

Thank you for the introduction.
Thanks to the conference organizers for the opportunity to talk here.

My name is Niklas.

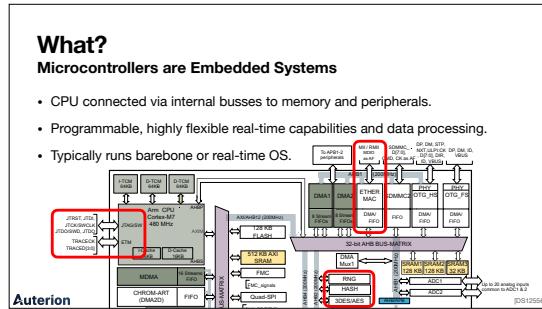
- I started studying Computer Science some time ago.
- I began building autonomous robots in 2010.
- We created a C++ library which is known as [modm.io](#), a C++23 library generator that supports 3700+ Cortex-M devices.
- I then started at ARM working on Cortex-M sandboxing, before returning to the university to study for my masters degree.
- There, I worked on a digital modular signalling system for railways.
- I finished my masters degree and now work at Auterion debugging the open-source PX4 Autopilot for commercial drones.

In particular, I'm debugging the Skynode, which contains a Linux system and a flight management unit, which runs on the PX4 Autopilot software.

- PX4 is based on the NuttX RTOS which is complex and has some subtle bugs now and then.
- My job is to debug and improve PX4 and NuttX.
- Difficult because large code base and fast processor, which limits printf

debugging.

- I want to share some of the tools I wrote over the last few months to help me debug PX4.



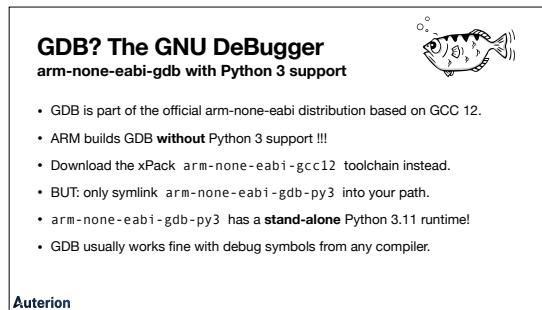
This talk is about microcontrollers, specifically with the ARM Cortex-M architecture.

Microcontrollers contain a microprocessor, here a Cortex-M7 in light green on the left, connected via a bus system in gray to non-volatile memories, like Flash, and volatile memories like SRAM (yellow), as well as a number of special purpose peripherals.

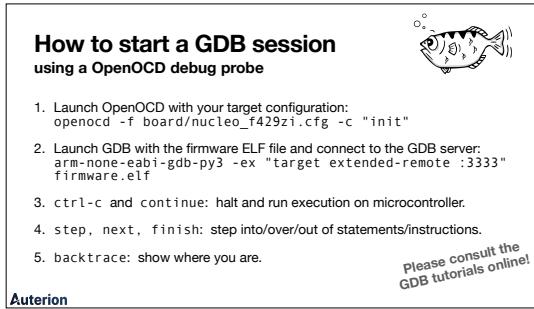
Peripherals can be internal, like the Random Number Generator (RNG) down here,

or external, like the Ethernet MAC up there which connects over Media Independent-Interface (MII) to an external PHY via the microcontroller pins.

The CPU itself can be debugged using the Serial Wire Debug connection here on the left.



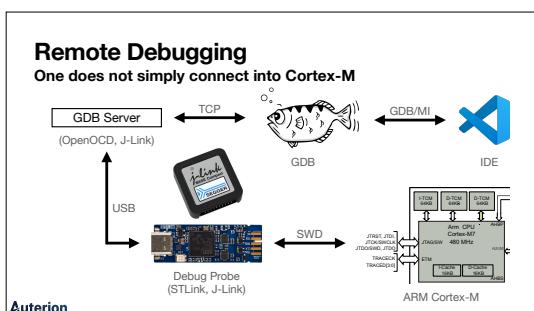
- GDB is the debugger that comes with your arm-none-eabi toolchain
- ARM provides an official and tested version, use that one for compilation.
- But the GDB is not compiled with Python3 support.
- So you need to install the xPack version, but only symlink the GDB, not the rest
- Alternatively use the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



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.

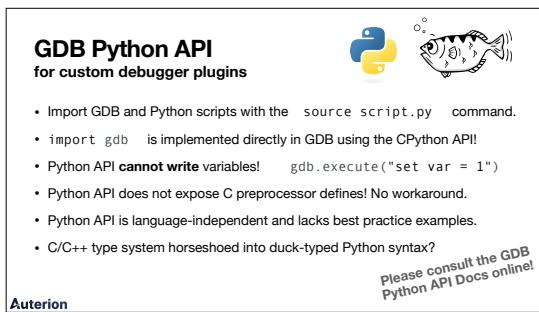
There are many GDB tutorials online, please refer to them if you're a beginner.



Debugging microcontrollers requires some extra steps.

- You need to connect the Serial Wire Debug (SWD) signals to a hardware debug probe
- For example a J-Link or a STLink
- The debug probe then communicates over USB to the driver software
- Typically this is OpenOCD or JLinkGDBServer

- Which 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
-



GDB has its 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 independent API can be a bit wonky for C/C++ semantics

So I wrote a bunch of tools and plugins for GDB for NuttX.

Embedded Debug Tools: emdbg
a modular toolbox for scripting GDB

BSD

- 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 PX4+NuttX+STM32, 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!

Auterion reference to module: [emdbg.debug.gdb#px4_tasks]

All tools in this talk are available as an open-source project called the 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 simple GDB tools to start.

Inspecting NuttX tasks
with PX4 extensions for CPU usage

NSH shows this via top! 

```
(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	0x20002a00
699	wq:SP13	7.2	1336/ 2336	w:sem	0x20005460
718	wq:I2C4	0.4	912/ 2336	w:sem	0x2000fd80
839	wq:nav_ctrlr	4.1	1276/ 2280	w:sem	0x2000c300
840	wq:rate_ctrl	7.7	1492/ 3152	w:sem	0x20016420
842	wq:IN50	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

Auterion [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.

NuttX has a similar tool, but PX4 has an external CPU usage monitor, so we implement our own.

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)
        self.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, ...
=> Easier to implement fully in GDB than via RTOS plugin of debug probe.
```

Auterion [emdbg.debug.gdb#px4_tasks]

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 scaled up or down



Inspecting Interrupt State
NuttX intercepts NVIC to implement IRQs

```
(gdb) px4_interrupts
IRQ EPA P ADDRESS = FUNCTION ARGUMENT
-1 e - 0x8020c4 = arm_hardfault
-1 e 0 0x80221c = arm_nmi_call
-1 e 80 0x8011efc = stm32_timerisr
1 e 80 0x800936c = stm32_dmainterrupt 0x20020740 <g_dma>
12 e 80 0x800936c = stm32_dmainterrupt 0x20020758 <g_dma+24>
21 e 80 0x816d8f8 = io_time_handler0
31 e 80 0x800936c = up_i2c_isr 0x20020944 <sm32_i2c1_priv>
37 e 80 0x808dd4 = up_interrupt 0x200203a0 <g_usart1priv>
40 e 80 0x8167df8 = stm32_ext1510_isr
59 e a 80 0x800936c = stm32_dmainterrupt 0x20020848 <g_dma+264>
65 ep 80 0x813004 = can2_Trig
103 e 80 0x816c1f64 = sm32_sdmmc_interrupt

Auterion [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_3V3_SENSORS1
A3 IN USART2_RX USART2_RX_TELEM3
A5 ALT+H X 0 SPI1_SCK SPI1_SCK_SENSORS1_ICM20602
A6 IN SPI1_MISO SPI1_MISO_EXTERNAL1
A8 IN+PU ^ TIM1_CH1 FMU_CH4
A9 OUT USB_OTG_FS_VBUS VBUS_SENSE
A11 ALT+VH - 0 USB_OTG_FS_DM USB_D_P
A12 ALT+VH - 0 USB_OTG_FS_DP USB_D_P
A13 ALT+PU+VH S SWDIO FMU_SWDIO
A14 ALT+PD X 0 SWCLK FMU_SWCLK
A15 OUT ^ ^ SPI16_NCS2_EXTERNAL1

Auterion [emdbg.debug.gdb#px4_gpios]
```

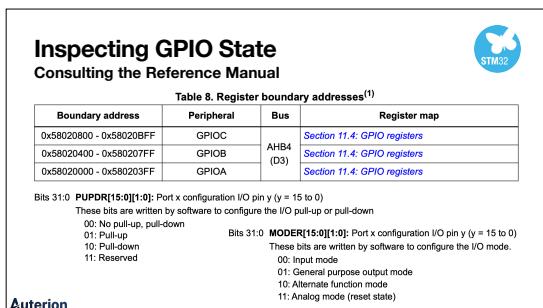
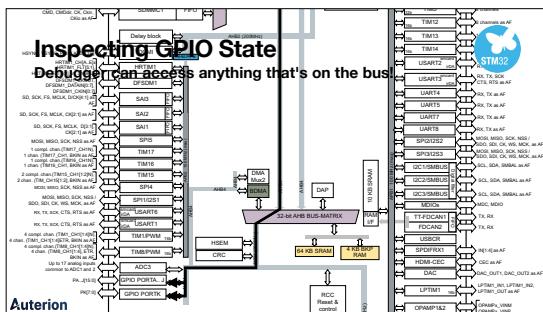
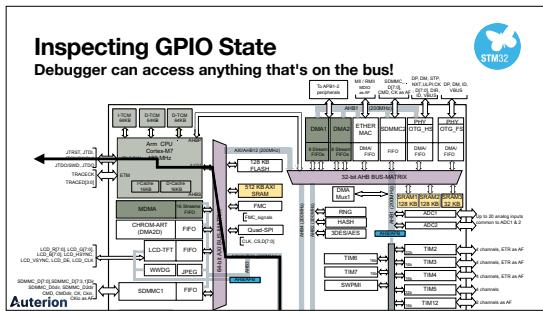
We also have some STM32 specific tools.

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).

But of course this is a custom implementation that manually interprets the



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

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

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?

The slide has a CMSIS logo in the top right corner. The title is 'Inspecting Any Peripheral using System View Description (SVD) files'. Below the title is a bulleted list of benefits:

- 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.

At the bottom left is the 'Auterion' logo, and at the bottom right is '[emdbg.debug.gdb#px4_pshow]'

So instead we are using the System View Description files to automatically interpret peripherals.

SVD files are 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.

The slide has a CMSIS logo in the top right corner. The title is 'Inspecting Any Register using System View Description (SVD) files'. Below the title is a screenshot of a GDB session showing the raw register value and its bitfield descriptions:

```
(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
MIE .....0.. - 0 // Memory increment interrupt enable
TCIE .....0.. - 0 // Transfer complete interrupt enable
PFCCTRL .....0.. - 0 // Peripheral flow controller
DIR .....01.... - 0 // 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
```

At the bottom left is the 'Auterion' logo, and at the bottom right is '[emdbg.debug.gdb#px4_pshow]'

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 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.  
    at 0x00000208 <exception_common+0>: ef f3 05 80 mrs r0, IPSR HardFault triggered  
    (gdb) arm scb /h  
    CFSR      = 00008200  
    BFRVALID  = ...8... - 1  
    PRECISERR = 00000000  
    BFRAR     = 00000008  
    BusFault at address 8!  
#0 exception_common ()  
#1 <signal handler called>  
#2 inode_insert (parent=0x0, peer=0x0, node=0x2007c010) Backtrace works beyond exception!  
    (gdb) frame 2  
11:     node->_1_peer  = parent->_1_child;  
    => 0x0000ad14 <inode_reserve+108>: a3 68 ldr r3, [r4, #8] r4=0 +8 Loading  
    Autonion [emdbg.debug,gdb#debugging-hardfaults]
```

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.

However, on Cortex-M7 the write may be delayed relative to the execution, so it doesn't always show the right location. In NuttX, all register access are typically done via a function, onto which you can set a breakpoint instead. Also DMA writes in the background may result in weird backtraces.

HardFault Debugging
using GDB with vector catch in DEMCR

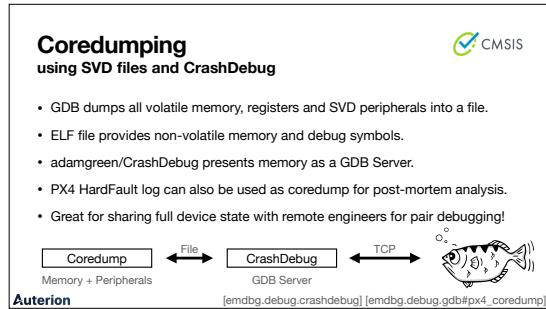
CMSIS

```
Program received signal SIGTRAP, Trace/breakpoint trap.  
=> 0x00000208 <exception_common+0>: ef f3 05 80 mrs r0, IPSR HardFault triggered  
    (gdb) arm scb /h  
    CFSR      = 00008200  
    BFRVALID  = ...8... - 1  
    PRECISERR = 00000000  
    BFRAR     = 00000008  
    BusFault at address 8!  
#0 exception_common ()  
#1 <signal handler called>  
#2 inode_insert (parent=0x0, peer=0x0, node=0x2007c010) Backtrace works beyond exception!  
    (gdb) frame 2  
11:     node->_1_peer  = parent->_1_child;  
    => 0x0000ad14 <inode_reserve+108>: a3 68 ldr r3, [r4, #8] r4=0 +8 Loading  
    Autonion [emdbg.debug,gdb#debugging-hardfaults]
```

We can also investigate hardfaults while debugging using the SVD files: We enable the vector catch bits to trap all fault exceptions and then the device halts **before** the NuttX interrupt dispatcher breaks the backtrace.

You can inspect the fault registers via the SVD plugin: here it's a precise bus fault at address 8.

The backtrace is intact, so we can find the exact location.
Here, the parent pointer is zero, thus the load + 8 fails.



Related to hardfaulting is coredumping support.

Not natively implemented, so we need to do it ourselves:

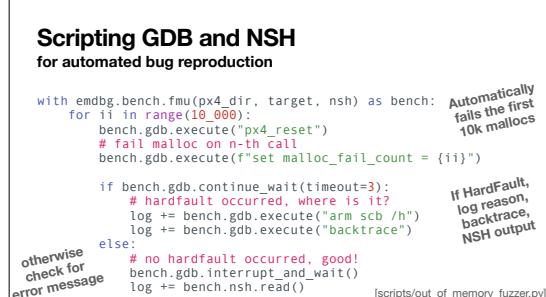
read out all volatile memory like SRAM and peripherals and store them in a file.

We also use the SVD files to read out the peripherals as well and then all peripheral tools like the GPIOs tool also works.

The CrashDebug utility from Adam Green then pretends to be a GDB server and serves the memory.

Works really well for post-mortem debugging of hardfaults to get a backtrace and reason for the fault.

This also works with PX4 hardfault logs!



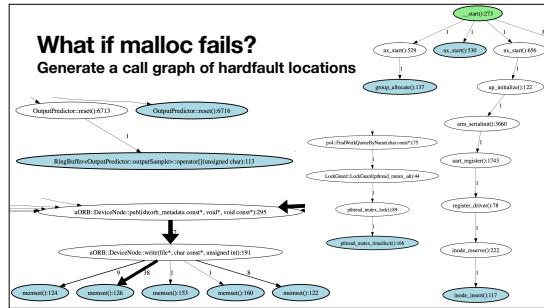
I've also implemented a basic GDB/MI interface, so that you can create scripts with GDB and NSH interaction.

Here we systematically fail the n-th malloc call to see what code does not check the returned pointer for NULL.

If the vector catch works, then we can print the fault registers and the backtrace.

Otherwise we read the NSH output, since there may be an error log.

We then convert the backtraces to a call graph.



We visualize the call graph using graphviz, to show where all the hardfault locations are.

We already know about inode_insert, but there are a bunch of NULL pointers passed to memset in uORB.

Much easier to test and fix OOM conditions with GDB scripting.

All of these tools are running, while the CPU is halted. What if you cannot halt the CPU? Let's talk about profiling!

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.

Auterion

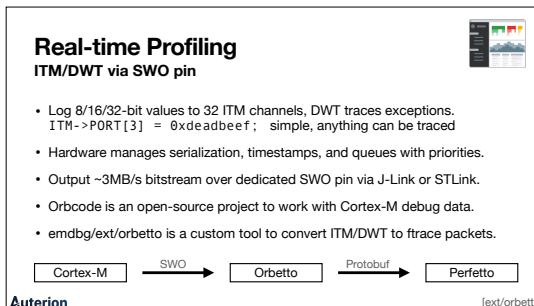
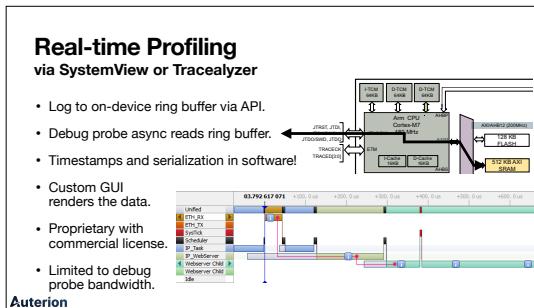
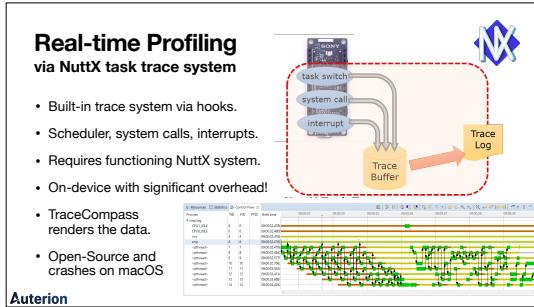
The simplest profiling method is logging, usually over Serial link.

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.



NuttX has a built-in task trace system.
It logs events to RAM and then to a file.
But it renders very nicely in TraceCompass.

Unfortunately TraceCompass crashes on ARM64 macOS.
It also runs on-devices, so it modifies timings and uses CPU time and program

So, can we externalize the profiling?

Simple idea: log to ring buffer in SRAM and let the debug probe do the transfer.
This is the idea behind SEGGER's 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 is not extensible.

And, still actually a fairly high overhead.

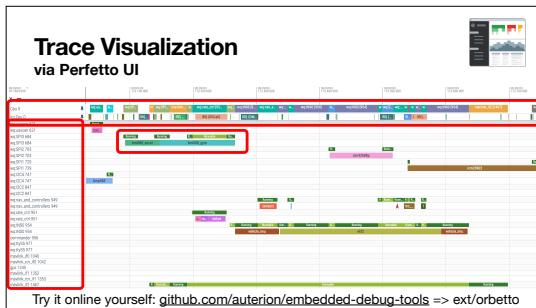
Wouldn't it be great if we could instead let the hardware do the serialization and timestamping?

Well, this is exactly what the the built-in ITM and DWT peripherals do.
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.

CPU overhead (<5%) only for waiting for space in the buffer, uses barely any program space (~170B).

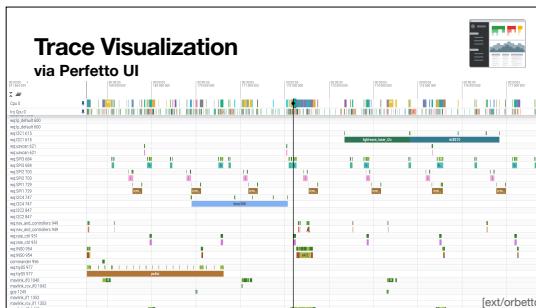
Serial Wire Output is basically a very fast UART. J-Link can read up to 3MB/s



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 100 μ s. NuttX schedules a lot, because it is an RTOS!
- On the left you can see 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.

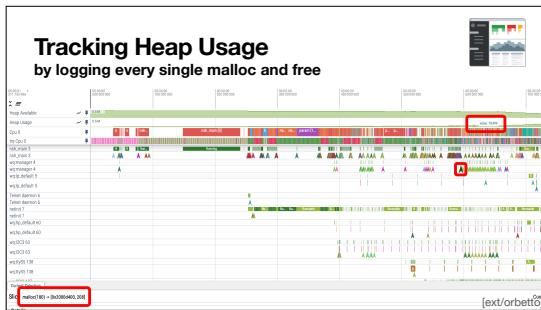
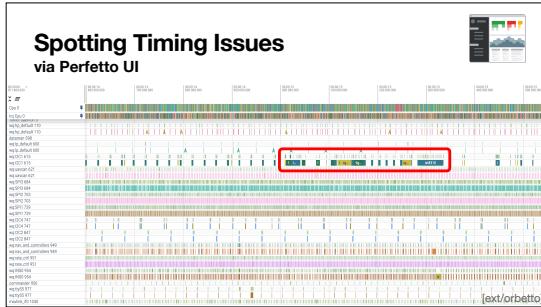
This view is incredibly educational to see how an RTOS actually works.



But it actually becomes more interesting if we zoom out, we can see some more patterns.

Here every tick is 1ms.

You can see the sensors on the SPI busses getting read periodically.



And even further: here every tick is 100ms.

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

This kind of visual debugging with your eyes is incredibly fast to spot timing issues.

You can really track **anything** over time.

For example, every malloc/free call, which helps you understand the heap usage.

Here you can see a single malloc call with the requested size and the returned pointer and allocated size including overhead.

By adding mallocs and subtracting frees, you can compute the total heap usage over time.

I didn't find a good UI for this, but you could even analyze heap fragmentation using this information.

Also create a histogram of allocation sizes for optimizing a binning block allocator for your application.

Incredibly useful and it also makes you look like a wizard if you just whip this out and show other people how PX4/NuttX works.

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



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. Does not give you data flow, you must still instrument your code via ITM for that.

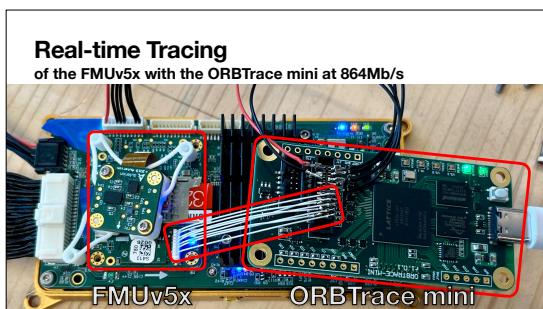
The J-Trace costs a lot of money.

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, particularly if you're new to tracing!

Derived information: Full stack traces, code coverage, timing analysis, complete RTOS state, branching information.

Goal is to combine this with GDB scripting and automated testing in the CI. Somehow.



I got this working only last week, so this is what the latest state of the debug tools development.

Here the Skynode FMU is connected to the Orbtrace to transfer around 864Mb/s of debug information, which is great.

- On the left is the FMUv5x on the Auterion Skynode.
- One right right is the ORBTrace mini

- And in the middle is the 4-bit parallel TRACE connection. This required a custom short cable to maintain the signal integrity, so that's why it looks so hacky.

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 + DWT sources	32 ITM channels + DWT/ETM sources
Timestamp	Manual	Manual	Hardware cycle counter from ITM	Hardware cycle counter from ETM
Exceptions	Manual	Manual	Any exception entry/exit via DWT	DWT exceptions + ETM instructions
Buffers	Depends on UART driver (~11KB/s (115200 baud))	>1kB ring buffer	10B (l) hardware buffer	4kB hardware buffer
Speed		<4MB/s if using J-Link	<3MB/s via SWO	<133MB/s via TRACE
Overhead	Very large	Large	Small	Very small
External Support	Cheap USB-Serial	SWD debug probe	Very fast USB-Serial	ORBTrace or J-Trace

Auterion

Using more specialized hardware for profiling is better, what a surprise!1!!

Future Work

- Add support for TraceCompass/Tracy since it has better analysis tools.
- Synchronize traces with logic analyzer captures via sigrok?
- Integrate parallel trace into emdbg for massive bandwidth increase:
 - Complete call stacks and automated timing analysis on them.
 - Extract all RTOS state via ITM: can we use Valgrind for analysis?
 - Fuzzing guided by instruction trace: systematically inject failures.

Auterion

In the future, I want to support TraceCompass or the Tracy Profiler, since it contains better analysis tools than perfetto.

We also display logic analyzer traces in Perfetto for a better overview of what happens outside the device.

Parallel trace needs to be integrated:

- Perfetto with full call stacks in real-time.
- Complete RTOS state: semaphores, spinlocks, critical sections, etc. in real-time
- Use Instruction trace to guide american fuzzy lop in conjunction with logic analyzer.

Conclusion

- Debugging and profiling are a very useful tools to have!
- 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.

Auterion

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?