Auterion

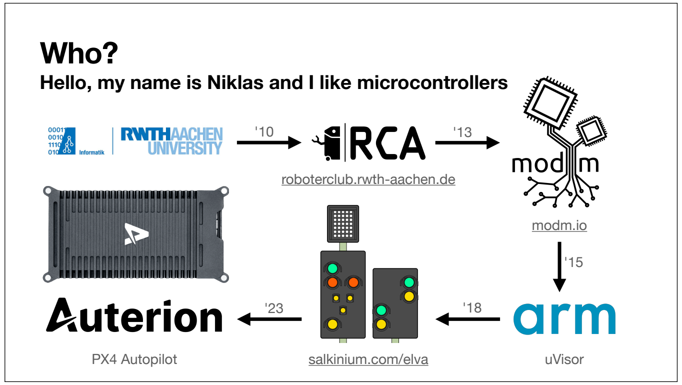
Analyzing Cortex-M Firmware

with the Perfetto Trace Processor

Niklas Hauser
Senior Embedded Software Engineer

emBO++

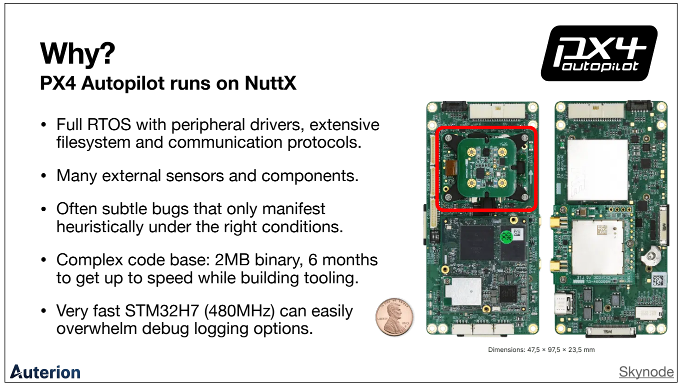
Thank you for the introduction.
Thanks to the conference organizers for the opportunity to talk here.



Who?

Hello, my name is Niklas and I like microcontrollers

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.



Why?

PX4 Autopilot runs on NuttX

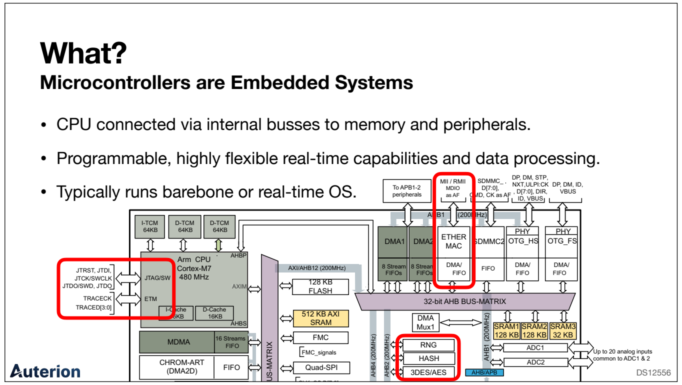
- Full RTOS with peripheral drivers, extensive filesystem and communication protocols.
- Many external sensors and components.
- Often subtle bugs that only manifest heuristically under the right conditions.
- Complex code base: 2MB binary, 6 months to get up to speed while building tooling.
- Very fast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

Auterion

Skynode

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 trace PX4.



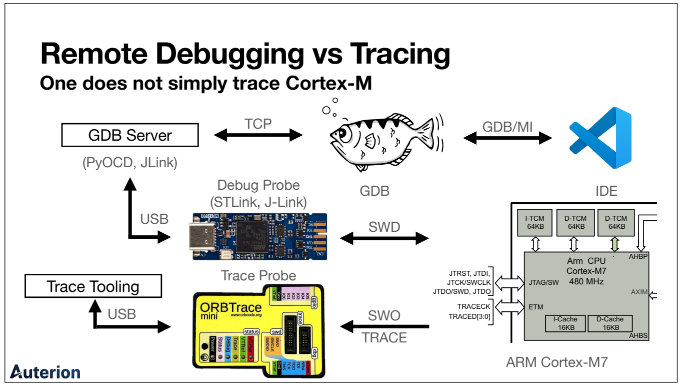
What?

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.

Auterion

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 Intependent-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. Tracing uses the SWO and TRACE pins.

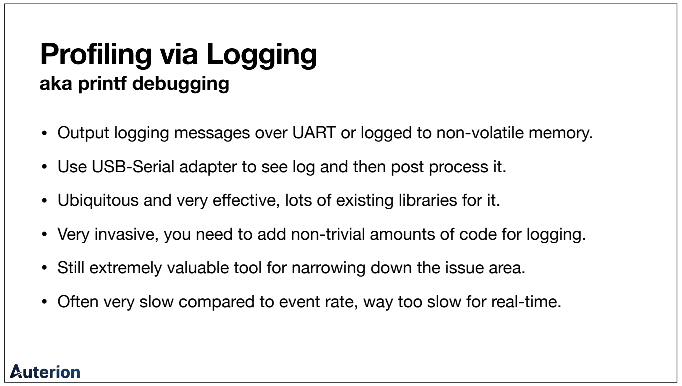


Remote Debugging vs Tracing

One does not simply trace Cortex-M

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, PyOCD 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

For tracing, you connect to the output-only SWO and TRACE pins. However, after that no standardized infrastructure exists. So let's have a look at tracing approaches.

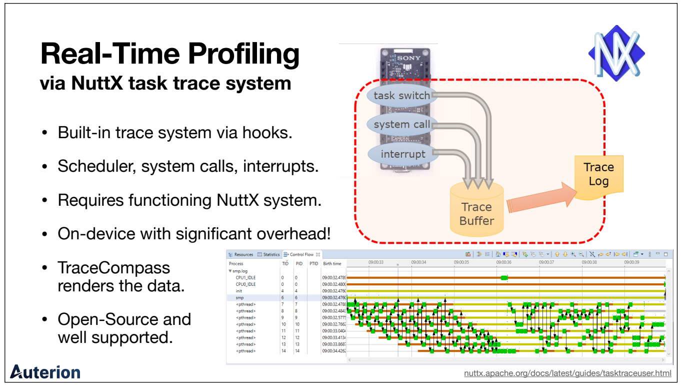


Profiling via Logging

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.

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

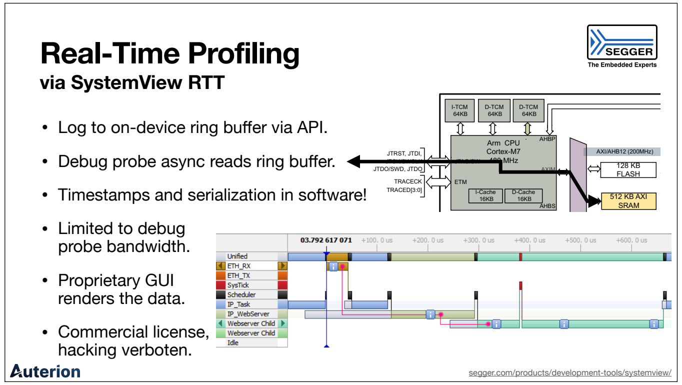
via NuttX task trace system

- Built-in trace system via hooks.
- Scheduler, system calls, interrupts.
- Requires functioning NuttX system.
- On-device with significant overhead!
- TraceCompass renders the data.
- Open-Source and well supported.

Auterion

nuttx.apache.org/docs/latest/guides/tasktraceuser.html

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. It also runs on-devices, so it modifies timings and uses CPU time and program space.



Real-Time Profiling

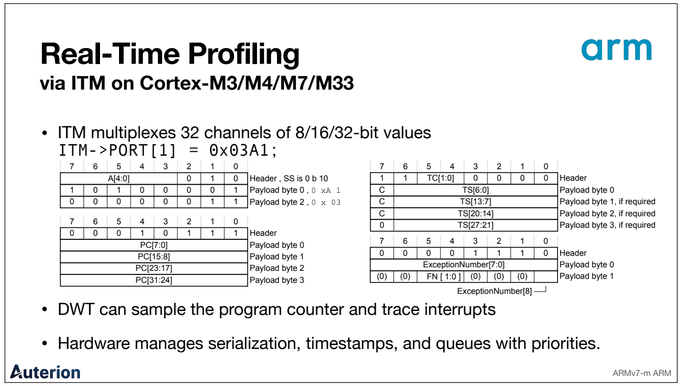
via SystemView RTT

- Log to on-device ring buffer via API.
- Debug probe async reads ring buffer.
- Timestamps and serialization in software!
- Limited to debug probe bandwidth.
- Proprietary GUI renders the data.
- Commercial license, hacking verboten.

Auterion

segger.com/products/development-tools/systemview/

So, can we externalize the profiling cost? 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 is not extensible.



Real-Time Profiling

via ITM on Cortex-M3/M4/M7/M33

- ITM multiplexes 32 channels of 8/16/32-bit values
  `ITM->PORT[1] = 0x03A1;`

7	6	5	4	3	2	1	0	
A[4:0]					0	1	0	Header , SS is 0 b 10
1	0	1	0	0	0	0	1	Payload byte 0 , `0 xA 1`
0	0	0	0	0	0	1	1	Payload byte 2 , `0 x 03`

7	6	5	4	3	2	1	0	
0	0	0	1	0	1	1	1	Header
PC[7:0]								Payload byte 0
PC[15:8]								Payload byte 1
PC[23:17]								Payload byte 2
PC[31:24]								Payload byte 3

7	6	5	4	3	2	1	0	
1	1	TC[1:0]		0	0	0	0	Header
C	TS[6:0]							Payload byte 0
C	TS[13:7]							Payload byte 1, if required
C	TS[20:14]							Payload byte 2, if required
0	TS[27:21]							Payload byte 3, if required

7	6	5	4	3	2	1	0	
0	0	0	0	1	1	1	0	Header
ExceptionNumber[7:0]								Payload byte 0
(0)	(0)	FN [1:0]		(0)	(0)	(0)		Payload byte 1

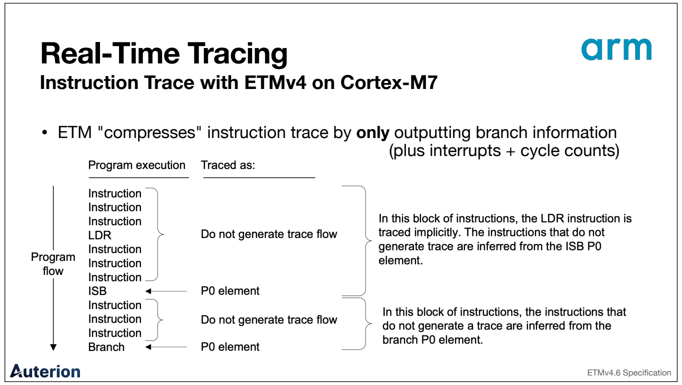
ExceptionNumber[8]

- DWT can sample the program counter and trace interrupts
- Hardware manages serialization, timestamps, and queues with priorities.

Auterion

ARMv7-m ARM

Wouldn't it be great if we could instead let the hardware do the serialization and timestamping? Well, this is exactly what the built-in ITM peripheral does. 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 only for waiting for space in the buffer, uses much less program space. The DWT peripheral can also output interrupts entry/exit and program counter samples. The overhead is only one header byte, and for this you get reliable framing and prioritization for free in hardware.



Real-Time Tracing

Instruction Trace with ETMv4 on Cortex-M7

arm

- ETM "compresses" instruction trace by **only** outputting branch information (plus interrupts + cycle counts)

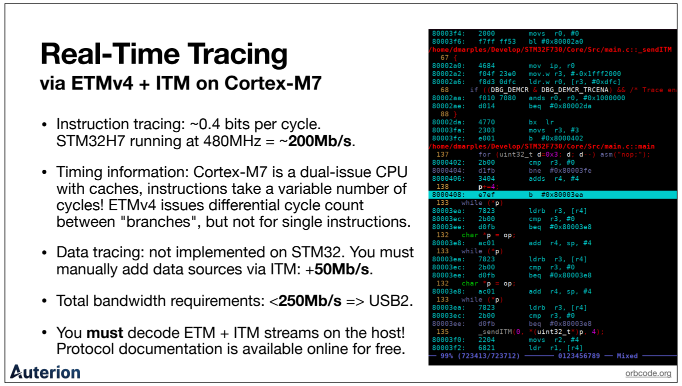
Program flow

Program execution	Traced as:	
Instruction Instruction Instruction LDR Instruction Instruction Instruction	Do not generate trace flow	In this block of instructions, the LDR instruction is traced implicitly. The instructions that do not generate trace are inferred from the ISB P0 element.
ISB	P0 element	
Instruction Instruction Instruction	Do not generate trace flow	In this block of instructions, the instructions that do not generate a trace are inferred from the branch P0 element.
Branch	P0 element	

Auterion

ETMv4.6 Specification

Then finally there is the ETM peripheral which allows for instruction trace. This is by far the most complicated trace peripheral, with *many* configuration options. The basic idea is relatively simple: only output which branch was taken, the other instructions in between are known and can be "traced" off-device. In practice it's a bit more complicated when dealing with conditional execution, interrupts, and cycle counts.



Real-Time Tracing

via ETMv4 + ITM on Cortex-M7

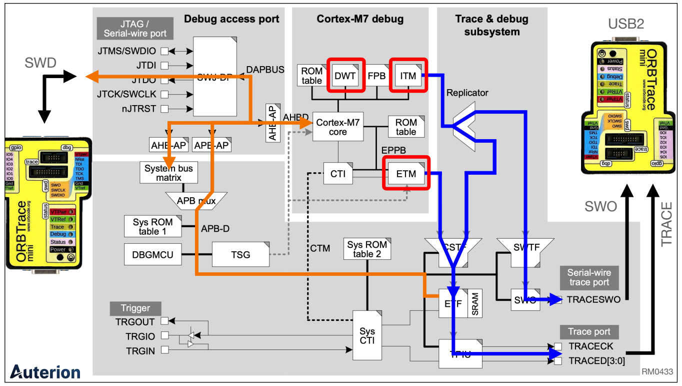
- Instruction tracing: ~0.4 bits per cycle.
  STM32H7 running at 480MHz = ~**200Mb/s**.
- Timing information: Cortex-M7 is a dual-issue CPU with caches, instructions take a variable number of cycles! ETMv4 issues differential cycle count between "branches", but not for single instructions.
- Data tracing: not implemented on STM32. You must manually add data sources via ITM: +**50Mb/s.**
- Total bandwidth requirements: <**250Mb/s** => USB2.
- You **must** decode ETM + ITM streams on the host! Protocol documentation is available online for free.

```
80003f4:   2000        movs  r0, #0
80003f6:   f7ff ff53   bl #0x80002a0
/home/dmarples/Develop/STM32F730/Core/Src/main.c::_sendITM
  67 {
80002a0:   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 en
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 ──────
```

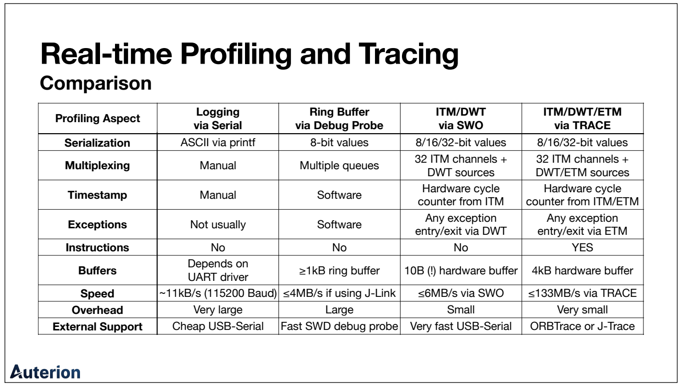
Auterion

orbcode.org

The required bandwidth is about 0.3-0.4 bits per cycle, so for a 480MHz STM32H7, we're looking at about 200Mb/s. Cortex-M7 is a beast: can process two instructions at the same time, has branch prediction, aggressive caching. Instruction timing is variable, thus ETM only gives use cycle counts between branch information, not for individual instructions. Data tracing is not implemented on STM32. So we must manually instrument any data we want with the ITM. Together we're looking at a 250Mb/s data stream, which fits comfortably into USB2. Reconstructing the program flow looks like on the right. This is done with the orbcode libraries.



So where does the data go? This is the debug and trace subsystem of the STM32H7. You can see the CPU code in the middle connected to the DWT, ITM, and ETM peripherals. You can connect your debugger on the left via SWD and then access the internals via the DAP. You can redirect the ITM output to an internal 4kB FIFO and then read this out via the SWD debugger. This is basically a hardware accelerated version of the RTT protocol. You can also redirect the ITM output to the SWO pin, which is a very fast UART. The super cheap STLinkv3 can trace this up to 2.4MB/s. ORBTrace mini can do 6MB/s, some (expensive) J-Links/J-Traces even higher. To output the ETM, you can only output it over the 4-bit parallel trace port with up to ~1Gb/s bandwidth (<133MB/s). For this you need a trace tool, here an open-source version called the ORBTrace mini, which is 10x cheaper than the J-Trace. Sadly there is no way to redirect the internal 4kB ETF buffer to SWO.

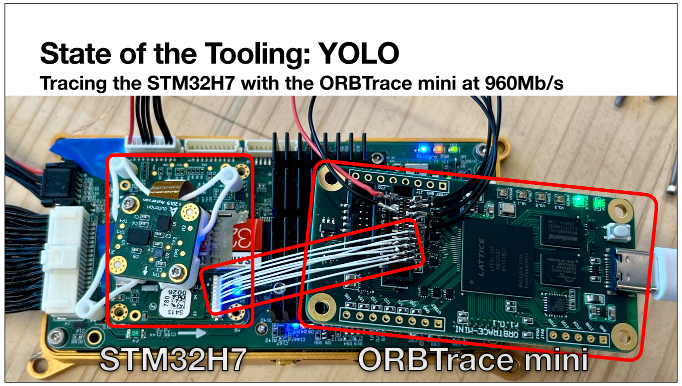


Real-time Profiling and Tracing

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	Software	Hardware cycle counter from ITM	Hardware cycle counter from ITM/ETM
Exceptions	Not usually	Software	Any exception entry/exit via DWT	Any exception entry/exit via ETM
Instructions	No	No	No	YES
Buffers	Depends on UART driver	≥1kB ring buffer	10B (!) hardware buffer	4kB hardware buffer
Speed	~11kB/s (115200 Baud)	≤4MB/s if using J-Link	≤6MB/s via SWO	≤133MB/s via TRACE
Overhead	Very large	Large	Small	Very small
External Support	Cheap USB-Serial	Fast SWD debug probe	Very fast USB-Serial	ORBTrace or J-Trace

Using more specialized hardware for profiling the better, what a surprise!1!! The 10B (yes!) hardware buffer for SWO requires busy-waiting the CPU when writing to the ITM in bursts.

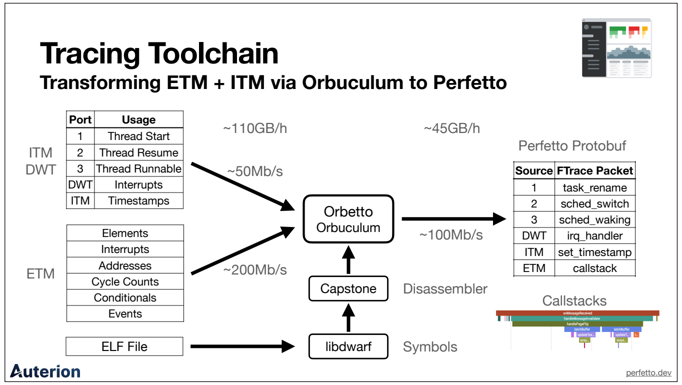


State of the Tooling: YOLO

Tracing the STM32H7 with the ORBTrace mini at 960Mb/s

Here the Skynode FMU is connected to the Orbtrace to transfer around 960Mb/s of debug information, which is great.

- On the left is the STM32H7 on the Auterion Skynode.
- One right right is the ORBTrace mini
- And in the middle is the 4-bit parallel TRACE connection. Not even length matched, still works fine.



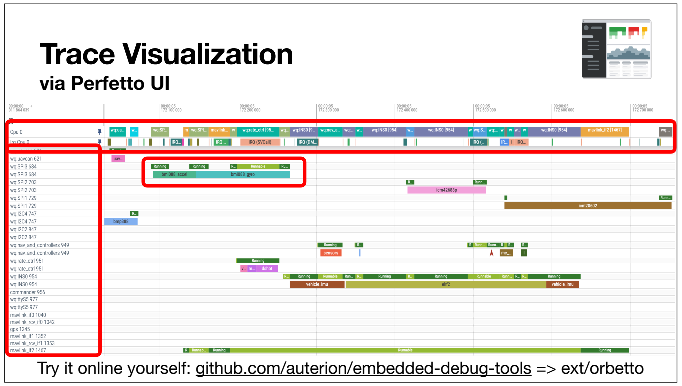
Tracing Toolchain

Transforming ETM + ITM via Orbuculum to Perfetto

Port	Usage
1	Thread Start
2	Thread Resume
3	Thread Runnable
DWT	Interrupts
ITM	Timestamps

Source	FTrace Packet
1	task_rename
2	sched_switch
3	sched_waking
DWT	irq_handler
ITM	set_timestamp
ETM	callstack

Ok, now what? We take the ITM, DWT, and ETM data streams and the data in the ELF file and convert them to a perfetto trace file. We convert the instruction trace to only function call stacks, which reduces the complexity significantly. Also reduces the file size significantly. Perfetto is based on FTrace, so you need to put the relevant data from your scheduler into the ITM to get threading support. ETM only gives you instructions, NOT data, so you will know that the scheduler has switched threads, but not WHICH thread! So ETM only makes sense in combination with ITM.



Trace Visualization

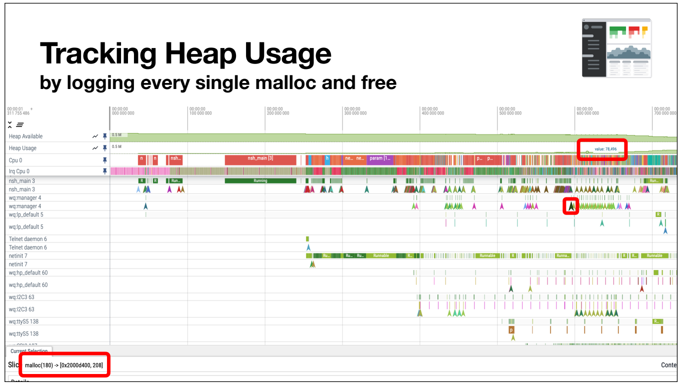
via Perfetto UI

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.

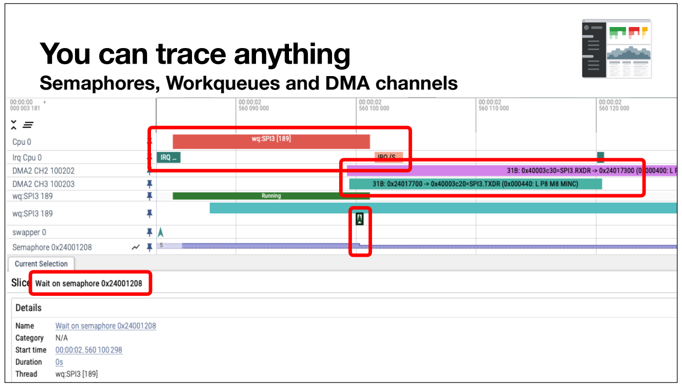


Tracking Heap Usage

by logging every single malloc and free

Slice malloc(180) -> [0x2000d400, 208]

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.



You can trace anything

Semaphores, Workqueues and DMA channels

31B: 0x40003c30=SPI3.RXDR -> 0x24017300 (0x000400: L P

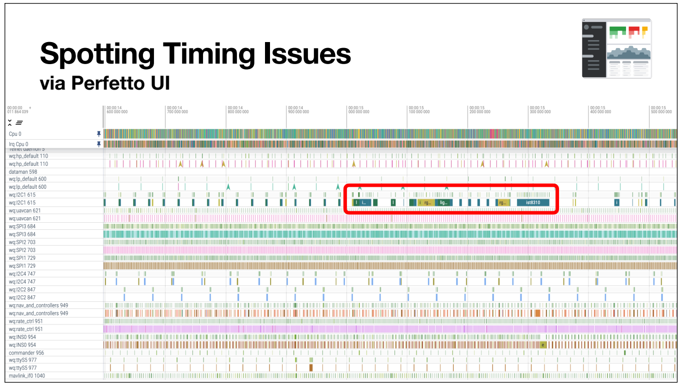
31B: 0x24017700 -> 0x40003c20=SPI3.TXDR (0x000440: L P8 M8 MINC)

Slice Wait on semaphore 0x24001208

Details

Name Wait on semaphore 0x24001208
Category N/A
Start time 00:00:02.560 100 298
Duration 0s
Thread wq:SPI3 [189]

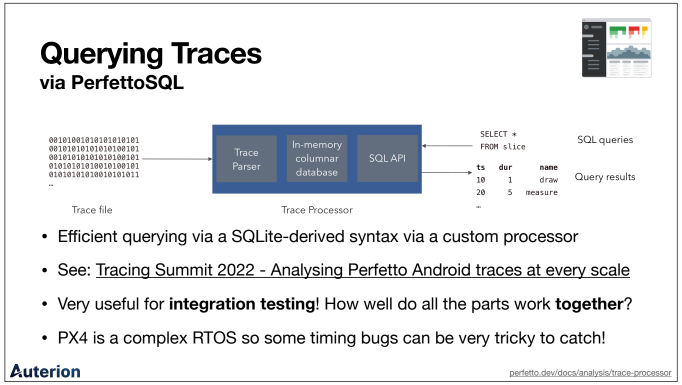
You can really track *anything* over time. Here we're tracing DMA channels and semaphores. You can see the SPI3 workqueue at the top. It starts two DMA transfers and then waits on a semaphore.



Spotting Timing Issues

via Perfetto UI

If we zoom out we can also see patterns. Here there was a brown out because we put too many sensors on the same power rail. And the sensors reinitialized. Would be nice if we could detect such issues automatically.



Querying Traces

via PerfettoSQL

```
0010100101010101010101
0010101010101010100101
0010101010101010100101
0101010101001010100101
0101010101001010101011
…
```

Trace file

Trace Parser | In-memory columnar database | SQL API

Trace Processor

```
SELECT *
FROM slice
```

SQL queries

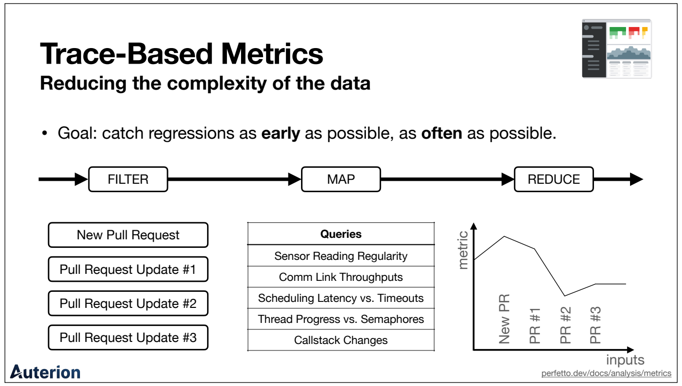
ts	dur	name
10	1	draw
20	5	measure
…		

Query results

- Efficient querying via a SQLite-derived syntax via a custom processor
- See: Tracing Summit 2022 - Analysing Perfetto Android traces at every scale
- Very useful for **integration testing**! How well do all the parts work **together**?
- PX4 is a complex RTOS so some timing bugs can be very tricky to catch!

perfetto.dev/docs/analysis/trace-processor

So perfetto also has a SQL interface to query your traces. This is very fast. There is a fantastic talk about this in more detail which I delegate to. The documentation is also fairly complete. We can write SQL queries to detect issues with our RTOS, which only show up when putting all the parts together. So this targets integration testing rather than unit testing.



Trace-Based Metrics

Reducing the complexity of the data

- Goal: catch regressions as **early** as possible, as **often** as possible.

FILTER → MAP → REDUCE

New Pull Request
Pull Request Update #1
Pull Request Update #2
Pull Request Update #3

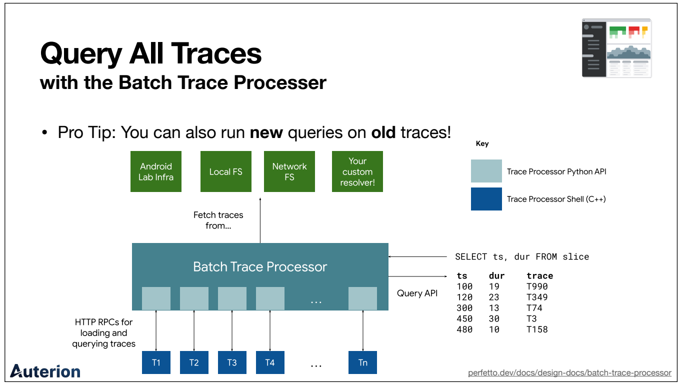
Queries
Sensor Reading Regularity
Comm Link Throughputs
Scheduling Latency vs. Timeouts
Thread Progress vs. Semaphores
Callstack Changes

perfetto.dev/docs/analysis/metrics

We want to have a typical filter-map-reduce pipeline that tests every PR or branch. Onto every trace we maps a set of queries:

- regular sensor readings is very important for a stable control loop.
- We need high communication link throughputs via DMA.
- Sometimes we have to wait on other threads, would be nice to know what the wait time distribution is like.
- Catching potential deadlocks, where semaphores are to blame.
- Some functions needs to be called in pairs (enable, disable). Is that balanced?

And then the reduce step renders these metrics into a graph.



Query All Traces

with the Batch Trace Processer

- Pro Tip: You can also run **new** queries on **old** traces!

Android Lab Infra

Local FS

Network FS

Your custom resolver!

Fetch traces from...

Batch Trace Processor

HTTP RPCs for loading and querying traces

T1 T2 T3 T4 … Tn

Query API

```
SELECT ts, dur FROM slice
```

```
ts    dur    trace
100   19     T990
120   23     T349
300   13     T74
450   30     T3
480   10     T158
```

Key

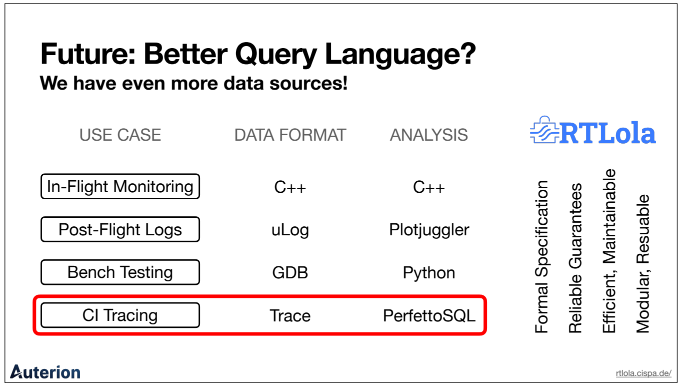
▨ Trace Processor Python API

■ Trace Processor Shell (C++)

Auterion

perfetto.dev/docs/design-docs/batch-trace-processor

Perfetto gives you a batch processor for the map-reduce pipeline. Why not do this processing on device? Because you need to know before hand what metrics you want. ETM catches *all* instructions, so you can go back in time and query old traces and get new insights. BUT: only works with instruction traces, not on ITM data, because you need to add the ITM tracing.



Future: Better Query Language?

We have even more data sources!

USE CASE	DATA FORMAT	ANALYSIS
In-Flight Monitoring	C++	C++
Post-Flight Logs	uLog	Plotjuggler
Bench Testing	GDB	Python
CI Tracing	Trace	PerfettoSQL

RTLola

Formal Specification

Reliable Guarantees

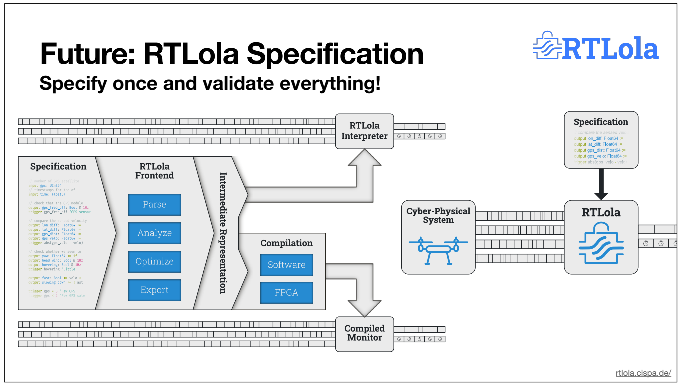
Efficient, Maintainable

Modular, Resuable

Auterion

rtlola.cispa.de/

Ok, so we have a lot more data sources than just traces. We also log on device and then evaluate this afterwards. Bandwidth is limited, thus we only log the important things. All of them have different tooling, the PerfettoSQL only works for the tracing part but what about the rest? Looking into research, the RTLola specification language may be a solution. The hope is to specify the queries once and then compile them to all our analysis tools. One specification to rule them all…

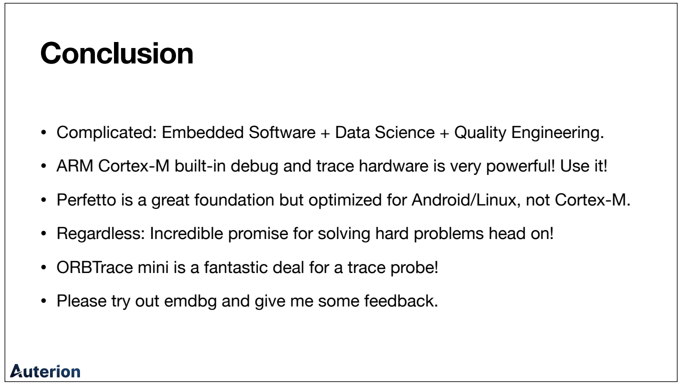


Future: RTLola Specification

Specify once and validate everything!

rtlola.cispa.de/

…and in darkness bind them! Or something like that. The idea is to interpret the off-device data streams against the specification, or alternatively compile the specification into a on-device monitor. Perhaps in future we can use RTLola to generate checks for our formats.

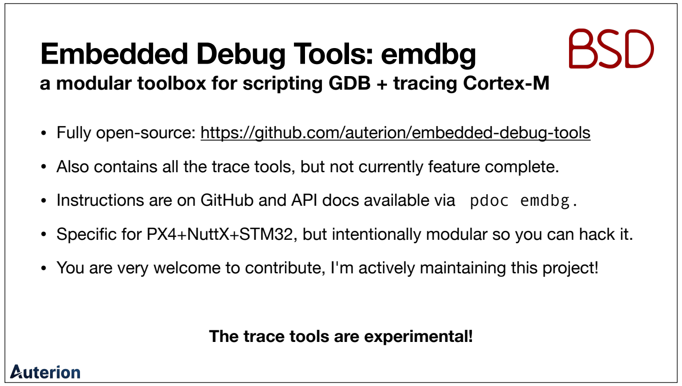


Conclusion

- Complicated: Embedded Software + Data Science + Quality Engineering.
- ARM Cortex-M built-in debug and trace hardware is very powerful! Use it!
- Perfetto is a great foundation but optimized for Android/Linux, not Cortex-M.
- Regardless: Incredible promise for solving hard problems head on!
- ORBTrace mini is a fantastic deal for a trace probe!
- Please try out emdbg and give me some feedback.

Auterion

This is fairly complicated. Perfetto is great and will become even better. ORBTrace mini is really good value and works fine even for STM32H7.



Embedded Debug Tools: emdbg

a modular toolbox for scripting GDB + tracing Cortex-M

BSD

- Fully open-source: https://github.com/auterion/embedded-debug-tools
- Also contains all the trace tools, but not currently feature complete.
- Instructions are on GitHub and API docs available 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!

The trace tools are experimental!

Auterion

Trace tools can be found on GitHub including examples. Not complete yet, we're still fighting a lot of bugs. There are also a much more mature GDB Python plugins, which I've given three separate talks on. We are actively developing the trace tools right now, hopefully much more complete by end of year.

Analyzing Cortex-M Firmware
with the Perfetto Trace Processor

Niklas Hauser: salkinium.com

Auterion: auterion.com

Orbcode: orbcode.org

RTLola: rtlola.cispa.de/

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

Thanks for your attention!

Thank you and do you have questions?