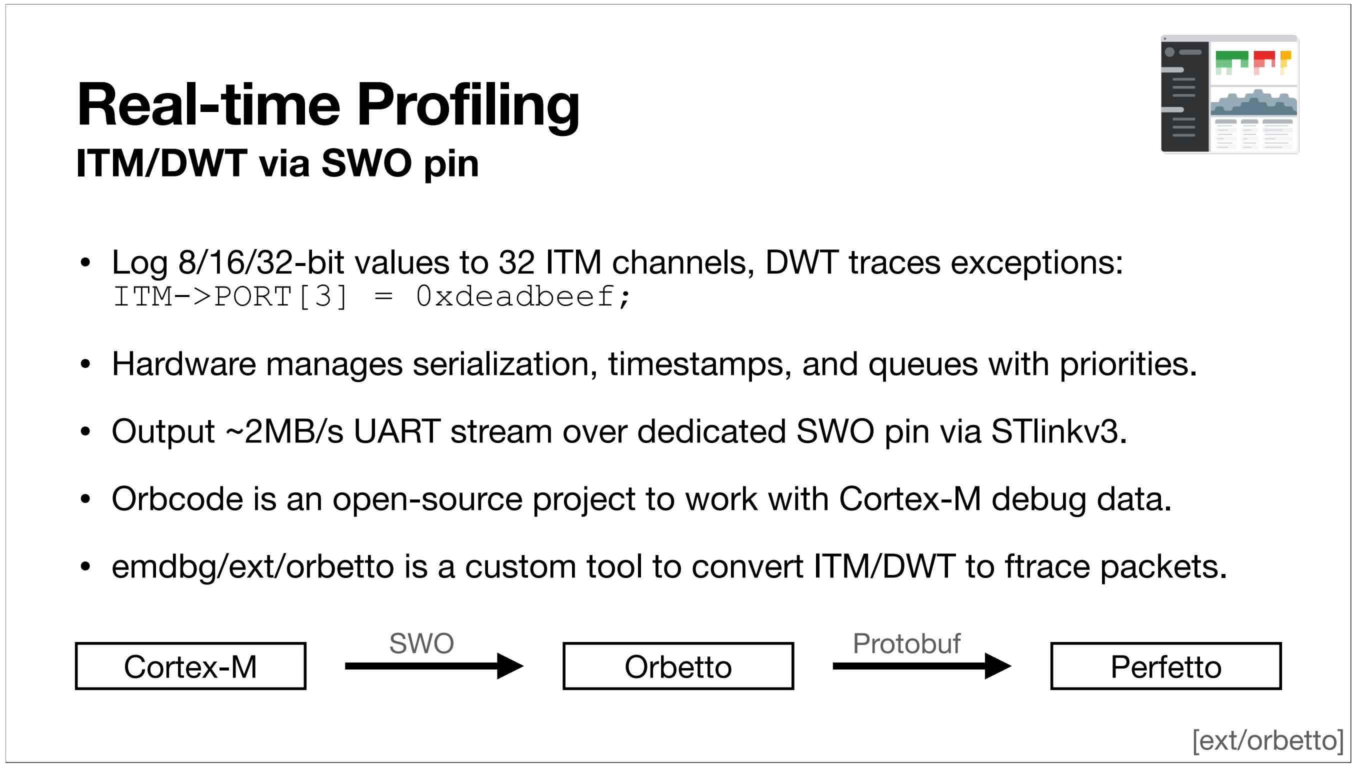
via Real-Time Transfer (SEGGER RTT)

- Log to on-device ring buffer via API.
- Debug probe async reads ring buffer.
- Timestamps and serialization in software!
- SEGGER SystemView to render the data.
- Proprietary with commercial license.
- No NuttX support!

We need a high-bandwidth connection:

Simple idea: log to ring buffer in SRAM and let the debug probe do the transfer.
This is the idea behind SEGGERs SystemView, which provides a library to serialize RTOS events and timestamp it in software.
threads, scheduling, semaphores, interrupts.
BUT: It's proprietary and costs money and it has no native NuttX support.

And, still actually a fairly high overhead.



Real-time Profiling

ITM/DWT via SWO pin

- Log 8/16/32-bit values to 32 ITM channels, DWT traces exceptions:
  `ITM->PORT[3] = 0xdeadbeef;`
- Hardware manages serialization, timestamps, and queues with priorities.
- Output ~2MB/s UART stream over dedicated SWO pin via STlinkv3.
- Orbcode is an open-source project to work with Cortex-M debug data.
- emdbg/ext/orbetto is a custom tool to convert ITM/DWT to ftrace packets.

[ext/orbetto]

Wouldn't it be great if we could instead let the hardware do the serialization and timestamping?

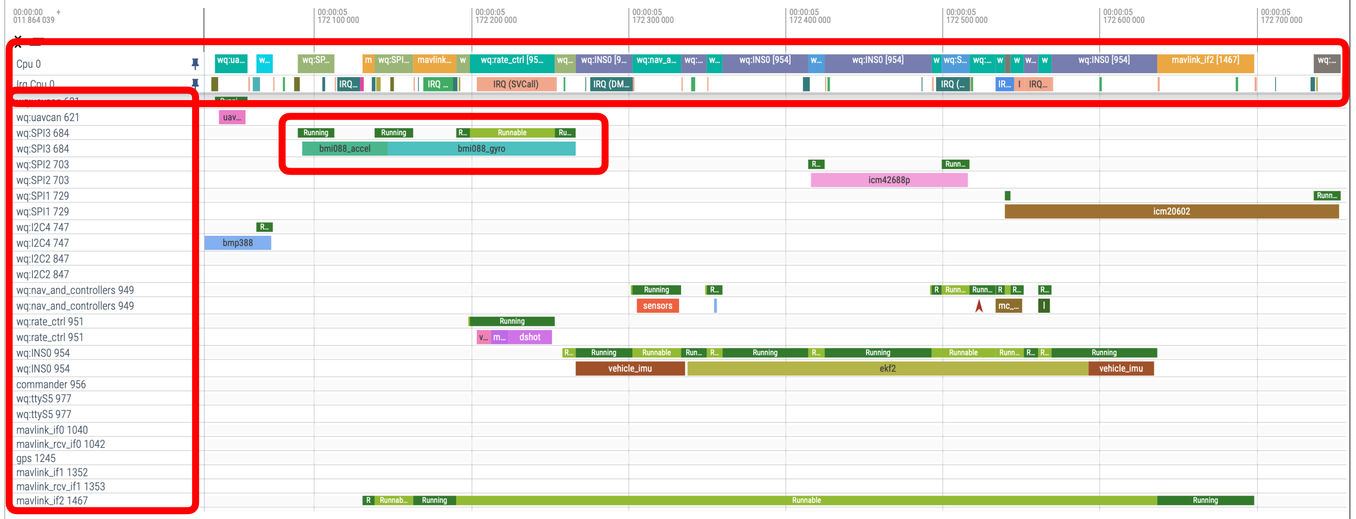
Well, look no further than the built-in Instrumentation Trace Macrocell (ITM) and Data Watchpoint and Trace (DWT) peripherals.
They provide 32 channels that you can write 8,16 or 32-bit values into and also logs exception entry and exit.
The whole thing is implemented in hardware, so you only need to add a single line statement to write to a ITM channel.

Serial Wire Output is basically a very fast UART. STLinkv3 can read up to 2MB/s via this link.

However, this is a compact binary format that we need to parse, which is what the Orbcode project provides.
With this parser we can demultiplex the bit stream and convert it to standard ftrace packets.
This tool is written in C++ and is called orbetto because.

Trace Visualization

via Perfetto UI



Try it online yourself: github.com/auterion/embedded-debug-tools => ext/orbetto

And this is then visualized by perfetto, which is actually meant to visualize Android and Linux traces.

At the top, you can see the CPU is multiplexing all the different threads, but you can also see the interrupts just below.
Note that is happening all within the same millisecond, each tick is 100µs.
NuttX schedules a lot, because it is an RTOS!

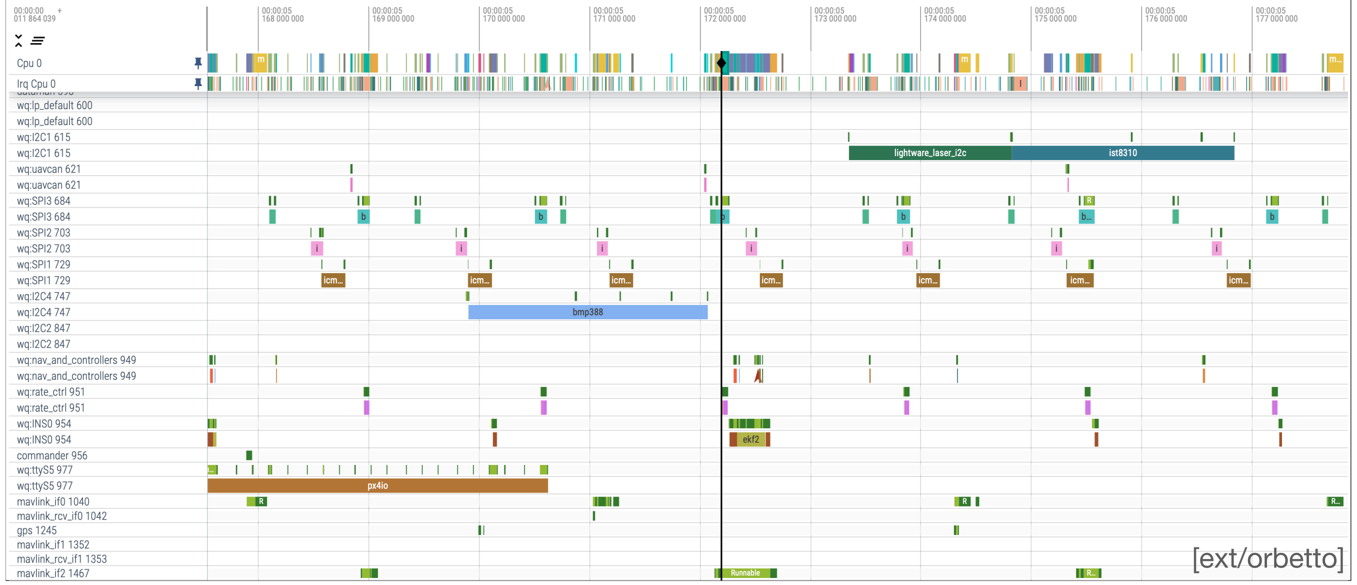
On the left you can the tasks with name and PID. PX4 has a lot of different threads.

We have a lot of work queues for all the sensors, which you can see when the workqueue item is called but often the thread actually gets interrupted a lot.

This view is incredibly educational to see how an RTOS actually works.

Trace Visualization

via Perfetto UI

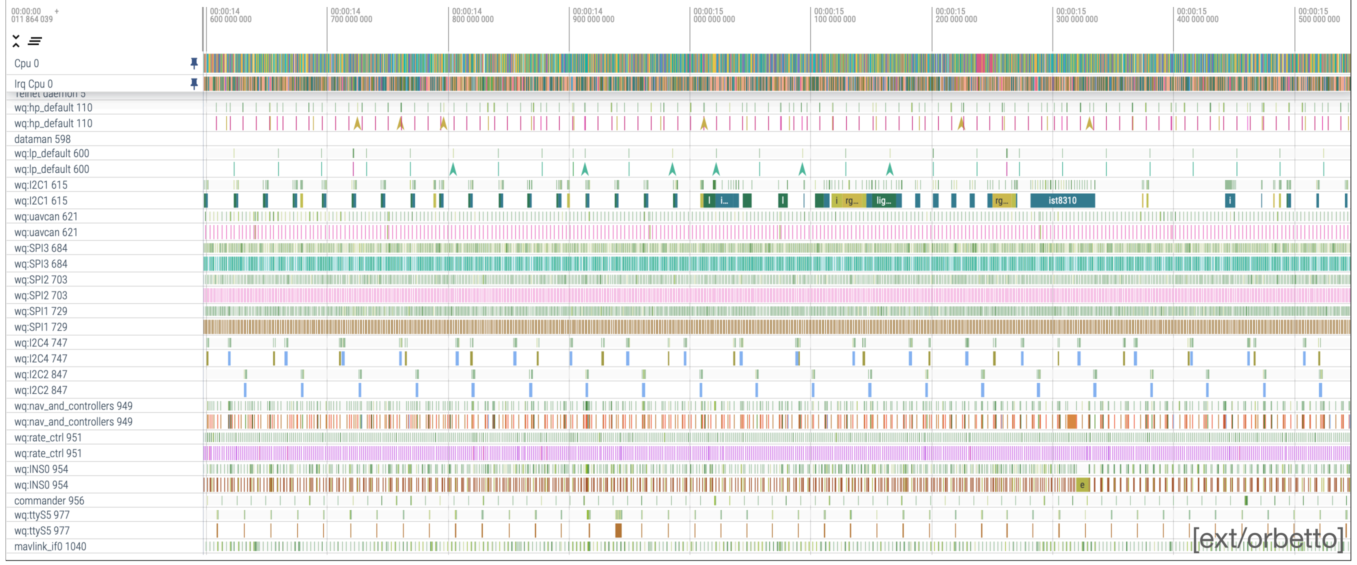


[ext/orbetto]

But it actually becomes more interesting if we zoom out, we can see some more patterns.
Here every tick is 1ms.

Spotting Timing Issues

via Perfetto UI



[ext/orbetto]

And even further: here every tick is 100ms.

And suddenly we see a hickup: The I2C1 task is irregular, because there was a power glitch and the sensors had to be reinitialized.

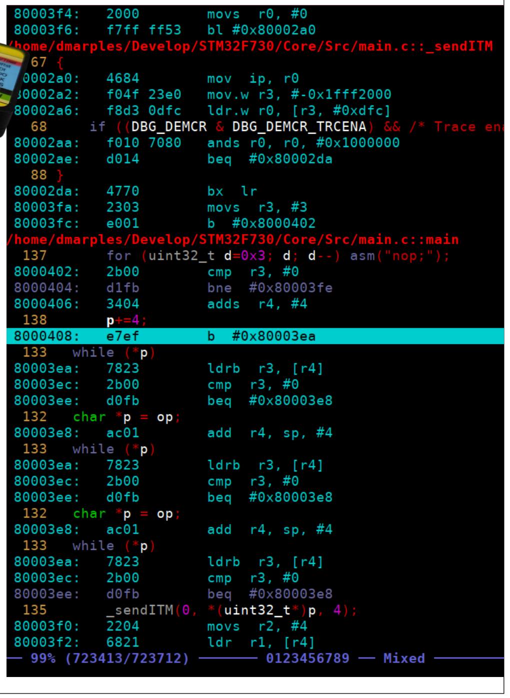
This kind of visual debugging with you eyes is incredibly fast to spot timing issues.
It also makes you look like a wizard if you just whip this out and show other people how PX4 works.

But SWO is still limited by bandwidth, so ARM provides you with even more hardware:

Real-time Tracing

ITM/DWT/ETM via TRACE pin

- ETM can trace all instructions.
- High-bandwidth 4-bit output interface ≤125MB/s requires FPGA and USB3
- J-Trace is ~2000€, ORBTrace is ~200€.
- ORBTrace mini is open-source! Go hack it!
- What do you do with so much data? Instrumentation for fuzzing: µAFL
- I'm looking for an intern to work on this at Auterion with me. Are you interested?



```
80003f4:   2000        movs  r0, #0
80003f6:   f7ff ff53   bl #0x80002a0
/home/dmarples/Develop/STM32F730/Core/Src/main.c::_sendITM
 67 {
0002a0:    4684        mov   ip, r0
80002a2:   f04f 23e0   mov.w r3, #-0x1fff2000
80002a6:   f8d3 0dfc   ldr.w r0, [r3, #0xdfc]
 68     if ((DBG_DEMCR & DBG_DEMCR_TRCENA) && /* Trace ena
80002aa:   f010 7080   ands r0, r0, #0x1000000
80002ae:   d014        beq  #0x80002da
 88 }
80002da:   4770        bx  lr
80003fa:   2303        movs  r3, #3
80003fc:   e001        b  #0x8000402
/home/dmarples/Develop/STM32F730/Core/Src/main.c::main
 137       for (uint32_t d=0x3; d; d--) asm("nop;");
8000402:   2b00        cmp  r3, #0
8000404:   d1fb        bne  #0x80003fe
8000406:   3404        adds  r4, #4
 138       p+=4;
8000408:   e7ef        b  #0x80003ea
 133   while (*p)
80003ea:   7823        ldrb  r3, [r4]
80003ec:   2b00        cmp  r3, #0
80003ee:   d0fb        beq  #0x80003e8
 132   char *p = op;
80003e8:   ac01        add  r4, sp, #4
 133   while (*p)
80003ea:   7823        ldrb  r3, [r4]
80003ec:   2b00        cmp  r3, #0
80003ee:   d0fb        beq  #0x80003e8
 132   char *p = op;
80003e8:   ac01        add  r4, sp, #4
 133   while (*p)
80003ea:   7823        ldrb  r3, [r4]
80003ec:   2b00        cmp  r3, #0
80003ee:   d0fb        beq  #0x80003e8
 135       _sendITM(0, *(uint32_t*)p, 4);
80003f0:   2204        movs  r2, #4
80003f2:   6821        ldr  r1, [r4]
— 99% (723413/723712) ——————— 0123456789 —— Mixed ————
```

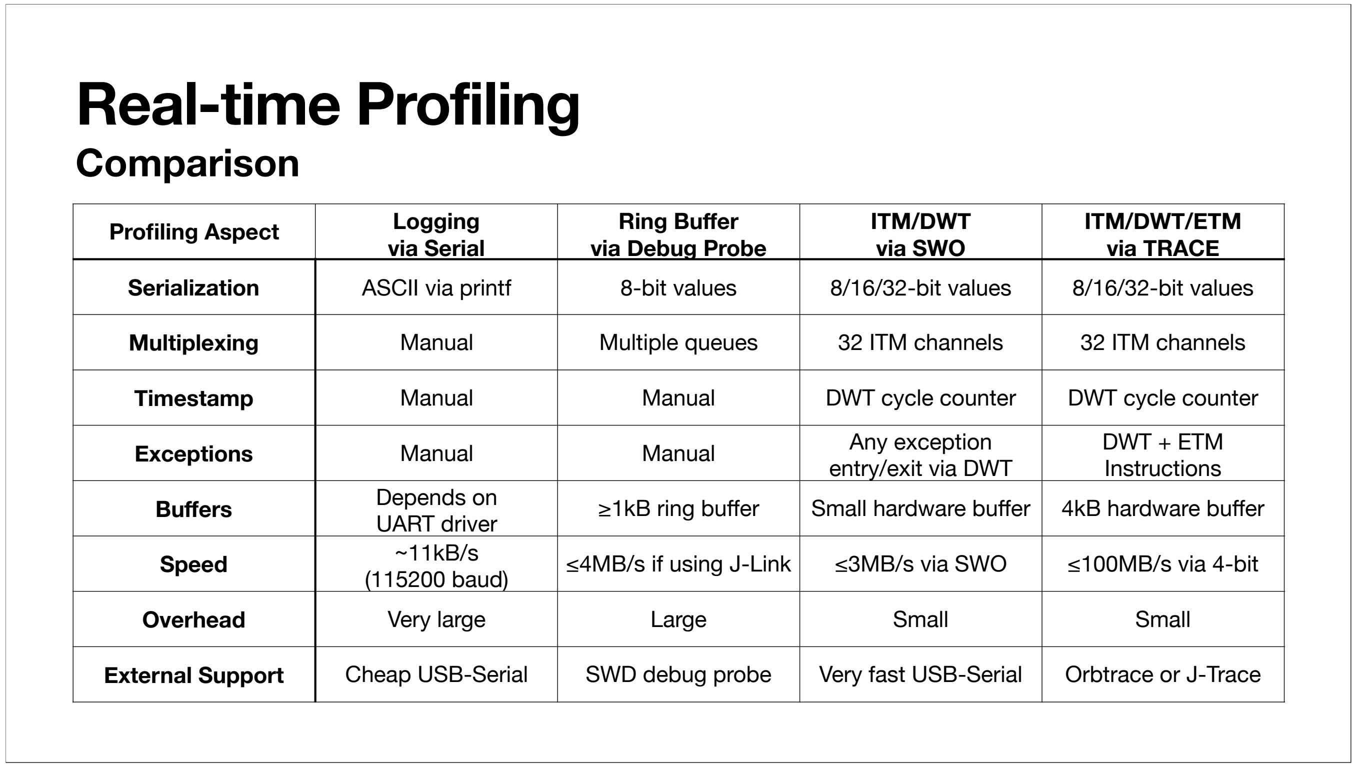
Parallel tracing gives you a 4-bit wide data bus up to 1GBit/s in theory.

In addition to all the previous functionality, there is also a compressed instruction trace, that allows you to reconstruct the program flow off device.
This requires custom FPGA hardware with a fast USB connection.

The J-Trace costs a lot of money. I got one for work, and it's not as well documented as I had hoped.
The Orbtrace is open-source, so it's much more hackable, and its 10x less expensive too.
There's a very helpful community around it, smart people work on this.
Go hack with it!

What do you do with this giant amount of data?
One research paper I saw used the branching information in the trace to run american fuzzy lop on the device and find lots of bugs in STM32 drivers.
It would be very helpful to find bugs in an Autopilot before they happen in flight!

I'm right now looking for an intern to work with me on this at Auterion!

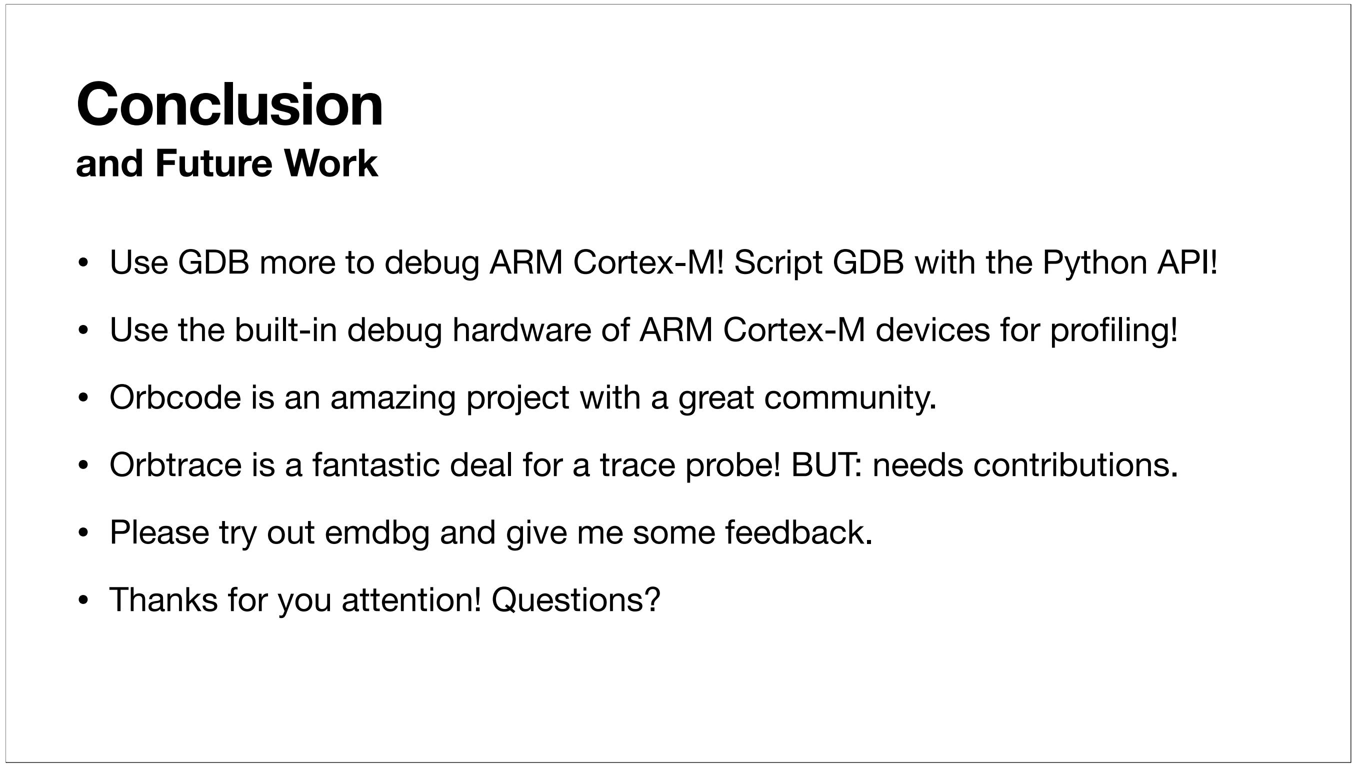


Real-time Profiling

Comparison

Profiling Aspect	Logging via Serial	Ring Buffer via Debug Probe	ITM/DWT via SWO	ITM/DWT/ETM via TRACE
Serialization	ASCII via printf	8-bit values	8/16/32-bit values	8/16/32-bit values
Multiplexing	Manual	Multiple queues	32 ITM channels	32 ITM channels
Timestamp	Manual	Manual	DWT cycle counter	DWT cycle counter
Exceptions	Manual	Manual	Any exception entry/exit via DWT	DWT + ETM Instructions
Buffers	Depends on UART driver	≥1kB ring buffer	Small hardware buffer	4kB hardware buffer
Speed	~11kB/s (115200 baud)	≤4MB/s if using J-Link	≤3MB/s via SWO	≤100MB/s via 4-bit
Overhead	Very large	Large	Small	Small
External Support	Cheap USB-Serial	SWD debug probe	Very fast USB-Serial	Orbtrace or J-Trace

SWO is the sweet spot regarding cost of debug probe and feature set.
Orbtrace mini is pretty good for high-bandwith Tracing ≤400Mbit/s.



Conclusion

and Future Work

- Use GDB more to debug ARM Cortex-M! Script GDB with the Python API!
- Use the built-in debug hardware of ARM Cortex-M devices for profiling!
- Orbcode is an amazing project with a great community.
- Orbtrace is a fantastic deal for a trace probe! BUT: needs contributions.
- Please try out emdbg and give me some feedback.
- Thanks for you attention! Questions?

So this is the end of the line, this is all the hardware and tooling I currently use for debugging.

In conclusion: Debug all the things!
Use more GDB, use more debug hardware!

Test and contribute to the Orbcode project, play around with the Orbtrace.
They are looking for competent embedded people to liberate the debug tools from commercial vendors!

Oh and also try out my embedded debug tools or maybe look at them for inspiration.

Thank you and do you have questions?

Debugging Microcontrollers

A debugging session is active. Quit anyway? (y or n) y

Niklas Hauser likes microcontrollers.

Homepage: salkinium.com

Fediverse: @salkinium@chaos.social

Code: github.com/salkinium

emdbg: github.com/auterion/embedded-debug-tools

Orbcode: orbcode.org

My name is Niklas and I like microcontrollers.