Auterion

Utilizing Instruction Tracing to Analyze PX4 at Runtime

with the Perfetto Trace Processor

Lukas von Briel
Embedded Software Intern

Who?

Hello, my name is Lukas

- My name is Lukas and I am currently studying electrical engineering at ETH in my masters.

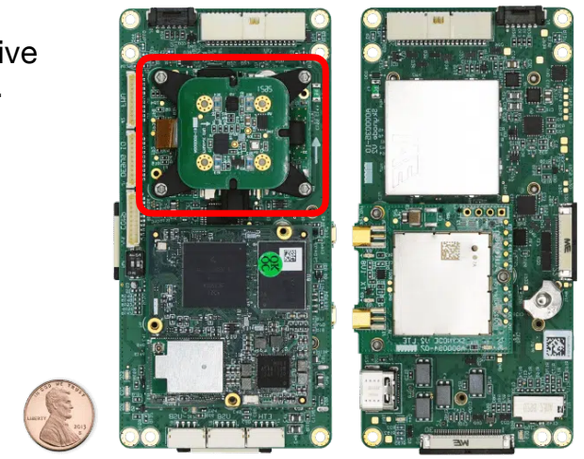
Let me tell you a bit about my background.

- During my bachelors I was part of a a student project called swissloop
    - Swissloop is a so called ‘Fokusproject’ where students develop a Hyperloop prototype over the course of a year
    - In this project I was responsible for developing an inverter module, which included power electric pcb design as well as an embedded controller design
    - This is the first time where I got hands on experience as an embedded engineer.
- For one year now I have worked 40% (2 days) as an embedded software intern under the guidance of Niklas to improve our debugging infrastructure

Why?

PX4 Autopilot runs on NuttX

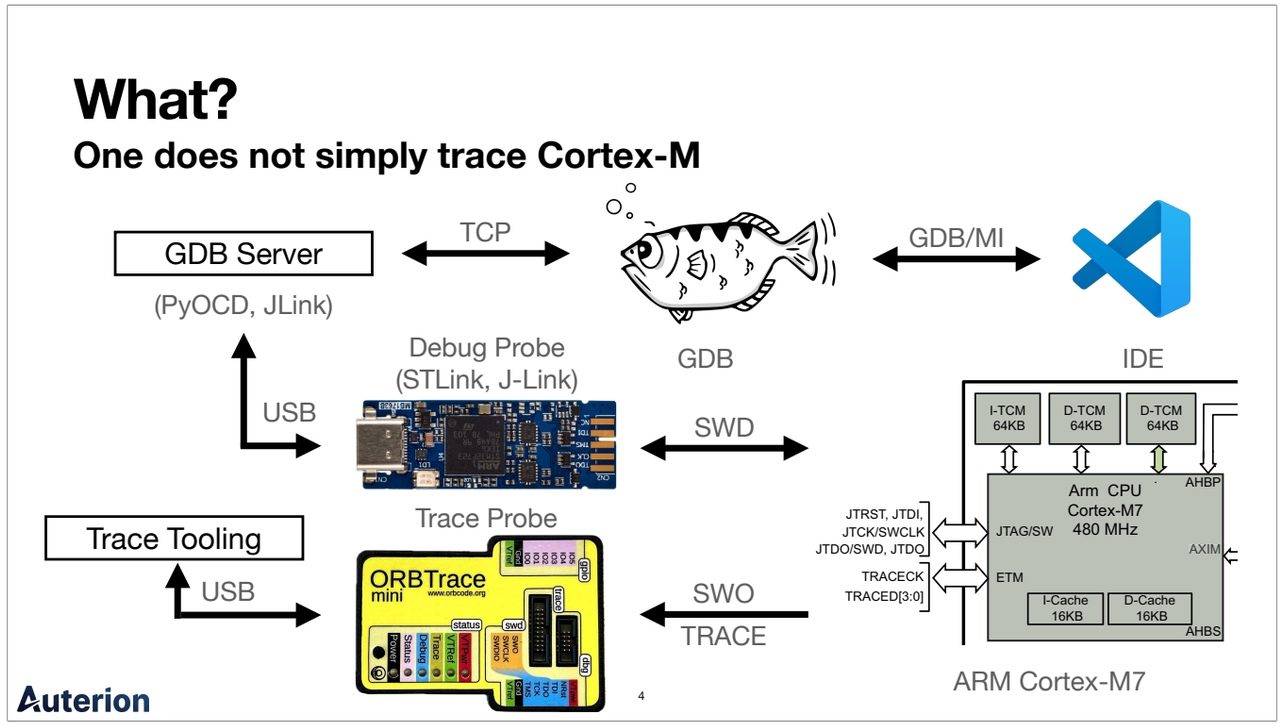
- Full RTOS with peripheral drivers, extensive filesystem and communication protocols.
- Many external sensors and components.
- Often subtle bugs that only manifest heuristically under the right conditions.
- Complex code base: 2MB binary
- Very fast STM32H7 (480MHz) can easily overwhelm debug logging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



Dimensions: 47,5 × 97,5 × 23,5 mm

First, let me explain why it is important that we need advanced debugging tools and cannot just print information in the console if needed or use basic gdb commands with a debug adapter like ST-Link:

- We are 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 really complex and has some subtle bugs now and then
    - The RTOS makes it very difficult to understand the context in which a bug is happening as many threads run simultaneously
    - Another point which makes it very difficult to finding bugs is the large size of the PX4 code base. Combined with the fast processors, which can operate at up to 500 MHz and many different peripherals, this makes classical printf style debugging very hard and time consuming
- That is why Niklas started developing the embedded debug tools to make debugging easier and to provide more advanced techniques for PX4 developers. This includes many different steps from structurally displaying raw debug information to things like regression testing.
- What we did to achieve that and what advances I made to enhance the capabilities of this tool is going to be the topic of this presentation.



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

For tracing, you connect to the output-only SWO and TRACE pins. However, after that no standardized infrastructure exists.

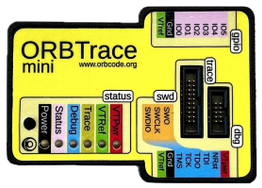
What?

One does not simply trace Cortex-M

SWO/
ITM

- ITM multiplexes 32 channels of 8/16/32-bit values, encoding custom messages
- DWT can sample the program counter and trace interrupts
- Hardware manages serialization, timestamps, and queues with priorities

Trace Probe



TRACE/
ETM

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

PE execution		Trace elements
0x2000	MOV	-
0x2004	LDR	-
0x2008	CMP	-
0x200C	BEQ -> 0x3000 (not taken)	atom_element(N)
0x2010	STR	-
-	IRQ	exception_element(IRQ, 0x2014)

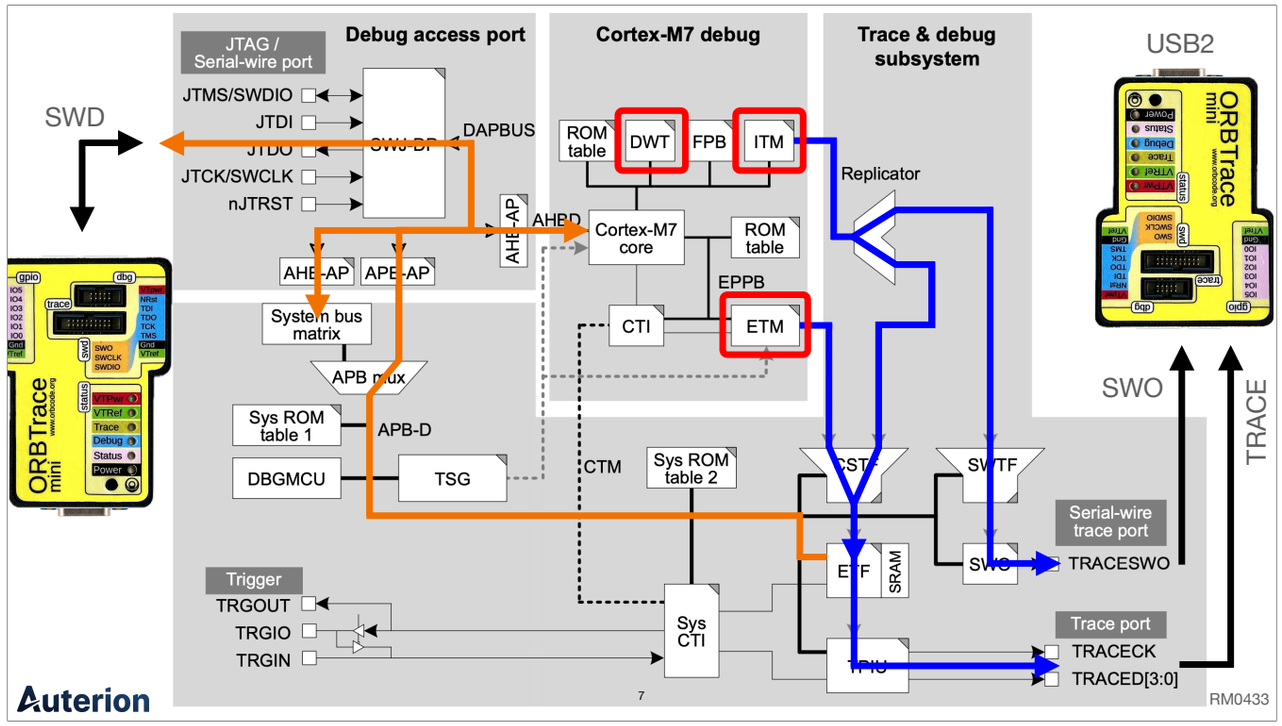
Auterion

The data that can be traced, categories into three parts, ITM, ETM and Data trace. Data tracing is not supported by STM so we only have access to ITM and ETM:
- ITM data is the tracing equivalent to printf. For this to work the software needs to me instrumented (what basically means adding the print in the codebase). These custom messages are then output out of the tracing pins. This does add some overhead to the execution depending on how many trace packets are generated.
- Additionally, the so called DWT module can be set to add program counter, timestamps and interrupt information to the trace stream.
- A disadvantage that I observed while adding new functionality to ITM and DWT tracing is the low compression on the output stream. This means we are not able to collect realtime information which only allows for a general idea of what has been executed and lacks a precise insight in what the processor is actually doing.
- Luckily, there is a saviour called ETM trace.
- This has been the biggest part of my work as intern, which is to enabling the use of this tracing data source and process the data to be useful for debugging in various ways.
- It is important to know that ETM is a highly compressed instruction trace, which mainly makes use of outputting branch information and precise cpu cycle count packets for timing information.
- This enabled me to reconstruct all executed instructions with a very precise timing.
- To get a feeling of what ‘only branch instructions mean’ lets have a quick look at this figure.

Real-Time Tracing
via ETMv4 + ITM on Cortex-M7

- Instruction tracing: ~0.4 bits per cycle.
  STM32H7 running at 480MHz = ~**200Mb/s**.
- Trace data can be very irregular by nature, the peak data rate can easily exceed the port capability, resulting in an overflow
- 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**.
- You **must** decode ETM + ITM streams on the host!
  Protocol documentation is available online for free.

Before we dive deeper into how the ETM data has been processed lets first look at some more high-level hardware facts.
- 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

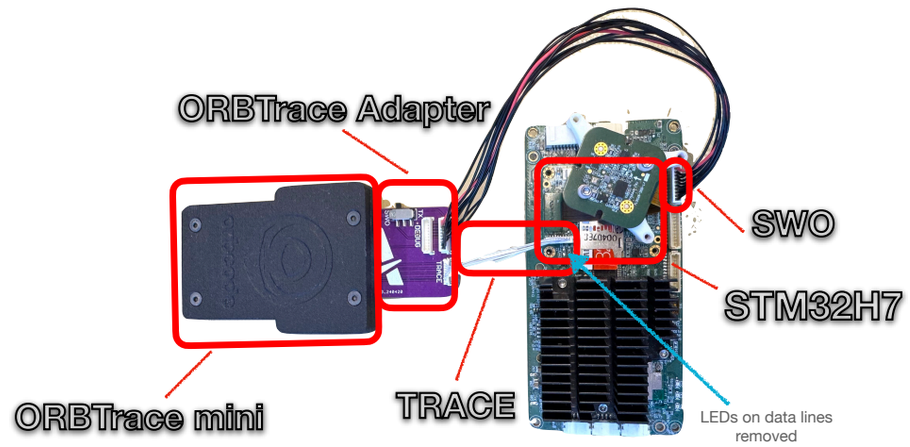


So where does the data go? This is the debug and trace subsystem of the STM32H7.

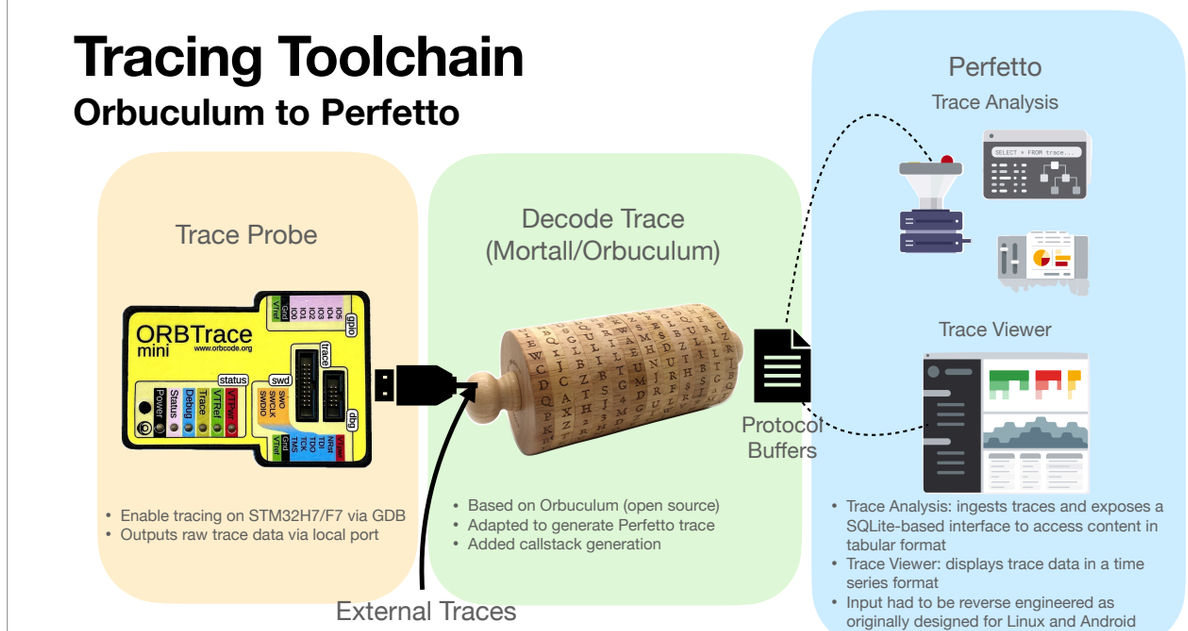
- You can see the CPU in the middle connected to the DWT, ITM, and ETM peripherals. You can connect your debugger on the left via SWD and then access the internals.
- You can redirect the ITM output to an internal 4kB FIFO (only true for stm32h7 we use on the Skynode V6x)and then read this out via the SWD debugger.
- 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, which in our case is the open-source ORBTrace mini.

State of the Tooling:

Tracing the STM32F7 with the ORBTrace mini



This is how the complete setup does look like.



Ok, now what? Lets have a closer look what we can do with ITM, DWT and ETM data streams:

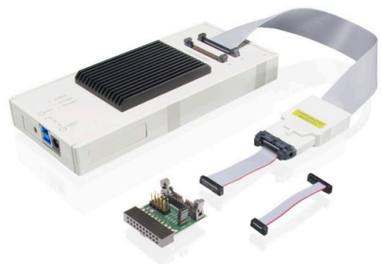
- To process ITM and DWT Data we used the open source library Orbuculum which already implements big parts of what we need.
- Additionally, to internal trace data only, we could also add External Traces from peripherals like SPI. A difficulty when adding external traces with a different time source is synchronization. I already experimented with that a bit and solved this issue for external SPI communication. However, the computation in Python was too slow and I dropped this part for now.
- Based on this we wrote the processing tool Orbetto which outputs a protobuf file.
- Protocol Buffers, which is similar to JSON, is a language and platform-neutral format for serializing structured data, that can be read by a tool called Perfetto.
- Perfetto is System-wide profiling for Linux and Android and is developed by google to do advanced trace analysis. Perfetto includes many different tools from which two where especially important to us:
    - Trace Viewer: Record, view and process trace data in the Perfetto UI
    - Trace Analysis Tool: it ingests traces and exposes a SQLite-based interface to access the contents of the trace in python
- To make use of this tool which is designed for Linux and Android we reverse engineered the input format to make it display PX4 trace data

To motivate why these tools are so really helpful let me quickly explain how they can be used.

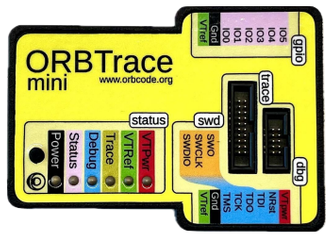
- The UI is kind of obvious. By visual inspection of the Callstack and other trace data you can basically look inside the cpu and verify the execution. This has already been proven really helpful to analyse and fix bugs in PX4. However, there are some down sights with this approach.
    - You need precise domain knowledge to know what to look for
    - Even given that it is still really hard to find the cause within this large amount of trace data. “Needle in a Haystack”
- To account for that we make us of the Trace Analysis tool as well. By utilising the SQL interface we computed general high-level metrics to compare different version of PX4. This statistical approach would enable us to find irregularities within PX4 before even deploying the change and with very little knowledge of the actual changes.

Full Stack Alternatives

µTrace All-In-One Debugging Solution - Lauterbach



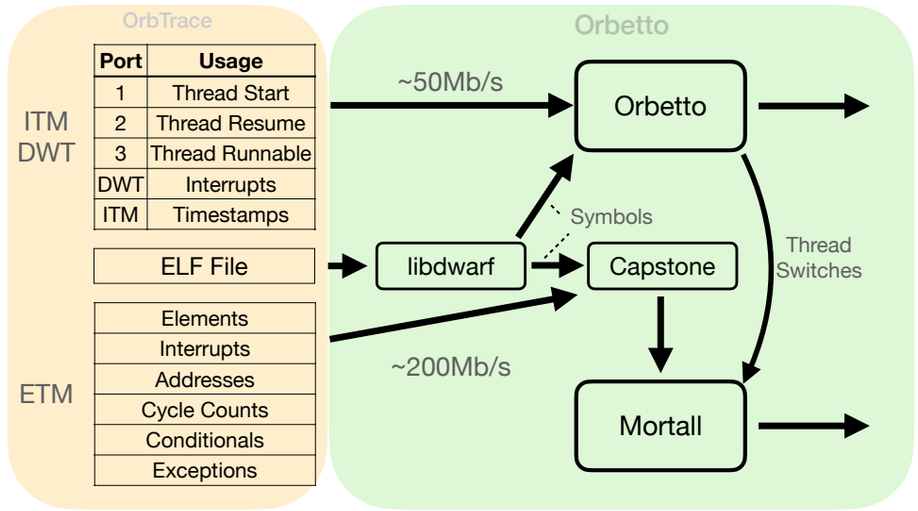
OrbTrace Mini



You might think what a genius idea of putting these different open source tools together. And you are right, it actually is genius and even so genius that other companies had the same idea and are selling this complete pipeline in a All-In-One Product.
So why don’t we just buy such a product.
The main reason in price. Comparing for example the mu trace from Lauterbach which is an all in one tracing solution to the OrbTrace mini it becomes obvious.
While Lauterbach is really expensive, it can cost the amount of a Kleinwagen, the orb trace mini only costs only around 200 bucks.
Furthermore, not only pricing is a disadvantage, it is also way less flexibel to go with such an all in one solution. As it is not open source it is barely customizable. It would be for example really hard to ingrate it into our own CI for customers or contributors to use it.
To sum it up, this means the goal is to build an open source customized version for PX4, which is then optimized for our needs.

Tracing Toolchain

Orbuculum to Perfetto



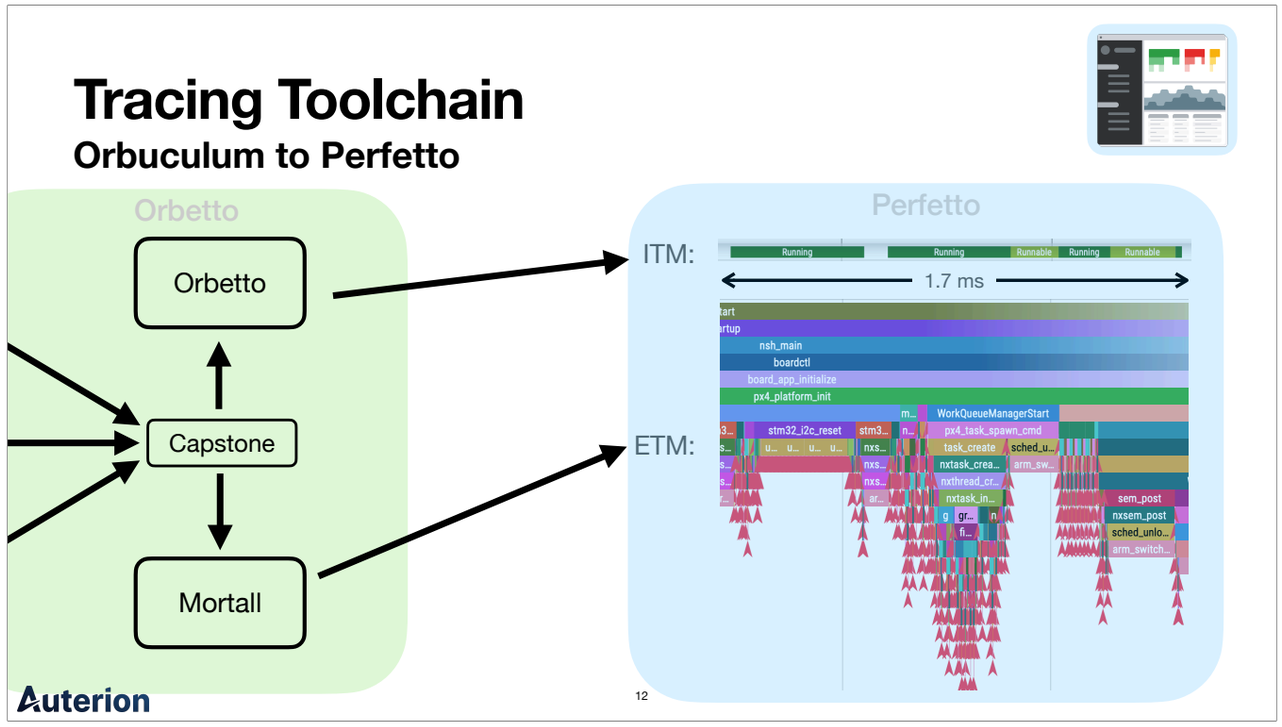
I hope the rough plan is clear now and we can have a closer look on how we implemented the pipeline.
As you have seen the first step is using a Trace Tool in our case OrbTrace mini to collect some trace data.

- To enable this, the code needs to be instrumented, which means actual debugging messages have to be added to NuttX source code, to track Thread starts, resume. This of course generates some overhead depending on the amount of debugging messages added.
- Additional the DWT (data watchpoint and trace unit) can output interrupts and programs counter values.
- The instruction trace stream generated by the ETM (Embedded Trace Macrocell) can be enabled without any overhead.
- It allows for reconstruction of every instruction executed in the cpu. Basically you can get exact image of what is happening outside of the cpu. Like I already said in the last slide this can be an overwhelming amount of information but if structured well also really helpful.

Like I explained this whole process in based on the open source library Orbuculum which already implements big parts of this pipeline.

- To reduce the throughput, ETM and ITM messaged are encoded and then multiplexed by the TPIU to be output on the Trace ports.
- The demultiplexing and decoding is done by the embedded debug tools Niklas started to develop two years ago.
- To decode PC values the elf file needs to converted to a mapping between pc values and function names. Therefore, we used the library libdwarf, which has already been implemented Orbuculum. To make it easier we call this mapping symbols.
- For ITM data these symbols are combined directly with the trace data to generate a rough overview of programs execution on the cpu.
- To get the ARM instructions from the ETM data we use Capstone which decodes ARM instructions.
- Having these instructions and information about the context switches from ITM, Mortrall follows the execution context and constructs a Callstack.
- All tracing information is the output as a protobuf file to be used in Perfetto.

My contribution was adding functionality to ITM decoding and build all instruction trace functionality which summarised under the tool called Mortrall.

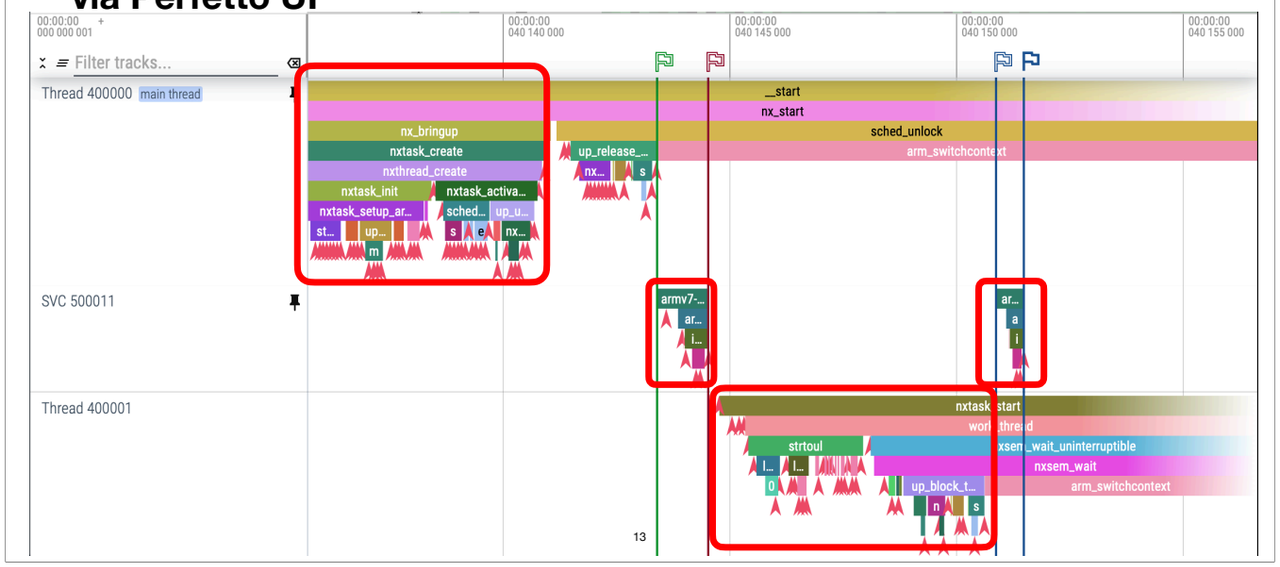


The tracing information, which is outputted by Orbetto into a protobuf file can than be displayed in the Perfetto UI.

- on the top you can see the thread state capture by ITM. There are three states of the thread. Either it is running, Runnable or sleeping.
- on the bottom we can see the call stack of the same thread over that same interval, which has been extract from ETM trace data.
- This plot also gives a feeling of the information density ETM vs ITM provides and why ETM can give way deeper insight into whats happening on the FMU.

Call Stack Visualisation

via Perfetto UI

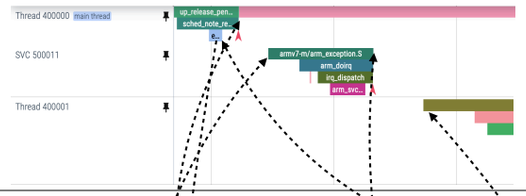


On the top you can see the a small slice of the Perfetto ui output. This is captured right at the startup of PX4 on the FMUv6x.
You can see that each thread and also each exception has its own call stack which stays idle when other threads are running.
Lets have a closer look on what happening here

- We can see that the main thread or thread 0 initializes a new NuttX thread.
- We also see in light blue the overhead ITM messages cause.
- Here we have an ITM packet that logs thread activations
- Then it starts the thread by first calling to SVC Exception and then continuing in a different context on thread 1
- We can see thread is called the first time an is set up by NuttX
- After the setup is finished again an SVC interrupt happens and context switches again

Call Stack Visualisation

via Perfetto UI



PE Execution	Trace Element	Function
bd10 pop {r4, pc}	f7 -> Atom Format 1 [b]	emdbg_itm32_block
f8d5 1088 ldr.w r1, [r5, #0x88]	-	sched_note_resume
…	-	sched_note_resume
-	10 -> Cycle Count (16)	sched_note_resume
-	06|97|00 -> Exception Info	sched_note_resume
-	96|d4|02 -> Short Addr	-
f3ef 8005 mrs r0, ipsr	-	arm_exception.S
…	…	-
2300 movs r3, #0	-	arm_exception.S
bd30 pop {r4, r5, pc}	f7 -> Atom Format 1 [b]	arm_exception.S
-	96|92|4c -> Short Addr	nxtask_start
b538 push {r3, r4, r5, lr}	-	nxtask_start
…	… 14	nxtask_start

Context:

Thread 0

SVC Interrupt

Thread 1

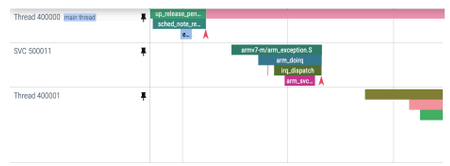
Auterion

To put you in my shoes and give you a feeling of how the ETM decoding works lets quickly step through a simplified trace and analyse how a call stack is generated:

….

Tracing Issues

- Function names could not be decoded
- Capstone wrongly classifies JUMP instructions
- Signal Integrity
  - Multipurpose pin usage
  - LED on high speed trace
  - Parallel traces not equally long



PE Execution	Trace Element	Function
bd10 pop {r4, pc}	f7 -> Atom Format 1 [b]	emdbg_itm32_block
f8d5 1088 ldr.w r1, [r5,	-	sched_note_resume
…	-	sched_note_resume
-	10 -> Cycle Count (16)	sched_note_resume
-	06|97|00 -> Exception Info	sched_note_resume
-	96|d4|02 -> Short Addr	-
f3ef 8005 mrs r0, ipsr	-	arm_exception.S
…	…	-
2300 movs r3, #0	-	arm_exception.S
bd30 pop {r4, r5, pc}	f7 -> Atom Format 1 [b]	arm_exception.S
-	96|92|4c -> Short Addr	nxtask_start
b538 push {r3, r4, r5, lr}	-	nxtask_start
…	…	nxtask_start

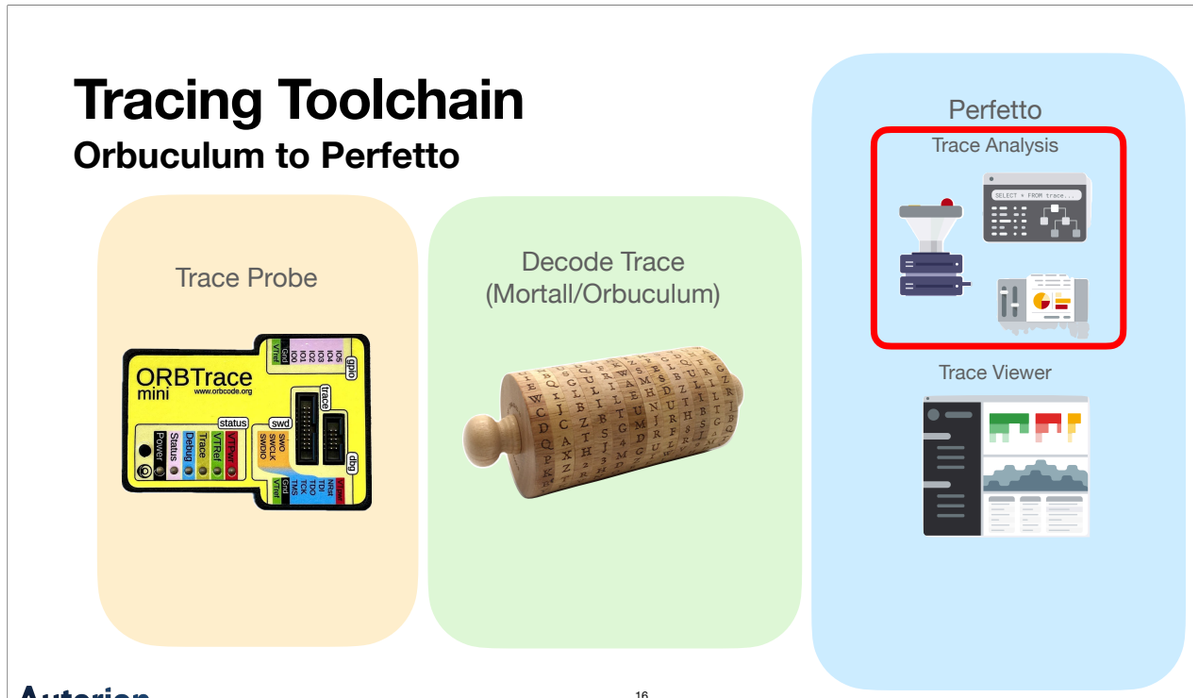
Contex

Thread

SVC

Thread

When understanding the concept it becomes obvious that this is a highly fragile framework as all future Trace Elements depend on the correctness of the current state. For Example if the exception Info packet would be missing, the context switch to the exception could not be detected and the next Atom element would be related to completely different jump instruction as the pc offset would be wrong.

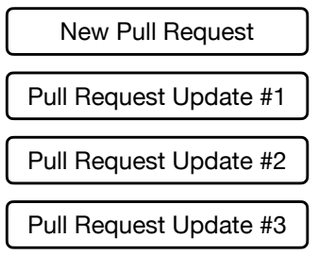


As we have just seen, the perfetto UI can be very helpful to put a structure into the large amount of data but it can still be quite overwhelming and time consuming to actually search for bugs in the UI.
Therefore instead of displaying we can also use the Trace Analysis tool of perfecto to get access to an SQLite database to compute metrics and reduce the complexity of our data.

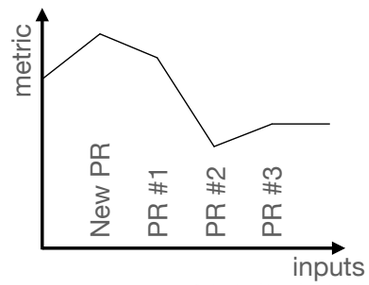
Trace-Based Metrics

Reducing the complexity of the data

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



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



Auterion

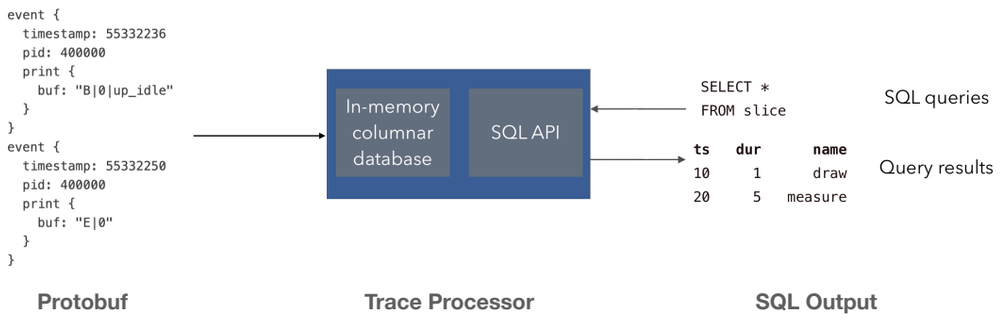
perfetto.dev/docs/analysis/metrics

What we ideally want is to have a typical filter-map-reduce pipeline that tests every PR or branch. Onto every trace we maps a set of queries. The first big question that comes up is what metrics would be most helpful to determine that functionality of PX4:

- regular sensor readings is very important for a stable control loop.
- We need high communication link throughputs via DMA.
- Another metric which could be very helpful especially when looking at the difference between two PRs is code coverage. When comparing code coverage the impact of changes can be easily measured. It can basically answer the question „did my PR only affect the system in the way I wanted it too be or might other areas be affected that I have not thought of.

Lastly, the reduce step renders all these metrics into a statistical representation of the current state of the PX4 to be compared between different PRs. This is called regression testing and would enable us to detect bugs in PX4 changes without even flying a drone.

Querying Traces
via PerfettoSQL



- Efficient querying via a SQLite-derived syntax via a custom processor
- Very useful for **integration testing through metrics!**

-Protobuf as input format
- Trace Processor
To calculate metrics, perfetto has a SQL interface to query your traces.
PerfettoSQL is also speed optimized which its very important when handling such large databases.
All the information we collected about our system has been combined by orbetto within this protobuf file and is then used as an input to the Trace processor.
Now we can used the SQLite based interface to perform custom queries on our system.
The received data can then be used for integration testing though metrics.

Perfetto SQL

Example

Write standard SQLite queries

Data is distributed over many different Tables

Optimized access to selected group of variables

Exported as Pandas dataframe for easy plotting

```
43  def get_function_distribution():
44      query = """
45      DROP VIEW IF EXISTS slice_with_thread_names;
46      CREATE VIEW slice_with_thread_names AS
47      SELECT
48          ts,
49          dur,
50          tid,
51          slice.name as slice_name,
52          slice.id as slice_id,utid,
53          thread.name as thread_name
54      FROM slice
55      JOIN thread_track ON thread_track.id = slice.track_id
56      JOIN thread USING (utid);
57
58      SELECT *
59      FROM (
60          SELECT
61              SUM(dur) AS cpu_time,
62              GROUP_CONCAT(DISTINCT ts) AS tss,
63              tid,
64              slice_name,
65              thread_name,
66              COUNT(*) AS count
67          FROM slice_with_thread_names
68          GROUP BY slice_name, tid
69      ) AS subquery
70      WHERE tid > 100000
71      ORDER BY cpu_time DESC;
72      """
73      return tp.query(query).as_pandas_dataframe()
```

Auterion

Here is an example of a query I wrote, which for those of you who use sql regularly probably looks familiar.
…

Trace-Based Metrics

ITM - Thread Overview

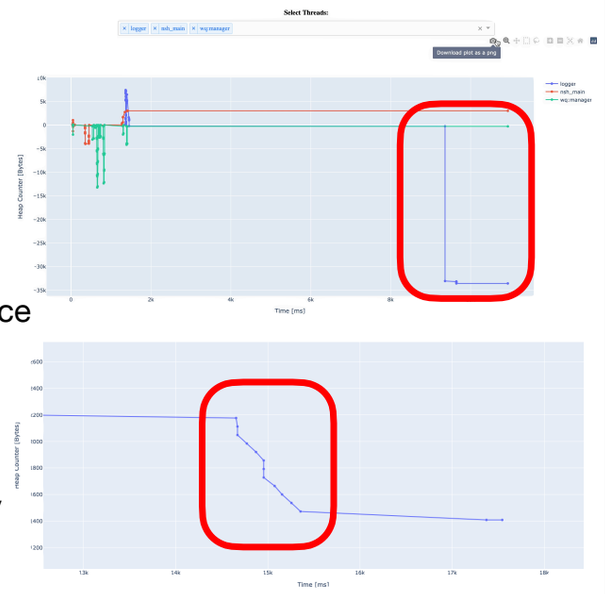
- Thread Status metrics over all threads
- Outlier detection
- Inconsistencies can be detected
- By getting an overview optimisation can be made

Having derived the query we could display it in the CI like this.
Here you can see an overview of the thread CPU time distributions.
I used the html tool dash to make a small webinterface where you can customize plots.
Choose work queue
In the Pie chart you can see for each function how long it ran, how long it slept and how long it waited for execution (runnable).
On the bottom you can you can see the histogram for each call of the function how long it was running.
For example when you look at board_adc you can see that most calls accumulate around 0.1 ms of runtime, however one called lasted longer.
So you could already implement some kind of outlier detection or use this information and compare it between different PRs.

Trace-Based Metrics

ITM - Diff Heap Profile

- Use case of Trace based metrics
- Heap allocation trace by ITM packets
- Based on SQL queries of Perfetto Trace processor
- Dash with plotly to display heap allocation between two PRs
- Effects of changes can be seen easily



Here you can see another metric that came in handy when looking at the heap usage of the Logger two months ago. This is an example where I plotted the difference of heap usage between two different traces. One with the Logger enabled and with the Logger disabled.
…
In the second graph we can see a similar behavior just on the uavcan thread now. It displays the effect of uavcan shrink which is supposed to reduce the heap usage of uavcan. We can easily that it save us about 1kByte of heap.
So already slightly processing the trace data and putting it in a nice CI tool can make debugging and gaining insight into the system way easier.

Trace-Based Metrics

ETM - Code Coverage

- Full instruction trace available, which allows for a Code Coverage Metric
- Helpful metric in regression testing
- Difference in code coverage shows what impact changes have

For now we only looked at metrics based on ITM data but why have we put in all that work to get the ETM too work as well. By tracing all instruction we can compute metric called code coverage, which basically lets us derive which part of the code has been reach and how often.
I think for every PX4 developer it is quite obvious why this can be really helpful as it gives deep insights into the systems. It is also very useful for regression testing and you can easily check the difference between two PRs.

Outlook

- Integrate into CI (improves usability)
- Regression testing
- **Goal**: AOS tracing <- Logic Analyser -> PX4 tracing
- End to end solution
- Automated insights

Coming to an end of my presentation I want to quickly mention what still needs to be done and actually will be done by our team.
As you have just seen many of these tools can already be used on their own to solve individual bugs on PX4 and will be added to the CI to be accessible for everyone.
However, to put this all together into a big regression debugging tool with a user friendly environment we want to do the following:
The idea would be to get a complete picture of our system on Skynode. This can be done by tracing on AOS like its already done with Perfetto, Tracing PX4 with ITM and ETM and tracing all external communication with some kind of logic analyzer.
This would enable us for example, to follow packets from AOS via MAVLINK to PX4 and debug thereby the whole pipeline.
To sum it up, we now have the proof of concept and a working implementation of data collection and processing part, which already helps with standard debugging, but the big goal would be to combine these into a automated end to end regression solution.

Conclusion

Situation:

- Embedded debug tools
- High-Level plan for advanced debugging

Task:

- Add functionality
- Improve visualization
- Work towards an end-to-end solution

Actions:

- Added ETM and conversion pipeline
- Investigated end-to-end strategies
- Step-by-step iterations increasing complexity
- Read all kinds of documentation

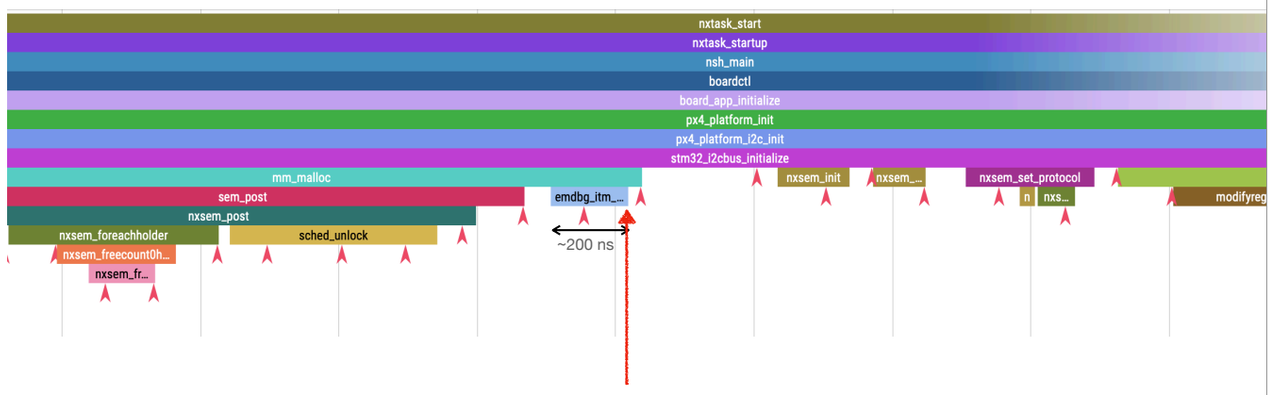
Result:

- Basic data generation and processing done successfully on STM32F7 and H7
- Still needs to be combined into a user-friendly environment

Last but not least I want to give you some conclusion on how my internship process was.
Basically the situation, when joining Auterion was, Niklas has already set up the embedded debug tools and enabled some itm traces and processed them with Orbetto to display them in the Perfetto UI. He had also developed a high-level plan of what the next steps should be to reach the just mentioned goal. However, because this has never been done in any open source project, it was lacking detail.
My Task was to integrate ETM and ITM and find a way to derive meaningful metrics to build up a regression tool.
Because of the novelty in combining these different open source tools, my work had a big research and investigation part. It was mostly about trying out a lot of different stuff and doing small step by step iterations.
I encountered many obstacles, for example bugs in these untested libraries and hardware issues with Skynode. Especially, getting the ETM tracing to work was really hard, as good documentation is hard to find and I had to reverse engineer many parts.
For Example, I implemented during first months all the debugging functionality on a simpler dummy task, than on PX4, to get a deeper insight into the topic without having the full complexity at once.
Looking back I am very happy with what I achieved, as there were also times where we were stuck and weren’t sure how to proceed.
All in all, we now have basic data generation and processing infrastructures for advanced debugging of PX4 and a way better knowledge of what still needs to be done to integrate and make it useful on all Auterion FMUs.

Extra: Call Stack Visualisation

via Perfetto UI



Auterion

Another thing I want to quickly mention, which I have not explained yet is the timing information:

- I added cycle count packets to the decoding process to get precise timing information for displaying the call stack in perfetto.
- In the plot you can see another example slice trace image from Perfetto. The red arrows mark cycle count packets.
- As you can see in the plot, we even estimate between cycle count packets by placing function switches relative to the instructions executed between the last cycle count and the next cycle count.
- Imagine if there would be 10 instructions than here there would be x before and y after.
- This allows to get a really precise display