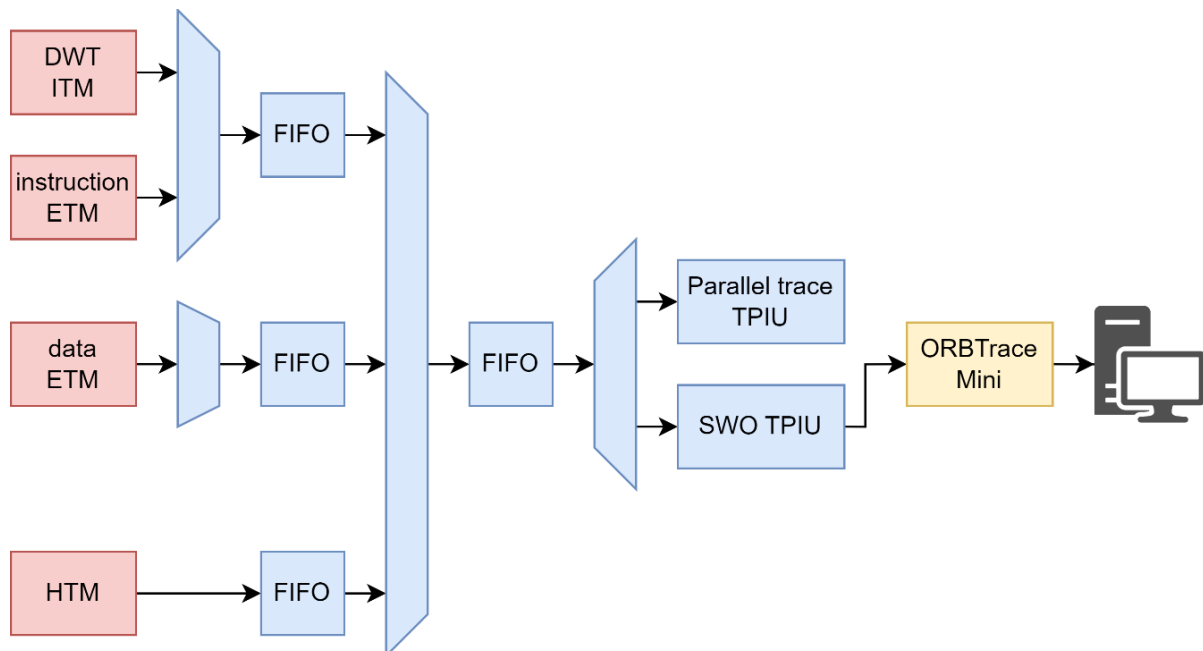


Enabling open-source trace tooling on NXP S32K344

Graduation report



Name student: Ian Baak
Student ID: 1735629

University: HU University of Applied Sciences
Degree programme: Electrical and Electronic Engineering
University supervisor: Franc van der Bent

Date of publication: 2023-05-22
Place of publication: Utrecht



Enabling open-source trace tooling on NXP S32K344

Graduation report

Name student: Ian Baak
Student ID: 1735629
Class: EV8

University: HU University of Applied Sciences
Institute: Institute for Design and Engineering
Degree programme: Electrical and Electronic Engineering

Course: Graduation Project
Course code: TEET-AGBACHEX-14
First assessor: Hubert Schuit
Second assessor/
university supervisor: Franc van der Bent

Project name: Investigation into methods and tooling for optimal performance execution of application software on ARM Cortex-M7
Internship company: NXP Semiconductors Netherlands B.V.
Company division: CTO System Innovations group, Mobile Robotics
Company supervisor: Peter van der Perk

Place of publication: Utrecht
Date of publication: 2023-05-22
Version: 1.0

Het bestuur van de Stichting Hogeschool Utrecht te Utrecht aanvaardt geen enkele aansprakelijkheid voor schade voortvloeiende uit het gebruik van enig gegeven, hulpmiddel, werkwijze of procedure in dit verslag beschreven. Vermenigvuldiging zonder toestemming van de auteur(s), de school of het bedrijf (indien van toepassing) is niet toegestaan.

Preface

After 4,5 years of working on my major in Electrical Engineering at HU University of Applied Sciences (Hogeschool Utrecht), I've arrived at the final course: the graduation internship. During this internship, the student is tasked with proving that they master the *competences* required of a Bachelor in Electrical Engineering: *Analyseren*, *Ontwerpen*, *Realiseren*, and *Professionaliseren* at level III, and *Managen* at level II.

I'm doing my graduation internship for NXP Mobile Robotics, which is a team inside the CTO division of NXP Semiconductors. NXP Mobile Robotics focusses on applying and promoting NXP's portfolio of semiconductor devices (microcontrollers, I/O transceivers, power management ICs, etc.) in autonomous vehicles (drones), such as UAVs and autonomous rovers. As part of this effort, NXP Mobile Robotics develops and supplies development boards which can perform various tasks in drones, such as acting as the vehicle management unit/board computer. The results of my graduation internship will aid in making the microcontrollers on these development boards more accessible to others by making their tracing features easier to use with tracing tools.

This graduation report is of interest to anyone who is not familiar with tracing features provided by ARM Cortex-M-based microcontrollers and wants to learn more about them, with a specific interest in using these on NXP S32K3 series microcontrollers with Cortex-M7 processors. This graduation report also serves as technical documentation for the tracing tools having been developed during the graduation project.

I would like to express my gratitude towards my project supervisors, Peter van der Perk and Joost van Doorn, for providing general guidance. This was especially important during the definition phases of my graduation project. I would also like to thank Dave Marples, one of the main developers of the Orbuculum project, for helping me understand and exploit the ARM tracing features and helping me select which trace features I should use for my purposes. Finally, I'd like to thank my university supervisor, Franc van der Bent, for providing feedback on my work when needed and for helping me when I struggled with university-specific aspects of the graduation project.

Utrecht, 2023-05-22, Ian baak

Abstract

Many ARM Cortex-M-based microcontrollers include tracing features which allow monitoring hardware events (e.g. exceptions occurring, a specific instruction being executed or a specific address range being accessed by the CPU). Tracing enables non-intrusive insight into software execution (i.e. software behavior and performance is preserved), as opposed to using intrusive debugging methods such as setting breakpoints and/or single-stepping. Unfortunately, tracing features are currently mostly exploited through high-end, expensive tooling from commercial vendors, making it hard to use tracing features in a cost-effective manner. A recent development is the Orbuculum suite, which is a collection of open-source tools that use ARM tracing features for various purposes.

NXP Mobile Robotics is currently transitioning towards using an NXP S32K3 series microcontroller with an ARM Cortex-M7 CPU for one of their development boards, and they are facing challenges with mapping an application to the M7 core and the various types of memory available in the system such that the application performs optimally. NXP Mobile Robotics is interested in making tracing features usable on S32K3 in a cost-effective manner, so that their users can use tracing to gain insight into how effectively an application exploits the hardware in the microcontroller. To this end, this graduation report focusses on enabling tracing features on an S32K344 MCU and enabling/developing tools that allow using these tracing features cost-effectively.

Based on the project goal and requirements defined in consultation with NXP Mobile Robotics, the tracing features available in S32K344 were investigated. The S32K344 features three types of trace sources that monitor various types of hardware events: the Data Watchpoint and Trace unit (DWT), the Instrumentation Trace Macrocell (ITM) and the Embedded Trace Macrocell (ETM). The S32K344 also contains two types of off-chip interfaces for receiving trace data: Single Wire Output (SWO; available on the debug headers of many ARM Cortex-M-based development kits) and parallel trace. SWO is capable of much lower maximum trace bandwidth than parallel trace due to using only a single pin instead of five, but this also makes SWO more cost-effective and less complex. Due to bandwidth requirements of the various types of trace sources, DWT and ITM tracing can be done over either SWO or parallel trace, while ETM tracing can practically only be done using the high-bandwidth parallel trace interface.

Based on the available trace sources, a list of tools to enable/develop was defined. In this graduation report, three tools which utilize DWT/ITM tracing and function over SWO were enabled/developed: (1) orbtap (from the Orbuculum suite), which allows reconstructing CPU load of an application; (2) an ITCM mapping tool, allowing automatic identification and mapping of high-load application functions to the S32K344's fastest memory; and (3) orbstat (from the Orbuculum suite), which allows generating call graphs of the running application to gain insight into frequency of and the relationship between function calls. A list of tracing tools that likely require ETM and/or DWT/ITM over the parallel trace interface was also defined, but at time of writing these have not yet been implemented due to prioritizing development of the tools that could function over the less complex SWO interface.

To demonstrate applicability of the three enabled/developed tracing tools, they were exercised with PX4 Autopilot (flight control software for autonomous vehicles) as an example application, running on an NXP MR-Buggy3 rover platform. Using the insights provided by the tracing tools, various optimizations were done to PX4 Autopilot.

Complementary to the enabled/developed tracing tools and in line with the intent to make them accessible to future users, a user guide was written that explains how to set up and use the tracing tools.

Contents

Preface	5
Abstract	6
Introduction	9
1 Assignment	10
1.1 Background	10
1.2 Problem statement and solution statement.....	10
1.3 Assignment description	11
1.4 Deliverables.....	11
1.5 Initial assignment description and motivation for changes	11
2 Requirements	13
2.1 Top-level requirements.....	13
2.2 User requirements.....	14
2.3 Non-user requirements	14
3 Research	17
3.1 Key points from analysis phase	17
3.2 Research design.....	19
3.3 Adding support for S32K344 to ORBTrace Mini.....	20
3.4 Setting up tracing over SWO and enabling orbtob	22
3.5 Automatically optimizing instruction mapping	24
3.6 Generating call graphs with ITM tracing.....	27
3.7 Letting a developer use tracing tools	29
4 Final designs	30
4.1 pyOCD configuration file for S32K344	30
4.2 Trace bus and SWO TPIU setup	31
4.3 orbtob DWT and ITM setup.....	31
4.4 ITCM mapping tool.....	32
4.5 orbstat instrumentation functions and setup process.....	33
4.6 User guide	35
5 Verification	36
5.1 Test setup.....	36
5.2 Optimizing PX4 Autopilot	37
5.3 Requirement compliance testing.....	42
6 Project management and execution.....	47
6.1 Management methodology.....	47
6.2 Phasing and milestones.....	47
6.3 Execution of the project schedule.....	49
6.4 Evaluation	50
7 Conclusion	53
7.1 Future work.....	53
Terms and abbreviations.....	55

References	56
Appendix A: testcases	58
Appendix B: pyOCD target configuration for S32K344	66
Appendix C: GDB files for setting up tracing on S32K344	68
<i>Trace component setup commands</i>	<i>68</i>
<i>GDB initialization script to set up tracing for orbtob</i>	<i>69</i>
<i>GDB initialization script to set up tracing for orbstat</i>	<i>70</i>
Appendix D: ITCM mapping tool source code	71
Appendix E: orbstat ITM instrumentation functions	74
Appendix F: S32K344 tracing tools user guide (external)	76

Introduction

In order to enable higher-performance flight control systems for unmanned, autonomous vehicles, NXP's Mobile Robotics team is transitioning from microcontrollers with ARM Cortex-M4 processors to using ARM Cortex-M7-based microcontrollers. During this transition, NXP Mobile Robotics has experienced an increase in difficulty with effectively mapping an application to the Cortex-M7's more complex microarchitecture and memory structure such that the application runs optimally. This poses a threat to NXP Mobile Robotics' goal of making their vehicle management unit (VMU) development boards accessible to other developers (both internal and external to NXP). To mitigate this issue, NXP Mobile Robotics is interested in using tracing features provided by most ARM Cortex-M-based microcontrollers. Tracing allows runtime monitoring of various hardware events without incurring software overhead or modifying the control flow of the CPU, thus providing deeper insight into execution of software. This could in turn aid in identifying and resolving suboptimal or undesired software behavior.

Unfortunately, the barrier to entry for using the tracing features in Cortex-M microcontrollers is relatively high due to host-side tooling for exploiting the trace features mostly being provided by high-end tool suites from commercial vendors. This leads to tracing tools usually being considered as expert-focused. The graduation project documented in this report aims to make using the tracing features more accessible to users of NXP Mobile Robotics' development boards by enabling/developing open-source, cost-effective tracing tools for the NXP S32K344 microcontroller utilizing an ARM Cortex-M7 processor.

This graduation report is structured as follows. Chapter 1 describes the graduation assignment in more detail by describing the background and defining the assignment goal. Chapter 2 documents the requirements package. Next, research into how trace tools can be developed/enabled on S32K344 is discussed in Chapter 3. This chapter also reiterates key points from research documented in the Project Initialization Document (PID)¹. In Chapter 4, the final designs of enabled/developed tracing tools based on the research results from the prior chapter are discussed. The tracing tools are subsequently verified in Chapter 5 by first exercising them with an example application (PX4 Autopilot) and then testing them for compliance to the requirements. Next, the way the graduation project was managed is discussed and evaluated in Chapter 6. Finally, Chapter 7 concludes the graduation report and discusses future work.

¹ The PID is a document created in the starting phases of the graduation project which describes the project goal, plan of approach, and initial project schedule.

1 Assignment

This chapter describes the internship assignment. The assignment has been redefined since finalization of the PID; the motivation for this is discussed at the end of the chapter.

1.1 Background

For software running on a microcontroller in an autonomous mobile vehicle, both ensuring the reliability of the application and achieving the necessary performance for meeting the real-time requirements of the vehicle are of high importance. When performance is not sufficient, but the specification of the vehicle's hardware platform is already frozen, increasing performance is usually only possible through optimization (i.e. effectively mapping the application to available execution resources). To aid in ensuring reliability and achieving sufficient performance, several features are provided by present-day ARM Cortex-M-based microcontrollers. These features range from basic debugging (single stepping, inspecting variables, etc.) to tracing of hardware structures (e.g. full reconstruction of instructions executed by the CPU) (ARM, 2015).

Host-side tooling for using tracing features are mostly exploited by high-end commercial tool suites from commercial vendors, making the use of tracing relatively rare in markets where open-source/cost-effective tooling is preferred (e.g. hobbyist-grade systems such as hobbyist drones/rovers). In an effort to make tracing features more accessible to users other than experts with access to commercial embedded tool suites, developers from the Orbcodes group have developed the open-source Orbcodes tool suite: which is a collection of host-side tools that allow parsing various types of trace data in useful ways (Orbcodes, 2022).

1.2 Problem statement and solution statement

NXP Mobile Robotics is currently transitioning from using NXP Kinetis K66 series microcontrollers with ARM Cortex-M4 cores for their vehicle management units (VMU) to NXP S32K3 series microcontrollers with ARM Cortex-M7 cores. The S32K3 series enable higher performance through the use of, among others, a more advanced CPU microarchitecture and various types of memory (Table 1.1). Unfortunately, the increase in complexity of this new platform also results in a higher difficulty of ensuring reliable software operation and optimal performance.

Table 1.1: feature comparison between M4 core in an NXP MK66FN2 and M7 core in an NXP S32K344. (ARM, 2022; NXP Semiconductors, 2017; NXP Semiconductors, 2022a; NXP Semiconductors, 2022b; NXP Semiconductors, 2018)

	Cortex-M4 in NXP MK66FN2	Cortex-M7 in NXP S32K344
Max. frequency	180 MHz	160 MHz
Pipeline	3-stage	6-stage, 2-way superscalar
Instruction cache	8 KiB, 2-way associative,	8 KiB, 2-way associative
Data cache	shared	8 KiB, 4-way associative
Instruction TCM	-	64 KiB (lockstep config)
Data TCM	-	128 KiB (lockstep config)
SRAM	256 KiB	320 KiB
Program flash	2 MiB	4 MiB
Data flash	-	128 KiB

NXP Mobile Robotics seeks to mitigate this increase in difficulty for developers by making the tracing features available in the S32K3 series accessible to developers, providing them deeper insight into the operation of their system without needing intricate knowledge of the tracing components themselves. NXP Mobile Robotics' goal thus aligns with the goals of the Orbcodes developers, but is focused specifically on NXP S32K3 series microcontrollers.

1.3 Assignment description

The goal of this graduation assignment is to develop new tools and enable existing open-source tools (e.g. Orbuculum) that exploit the tracing features of the S32K3 series for achieving specific goals, with the NXP S32K344 MCU used as the target device. The main research question is as follows:

"How can the tracing features provided by the NXP S32K344 MCU be made accessible to software developers for the purpose of profiling, optimizing, and debugging applications?"

The top-level requirements directly following from this assignment goal are listed in Section 2.1.

1.4 Deliverables

The following deliverables of interest to both HU University of Applied Sciences and NXP Mobile Robotics will be created during the graduation project:

1. **PID (completed as of 2023-03-13).** Describes the student's plan of approach from both a technical and process perspective.
2. **Graduation report (this document).** Describes the student's execution and results of the assignment from both a technical and process perspective.
3. **Final presentation.** Presents the student's execution and results of the assignment from a technical perspective. New developments since finalization of the graduation report are also disclosed.

Deliverables mainly of interest to NXP Mobile Robotics are as follows:

4. **Tools and configuration files/scripts for using trace features.** These are the deliverables of main interest, and are intended to be used by future developers utilizing S32K3-based NXP Mobile Robotics products. These deliverables are included in appendices B through E.
5. **User guide for tracing on S32K3.** This user guide allows future developers to setup and use the trace tools and supporting configuration files/scripts. This deliverable is included in appendix F.
6. **Insights gained from exercising an existing application with the tracing tools.** To demonstrate the applicability of trace tools enabled/developed, they will be used to analyze and optimize an existing application. This deliverable is included in Section 5.2.
7. **General knowledge about S32K3 tracing gained while working on the assignment.** This knowledge will be of use to the Mobile Robotics team going forward.

1.5 Initial assignment description and motivation for changes

Originally, the graduation assignment focused on the development of performance profiling methods targeting the M7 CPU inside the S32K344. These profiling methods would allow embedded software developers targeting the S32K344 to gain insight into performance bottlenecks of their application, with the intention that they could subsequently use these insights to optimize their application to better exploit the available resources in the S32K344, and in particular, its M7 core. The profiling methods were mainly targeted towards software developers that have relatively little experience with the process of performance optimization or the S32K3 series MCUs; making profiling methods accessible to these users would allow them to gain this experience more quickly.

The new assignment goal both narrows down and expands the scope of the original assignment: the new assignment focusses on enabling tracing features on S32K3 for the purpose of profiling, optimization, and

debug, while the original assignment only intended to incorporate tracing in profiling methods where this was relevant and didn't focus on either optimization or debugging.

The change in assignment goal was motivated by two factors. First, during the development of the second profiling method (measuring instruction fetch stalls experienced by the M7 core), it was concluded that implementing this method in an effective manner was infeasible (see Section 3.5.1), and a similar conclusion was subsequently drawn for the remaining profiling methods. Second, based on initial experiments conducted with orbttop (allowing measurement of CPU load through tracing), it was concluded that the main interest of the graduation project was actually in exploiting tracing features in general.

Ramifications of the assignment redefinition for other parts of this graduation report (e.g. research sub-questions, requirements, project schedule) are discussed in their respective chapters.

2 Requirements

This chapter documents the requirements package for the graduation assignment. Due to the nature of the assignment, where various tracing tools and a user guide are regarded as the main deliverables, requirements are categorized using a custom scheme:

1. **Top-level requirements:** requirements that have been defined at the start of the project.
2. **User requirements:** requirements that specify how the tracing tools are presented to a software developer using them.
3. **Non-user requirements:** requirements that specify what the tracing tools should do.

Both the user and non-user requirements have been derived from the top-level requirements while working on the graduation assignment. All requirements are prioritized using the MoSCoW method (Brush, 2023), which classifies requirements as follows:

1. **must have (M):** fulfillment of these requirements is considered mandatory for the graduation project to be deemed successful.
2. **should have (S):** these requirements are important and effort should be spent towards meeting them within the available time, but meeting them is not mandated for project success.
3. **could have (C):** these requirements denote features that are desirable but not necessary, and should thus be handled with less priority. These requirements can be regarded as ‘stretch goals’.
4. **won’t have (W):** these requirements are of the lowest priority. They also denote features that are desirable, but it’s not certain when, if ever, they will be implemented.

To accommodate the redefinition of the project goal (Section 1.5), most requirements have been updated since the PID. Newly defined requirements are marked with asterisks (*).

2.1 Top-level requirements

ID	Source	MoSCoW	Description
R.T.1.	Discussion with supervisors	M	Tools that assist in profiling, optimizing, and debugging software by using ARM tracing features must be developed for/enabled on NXP’S S32K344 microcontroller.
R.T.2.	Discussion with supervisors	M	The tracing tools must be made accessible to software developers that have no prior experience with these tools, ARM tracing features, the M7 CPU, or the S32K344 MCU.
R.T.3.	Discussion with supervisors	M	Applicability of the tracing tools must be demonstrated by applying them to a real-world application that is relevant to NXP Mobile Robotics’ activities.
R.T.4.	Discussion with supervisors	S	Tracing tools and any hardware (e.g. trace probes) necessary for using them should be cost-effective to aid in making them accessible.
R.T.5.*	Discussion with supervisors	M	GDB must be used when a debugger, running on a host computer, is needed to use a tracing tool.

2.2 User requirements

ID	Source	MoSCoW	Description
R.U.1.	R.T.2, PID	M	The documentation for each tool must provide the user with information on whether it is relevant for what the user is trying to achieve.
R.U.2.	R.T.2, PID	M	The documentation of each tool must describe its setup process in a step-by-step manner.
R.U.3.	R.T.2, PID	M	The documentation of each tool must state all prerequisites necessary to be able to set it up.
R.U.4.	R.T.2, PID	M	The documentation for each tool must provide the user with information on potentially useful next steps to perform after having interpreted retrieved results.
R.U.5.	R.T.2, PID	M	The user guide must be structured such that documentation for more tools can be added in the future.
R.U.6.	R.T.2, PID	M	Documentation for setting up any prerequisites used by tracing tools must be provided.
R.U.7.*	R.T.2, §3.7	M	The documentation for each tool must provide explanations on how retrieved results should be interpreted.
R.U.8.*	R.T.2, §3.7	M	The documentation for each tool must include a description of limitations associated with it.
R.U.9.*	R.T.2, §3.7	M	The documentation for each tool must direct the user to other sources for further reading on topics related to the tool.
R.U.10.*	R.T.2, §3.7	M	The user guide must describe an assumed hardware and software setup used, which the reader should adhere to as closely as possible.

2.3 Non-user requirements

Trace tools

ID	Source	MoSCoW	Description
R.N.1.	R.T.1, PID	M	A tool for measuring CPU load of an application running on S32K344 utilizing DWT trace data must be enabled/developed.
R.N.2.*	R.T.1, §3.2.1	M	A tool for automatically mapping an application's functions to the M7 core's ITCM based on CPU load data must be enabled/developed.
R.N.3.*	R.T.1, §3.2.1	M	A tool for generating call graphs of the application running on the S32K344 based on ITM trace data must be enabled/developed.
R.N.4.*	R.T.1, §3.2.1	S	A tool for reconstructing the M7 core's execution based on ETM trace data should be enabled/developed.
R.N.5.*	R.T.1, §3.6.1	S	orbstat should be augmented with ETM compatibility to allow for non-intrusive generation of call graphs.
R.N.6.*	R.T.1, §3.2.1	S	A tool for investigating CPU load transients based on trace data should be developed.

Trace data transfer

ID	Source	MoSCoW	Description
R.N.7.	R.T.1, PID	M	The SWO interface of the S32K344 must be made functional for retrieving trace data.
R.N.8.	R.T.1, PID	S	The parallel trace interface of the S32K344 should be made functional for retrieving trace data.
R.N.9.	R.T.4, PID	S	The Orbcodes ORBTrace Mini should be used as the trace probe for retrieving data from either SWO or parallel trace.
R.N.10.*	R.T.5, §3.4.2	M	Configuration of trace bus components (ETF, CSTF) and off-chip interfaces (SWO TPIU, parallel trace TPIU) must be done from GDB.
R.N.11.*	R.T.5, §3.4.2	M	Configuration of trace sources (DWT, ITM, ETM) must be done from GDB.

PX4 Autopilot trace tool test setup

ID	Source	MoSCoW	Description
R.N.12.	R.T.3, PID	M	PX4 Autopilot must be used as the real-world application that profiling methods are tested and demonstrated with.
R.N.13.*	R.N.12, PID	M	The PX4 Autopilot application must be run on an NXP MR-CANHUBK3 board, connected to the NXP MR-Buggy3 rover.
R.N.14.*	R.N.12, §5.1.2	M	PX4 Autopilot running on the NXP MR-Buggy3 rover must be functional such that it is capable of driving the rover in manual control mode.
R.N.15.*	R.N.12, §5.1.2	C	PX4 Autopilot running on the NXP MR-Buggy3 rover could be functional such that it is capable of driving the rover in automated/mission mode.

pyOCD for S32K344

ID	Source	MoSCoW	Description
R.N.16.*	R.N.9, §3.3.2	S	pyOCD should be used as the debugging toolchain for connecting a debugger to the S32K344.
R.N.17.*	R.N.16, R.T.5, §3.3.2	S	pyOCD should support the Cortex-M7-generic debug features in S32K344, including starting/stopping the CPU, setting/removing breakpoints, and accessing memory through the CPU.
R.N.18.*	R.N.16, R.T.5, §3.3.2	S	pyOCD should support programming and erasing the on-chip flash of the S32K344.

ITCM mapping tool

ID	Source	MoSCoW	Description
R.N.19.*	R.N.2, §3.5.3	M	The ITCM function mapping tool must keep track of the ITCM utilization while generating the linker script to ensure that ITCM does not become oversubscribed.
R.N.20.*	R.N.2, §3.5.3	M	The ITCM mapping tool must notify the user about functions with unknown sizes, as this could result in oversubscribing the ITCM.
R.N.21.*	R.N.2, §3.5.3	M	The ITCM function mapping tool must derive function sizes without user interaction by running GNU nm and parsing its output.
R.N.22.*	R.N.2, §3.5.3	M	The ITCM function mapping tool must notify the user about the orbtcp JSON containing demangled C++ function names.
R.N.23.*	R.N.2	M	The ITCM function mapping tool must be able to ignore functions listed in an ignore list.
R.N.24.*	R.N.2, §4.4	C	The ITCM mapping tool could automatically continue mapping functions to SRAM once ITCM is full.
R.N.25.*	R.N.2, §4.4	C	The ITCM mapping tool could automatically identify and map critical data structures of the application to DTCM.

3 Research

This chapter documents the research done during the design and implementation activities of the project. First, key points from the analysis phase and an updated research design (due to the assignment redefinition) are discussed. Subsequently, the research results are documented.

3.1 Key points from analysis phase

During the analysis phase of the project, research sub-questions 1.a through 1.c were answered (documented in the PID). The key points from this research are briefly reiterated here. As this research was conducted when the project still focused on developing profiling methods only, changes made to the original conclusions due to redefinition of the project goal are explained.

3.1.1 Selection of features and tools for performance profiling

Two types of potentially useful features for performance profiling on S32K344 were identified in the PID: performance monitoring registers and hardware tracing. Performance monitoring registers (provided by the Data Watchpoint and Trace unit; DWT) allow measuring e.g. cycle counts of instruction sequences and cycles spent stalling due to data loads/stores. These can be used to create microbenchmarks for performance profiling. Hardware tracing allows monitoring events occurring in the CPU at runtime and generating trace messages in response to these events. Three types of trace message sources are available: (1) the DWT can monitor miscellaneous types of events, such as data accesses/instruction fetches at specific addresses; (2) the Instrumentation Trace Macrocell (ITM) allows, among others, generation of trace messages from software and forwards trace messages generated by DWT; and (3) the Embedded Trace Macrocell (ETM) traces all instructions being executed and data accesses being done by the CPU such that exact execution of the program can be reconstructed using tooling on the host computer. Trace messages generated by any source are forwarded to a Trace Port Interface Unit (TPIU), which in turn transmits the data off-chip through either Single Wire Output (SWO) or the parallel trace interface (Figure 3.1).

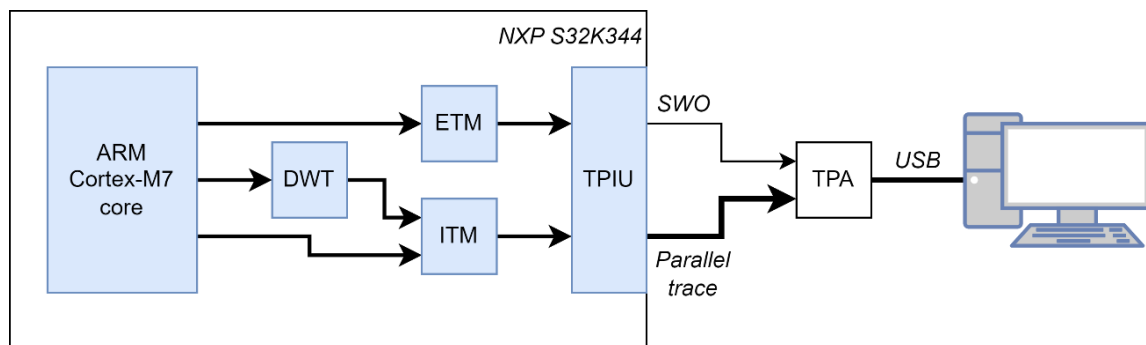


Figure 3.1: diagram from PID visualizing the flow of data through the trace components.

Trace messages output by the TPIU are received and forwarded to a host computer using a Trace Port Analyzer (TPA). Based on its attractive feature set and price, the ORBTrace Mini debug and trace probe was selected for this purpose.

Besides investigating TPAs, tools for parsing/visualizing received trace data were also explored. Of interest was the Orbculum tool suite, a fully open-source and extensible collection of tools that can parse trace data in different ways. For example, the tool orbttop allows reconstructing CPU usage of functions in an application by letting the DWT sample the program counter (PC). Another tool is orbstat, which allows generating call graphs by tracing each function entry and exit.

Based on the differences in complexity between SWO and parallel trace, it was decided to only use SWO with Manchester encoding at first. This was not expected to form a trace bandwidth bottleneck, since it was also decided that exploitation of the ETM, which is the most bandwidth-hungry trace source, would be a stretch goal for this project. These decisions still hold after redefining the project goal.

3.1.2 Definition of profiling methods to develop

Based on the identified features available in the S32K344, a list of profiling methods to develop was defined: profiling CPU usage of functions in an application (1), profiling impact of instruction fetch stalls on performance (2), profiling impact of memory access stalls on performance (3), profiling impact of peripheral access stalls on performance (4), and profiling impact of internal CPU stalls on performance (5).

The motivation behind each profiling method was explained, and a high-level description of how each method could be implemented was provided. In case of the first profiling method (profiling CPU usage), it was decided early on that making orbtap functional with S32K344 would be a fitting implementation. For other profiling methods, a combination of experimenting with the mapping of functions to various types of memory and running microbenchmarks was suggested. The potential use of tracing features for these profiling methods was yet unclear, and was left to be investigated during the development of the profiling methods.

After redefinition of the assignment goal, only the profiling of CPU usage was kept as a tool to develop/enable (sub-question 3.a). Profiling instruction fetch stalls (method 2) was dropped due to difficulty in finding an effective and accurate implementation (see Section 3.5.1), and all other profiling methods were dropped as a direct result of the project redefinition.

3.1.3 Selection of application for demonstrating profiling methods

To be able to test the effectiveness of developed profiling methods, it was decided that they should be applied to a real-world application that was representative of the workloads run by NXP Mobile Robotics. Setting up PX4 Autopilot on an MR-CANHUBK3 development board (containing an S32K344 MCU) in combination with an NXP MR-Buggy3 rover (Figure 3.2) was chosen for this purpose. This decision still holds after redefining the project goal: any tracing tools that were enabled/developed for using the S32K344's tracing features are exercised using this test setup as part of the verification process (Section 5.2).

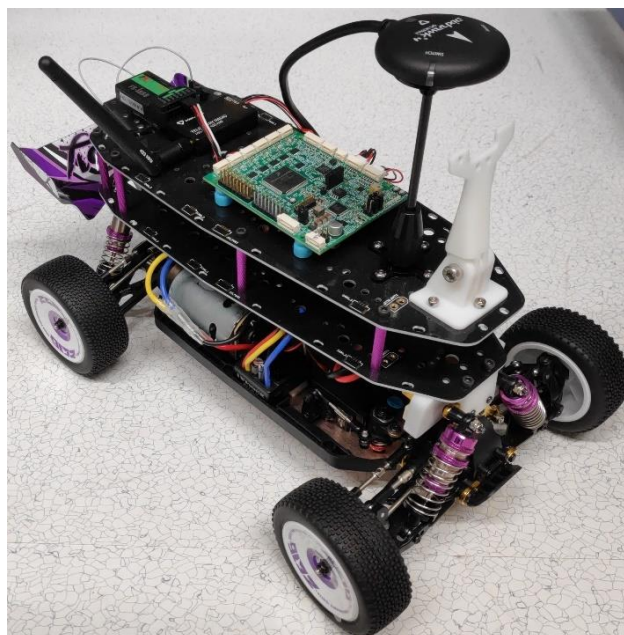


Figure 3.2: MR-Buggy3 rover.

3.2 Research design

The main research question from Section 1.3 has been divided into the sub-questions listed below. As sub-questions 1.a through 1.c were already answered in the PID, this chapter focusses on answering sub-questions 2.a, 2.b, 3.a through 3.c, and 4.a. Sub-questions 2.c, 2.d, and 3.d through 3.g will be answered after finalization of the graduation report.

1. Questions already answered in PID (before project redefinition)

- a. Which features and tools are available to gain insight into the way software executes on the ARM Cortex-M7 and which of these should be used?
- b. Based on the architectural features of the ARM Cortex-M7 and the NXP S32K344 MCU, what profiling methods should be developed?
- c. What application should be used to test and demonstrate the applicability of the developed profiling methods?

2. Setting up trace components

- a. How can support for using the ORBTrace Mini with S32K344 be added?
- b. How can DWT/ITM tracing on the S32K344 be enabled over SWO?
- c. How can DWT/ITM tracing on the S32K344 be enabled over parallel trace?
- d. How can ETM tracing on the S32K344 be enabled over parallel trace?

3. Using trace components for profiling, optimization, and debugging

- a. How can DWT tracing be used to measure the CPU load of an application?
- b. How can tracing be used to automatically optimize an application's instruction mapping?
- c. How can ITM tracing be used to generate a call graph of an application?
- d. How can ETM tracing be used to debug system crashes with an unknown cause?
- e. How can ETM tracing be used to generate a call graph of an application non-intrusively?
- f. How can tracing be used to investigate CPU load transients?
- g. How can tracing be used for profiling different kinds of CPU stalls effectively?

4. Letting a developer use tracing features

- a. How can tracing tools be made accessible to others developers, such that they can use them without prior experience?

3.2.1 Sub-question motivation and context

A diagram visualizing the relationship between sub-questions and the path they create towards answering the main research question is shown in Figure 3.3. While the motivation behind and dependencies between sub-questions are mostly straightforward, the following is noteworthy:

- Sub-questions 2.a, 2.b, and 3.a were adapted from the PID. Answering 2.a and 2.b is necessary for creating a setup that allows working on sub-question 3.a, hence the dependency.
- Sub-question 3.b is an extension to 3.a: CPU load measurement through DWT tracing is used for developing a tool that automatically optimizes instruction memory mapping.
- Sub-question 3.c focusses on enabling orbstat on S32K344, allowing generation of call graphs. This gives alternative insight into the application compared to measuring CPU load (sub-question 3.a).
- After the graduation report deadline, the research focus will shift from enabling tracing tools over SWO to enabling tracing tools over the parallel trace interface. First, the tracing tools already enabled over SWO are enabled over parallel trace as well (sub-question 2.c). Next, enablement of ETM over parallel trace is researched (sub-question 2.d).

- With ETM enabled, it becomes possible to enable/develop four additional trace tools for S32K344:
 - Sub-question 3.d: enabling orbmortem (from the Orbuculum suite) on S32K344 allows debugging system crashes/hard faults by tracing instruction execution of the M7 core.
 - Sub-question 3.e: using ETM instruction tracing, an alternative, non-invasive implementation of orbstat could be developed (see Section 3.6.1).
 - Sub-question 3.f: using either ETM or DWT, a tool for investigating short CPU load spikes (transients) will be developed. This idea stems from a bug currently present in PX4.
 - Sub-question 3.g: as a nod to the original project goal, an investigation into profiling possibilities with ETM tracing is conducted. This might result in additional tracing tools.

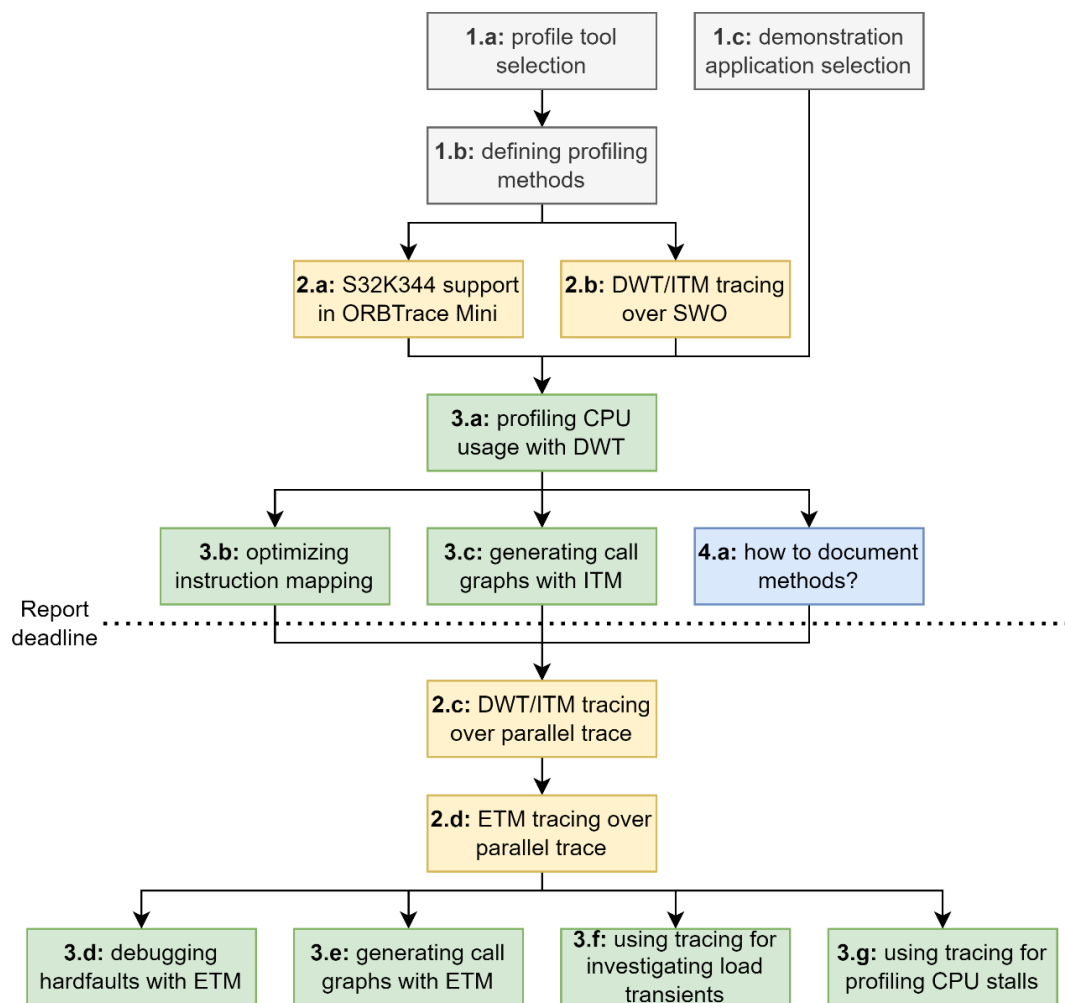


Figure 3.3: diagram showing the relationship between sub-questions. A sub-question can be researched once all preceding sub-question that point to it have been answered. Sub-questions listed side-by-side have no interdependency and can be researched in parallel. Greyed-out sub-questions have already been answered in the PID.

3.3 Adding support for S32K344 to ORBTrace Mini

To enable the tracing features of the S32K344, access to the configuration registers of the trace components is needed. These registers could be accessed programmatically by the CPU, but this requires recompiling and reflashing the application for each configuration change. Instead, trace components should be configured through the S32K344's debug subsystem (which provides access to memory-mapped peripheral registers). Since ORBTrace Mini is already intended to be used as the TPA (Section 3.1.1), it makes sense to also use it as the debug probe. Unfortunately, the S32K344 is officially only supported by commercially-developed debug probes, such as those manufactured by SEGGER or PEmicro (PEmicro, n.d.; SEGGER, 2023). Support for exploiting the S32K344's debug features using ORBTrace Mini must thus be implemented first. This section focusses on this effort by answering sub-question 2.a.

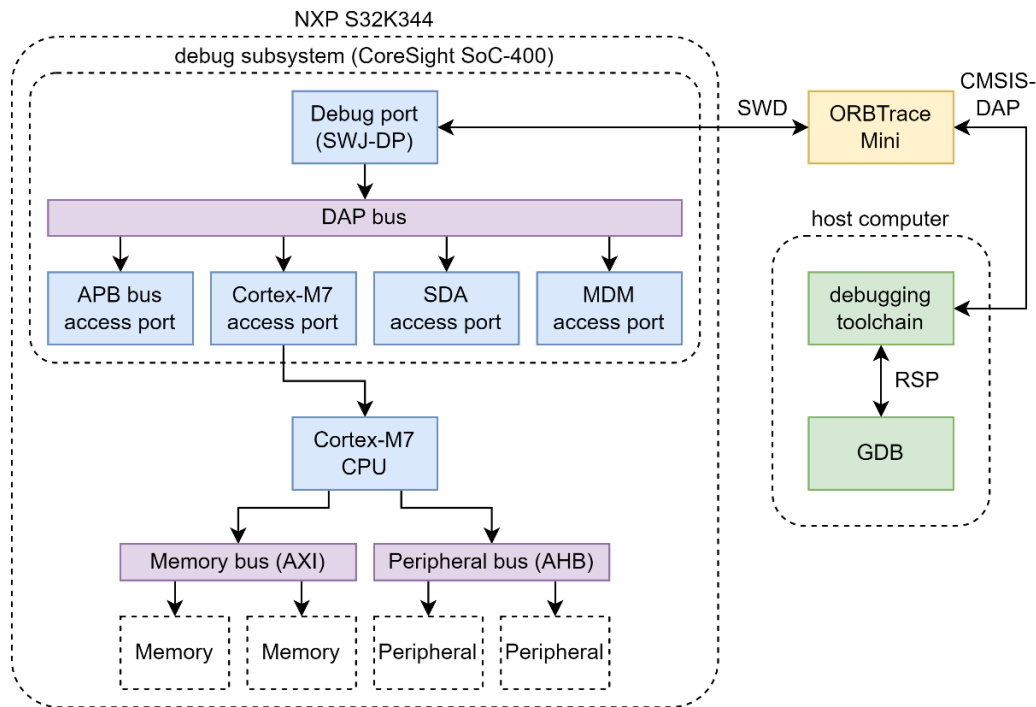


Figure 3.4: connection diagram of components involved in debugging the M7 core inside the S32K344 using GDB.

3.3.1 Debug connection chain overview

To enable S32K344 debugging with ORBTrace Mini, it is crucial to understand what components are involved in connecting a debugger to an ARM microcontroller. Figure 3.4 shows a connection diagram for these components. In the context of this project, GDB is used as the debugger (requirement R.T.5).

The connection diagram starts with GDB, running on a host computer. Since GDB will debug an application running on another system, it must connect to it using the GDB remote serial protocol (RSP) (Stallman, Pesch, & Shebs, Debugging Remote Programs, 2023a). The S32K344 doesn't expose a GDB server directly, so instead GDB connects to a debugging toolchain that translates RSP commands received from GDB into corresponding actions in the debug subsystem of the S32K344. The debugging toolchain accesses the debug subsystem by connecting to ORBTrace Mini using the ARM CMSIS-DAP protocol over USB, and ORBTrace Mini 'forwards' this connection to the debug port of the S32K344 through ARM SWD (Orbcode, 2021).

The debug port allows an external debug probe access to the DAP bus, and the DAP bus in turn allows access to the access ports (AP) of the debug subsystem (ARM, 2015). Finally, the APs provide access to the various debug and trace features inside the S32K344. For example, the Cortex-M7 AP provides access to the registers of the debug components inside the M7 core, allowing configuration of e.g. hardware breakpoints and starting/stopping the CPU (ARM, 2018). The Cortex-M7 AP also provides access to the memory and peripheral busses of the S32K344, allowing modification of memory contents or peripheral access directly from GDB. The APB bus AP provides access to a small bus hosting various chip-wide debug components. The system peripheral bus also has access to this small APB bus through an AHB-APB bus translator, making it so that the chip-wide debug components can either be accessed directly through the APB bus AP or indirectly through the M7 core and peripheral bus (NXP Semiconductors, 2022c). Further functionalities provided by the APs in the S32K344 are discussed when relevant.

3.3.2 Debugging toolchain selection

In Figure 3.4, the debugging toolchain is the component that deals with the characteristics and peculiarities unique to each microcontroller: it receives RSP commands from GDB and translates this into a sequence of actions compatible with the APs and debug features provided by the specific microcontroller. For example,

when GDB wants to upload an application to the internal flash of the S32K344, the debugging toolchain receives an 'upload' command from GDB and executes it using the algorithm expected by the on-chip flash controller. Since the communication protocols between the components in the debug chain are standardized (SWD, CMSIS-DAP), the only component that needs modification to enable S32K344 compatibility is the debugging toolchain.

Various open-source debugging toolchains are available, such as openOCD, pyOCD, Black Magic Probe software, and probe.rs. Of these, pyOCD provides a step-by-step guide on how to add support for new MCU targets using CMSIS-Packs (Reed, 2020). CMSIS-Packs are standardized configuration files documenting unique aspects of a specific microcontroller, such as the memory map and the algorithms used to program flash memory (ARM, n.d.). NXP has made a CMSIS-Pack for the S32K3 series publicly available, making fast development of an S32K344 target configuration for pyOCD *with* support for the S32K344's flash algorithms possible. This would satisfy requirements R.N.10 and R.N.11. Based on this, pyOCD is selected as the debugging toolchain.

3.3.3 Fixing the AP discovery sequence

Development of a functional pyOCD target configuration for S32K344 is more involved than just following the pyOCD-provided guide mentioned above. This is because pyOCD's initialization sequence for connecting to ARM microcontrollers contains an AP discovery sequence that is not fully compatible with the S32K344. By default, this AP discovery sequence goes roughly as follows (Reed, 2022):

1. Connect to debug port in S32K344 through ORBTrace Mini.
2. Scan the DAP bus for APs by probing all AP addresses; add addresses of detected APs to a list.
3. Scan each detected AP for debug components connected to it by reading the AP's ROM table and the identification registers of each debug component.
4. Store base addresses and type information of discovered debug components in each AP in a list.
5. Store which APs were discovered as being dedicated to a CPU core. These APs will be used to manage execution and debugging of these CPU cores.

On S32K344, probing an AP that does not exist results in a memory read fault, causing pyOCD to crash during step 2. To solve this, the AP probing must be skipped or reimplemented in a compatible manner. Furthermore, access to AP registers in S32K344 is disabled after a power cycle/hardware reset, causing step 3 in the above sequence to fail. Debug features can be enabled using a specific control register in the SDA AP (See DAP bus in Figure 3.4; this AP is also used to configure other debug-related features.). Section 4.1 discusses the design of a modified AP discovery sequence with S32K344 compatibility.

3.4 Setting up tracing over SWO and enabling orbtap

This section answers sub-questions 2.b and 3.a by documenting the research done for enabling orbtap on S32K344. As mentioned in Section 3.1.1, orbtap reconstructs the CPU load of an application (Figure 3.5) at function level by matching PC samples retrieved against the addresses of instructions inside each function. This address information is provided by an ELF application binary with DWARF-compliant debug symbols (IBM, 2013). PC sampling is a dedicated feature provided by the DWT: at a configured interval, the DWT samples the current PC value of the M7 core and sends out a trace message containing this sample. This trace message is subsequently sent to the host computer (running orbtap) through SWO.

```

47.44%    9283 delay
18.60%    3641 Lpuart_Uart_Ip_GetStatusFlag
13.22%    2587 Lpuart_Uart_Ip_CheckTimeout
 9.22%    1804 OsIf_GetElapsed
 5.84%    1144 Lpuart_Uart_Ip_SyncSend
 3.07%     602 bench
 1.97%     386 OsIf_Timer_Dummy_GetElapsed
 0.26%      52 Lpuart_Uart_Ip_PutData
-----
99.62%   19499 of 19566 Samples
[---H] Interval = 1001mS / 0 (~0 Ticks/mS)

```

Figure 3.5: example output of `orbttop`. `delay()` is shown as using 47,4% of CPU time within the measurement interval.

3.4.1 S32K344 trace bus overview

The trace infrastructure of the S32K344 must be understood before being able to exploit the PC sampling feature of the DWT for `orbttop`. Figure 3.1 shows a simplified trace bus to illustrate how trace messages are conveyed to a host computer. Figure 3.6 shows the actual trace bus of the S32K344, which is considerably more complex and contains a variety of component types:

1. **DWT, ITM, and ETM:** these are the trace sources, as already discussed in Section 3.1.1.
2. **CoreSight Trace Funnel (CSTF):** this component combines multiple incoming trace streams into a single output stream (ARM, 2015). The S32K344 contains three funnels. By default, they are configured to block all incoming streams.
3. **Embedded Trace FIFO (ETF):** this component is a hardware FIFO that stores and forwards the incoming trace stream on its output. Its main function is to absorb load peaks: when the incoming trace stream temporarily has uses more bandwidth than can be drained at the ETF's output, the ETF locally buffers the trace messages at the input rate, but keeps outputting them at the output rate. This results in no loss of trace data as long as the FIFO buffer does not overflow (ARM, 2010).
 - a. While (NXP Semiconductors, 2022c) refers to this component as 'ETF' only, at power on the ETFs are actually configured as Embedded Trace Buffers (ETB). This mode of operation behaves as a circular buffer that stores all incoming trace messages and doesn't forward them. This allows first acquiring trace data and later retrieving the buffered data through a memory-mapped interface. For this project, this default operation mode is not preferred and the component must thus be reconfigured to act as an ETF.
4. **Replicator, parallel trace TPIU, and SWO TPIU.** The S32K344 uses two separate TPIUs, one dedicated to the parallel trace interface and one dedicated to the SWO interface. While a single TPIU could also switch between both interface types, this setup allows using both trace output interfaces simultaneously. The replicator functions as a demultiplexer that sends each trace message to both TPIUs. To enable PC sample tracing over SWO, these three components must be configured such that trace messages are forwarded to the SWO TPIU only.

The complexity of the trace bus stems from its ability to combine trace streams from many sources into a single stream going off-chip (through parallel trace or SWO). While the trace bus seems overbuilt for the S32K344's relatively small amount of trace sources, other MCU variants in the S32K3 family contain more trace sources by way of having multiple independent CPU cores, each with their own DWT, ITM, and ETM.

3.4.2 Trace bus and DWT setup considerations

To enable periodic PC sample tracing, the DWT, ITM, SWO TPIU and all intermediate trace bus components must be configured such that trace messages are transmitted through the red-highlighted path in Figure 3.6. Since GDB has direct access to S32K344 peripheral registers through the Cortex-M7 AP, it makes sense to implement the configuration of tracing components using GDB commands. GDB can be configured to

automatically invoke these configuration commands at startup, improving ease of use and accessibility (requirement R.T.2). Sections 4.2 and 4.3 discuss the design of the GDB commands for setting up the trace bus and the DWT/ITM for use with orbtap, and orbtap is exercised with PX4 Autopilot in Section 5.2.1.

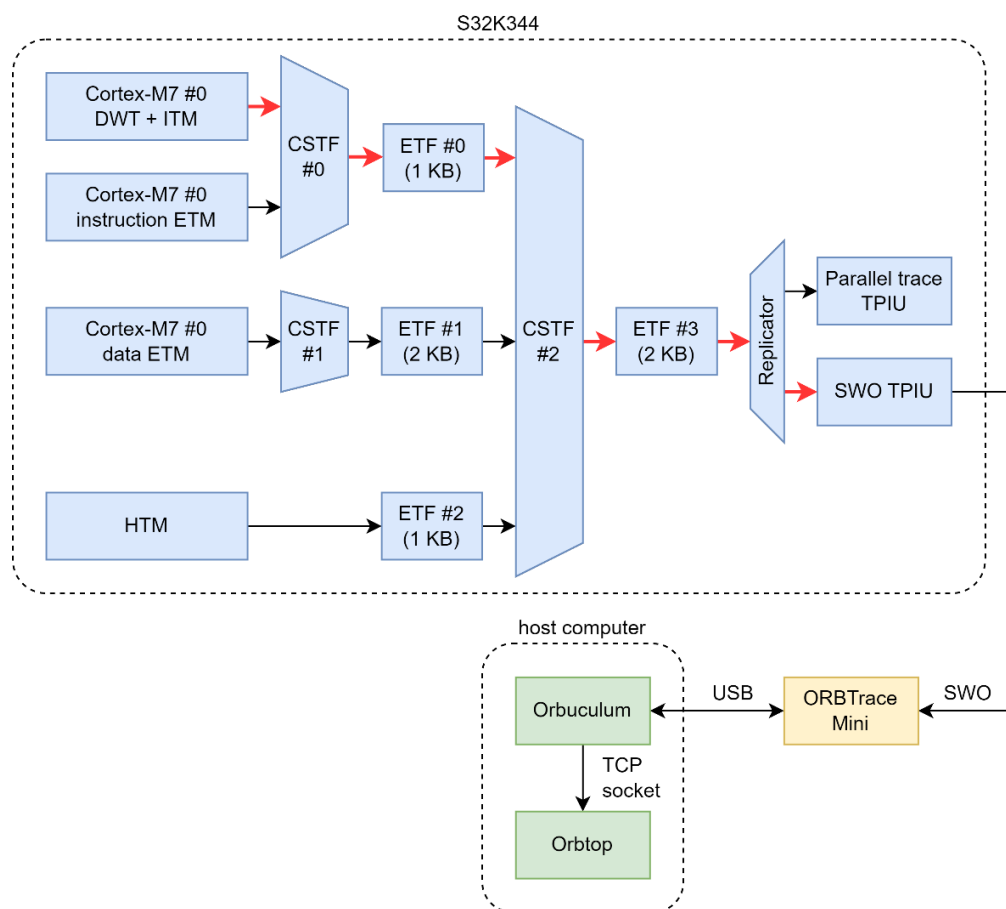


Figure 3.6: diagram showing the connection of the DWT + ITM to a desktop computer through the S32K344's trace bus.

3.5 Automatically optimizing instruction mapping

As originally mentioned in the PID, the M7 core in the S32K344 can execute instructions from various types of memory. Inside the M7 core itself are the instruction tightly coupled memory (ITCM, 64 KiB) and an instruction cache (8 KiB), both with the same access latency (NXP Semiconductors, 2022a). Through its external AXI memory interface, the M7 core can also fetch instructions from 320 KiB of SRAM or 4 MiB of on-chip flash. Fetching from SRAM is slower than TCMs and caches, while flash access time is even slower.

Due to its large capacity and non-volatile nature, the flash is used to store all application instructions. For program sections which are bound by instruction fetch latency, this imposes a performance bottleneck: for each instruction, the CPU pipeline has to stall for the amount of cycles necessary to read from flash. The instruction cache alleviates this bottleneck by dynamically storing instructions which are fetched often, but this is only effective for small code footprints due to its relatively small capacity.

To alleviate the instruction fetch bottleneck, parts of the application should be fetched from ITCM instead. Not only does this guarantee that these instructions are fetched with the lowest possible latency, but it also reduces pressure on the instruction cache: instructions fetched from ITCM aren't cached anymore (ARM, 2018), allowing the cache to more effectively cache instructions that do still execute from flash. Manually mapping functions into ITCM is time-consuming and hard to maintain. To automate this process, a tool that automatically maps functions of an application to ITCM should be created. This section answers sub-question 3.b by researching how such a tool can be developed.

3.5.1 Identifying eligible functions

Due to the ITCM's limited size, only functions that would benefit most from executing from ITCM should be mapped to it. These would be functions that are actually bottlenecked by instruction fetch stalls. Various methods of identifying eligible functions for remapping were considered:

1. **orbttop can generate a JSON file with all function names**, sorted by CPU load from highest to lowest (Marples, et al., 2022). By naively assuming that all functions are somewhat bottlenecked by instruction fetch, all high-load functions measured by orbttop could be identified as eligible for remapping.
2. **A GDB script could be written that automatically profiles a function for fetch stalls**. This script would read `CPICNT` (a DWT profiling counter that increments for each fetch stall cycle or execution stall cycle (ARM, 2021)) at function entry and exit (using breakpoints) and calculate the amount of stall cycles that occurred. These stall cycles could then be compared to the amount of instructions in the function. This method measures exactly the information that is needed for effective remapping to ITCM, but experiences major problems:
 - a. `CPICNT` is only 8 bits wide, making its use impractical for profiling functions that experience more than 256 stall cycles.
 - b. Interrupts and/or calls to other functions could occur during function execution, which makes it hard to limit the stall cycle measurement to only the function being profiled.
 - c. The GDB script would have to be run for each function, which requires intensive user interaction (and is thus not automated) or requires the development of another, more complex script that manages this.
 - d. Comparing the stall cycles to the function's instruction count could be inaccurate, since the M7's dual-issue pipeline often executes pairs of 2 instructions in one cycle.
3. **The DWT could be configured to emit trace messages at rollover of `CPICNT`**. The frequency of the trace messages would then give an indication of the intensity of fetch stalls within a function. This method also faces major problems:
 - a. A trace message only occurs at `CPICNT` rollover, so profiling stalls of functions with less than 256 stall cycles becomes impractical.
 - b. Limiting `CPICNT` to only incrementing in the function of interest is not possible without intervention of e.g. GDB, which is impractical for automation.

Based on the above, naively using orbttop JSON output for identifying high-load functions was chosen. The other two methods would allow more definitive identification of functions that are actually bottlenecked by fetch stalls, but they come with major challenges in making them work reliably and/or accurately. These same implementation problems were what led to a reconsideration of the project goal: originally a method for profiling instruction fetch stalls was intended to be developed, but the impractical implementation strategies mentioned above led to this being dropped in favor of developing an ITCM mapping tool.

3.5.2 Placing functions into ITCM

Placement of eligible functions into the memories available in a microcontroller is handled during the linking phase of building an application: the linker takes a range of input sections (containing e.g. functions or initialization data) from compiled/assembled object files and places them in output sections (which can be regarded as memory ranges within this context) of an executable output file (Baldassari, 2019; Pyeatt, 2016). During this process, each instruction receives a definitive address, and target addresses for e.g. subroutine calls and jumps are resolved based on these. The manner in which the linker maps input sections to output sections is governed by a *linker script*; Listing 3.1 shows a basic example.

To map functions into ITCM, this memory must be defined in the linker script. Subsequently, specific functions can be mapped into this memory range by specifying the function name in the linker script; see Listing 3.2. The mapping of the `itcmfunc` output section to memories shows an important detail: `itcmfunc` is mapped to both the `itcm` and `flash` memory regions. The `AT` keyword is used to indicate to the linker that the contents of `itcmfunc` are located in ITCM at runtime (Chamberlain & Taylor, 2023), but are to actually be stored in flash (the only non-volatile memory in the system). This allows the CPU to load the contents of `itcmfunc` from flash into ITCM during application startup.

```
MEMORY
{
    flash (rx) : ORIGIN = 0x400000, LENGTH = 0x3dffff
}

SECTIONS
{
    .text :
    {
        *(.text .text.*)
    } > flash
}
```

Listing 3.1: example linker script that maps all functions (by default placed in section `text`) to flash.

```
MEMORY
{
    itcm (rwx) : ORIGIN = 0x0, LENGTH = 0x20000
    flash (rx) : ORIGIN = 0x400000, LENGTH = 0x3dffff
}

SECTIONS
{
    .itcmfunc :
    {
        *(.text.memset)
    } > itcm AT > flash

    .text :
    {
        *(.text .text.*)
    } > flash
}
```

Listing 3.2: linker script that first maps the function `memset()` to ITCM and maps all other functions to flash.

3.5.3 Implementation considerations

This section discusses various implementation considerations for the mapping tool.

For automating function placement, the tool needs to keep track of the size of functions it maps to ensure it doesn't oversubscribe the ITCM. Various tools for retrieving function sizes are available, such as GNU nm and GNU objdump (Pesch, Osier, & Cygnus Support, 2023). Listing 3.3 and Listing 3.4 show example output of both tools. The output of nm is shown to be simpler and cleaner, though the difference is small. Based on this, nm is chosen to retrieve function sizes. In line with the automation goal, the mapping tool should independently invoke nm and parse its output to generate an internal list of function names and sizes.

nm isn't always successful in providing sizes for functions. Not knowing the size of a function being mapped to ITCM could cause the ITCM to become oversubscribed, so the user should be warned when this occurs.

```
00000100 0000004c T hrt_absolute_time
00549598 0000002c T hrt_call_after
0000a9f0 0000004c t hrt_call_enter
005495c4 0000002e T hrt_call_every
0000d1d0 00000048 t hrt_call_internal
005494e4 00000054 t hrt_call_reschedule
005495f4 0000002c T hrt_cancel
00549558 00000040 T hrt_init
00549538 0000001e T hrt_store_absolute_time
0000cd20 000000d4 t hrt_tim_isr
```

Listing 3.3: example output of GNU nm with the `-S` flag. The second column shows function size.

005494e4	l	F .text	00000054	hrt_call_reschedule
0000a9f0	l	F .itcmfunc	0000004c	hrt_call_enter
0000d1d0	l	F .itcmfunc	00000048	hrt_call_internal
0000cd20	l	F .itcmfunc	000000d4	hrt_tim_isr
005495c4	g	F .text	0000002e	hrt_call_every
00549538	g	F .text	0000001e	hrt_store_absolute_time
005495f4	g	F .text	0000002c	hrt_cancel
00000100	g	F .itcmfunc	0000004c	hrt_absolute_time
00549598	g	F .text	0000002c	hrt_call_after
00549558	g	F .text	00000040	hrt_init

Listing 3.4: example output of GNU objdump invoked with the `-t` flag. Function sizes are shown in the fourth column.

Instead of letting the mapping tool generate an entire linker script, it makes sense from an implementation standpoint to only let the tool generate a list in linker script-compatible format, which the user can subsequently include into their own linker script with GNU LD's `INCLUDE` command (Chamberlain & Taylor, 2023). This way, the user is still able to control other aspects of their linker script without intervention of the mapping tool.

Object files generated from C++ sources contain *mangled* functions, which are used to implement various features of the C++ language (IBM, 2023). For legibility purposes, mangled function names can be demangled. For example, the mangled function `_ZN7sensors22VehicleAngularVelocity3RunEv` from PX4 Autopilot demangles into `sensors::VehicleAngularVelocity::Run()`. orbttop is able to show mangled or demangled function names, but GNU LD (the linker) can only handle mangled function names. To ensure that users only provide orbttop JSON files with mangled function names, the mapping tool must show a warning when demangled function names are detected.

Section 4.4 discusses the design of the mapping tool based on the considerations made here, while Section 5.2.2 discusses exercising the mapping tool with PX4 Autopilot.

3.6 Generating call graphs with ITM tracing

As mentioned in Section 3.1.1, orbstat allows generating call graphs of an application by tracing entry and exit of functions. A call graph, such as the one shown in Figure 3.7, shows the relationships between functions of an application. This can provide insights such as:

1. Understanding the architecture of an existing application (e.g. PX4 Autopilot);
2. Identifying critical paths (i.e. function calls occurring often) in the application;
3. Identifying functions belonging to a subsystem of the application.

These insights could also be used to identify optimization opportunities that aren't obvious based on CPU load measurement (orbttop). For example, the interrupt stack of an application could be identified in the call graph and subsequently mapped to ITCM. This would ensure deterministic fetching of the interrupt handlers, which could be an important property for certain real-time applications. The same could apply to the scheduling functions of an RTOS running on the MCU. This section answers sub-question 3.c by researching how orbststat can be enabled on S32K344.

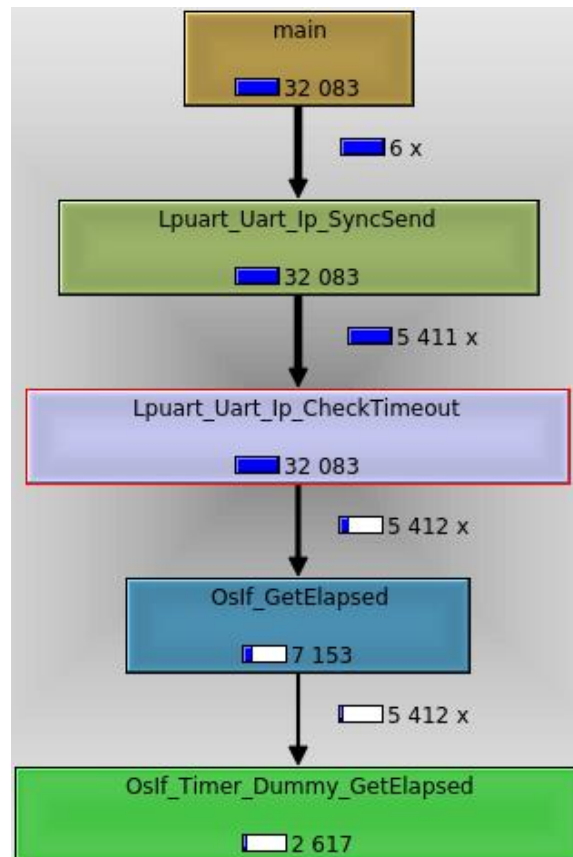


Figure 3.7: call graph of a function involved in transmitting data using a UART peripheral.

3.6.1 orbststat setup considerations

orbststat uses two features for generating call graphs (Marples, 2022a): the GCC `-finstrument-functions` compilation flag and ITM software trace channels. Enabling `-finstrument-functions` during software compilation makes GCC insert calls to instrumentation functions for each application function: at the start of an application function, `__cyg_profile_func_enter(void *this_fn, void *call_site)` is called, and before the end of an application function, `__cyg_profile_func_exit(void *this_fn, void *call_site)` is called (Stallman & GCC Developer Community, 2023b). ITM software trace channels allow programmatically generating ITM trace messages, containing arbitrary data written by the software (usually application-specific debug information) (ARM, 2021).

By using the instrumentation functions to send data over an ITM software channel at each function entry and exit, the data needed by orbststat can be generated. For each instrumentation function call, orbststat expects to receive the following data from ITM:

1. The current value of `DWT_CYCCNT` (a cycle counter in the DWT counting at the CPU clock) along with whether a function entry or exit is currently occurring.
2. The address from where the current application function being instrumented was called from (`void *call_site`).
3. The address of the current application function being instrumented (`void *this_fn`).

Section 4.5 discusses the design of the instrumentation functions based on the above considerations.

In contrast to the PC sample tracing used by `orbttop`, generating the trace data needed by `orbstat` imposes additional CPU load: for each normal application function, two instrumentation functions are executed. Depending on the amount of functions in the application, this can lead to application performance becoming insufficient and/or real-time software requirements not being met. Using `orbstat` with PX4 Autopilot (Section 5.2.3) demonstrates this clearly: instrumenting a single peripheral driver already imposes ~74,3% of CPU load resulting from instrumentation. Conceptually, ETM tracing (used for full instruction tracing of the CPU) could also be used to provide the information needed by `orbstat` without imposing CPU load. Sub-question 3.e focusses on developing a non-intrusive implementation of `orbstat` using ETM trace.

3.7 Letting a developer use tracing tools

This section discusses how developed/enabled tracing tools should be documented in order to make them accessible to interested developers (sub-question 4.a). The preliminary design discussed in the PID gave an indication of how to-be-developed profiling methods could be presented to other developers, which resulted in the definition of user requirements R.U.1 through R.U.6. While the focus has since shifted away from developing profiling methods, these user requirements still hold for documenting the tracing tools and largely answer sub-question 4.a already: documentation for tools should describe their intended use-cases, describe how to set them up, suggest potential next steps after having used the tool, etc. Since defining these user requirements, some additional considerations were made for the user guide:

1. As part of listing prerequisites for each tool and explaining their setup process, an assumed setup with which the tools were developed and tested should also be described in the user guide. This provides users with information on how to connect their hardware appropriately for using the trace tools. Describing the assumed setup also allows a user to take differences with their setup into account.
2. Requirement R.U.4 states that the user should be informed of potentially useful next steps to perform after using a tool. What next steps are relevant depends on the results the user retrieved, so the user guide must also explain how to interpret these results.
3. Some tracing tools allow changing settings to tailor the output to user needs. `orbttop`, for example, allows showing CPU load at function or line level, but only the former option is selected by default. The user guide should mention configuration options like these.
4. Some trace tools have limitations with regards to what they can measure. For example, since `orbttop` only samples a fraction of the PC values the M7 core actually executes, instruction sequences/functions which execute much faster than the sampling interval will be missed. This limits measurement granularity. Limitations like these should also be explained in the user guide.
5. The reader must be directed to external sources for further reading (if desired) related to the tracing tool discussed. For example, the chapter for the ITCM mapping tool could redirect the user to linker script documentation.

The final design of the user guide, based on the user requirements and the above considerations, is summarized in Section 4.6.

4 Final designs

This chapter discusses the final designs of any tracing tools developed for and configuration files/initialization sequences created to enable compatibility of existing tools (e.g. pyOCD, orbtopy) with S32K344. For each design, a short description is also given on how it adheres to relevant requirements. Systemic verification of the implemented designs through compliance tests is the subject of Section 5.3.

4.1 pyOCD configuration file for S32K344

Based on the findings from Section 3.3.2 and 3.3.3, a pyOCD target configuration was developed that allows access to all APs and their connected components, enables the debug features in the Cortex-M7 AP for use with GDB, and is capable of uploading applications to the S32K344's on-chip flash. With this, the design meets requirements R.N.16, R.N.17, and R.N.18. The full pyOCD configuration file is shown in Appendix B.

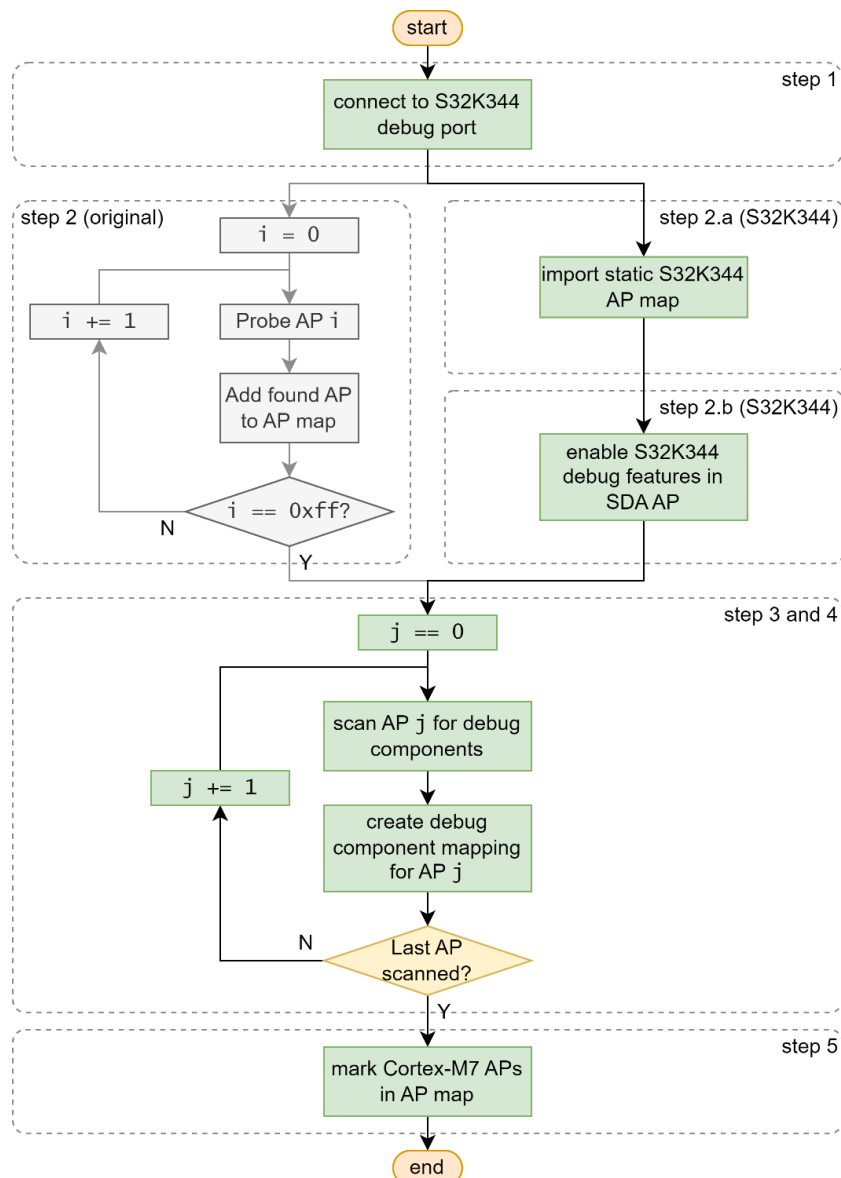


Figure 4.1: flowchart of the AP discovery sequence based on the steps listed in Section 3.3.3. For steps that were modified for compatibility with S32K344, both the original (greyed out) and the modified (in color) variants are shown.

The AP discovery sequence and the modifications done for compatibility with S32K344 are shown in Figure 4.1. Since this pyOCD configuration file only targets the S32K344, providing a static AP map was sufficient for the purposes of this project. In the future, a dynamic AP probing loop that is compatible with all S32K3 series MCUs should be implemented.

The pyOCD target configuration is not capable of gracefully handling a hardware reset of the S32K344 after application upload. Working around this requires a manual process of running the `pyocd flash` command, pressing the reset button on the development kit, and finally starting the pyOCD GDB server. Ideally after flashing an application, pyOCD would autonomously send a hardware reset command, wait for the S32K344 to restart, and reconnect. This improvement was left as a future exercise in the interest of time.

4.2 Trace bus and SWO TPIU setup

As discussed in Section 3.4.2, the trace bus of the S32K344 must be set up correctly to allow transmitting trace data from DWT or ITM over SWO. A GDB command has been implemented that performs this process and can automatically be run at GDB startup. The command performs the following sequence:

1. Write the magic value `0xc5acce55` to the `Lock Access Register` of the SWO TPIU, the CSTFs, and the ETFs to get write access to the configuration registers of all trace bus components.
2. Enable all input ports of CSTF #0 and CSTF #2 (depicted in Figure 3.6), as the CSTFs block all input ports by default.
 - a. While enabling all input ports of the CSTFs is not necessary for only using DWT/ITM, this ensures that additional trace sources (e.g. the instruction ETM) can be used without further CSTF configuration.
3. Configure ETF #0 and ETF #3 in hardware FIFO mode, since they are configured in circular buffer mode by default.
4. Configure the replicator to not allow the parallel trace TPIU to apply backpressure to the trace bus, since only the SWO TPIU is used.
5. Configure the PTA10 I/O pad of S32K344 for SWO operation: output mode, high slew rate, etc.
6. Configure the SWO TPIU by setting the pin encoding (Manchester or UART) and clock speed/data rate requested by the user.

The trace bus and SWO TPIU configuration sequence adheres to requirements R.N.7 (SWO interface must be made functional), R.N.10 (setup must be done from GDB). The full trace setup GDB command is documented in Appendix C.

4.3 orbttop DWT and ITM setup

After configuring the trace bus for forwarding trace messages off-chip through SWO, the DWT and ITM can be configured to enable periodic PC sample tracing for orbttop. This setup process has been implemented as the following sequence in a GDB initialization script:

1. Write the magic value `0xc5acce55` to the `Lock Access Register` of the M7's ITM to allow write access to its configuration registers.
2. Set the `Trace Enable` bit in the M7's `DEMCR` register to enable the DWT and ITM.
3. Configure the DWT for PC sampling:
 - a. Set PC sampling enable bit.
 - b. Enable the DWT's `CYCCNT` counter, which is used as the timing base for the PC sampling rate.
 - c. Configure the PC sampling rate based on user settings.
4. Configure the ITM to forward trace messages generated by DWT to the trace bus.

The above DWT and ITM setup process ensures that requirement R.N.1 is met and also adheres to requirement R.N.11. Like the trace bus setup command, the GDB initialization script for orbttop is documented in appendix C.

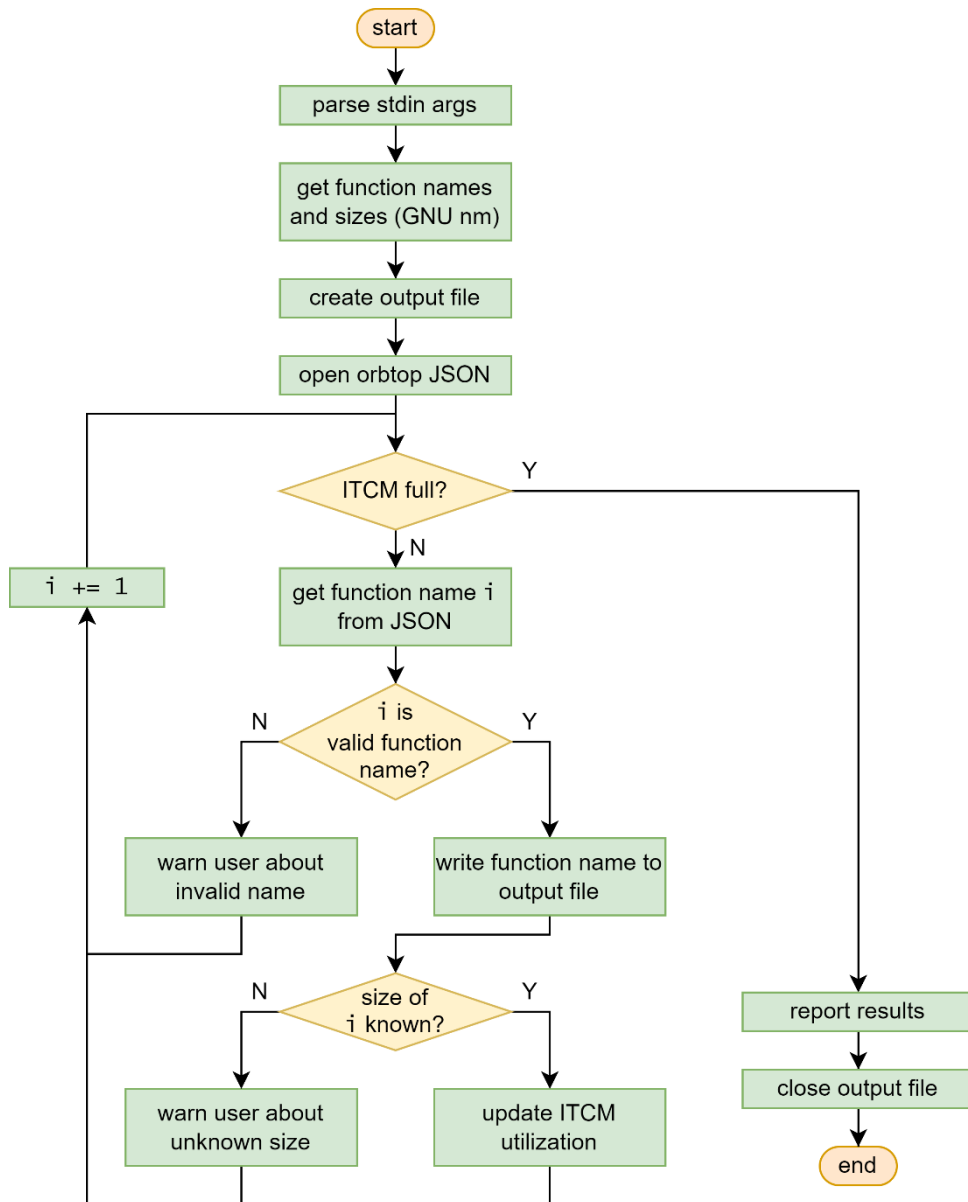


Figure 4.2: flowchart of the ITCM function mapping tool.

```

*(.text.hrt_absolute_time)
*(.text.s32k3xx_lpspi_send)
*(.text.memset)
*(.text.nxsem_post)
*(.text.sched_unlock)
*(.text.exception_common)
*(.text.nxsem_wait)
*(.text._ZN3px46logger6Logger3runEv)
*(.text._ZN7sensors22VehicleAngularVelocity3RunEv)
*(.text._ZN4uORB7Manager13orb_data_copyEPvS1_Rjb)

```

Listing 4.1: example output of the ITCM mapping tool.

4.4 ITCM mapping tool

Figure 4.2 shows the design of the ITCM mapping tool, which automatically generates a list of high-load functions (as measured by orbtap) that should be placed in ITCM.

The tool, written in Python, first parses four filename arguments specified by the user: the output file filename, the application ELF filename, the orbtob JSON filename, and (optionally) a function ignore list file. Next, the tool calls GNU nm in a subprocess to generate a list of all symbols and symbol sizes in the ELF. The output of GNU nm is filtered and parsed so that only a list of Python dictionaries remains, with each entry containing the name and size of a function in the application. With the names and sizes of all functions known, the tool starts iterating over the functions listed in the orbtob JSON file (which is sorted from highest CPU load to lowest CPU load) and writes each valid function name (i.e. the function name is not demangled and is not on the ignore list) to the output file in linker script format (Listing 4.1). The cumulative size of functions written to the output file is tracked, and the tool stops once ITCM is full. The resulting output file can be added to the user's existing linker script using the `INCLUDE` command.

The mapping tool currently only maps functions to ITCM. The script could be expanded further by also having it automatically map functions to SRAM once ITCM is full, or by having it map critical data structures to DTCM. In the interest of enabling other tracing tools with the S32K344 instead, these additions are considered as future exercises, perhaps to be implemented by future users.

The design of the mapping tool meets all requirements relevant to it: it keeps track of functions committed to ITCM and stops when ITCM is full (R.N.19), the function sizes are retrieved automatically using GNU nm (R.N.21), the user is warned about unknown function sizes and demangled function names in the orbtob JSON (R.N.20 and R.N.22), and function names on an ignore list are not added to the output file (R.N.23).

The full ITCM mapping tool source code is provided in Appendix D.

4.5 orbtob instrumentation functions and setup process

Figure 4.3 shows the common design of the entry and exit instrumentation functions used to transmit the three data values required by orbtob (listed in Section 3.6.1). After ensuring the ITM software channel is enabled, interrupts are disabled, followed by the transmission of the orbtob data. Finally, interrupts are enabled again and the function exits. Enabling and disabling of interrupts is left for the user to insert into the instrumentation functions, since the procedure for doing this safely is dependent on the application.

Before transmitting each data value to an ITM software channel, its transmission FIFO is checked for available space. These checks were implemented to ensure safe transmission of the orbtob data, but they could be unnecessary if the transmission FIFO is emptied faster than software can fill it. Removal of the free space checks would reduce CPU load imposed by the instrumentation functions; investigation into this is regarded as a future exercise.

The setup process of the ITM for use with the instrumentation functions and orbtob is similar to the DWT/ITM setup process for orbtob. A full GDB script for orbtob is shown in Appendix C, and the sources for the ITM instrumentation functions are shown in appendix E.

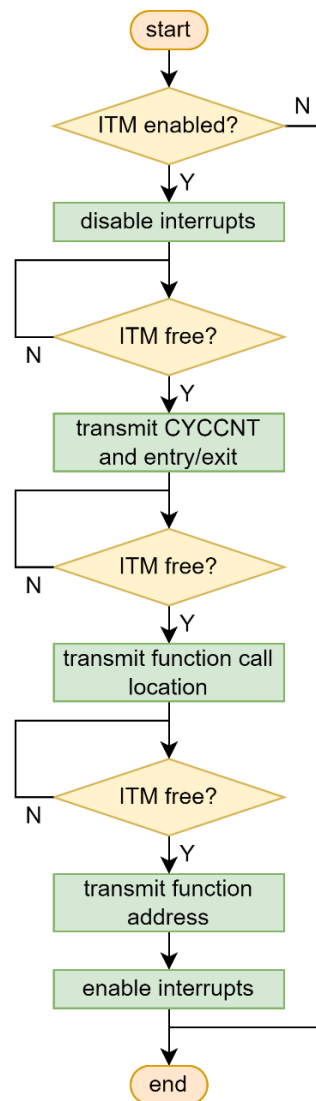


Figure 4.3: flowchart of the orbstat ITM instrumentation functions.

4.6 User guide

The design of the user guide is discussed based on its document outline, listed below. User guide sections that ensure compliance with user requirements are indicated.

1. **Introduction:** describe goal and intended use of user guide.
2. **Background information on tracing:** since the intended reader is assumed to never have used ARM tracing features before, an introductory explanation on tracing features and components is first given.
3. **Prerequisite setup:**
 - 3.1. **Assumed setup (requirement R.U.10):** describe the hardware/software setup that is assumed throughout the user guide.
 - 3.2. **Setting up pyOCD for S32K344 (requirement R.U.6)**
 - 3.3. **Setting up GDB (idem)**
 - 3.4. **Setting up Orbuculum tools (idem)**
4. **Measuring CPU load using orbttop and DWT trace**
 - 4.1. **Description**
 - 4.1.1. **Relevancy (requirement R.U.1):** explain when this tracing tool is relevant for the reader.
 - 4.1.2. **Working principle:** give a brief explanation of how the tracing tool uses the tracing features inside the S32K344, so that the reader understands the intention of the setup steps that follow.
 - 4.2. **Setup:**
 - 4.2.1. **Prerequisites (requirement R.U.3):** list any prerequisite tools that must be set up, and refer to the relevant sections in Chapter 3 of the user guide.
 - 4.2.2. **Steps (requirement R.U.2):** describe step-by-step how the tracing tool can be set up.
 - 4.2.3. **Optional steps**
 - 4.3. **Interpreting results (requirement R.U.7)**
 - 4.4. **Next steps (requirement R.U.4):** based on the result interpretation, suggest next steps for the user to undertake.
 - 4.5. **Limitations (requirement R.U.8)**
 - 4.6. **Further reading (requirement R.U.9)**
5. **Tool for mapping functions to ITCM**
 - 5.1. ...
 - 5.2. ...
6. **Generating call graphs using orbstat and ITM trace**
 - 6.1. ...
 - 6.2. ...

5 Verification

This chapter covers the verification of tracing tools, which is split into two phases. First, the tracing tools enabled/developed so far are exercised on PX4 Autopilot. This demonstrates the applicability of the tools in a real-world situation. Afterwards, compliance to the requirements listed in Chapter 2 is assessed by way of test procedures.

5.1 Test setup

The same test setup is used throughout this chapter. It consists of the NXP MR-CANHUBK3 development board, containing an S32K344 MCU, connected to the sensors and actuators of an NXP MR-Buggy3 rover. The S32K344 runs an experimental version of PX4 Autopilot that is configured to act as a vehicle management unit (VMU) for the rover (Sidrane, van der Perk, & Agar, 2023). The test setup described here does not detail how the MR-Buggy3 rover must be built and configured. This information can be found in (Galloway & Haugh, 2022).

5.1.1 Component overview

All test setup components and the connections between them are shown in Figure 5.1. The MR-CANHUBK3 board is connected to various sensors and actuators of the MR-Buggy3, which are needed to control the rover.

PX4 Autopilot can be configured using the QGroundControl application running on the host computer (Figure 5.2). This software allows configuring flight/drive paths, starting/stopping a flight/mission, calibrating sensors, etc. The communication between the S32K344 and the host computer for running QGroundControl is done using a protocol tunneled over UART, and the UART data is transmitted and received wirelessly using two HGD-TELEM433 telemetry radios. Instead of defining flight paths through QGroundControl for autonomous driving, the rover can also be driven manually. Manual control of the rover is handled by a FlySky FS-I6S remote controller, which connects to the S32K344 through an FS-iA6B receiver. The NuttX shell of the PX4 firmware can also be accessed directly through a console UART, which is connected to the host computer through an FTDI FT232R UART-USB converter.

Finally, the ORBTrace Mini is connected to the S32K344 to allow debugging and tracing from the host computer. This is the most important component of the test setup for the purposes of this project, as it facilitates using the tracing tools. All other components are instead present mostly to create a representative, real-world workload for the S32K344.

5.1.2 Conditions and scoping

The PX4 Autopilot firmware used for letting the MR-CANHUBK3 act as a VMU is built using the `nxp_mr-canhubk3_fmu` target provided in the PX4 repository. This build target is still in development and is thus unoptimized. Most notably, the firmware runs completely from program flash (van der Perk, 2022), which imposes a major performance bottleneck. The M7's instruction cache is enabled, however, to partially mitigate this.

When applying optimizations and measuring performance improvements, it is crucial that testing conditions (and thus, load imposed on the CPU) are consistent between tests. Through experimentation with orbttop, it was discovered that CPU load imposed by PX4 is consistent when arming the rover in manual driving mode, regardless of whether the rover is actually moving or not. This is likely because PX4 always runs the same data aggregation tasks and control loops, and the actual data being processed do not matter. Based on this, it was decided to carry out all verification with the rover armed in manual control mode while at standstill.

Only doing verification in manual control mode, while practical and convenient, creates a bias in testing: the tracing tools will only be exercised and tested on part of the PX4 stack, since functionality for e.g. autonomously controlling the rover during a flight mission is not used. Profiling and optimization of PX4 while doing a flight mission is regarded as a future exercise.

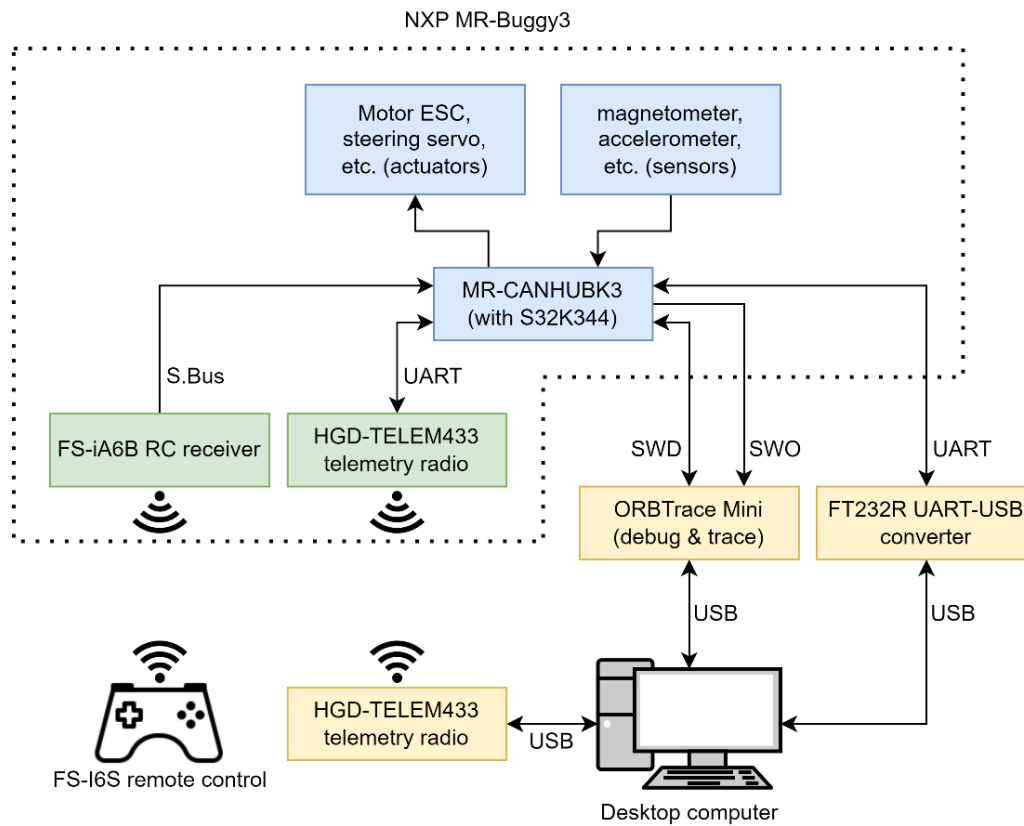


Figure 5.1: diagram of the test setup.

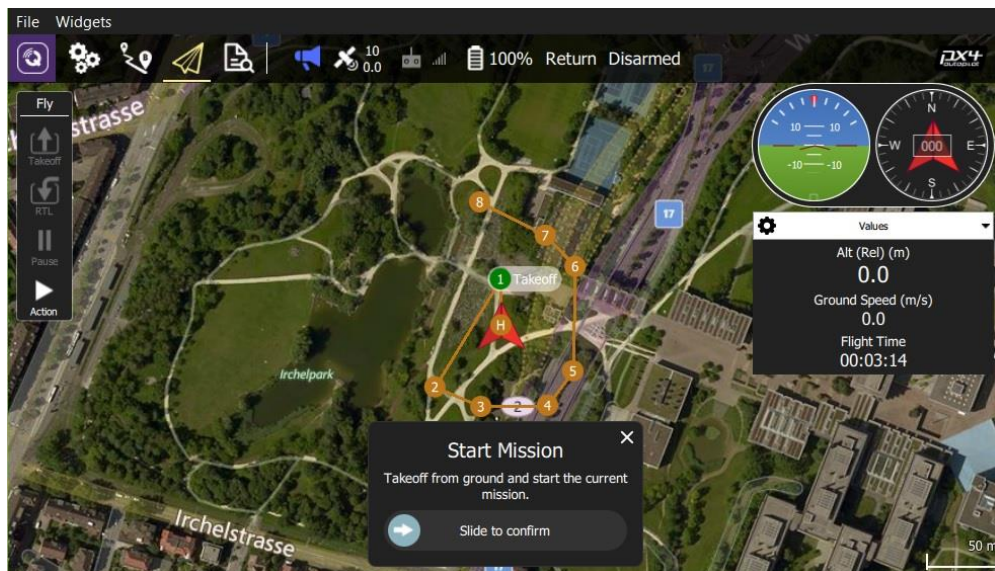


Figure 5.2: example of the QGroundControl application, which shows a flight path (shown in orange) being configured for some drone (Willee, Galvani, & aamirglb, 2021).

5.2 Optimizing PX4 Autopilot

This sections covers optimizations done to PX4 Autopilot based on insights retrieved using the three trace tools enabled/developed so far. Future optimizations done using trace tools to be developed after finalizing the graduation report will be demonstrated during the final presentation.

5.2.1 Optimization based on CPU load measurements (orbttop)

Using orbttop, an overview of the highest-load PX4 functions while the rover is armed was generated (Table 5.1), showing which functions should be optimized first to achieve the largest performance improvement (i.e. reduction in CPU load). Due to lack of experience with the PX4/NuttX codebase, optimization at this stage was done by experimenting with optimization levels provided by the compiler.

At first, optimization was attempted on a function-by-function basis by repeatedly setting an optimization level for a function, recompiling, and measuring what the change in CPU load for this function was. This strategy didn't yield clear results for most functions and was time-consuming. Instead, a different approach was used: experiment with the global optimization settings of the PX4 project, and measure the change in CPU load of all functions at once.

PX4 provides a global optimization setting in its compilation scripts: `MAX_CUSTOM_OPT_LEVEL` (Agar & K ng, 2023). This setting influences the optimization levels of all sources by either using the specified optimization level as default (in case modules/sources don't explicitly set their own optimization levels) or by overriding locally-set optimization levels if these are too high (e.g. if a source requests to be compiled at level `O3` but `MAX_CUSTOM_OPT_LEVEL` is set to `O1`, the optimization level of this source will be reduced to `O1`). By varying `MAX_CUSTOM_OPT_LEVEL`, letting orbttop measure the CPU load of all functions, and calculating the change in CPU load for each function, an overview of which optimization level is best for each function was created. Optimization levels `O2` ('optimize more', default for PX4) and `O0s` ('optimize for size'; same as `O2`, but without optimizations that often increase code size) have been compared using this strategy (Stallman & GCC Developer Community, 2023b). `MAX_CUSTOM_OPT_LEVEL = O2` resulted in ~140 sources being compiled at `O2` and ~700 at `O0s` with a resulting program flash utilization of ~1,8 MB, while `MAX_CUSTOM_OPT_LEVEL = O0s` resulted in all sources being compiled at `O0s` and flash utilization of ~1,73 MB. `MAX_CUSTOM_OPT_LEVEL = O3` was also attempted, but this resulted in non-functional firmware. The results are shown in Table 5.1.

The results show both major reductions and increases in CPU load when changing the maximum optimization level from `O2` to `O0s`, depending on function. For example:

1. `sensors::VehicleAngularVelocity::Run()`'s CPU load decreased by 60,0%. This is likely because of code size: the `O2`-optimized variant uses 608 instructions, while the `O0s`-optimized variant uses 256 instructions (a 57,9% reduction).
2. `sched_note_resume()`, `sched_note_suspend()`, and `memset()` similarly see large decreases in CPU load. The cause of this is unclear, since these functions don't experience any reduction in code size. Changes in CPU load could be caused by them not being called as often anymore by other functions.

Overall, compiling everything using `O0s` is suboptimal: the CPU's idle time decreased from 50,9% to 38,2%. It is clear, however, that using `O0s` over `O2` for some functions can result in major improvements. Using orbttop and varying global optimization options allowed finding these functions in an effective manner.

The load increase of `memset()` inspired another optimization: there could be an optimized implementation, handcrafted in ARM assembly, available for the M7 core. Coincidentally, an optimized variant of `memset()` was recently merged into the NuttX codebase (zyfeier, Xiao, & Karashchenko, 2023). Replacing the original (C-based) `memset()` with this optimized variant yielded a reduction in `memset()`'s CPU load by 25,1% and an increase in CPU idle time by 4,67% (both tested with PX4 default optimization settings and measured with orbttop). This optimization is another example of the valuable insights provided by orbttop.

While orbttop's effectiveness has been shown, it also has some shortcomings. One of these is that orbttop does not provide a way to distinguish between functions that impose high CPU load because of being called often and functions imposing high load due to being large. This information is desirable because it helps with choosing an optimization strategy. For example, a small function could be optimized by inlining it, since this removes CPU overhead associated with calling and returning the function. Inlining is unlikely to be useful for a large function, however, due to call/return overhead being comparatively small.

Table 5.1: CPU load of 15 highest-load functions in PX4 Autopilot for maximum optimization levels O2 and Os.

Function	CPU load (O2, default)	CPU load (Os)	% difference
Idle	50,87%	38,21%	-24,88%
hrt_absolute_time	5,65%	5,34%	-5,44%
s32k3xx_lpspi_send	3,41%	3,31%	-2,86%
sched_note_suspend	1,29%	0,37%	-71,40%
nxsem_post	1,08%	1,27%	17,73%
memcpy	1,06%	1,15%	8,50%
memset	1,02%	1,53%	49,70%
uORB::DeviceNode::write	0,98%	0,80%	-18,33%
sched_note_resume	0,91%	0,31%	-66,48%
void sym::PredictCovariance<float	0,86%	0,69%	-19,39%
sched_unlock	0,77%	0,84%	9,04%
uORB::Manager::orb_add_internal_subscriber	0,74%	0,59%	-20,56%
px4::logger::Logger::run	0,72%	0,58%	-19,31%
sensors::VehicleAngularVelocity::Run	0,69%	0,28%	-59,99%
exception_common	0,66%	0,81%	22,97%
nxsem_wait	0,63%	0,76%	19,84%

5.2.2 Mapping functions with ITCM mapping tool

By providing an orbttop-generated JSON file of PX4 Autopilot in an armed state to the ITCM mapping tool, a linker script-compatible list of high-load functions to map to ITCM was generated. To ensure this list of functions is linked into ITCM during the PX4 compilation process, the linker script of the `nxp_mr-canhbk3_fmu` target was modified as shown in Listing 5.1. As well, the initialization function of the application (`s32k3xx_start`) was modified to copy ITCM contents from flash to ITCM at startup, as shown in Listing 5.2. After compiling with these changes (all other compilation settings left at default), the build scripts reported the ITCM as being almost fully utilized (Listing 5.3).

Table 5.2 shows the CPU load measured by orbttop before and after applying the ITCM mapping tool. The amount of idle time increased noticeably, from 47,3% to 54,0%. Likewise, 13 of the 15 functions listed show a clear decrease in CPU load of at least 10% (relative). The two highest-load functions (`hrt_absolute_time` and `s32k3xx_lpspi_send`), however, show a difference in CPU load that can be considered within the margin of error. Unsurprisingly, both these functions access peripherals: `hrt_absolute_time` reads from an on-chip counter, while `s32k3xx_lpspi_send` transmits data using an SPI peripheral. These two functions demonstrate the disadvantage of the ITCM mapping tool assuming that all high-load functions are bottlenecked by fetch stalls.

A future optimization of the ITCM mapping tool could be to make it 'learn' from its mistakes: the tool could analyze the performance improvement measured for each function it mapped to ITCM, and remove functions that didn't experience much performance uplift. This would subsequently free up space in the ITCM for

mapping other functions into it. This iterative optimization of the ITCM mapping could compensate for incorrect assumptions done during the selection of eligible functions for mapping to ITCM.

```
...
.itcmfunc :
{
    . = ALIGN(8);
    _sitcmfuncs = ABSOLUTE(.);
    INCLUDE "itcmfunc.ld"
    . = ALIGN(8);
    _eitcmfuncs = ABSOLUTE(.);
} > itcm AT > flash

_fitcmfuncs = LOADADDR(.itcmfunc);

.text :
{
    _stext = ABSOLUTE(.);
    *(.text.__start)
    *(.text .text.*)
...

```

Listing 5.1: in the `nxp_mr-canhubk3_fm` target linker script, an extra section was added that places all functions listed in `itcmfunc.ld` into ITCM.

```
/* Copy any necessary code sections from FLASH to ITCM. The process is the
 * same as the code copying from FLASH to RAM above. */
for (src = (uint64_t *)&_fitcmfuncs, dest = (uint64_t *)&_sitcmfuncs;
     dest < (uint64_t *)&_eitcmfuncs; )
{
    *dest++ = *src++;
}

```

Listing 5.2: code snippet that copies the ITCM contents from flash to ITCM at PX4 startup.

Memory region	Used Size	Region Size	%age Used
BOOT_HEADER:	256 B	4 KB	6.25%
VECTORS:	992 B	4 KB	24.22%
flash:	1803984 B	4055039 B	44.49%
sram0_stdby:	0 GB	32 KB	0.00%
sram:	55856 B	272 KB	20.05%
itcm:	65072 B	65280 B	99.68%
dtcm:	0 GB	128 KB	0.00%

Listing 5.3: memory utilization of PX4 on S32K344 after mapping functions to ITCM.

Table 5.2: CPU load of 15 highest-load functions in PX4 Autopilot before and after applying ITCM mapping tool.

Function	CPU load (no ITCM)	CPU load (with ITCM)	% change
Idle	47,30%	54,03%	14,23%
hrt_absolute_time	5,27%	5,20%	-1,50%
s32k3xx_lpspi_send	1,94%	1,90%	-1,83%
memset	1,36%	1,20%	-12,23%
nxsem_post	1,19%	1,05%	-11,92%
uORB::DeviceNode::write	1,08%	0,90%	-16,76%
sched_unlock	0,86%	0,77%	-10,05%
void sym::PredictCovariance<float	0,86%	0,45%	-47,83%
exception_common	0,83%	0,74%	-10,80%
uORB::Manager::orb_add_internal_subscriber	0,75%	0,66%	-11,45%
nxsem_wait	0,73%	0,70%	-5,09%
px4::logger::Logger::run	0,73%	0,63%	-13,07%
math::WelfordMeanVector<float, 3u	0,72%	0,58%	-19,27%
sensors::VehicleAngularVelocity::Run	0,69%	0,53%	-22,68%
uORB::Manager::orb_data_copy	0,68%	0,56%	-18,34%
s32k3xx_dmach_xfrsetup	0,66%	0,57%	-13,10%

5.2.3 Optimization based on insights from call graph (orbstat)

Setting up PX4 with the ITM instrumentation functions necessary for orbstat immediately demonstrated a major limitation of its ITM software tracing approach: instrumenting all application functions resulted in an increase of program flash utilization from ~42% (~1,72 MB) to ~79% (~3,23 MB), resulting purely from the additional instrumentation function calls. This increase in size was exacerbated by the fact that GCC also instruments inlined functions; there is currently no compiler flag to prevent this behavior (Dubach, 2022). PX4 wouldn't run in this state, instead immediately experiencing a hard fault at startup due to a stack overflow. Due to this, only parts of PX4 were instrumented instead.

Since the NuttX OS functions (e.g. `nxsem_post`, `sched_note_resume`) impose major CPU load (Table 5.1), it was attempted to only instrument the OS portion of PX4. With this, PX4 was started successfully. Upon enabling the ITM for orbstat data retrieval, however, PX4 became unresponsive. Enabling orbttop showed why: the instrumentation functions were imposing a cumulative load of ~97,65% (Figure 5.3). Since the instrumentation functions were actually running, however, it was possible to generate a call graph of the NuttX OS. Insights from this are discussed later in this section.

```

49.15%  48005 __cyg_profile_func_exit
48.41%  47281 __cyg_profile_func_enter
0.23%   230 hrt_absolute_time
0.16%   159 hrt_tim_isr
0.15%   152 strcmp
0.12%   121 exception_common
-----
98.22%  95948 of 97655 Samples
[-S-H] Interval = 5000mS / 0 (~0 Ticks/mS)

```

Figure 5.3: orbttop output when instrumenting only the NuttX OS of PX4.

In a last attempt to use orbstat without PX4 breaking, only a single peripheral driver was instrumented: the driver for the ICM42688P gyroscope and accelerometer. PX4 stayed functional after enabling the ITM with only this driver instrumented, but the imposed CPU load by the instrumentation functions is still too high at ~74,3%.

The CPU usage of the idle task was only 4,5%. Nonetheless, it was possible to generate a call graph of the ICM42688P driver in this state; see Figure 5.4. This call graph could aid a developer in understanding how this driver works.

The call graph of the NuttX OS allowed making the IRQ stack (Figure 5.5) and OS scheduling routines visible. As an optimization exercise, the IRQ stack was mapped to ITCM, followed by filling up the remaining ITCM space using the ITCM mapping tool. This resulted in an idle CPU load of ~53,1%, which falls short of the ~54,0% idle load achieved when only mapping ITCM with the mapping tool. Nonetheless, performance could still have improved in other, less visible metrics, such as IRQ servicing latency. Based on this, a second optimization exercise was done which gave all OS routines (identified through the NuttX call graph) priority mapping to ITCM. The NuttX top application shows OS scheduling overhead. Hypothetically, this overhead should decrease when the OS routines are fetched from ITCM instead of flash and the instruction cache. The result was a reduction in OS scheduling overhead from ~2,82% to ~2,78. While the small difference in OS scheduling overhead was consistent, the difference is also too small to be considered more effective than just mapping functions to ITCM based on CPU load only.

In the above experimentation, orbstat has been demonstrated as a useful tool for making informed decisions when memory mapping functions, but the improvements in performance measured are not as obvious (e.g. reductions in total CPU load) compared to naively mapping functions using the mapping tool. orbstat is thus mainly considered useful for identifying remapping opportunities for real-time components of an application, which can still require fast execution while not imposing high CPU load. orbstat has also been demonstrated as an effective tool for showing relationships between application functions; something which is not always obvious from source code inspection. If orbstat could be implemented in a non-intrusive manner (e.g. using ETM; see Section 3.6.1), more comprehensive investigation of PX4 using call graphs could be performed.

5.3 Requirement compliance testing

In this section, compliance of the tracing tools and user guide to the requirements is tested. This is done either through manual inspection (e.g. inspecting source code, assessing the presence of some component) or through testcases (i.e. conducting a test procedure and comparing retrieved results against passing criteria). For testcases, the template shown in Table 5.3 is used.

Table 5.3: template for a testcase.

Description	<i>A short description of the testcase is given here.</i>
Requirement(s)	<i>This field summarizes which requirements are covered by this test.</i>
Test setup modifications	<i>This field describes any deviations from the default state of the test setup for the purposes of this test (e.g. using a specific power supply voltage).</i>
Passing criteria	<i>The passing criteria that have to be met for the test procedure to be deemed successful are described here.</i>
Protocol	<i>This field describes how the test must be conducted. The protocol is usually described as a list of steps that are to be executed in order.</i>
Results	<i>This field describes the test results retrieved while executing the test protocol. This can be a list of values measured or a summary of observations, depending on what is appropriate for the requirements being verified.</i>
Conclusion	<i>Based on the test results retrieved, a conclusion is made: have all requirements that this test covers been met?</i>

5.3.1 Results

The results of the manual inspections and all testcases are shown in Table 5.4, while the testcases themselves are documented in appendix A for the sake of brevity. All testcases have been predefined during the design and implementation of each instrumentation tool, while the testing has been conducted during the testing phase of the project.

Table 5.4: results of all compliance testing carried out.

Top-level			
Req. ID	Short description (MoSCoW)	Met? (Y/N)	Comment/proof
R.T.1	Trace tools for S32K344 must be developed/enabled (M)	Y	R.N.1 through R.N.3 met (must); R.N.4 through R.N.6 not met (should).
R.T.2	Trace tools must be accessible (M)	Y	User guide (Appendix F) written, all user requirements met.
R.T.3	Trace tools must be demonstrated with example application (M)	Y	Section 5.2 discusses demonstration of tools with example application.
R.T.4	Hardware needed is cost-effective (S)	Y	R.N.9 met.
R.T.5	GDB must be used as the debugger (M)	Y	GDB used for config files (Appendix C) and in testcase 5 (Appendix A).

User			
Req. ID	Short description (MoSCoW)	Met? (Y/N)	Comment/proof
R.U.1	Relevancy of tools (M)	Y	See user guide.
R.U.2	Setup of tools (M)	Y	Idem
R.U.3	Prerequisites of tools (M)	Y	Idem
R.U.4	Next steps for tools (M)	Y	Idem
R.U.5	Future additions to user guide (M)	Y	Idem
R.U.6	Setup of prerequisite tools (M)	Y	Idem
R.U.7	Interpretation of tool results (M)	Y	Idem
R.U.8	Tool limitations (M)	Y	Idem
R.U.9	Directions to further reading (M)	Y	Idem
R.U.10	Assumed HW + SW setup (M)	Y	Idem

Non-user: trace tools			
Req. ID	Short description (MoSCoW)	Met? (Y/N)	Comment/proof
R.N.1	Tool for CPU load measurement (M)	Y	See testcase 1 in Appendix A.
R.N.2	Tool for mapping ITCM (M)	Y	See testcase 2 in Appendix A.
R.N.3	Tool for generating call graphs (M)	Y	See testcase 4 in Appendix A.
R.N.4	Tool for tracing M7 execution (S)	N	Not developed yet; left as future exercise.
R.N.5	Non-intrusive orbstat with ETM (S)	N	Idem
R.N.6	Tool for profiling load transients (S)	N	Idem

Non-user: trace data transfer			
Req. ID	Short description	Met? (Y/N)	Comment/proof
R.N.7	SWO must be made functional (M)	Y	Testcases R.N.1 and R.N.3 demonstrate SWO being functional.

R.N.8	Parallel trace should be made functional (S)	N	Not enabled yet; planned for after graduation report.
R.N.9	ORBTrace Mini as TPA (S)	Y	All testcases use ORBTrace Mini as TPA.
R.N.10	Trace bus configuration with GDB (M)	Y	Appendix C shows configuration being done through GDB.
R.N.11	Trace source configuration with GDB (M)	Y	Idem

Non-user: PX4 Autopilot

Req. ID	Short description (MoSCoW)	Met? (Y/N)	Comment/proof
R.N.12	PX4 as demonstration application (M)	Y	See Section 5.1.1.
R.N.13	PX4 is run on MR-CANHUBK3 devkit, connected to MR-Buggy3 (M)	Y	Idem
R.N.14	PX4 in manual control mode (M)	Y	See Section 5.1.2.
R.N.15	PX4 in mission mode (C)	N	Idem

Non-user: pyOCD

Req. ID	Short description (MoSCoW)	Met? (Y/N)	Comment/proof
R.N.16	pyOCD is debugging toolchain (S)	Y	R.N.17 and R.N.18 met.
R.N.17	pyOCD supports S32K344 debug (S)	Y	See testcase 5 in appendix A.
R.N.18	pyOCD support S32K344 flashing (S)	Y	See testcase 6 in appendix A.

Non-user: ITCM mapping tool

Req. ID	Short description (MoSCoW)	Met? (Y/N)	Comment/proof
R.N.19	Tool does not oversubscribe ITCM (M)	Y	See testcase 3 in appendix A.
R.N.20	Notification about unknown sizes (M)	Y	Idem
R.N.21	Automatically deriving function sizes (M)	Y	See mapping tool source code (Appendix D).
R.N.22	Notification about demangled names (M)	Y	See test case3 in appendix A.
R.N.23	Ignore functions on ignore list (M)	Y	Idem
R.N.24	Map functions to SRAM when ITCM is full (C)	N	Not developed yet; left as future exercise.
R.N.25	Identify and map data structures to DTCM (C)	N	Idem

5.3.2 Evaluation

As shown above, all ‘must’ requirements have been met, 5 out of 9 ‘should’ requirements have been met and 0 ‘could’ requirements have been met. The ‘should’ requirements that were not met are only relevant to trace tools that have not been enabled yet. Any requirements not met thus didn’t impact the end products developed and tested in this report.

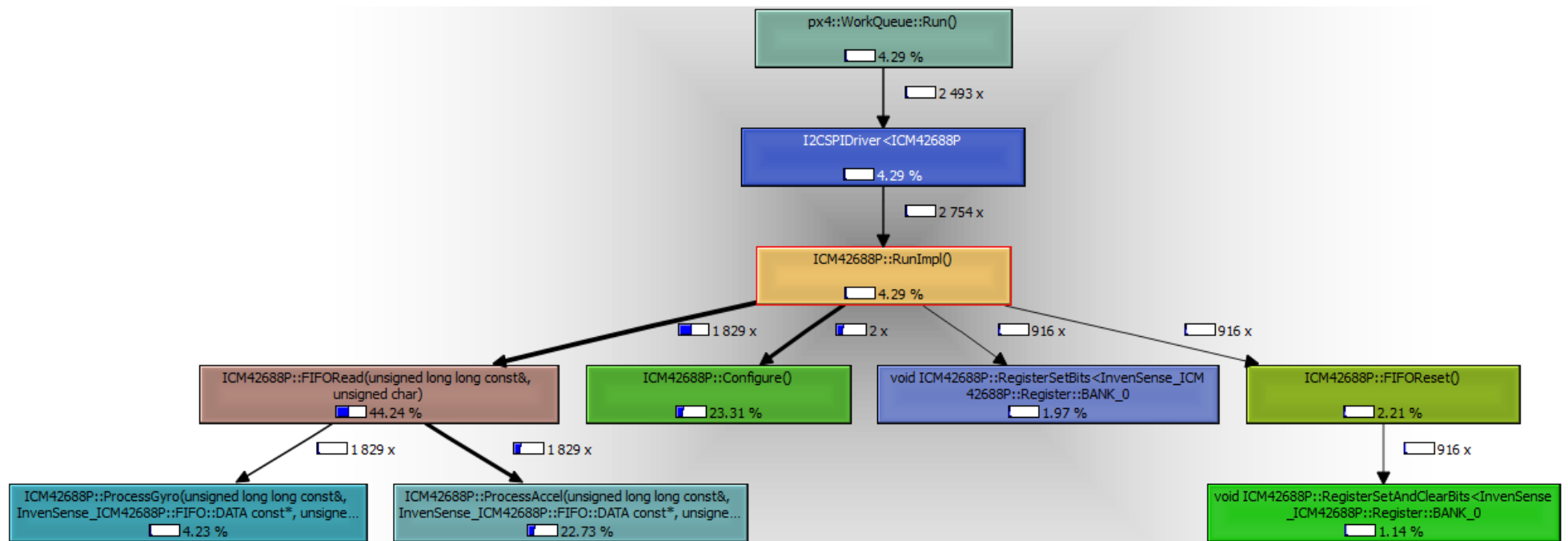


Figure 5.4: part of the call graph generated of the ICM42688P driver.

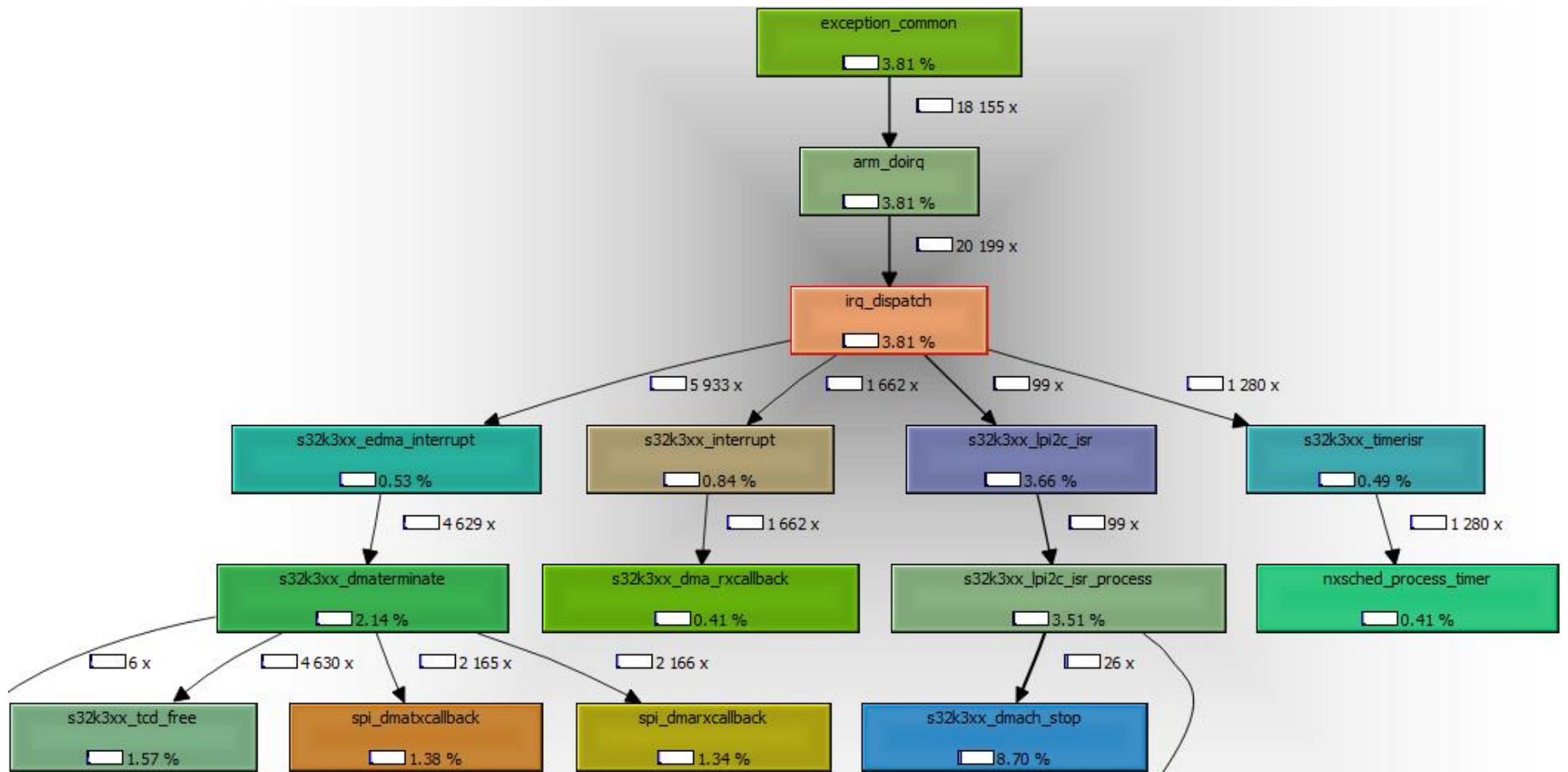


Figure 5.5: part of the IRQ stack made visible using a call graph of the NuttX OS.

6 Project management and execution

This chapter evaluates how the chosen management methodology and project schedule were executed.

6.1 Management methodology

6.1.1 Intended methodology

A modified variant of the waterfall model was chosen as the management methodology in the PID. This decision was motivated by the fact that the waterfall model starts a project with a rigorous analysis phase to clearly define the goals, requirements, and deliverables of a project, which was deemed important for an investigation-oriented project like this one. The waterfall model was modified by introducing an iterative element to the project schedule: each tracing tool was to be designed, implemented, and exercised with PX4 in succession instead of working on all tools in parallel during a design phase first, an implementation phase second, and a testing phase last.

Based on the methodology, project phases and milestones were defined and laid out in a project schedule. The project schedule was to be kept up-to-date each week during the weekly status meeting with the project supervisors. The original phasing, milestones and schedule are documented in the PID.

6.1.2 Execution of the methodology

The intended management methodology was followed diligently during execution of the graduation project. The project schedule was kept up-to-date and discussed with the project supervisors each week. This ensured that a clear overview of the project state was always available, and that possible action in response to delays was always taken on time.

The intended management methodology was also upheld when the need for redefining the project goal arose (Section 1.5): after defining the new project goal with the supervisors, research questions and requirements were updated. As well, the project schedule was updated with updated milestones in a way that still adheres to the iterative nature of the intended methodology. In essence, the process of redefining the project goal was a secondary analysis phase, which was completed before continuing with subsequent design/implementation phases. The updated project phasing and milestones is documented in Section 6.2. Section 6.3 discusses the updated project schedule and its execution.

6.2 Phasing and milestones

The list below summarizes the project phases (denoted by 'P') and milestones (denoted by 'M').

Phases/milestones that were modified or newly defined due to redefinition of the project goal are indicated using asterisks (*). Milestones in the introductory and analysis phases are not clarified further, as these were already completed as of finalization of the PID.

P.1. Introductory phase

- M.1. Student and supervisor have established what will be researched during the analysis phase.**
- M.2. First experiments with the NXP S32K344/M7 have been completed.**
- M.3. PID: assignment description and research design complete.**

P.2. Analysis phase.

- M.4. Selection of profiling features and profiling tools to use has been made (sub-question 1.a).**
- M.5. Profiling methods to develop have been defined (sub-question 1.b).**
- M.6. A representative demonstration application for testing has been selected (sub-question 1.c).**
- M.7. PID: research for sub-questions 1.a through 1.c documented.**
- M.8. PID: complete.**

P.3. First design + implementation phase. This phase focusses on developing the setup/workflow that will be used to conduct the activities of phase P.4.

- M.9. S32K344 support has been added to ORBTrace Mini (sub-question 2.a).** Since the possibility of using the ORBTrace Mini with S32K344 is uncertain, this milestone focusses on adding the necessary tooling support for this.
- M.10. DWT/ITM tracing over SWO on S32K344 is functional (sub-question 2.b).** A functioning tracing setup must be created to use DWT/ITM tracing for performance profiling. This milestone focusses on setting up DWT/ITM tracing over SWO.
- M.11. The MR-Buggy3 with PX4 is up and running.** Since the PX4 application is used to test the effectiveness of any profiling methods developed, it is crucial that this application is brought to a functional state in the test/development setup first.
- M.12. Graduation report: research for sub-questions 2.a and 2.b documented.**

P.4. * Second design + implementation phase. In this phase, three tracing tools for S32K344 that can function over the SWO interface will be enabled/developed, documented, and tested.

- M.13. * Tool for measuring CPU load per function (orbtcp) is functional, documented, and exercised with PX4 (sub-question 3.a).** This milestone focusses on enabling orbtcp on S32K344 and is achieved when three criteria have been met: the tool is functional with S32K344 (1), the tool has been exercised with PX4 as an optimization tool (2), and the tool has been documented in the user guide (3).
- M.14. Profiling method for instruction fetch stall impact is finished and documented.** This milestone was scrapped; see Section 6.3.2.
- M.15. * ITCM mapping tool is functional, documented and exercised with PX4 (sub-question 3.b).**
- M.16. * Tool for generating call graphs (orbstat) is functional, documented, and exercised with PX4 (sub-question 3.c).**
- M.17. * The user guide documenting the trace tools enabled/developed so far is complete and approved by project supervisors (sub-question 4.a).**
- M.18. * Graduation report: research for sub-questions 3.a, 3.b, 3.c, and 4.a documented.**

P.5. Testing + report phase. In this phase, all compliance testing defined during design and implementation phases is conducted. The graduation report is also finished in this phase.

- M.19. * Trace tools and user guide have been verified against requirements.**
- M.20. Graduation report: complete.**

P.6. Closing phase. This phase focusses on preparing the final presentation and enabling/developing additional trace tools that may require the high-bandwidth parallel trace interface and/or ETM trace source. Enablement/development of these tools has deliberately been moved to after completion of the graduation report, based on the decision to consider use of parallel trace and/or ETM as stretch goals during the analysis phase.

M.21. * DWT/ITM tracing over parallel trace on S32K344 is functional (sub-question 2.c).

Before attempting use of ETM instruction tracing, the parallel trace interface is made functional. Already enabled trace tools (e.g. orbttop) will be used to test this, but using parallel trace instead of SWO.

M.22. * ETM tracing over parallel trace in combination with orbmortem has been set up and demonstrated with PX4 (sub-questions 2.d and 3.d). Next, ETM instruction tracing will be set up, with orbmortem (tool for reconstructing CPU execution using ETM tracing) used as the trace tool to verify ETM tracing with.

M.23. * orbstat has been extended with the ability to generate call graphs from ETM tracing (sub-question 3.e). With ETM tracing functional, research and development into enabling orbstat in a non-intrusive manner can begin.

M.24. * A tool for investigating CPU load transients using trace data has been developed and exercised with PX4 (sub-question 3.f).

M.25. Final presentation: complete.

6.3 Execution of the project schedule

This section describes the execution of the graduation project with regards to the project schedule (visualized using the Gantt chart in Table 6.1 and Table 6.2). The execution of each phase is briefly discussed, with emphasis placed on impediments/delays that occurred and how they were resolved.

6.3.1 First design + implementation phase

As shown in table Table 6.1, this phase started with adding S32K344 support to ORBTrace Mini (M.9), which turned into adding S32K344 support to pyOCD. Execution for this milestone went according to schedule, and was a process of identifying, understanding, and resolving issues with the pyOCD initialization sequence on S32K344. A pyOCD target configuration for S32K344 was created as a deliverable.

Next, DWT tracing was enabled over SWO (M.10). For this, the trace bus components, SWO TPIU, DWT, and ITM were researched and experimented with through trial-and-error until the first DWT trace message was retrieved from the SWO pin. This milestone was completed one week ahead of schedule, and a trace bus + SWO TPIU initialization function was a resulting deliverable.

To finish the test + development setup for trace tools, an MR-Buggy3 with MR-CANHUBK3 as VMU was assembled (M.11). PX4 was subsequently compiled and uploaded to S32K344, followed by configuration of PX4 through QGroundControl for correct vehicle operation in manual control mode. Execution went as scheduled; the resulting deliverable was a functional and representative test setup.

Finally, the research done during this phase was documented in the graduation report and testcases were written in advance for relevant requirements (M.12). Start of execution was slightly delayed, but this didn't impact the project schedule.

6.3.2 Second design + implementation phase

After bringing DWT trace over SWO in a functional state (M.10), setup of orbttop with S32K344 (M.13) was a quick process of enabling the DWT PC sampling feature. Nonetheless, completion of this milestone occurred

one week later than planned due to milestone M.11 requiring more time than expected in week 8. Deliverables for milestone M.13 were a chapter on orbtcp in the user guide, a GDB initialization script, and a testcase written in advance.

The need to redefine the project goal arose while working on milestone M.14: based on research, it was soon concluded that implementing an effective profiling method for instruction fetch stalls was impractical. The red line in the Gantt chart indicates when the project goal was redefined, and milestone M.14 was scrapped.

After project goal and schedule redefinition, design and implementation of the ITCM mapping tool started (M.15). This went according to schedule: the tool was implemented in week 10, and it was documented and exercised with PX4 in week 11. Deliverables were the ITCM mapping tool itself, an accompanying chapter in the user guide, and testcases for the mapping tool written in advance.

Enablement of orbstat on S32K344 (M.16) started in week 11, but was not formally completed until week 13 due to delays in writing a user guide chapter for it. Exercising orbstat with PX4 and documenting the results was completed on time by week 12, however. Deliverables were a user guide chapter for orbstat, ITM instrumentation function templates, a GDB initialization script, and a testcase written in advance.

Like milestone M.16, completion of the user guide (M.17) experienced one week of delay. Both of these delays with the user guide were caused due to priority being given to finishing the research documentation in the graduation report (M.18). Milestone M.18 was completed according to schedule due to this.

6.3.3 Testing + report phase

In this phase, all testing through manual inspection and conducting the testcases written in prior phases was done (M.19), followed by completion of the graduation report (M.20). This phase coincided with completing delayed milestones M.16 and M.17.

6.3.4 Closing phase

The closing phase is planned for execution in weeks 14 through 20. The technical activities/milestones for this phase are discussed further in Section 7.1. A notable addition to this phase is the presence of 'free space'. The activities to be conducted during this free space are still left to be decided, and could encompass tasks like upstreaming e.g. configuration files made back into the pyOCD or Orbuculum projects, or developing further tracing tools.

6.4 Evaluation

Section 6.3 demonstrates that the project schedule was (so far) executed on as intended. While there were some delays, these were minor in nature and never resulted in further delays for other milestones. A major factor in being able to adhere to the schedule effectively was the weekly monitoring and updating of the schedule. By doing so, the need to make major changes to the project goal (and by extension, the project schedule) was identified early. Were the project goal not redefined on time, a major delay could have been encountered due to attempting to achieve milestones that were not feasible to begin with.

Based on the above, it is concluded that this project was managed effectively. The intent is continue managing the project in this manner during the closing phase.

Table 6.1: Gantt chart of the project as executed, upper half.

Phase	Milestone	Week number																			
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
P.1: introductory phase	M.1: student and supervisor have established what will be researched during the analysis phase																				
	M.2: first experiments with the NXP S32K344/M7 have been completed																				
	M.3: PID: assignment description and research design complete																				
P.2: Analysis phase	M.4: selection of profiling features and profiling tools to use has been made (sub-question 1.a)																				
	M.5: profiling methods to develop have been defined (sub-question 1.b)																				
	M.6: a representative demonstration application for testing has been selected (sub-question 1.c)																				
	M.7: PID: research for sub-questions 1.a through 1.c documented																				
	M.8: PID: complete																				
P.3: first design + implementation phase	M.9: S32K344 support has been added to ORBTrace Mini (sub-question 2.a)																				
	M.10: DWT/ITM tracing over SWO on S32K344 is functional (sub-question 2.b)																				
	M.11: the MR-Buggy3 with PX4 is up and running																				
	M.12: graduation report: research for sub-questions 2.a and 2.b documented																				
P.4: second design + implementation phase	M.13: tool for measuring CPU load per function (orbtcp) is functional, documented, and exercised with PX4 (sub-question 3.a)																				
	M.14: profiling method for instruction fetch stall impact is finished, documented, and exercised with PX4																				
	* M.15: ITCM mapping tool is functional, documented, and exercised with PX4 (sub-question 3.b)																				

Table 6.2: Gantt chart of the project as executed, lower half.

Phase	Milestone	Week number																			
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
P.4: second design + implementation phase	* M.16: tool for generating a call graph of an application running on S32K344 is approved, documented, and exercised with PX4 (sub-question 3.c)																				
	* M.17: the user guide documenting the trace tools enabled/developed so far is complete and approved by project supervisors (sub-question 4.a)																				
	* M.18: graduation report: research for sub-questions 3.a, 3.b, 3.c, and 4.a documented																				
P.5: testing + report phase	* M.19: trace tools and user guide have been verified against requirements																				
	* M.20: graduation report: complete																				
P.6: closing phase	* M.21: DWT/ITM tracing over parallel trace on S32K344 is functional (sub-question 2.c)																				
	* M.22: ETM tracing over parallel trace in combination with orbmortem has been set up and demonstrated with PX4 (sub-questions 2.d and 3.d)																				
	* M.23: orbstat has been extended with the ability to generate call graphs from ETM tracing (sub-question 3.e)																				
	* M.24: a tool for investigating CPU load transients using trace data has been developed and exercised with PX4 (sub-question 3.f)																				
	* M.25: final presentation: complete																				
	Free space																				

planned	in progress	done	scrapped	delay
---------	-------------	------	----------	-------

Figure 6.1: legend for the Gantt chart in Table 6.1 and Table 6.2.

7 Conclusion

This graduation project focused on making the tracing features provided by the NXP S32K344 accessible to other software developers for the purpose of profiling, optimizing, and debugging their applications. To this end, a range of research sub-questions were defined which paved the way towards understanding and enabling the S32K344's tracing features, as well as developing new tools or enabling existing tools (from the Orbuculum project) for making the S32K344's tracing features usable in various ways without requiring intricate knowledge of the trace subsystem's inner workings. Researching these sub-questions and subsequent design and implementation work have resulted in the following deliverables:

1. pyOCD was made compatible with S32K344 by creating S32K344-specific target configuration. This target configuration supports using the M7 core's common debug features and programming on-chip flash.
2. GDB scripting was written for automatically setting up the S32K344 trace bus and SWO TPIU, allowing trace messages generated by DWT or ITM to be transmitted off-chip through SWO.
3. Three tracing tools that can function with the bandwidth available through SWO were enabled on S32K344:
 - a. orbttop (from the Orbuculum suite), allowing reconstruction of CPU load through DWT tracing.
 - b. ITCM mapping tool, allowing automatic generation of linker script-compatible lists of functions which should be mapped into the S32K344's ITCM.
 - c. orbstat (from the Orbuculum suite), allowing generation of call graphs through ITM software tracing.
4. All tracing tools enabled/developed so far were exercised with PX4 Autopilot, an application representative of NXP Mobile Robotics' activities. Based on tracing tool insights, optimizations were done to more effectively map PX4 to the S32K344's hardware. In the best case, an increase in idle CPU time from ~47% to ~54% was observed as a result of optimizations.
5. A user guide for the three tracing tools enabled/developed so far was written, allowing other developers new to ARM tracing features to use these tools and become familiar with tracing in general.
6. Finally, all deliverables were tested against requirements and found to be compliant.

7.1 Future work

As shown in Table 6.2, further activities are planned for after finalization of this report. These additional activities mainly encompass enabling/developing additional tracing tools which are dependent on the parallel trace interface (as opposed to SWO) being functional, in turn allowing exploitation of the high-bandwidth ETM trace source for full instruction execution tracing of the M7 core. This allows the following:

1. With parallel trace functional, the DWT can be configured to sample the PC at a higher rate. This in turn allows orbttop to determine CPU load accurately across smaller measurement intervals and allows it to measure CPU load of short/fast functions.
2. With ETM, orbmortem can be enabled on S32K344. This would be a very effective tool for a developer when determining the root-cause of system crashes, since orbmortem tracks and makes visible dynamic execution of the program up until the occurrence of a halt.
3. With ETM, orbstat could be reimplemented to generate call graphs without requiring modification of the application with intensive instrumentation functions that programmatically write to the ITM software channels.

4. A tool to investigate undesired transients/spikes in CPU load (e.g. in response to certain events) could be developed. Whether this would use ETM or DWT trace data (or both) is yet to be determined. A possible implementation could be to use the DWT to sample the PC at a high rate (over the parallel trace interface), allowing the creation of a CPU load graph that shows the duration and intensity of the CPU load spike. This is akin to using orbttop with a very short measurement interval, with each measurement interval representing a point on the horizontal axis of the graph.
5. Using either high-speed DWT or ETM tracing, profiling methods for microarchitectural CPU behaviors akin to the ones that were originally intended to be developed during the graduation project could still be enabled. Research has to be conducted into what profiling methods are possible with the data provided by the trace sources.

Besides the milestones planned for the closing phase, the following miscellaneous activities could be conducted in parallel during the closing phase or during the 'free space' allocated in the schedule:

1. Set up flight missions in PX4 Autopilot and exercise the tracing tools on it in this state. Enabling flight missions will likely result in other PX4 subsystems becoming active, resulting in different insights from the tracing tools.
2. Develop the ITCM mapping tool further, e.g. by adding the following:
 - a. Let the tool automatically continue mapping functions to SRAM once ITCM is full.
 - b. Let the tool automatically identify and map important program data structures to DTCM.
 - c. Let the tool determine suitable functions for remapping in a more effective manner by providing it information on instruction stalls occurred in each function. This could potentially be measured based on ETM trace data.
3. pyOCD:
 - a. Fix the way pyOCD handles an S32K344 hardware reset.
 - b. Implement a dynamic AP probing loop compatible with S32K3 series MCUs, so that the pyOCD target configuration is not only limited to S32K344.
 - c. Upstream the S32K344 target configuration back into the pyOCD project.
4. Orbuculum:
 - a. Upstream the trace initialization sequence for S32K344 into the Orbuculum project.

Terms and abbreviations

Terms and abbreviations commonly used throughout this document are clarified below.

Abbreviation/term	Meaning
AP	Access Port; a component of an ARM CoreSight-based debug subsystem that allows access to a local bus hosting various types of debug components.
cache	Small, fast memory that stores (i.e. 'caches') a subset of a larger, slower memory to improve access time of memory locations that are accessed often. Caches are managed by a hardware-implemented algorithm and cannot be addressed directly.
CPU	Central Processing Unit; used synonymously with 'processor', 'processor core', and 'core'.
DAP bus	Debug Access Port bus; the bus of an ARM CoreSight-based debug subsystem that connects the top-level debug port to all APs.
DTCM	Data Tightly Coupled Memory
DWT	Data Watchpoint and Trace unit; allows monitoring specific hardware structures and generates trace messages based on configured conditions.
ETM	Embedded Trace Macrocell; produces trace data that allows reconstructing exact CPU behavior.
flash	A non-volatile/persistent memory technology. Is available on the NXP S32K344's memory bus as the largest but slowest tier of memory.
ITCM	Instruction Tightly Coupled Memory
ITM	Instrumentation Trace Macrocell; generates software-generated trace messages and forwards trace messages generated by the DWT.
M4	Shorthand for referring to the ARM Cortex-M4 processor
M7	Shorthand for referring to the ARM Cortex-M7 processor
MCU	Microcontroller Unit
parallel trace	High-speed interface between TPIU and TPA, utilizing four DDR data pins and a dedicated clock pin.
PC	Program counter; a register in a CPU that stores the current (or next, depending on instruction set architecture) instruction being executed.
SRAM	Static Random Access Memory. Although conventionally used to refer to a specific memory technology, it refers to a tier of memory available in the NXP S32K344 in this report.
SWO	Single Wire Output; low-speed interface between TPIU and TPA over a single pin.
TCM	Tightly Coupled Memory; a low-latency, high-bandwidth memory offered by the ARM Cortex-M7.
TPA	Trace Port Analyzer; receives trace stream from TPIU and forwards it to a PC for further analysis.
TPIU	Trace Port Interface Unit; retrieves merged stream of trace data and outputs it off-chip through either SWO or parallel trace.
VMU	Vehicle Management Unit; the hardware implementing the 'brains' of an autonomous vehicle.

References

- Agar, D., & Küng, B. (2023, January 17). *CMakeLists.txt*. Retrieved from github.com:
<https://github.com/PX4/PX4-Autopilot/blob/main/CMakeLists.txt>
- ARM. (2010, December 10). *CoreSight Trace Memory Controller Technical Reference Manual*. Retrieved from developer.arm.com: <https://developer.arm.com/documentation/ddi0461/b>
- ARM. (2015, March 16). *ARM CoreSight SoC-400 Technical Reference Manual*. Retrieved from developer.arm.com: <https://developer.arm.com/documentation/ddi0480/latest/>
- ARM. (2018, November 15). *ARM Cortex-M7 Processor - Technical Reference Manual*. Retrieved from developer.arm.com: <https://developer.arm.com/documentation/ddi0489/latest/>
- ARM. (2021, February 15). *ARMv7-M Architecture Reference Manual*. Retrieved from developer.arm.com: <https://developer.arm.com/documentation/ddi0403/latest/>
- ARM. (2022). *Arm Cortex-M Processor Comparison Table*. Retrieved from developer.arm.com: <https://developer.arm.com/documentation/102787/latest>
- ARM. (n.d.). *CMSIS-Packs*. Retrieved from developer.arm.com: <https://developer.arm.com/tools-and-software/embedded/cmsis/cmsis-packs>
- Baldassari, F. (2019, June 25). *From Zero to main(): Demystifying Firmware Linker Scripts*. Retrieved from interrupt.memfault.com: <https://interrupt.memfault.com/blog/how-to-write-linker-scripts-for-firmware>
- Brush, K. (2023, March 29). *MoSCoW method*. Retrieved from www.techtarget.com:
<https://www.techtarget.com/searchsoftwarequality/definition/MoSCoW-method>
- Chamberlain, S., & Taylor, I. L. (2023). *The GNU Linker*. Retrieved from sourceware.org:
<https://sourceware.org/binutils/docs/ld.pdf>
- Dubach, J. (2022, April 13). *Request an option to make -finstrument-functions not apply to inlined function calls*. Retrieved from gcc.gnu.org: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=28205
- Galloway, I., & Haugh, L. (2022, December 22). *MR-Buggy3 build guide*. Retrieved from nxp.gitbook.io:
<https://nxp.gitbook.io/nxp-cup/mr-buggy3-developer-guide/mr-buggy3-build-guide>
- IBM. (2013, August 11). *Exploring the DWARF debug format information*. Retrieved from developer.ibm.com:
<https://developer.ibm.com/articles/au-dwarf-debug-format/>
- IBM. (2023, April 11). *Name mangling (C++ only)*. Retrieved from www.ibm.com:
<https://www.ibm.com/docs/en/i/7.3?topic=linkage-name-mangling-c-only>
- Marples, D. (2022a, March 21). *SWO – Code Instrumentation*. Retrieved from orbcodes.org:
<https://orbcodes.org/orbuculum/swo-code-instrumentation/>
- Marples, D. (2022b, July 18). *Using Single Wire Output when parallel trace isn't available*. Retrieved from orbcodes.org: <https://orbcodes.org/orbtrace/using-single-wire-output-when-parallel-trace-isnt-available/>
- Marples, D., Bernack, J., Palsson, K., Nowak, M., Huesmann, P., Martens, A., & bissonex. (2022, December 05). *README.md*. Retrieved from github.com:
<https://github.com/orbcodes/orbuculum/blob/main/README.md>
- Marples, D., Palsson, K., Girault, D., & Izptr. (2022, December 01). *gdbtrace.init*. Retrieved from github.com:
<https://github.com/orbcodes/orbuculum/blob/main/Support/gdbtrace.init>
- NXP Semiconductors. (2017, April). *Kinetis K66 Sub-Family*. Retrieved from www.nxp.com:
<https://www.nxp.com/docs/en/data-sheet/K66P144M180SF5V2.pdf>
- NXP Semiconductors. (2018, August). *K66 Sub-Family Reference Manual*. Retrieved from www.nxp.com:
<https://www.nxp.com/webapp/Download?colCode=K66P144M180SF5RMV2>
- NXP Semiconductors. (2022a, July). *S32K3 Memories Guide*. Retrieved from www.nxp.com:
<https://www.nxp.com/webapp/Download?colCode=AN13388>

NXP Semiconductors. (2022b, November). *S32K3xx Data Sheet*. Retrieved from [www.nxp.com](https://www.nxp.com/docs/en/data-sheet/S32K3xxDS.pdf):
<https://www.nxp.com/docs/en/data-sheet/S32K3xxDS.pdf>

NXP Semiconductors. (2022c, September). *S32K3xx Reference Manual*. Retrieved from [www.nxp.com](https://www.nxp.com/webapp/Download?colCode=S32K3XXRM):
<https://www.nxp.com/webapp/Download?colCode=S32K3XXRM>

Orbcode. (2021, November). *ORBTrace Mini - A low cost, open source, Debug and Parallel Trace interface for CORTEX-M class embedded micro controllers*. Retrieved from orbcode.org:
<https://orbcode.org/orbtrace-mini/>

Orbcode. (2022). *Orbuculum*. Retrieved from [orbcode.org](https://orbcode.org/orbuculum/): <https://orbcode.org/orbuculum/>

PEmicro. (n.d.). *NXP Automotive S32K344 is supported!* Retrieved from www.pemicro.com:
https://www.pemicro.com/arm/device_support/NXP_Automotive/S32K3xx/S32K344/index.cfm#device

Pesch, R. H., Osier, J. M., & Cygnus Support. (2023, January). *The GNU Binary Utilities*. Retrieved from [sourceware.org](https://sourceware.org/binutils/docs/binutils.pdf): <https://sourceware.org/binutils/docs/binutils.pdf>

Pyeatt, L. D. (2016). The Linker Script. In L. D. Pyeatt, *Modern Assembly Language Programming with the ARM Processor* (p. 504). Newnes.

Reed, C. (2020, September 20). *Adding a new built-in target*. Retrieved from github.com:
https://github.com/pyocd/pyOCD/blob/main/docs/adding_new_targets.md

Reed, C. (2022, May 09). *discovery.py*. Retrieved from github.com:
<https://github.com/pyocd/pyOCD/blob/main/pyocd/coresight/discovery.py>

SEGGER. (2023, March 16). *NXP S32K3xx*. Retrieved from wiki.segger.com:
https://wiki.segger.com/NXP_S32K3xx

Sidrane, D., van der Perk, P., & Agar, D. (2023, February 23). *mr-canhubk3*. Retrieved from github.com:
<https://github.com/PX4/PX4-Autopilot/tree/main/boards/nxp/mr-canhubk3>

Stallman, R., & GCC Developer Community. (2023b). *Using the GNU Compiler Collection*. Retrieved from gcc.gnu.org: <https://gcc.gnu.org/onlinedocs/gcc-13.1.0/gcc.pdf>

Stallman, R., Pesch, R., & Shebs, S. (2023a, February 22). *Debugging Remote Programs*. Retrieved from sourceware.org: <https://sourceware.org/gdb/current/onlinedocs/gdb#Remote-Debugging>

van der Perk, P. (2022, December 5). *script.ld*. Retrieved from github.com: <https://github.com/PX4/PX4-Autopilot/blob/94fb334d8f6625a00ad28fe456c98c11ddc315c8/boards/nxp/mr-canhubk3/nuttx-config/scripts/script.ld#L62>

Willee, H., Galvani, W., & aamirglb. (2021, September 22). *Fly View*. Retrieved from github.com:
<https://github.com/mavlink/qgc-user-guide/blob/master/en/FlyView/FlyView.md>

zyfeier, Xiao, X., & Karashchenko, P. (2023, January 10). *Libc: Add more libc function for arm and riscv*. Retrieved from github.com: <https://github.com/apache/nuttx/pull/8042>

Appendix A: testcases

This appendix contains testcases for requirements that could be tested through the process of manual inspection. Refer to Section 5.3 for more information, such as the template used for each testcase.

Test 1: measurement of CPU load with orbtcp

Description	This test verifies if measuring CPU load on S32K344 with DWT/ITM PC sampling and orbtcp is functional.
Requirement	R.N.1
Test setup modifications	N/A
Passing criteria	<ul style="list-style-type: none">The idle CPU percentages measured by orbtcp and NuttX top while the rover is unarmed differ by no more than $\pm 5\%$. This margin for error accounts for different methods used by both tools.When the rover is armed (i.e. flight/drive mode is enabled), functions visibly start using more CPU time in an expected manner.
Protocol	<ol style="list-style-type: none">Flash the PX4 application to S32K344 and enable all necessary tracing features for periodic PC sampling. Refer to Chapter 4 of the user guide (Appendix F) for instructions.Start orbtcp with the <code>-l 10000</code> argument to average CPU load over a period of 10 seconds.Open the NuttX shell and start its top application: <code>nsh> top</code>Identify and record idle CPU percentages shown by orbtcp (denoted by <code>** Sleeping **</code>) and NuttX top.Arm the MR-Buggy3 using the remote control. Identify resulting changes in CPU load of functions (in orbtcp) and record the names of these functions.
Results	<p>Idle percentages measured while rover unarmed:</p> <p>PX4 top: ~49,77%</p> <p>orbtcp: ~46,47%</p> <p>Changes in CPU load after arming the rover:</p> <p>Idle percentage: ~46,47% (unarmed) → ~41,33% (armed)</p> <p><code>hrt_absolute_time()</code>: ~5,31% (unarmed) → ~6,22% (armed)</p> <p><code>s32k3xx_lpspi_send()</code>: 0% (unarmed) → ~3,09% (armed)</p>
Conclusion	Test successful: passing criteria met.

Test 2: ITCM mapping tool functional compliance

Description	This test verifies if the ITCM mapping tool correctly maps functions to ITCM and whether its output file is correct.																																
Requirement	R.N.2																																
Test setup modifications	N/A																																
Passing criteria	<ul style="list-style-type: none">• GNU LD does not show any errors when including the output file of the ITCM mapping tool.• The memory usage report shows ITCM being (nearly) fully utilized.• The mapping report generated by GNU LD shows all functions listed in the ITCM output file as being remapped to ITCM.																																
Protocol	<ol style="list-style-type: none">1. Flash the PX4 application to S32K344 and enable the necessary tracing features for periodic PC sampling. Refer to Chapter 4 of the user guide (Appendix F) for instructions.2. Arm the MR-Buggy3 using the remote control.3. Generate a JSON file of the CPU load (refer to Chapter 5 of the user guide).4. Run the ITCM mapping tool using the PX4 application binary and the generated orbtcp JSON file.5. Make the necessary changes to the PX4 MR-CANHUBK3 linker script to map all functions in the ITCM output file into the ITCM memory range (refer to Chapter 5 of the user guide).6. Rebuild the nxp_mr-canhubk3_fmu PX4 target.7. Take note of the memory usage report and the GND LD mapping report upon build completion.																																
Results	<p>GNU LD encounters errors?</p> <p>No error shown in console output.</p> <p>Memory usage:</p> <table><tr><td>Memory region</td><td>Used Size</td><td>Region Size</td><td>%age Used</td></tr><tr><td>BOOT_HEADER:</td><td>256 B</td><td>4 KB</td><td>6.25%</td></tr><tr><td>VECTORS:</td><td>992 B</td><td>4 KB</td><td>24.22%</td></tr><tr><td>flash:</td><td>1803984 B</td><td>4055039 B</td><td>44.49%</td></tr><tr><td>sram0_stdby:</td><td>0 GB</td><td>32 KB</td><td>0.00%</td></tr><tr><td>sram:</td><td>55856 B</td><td>272 KB</td><td>20.05%</td></tr><tr><td>itcm:</td><td>64624 B</td><td>65280 B</td><td>99.00%</td></tr><tr><td>dtdcm:</td><td>0 GB</td><td>128 KB</td><td>0.00%</td></tr></table>	Memory region	Used Size	Region Size	%age Used	BOOT_HEADER:	256 B	4 KB	6.25%	VECTORS:	992 B	4 KB	24.22%	flash:	1803984 B	4055039 B	44.49%	sram0_stdby:	0 GB	32 KB	0.00%	sram:	55856 B	272 KB	20.05%	itcm:	64624 B	65280 B	99.00%	dtdcm:	0 GB	128 KB	0.00%
Memory region	Used Size	Region Size	%age Used																														
BOOT_HEADER:	256 B	4 KB	6.25%																														
VECTORS:	992 B	4 KB	24.22%																														
flash:	1803984 B	4055039 B	44.49%																														
sram0_stdby:	0 GB	32 KB	0.00%																														
sram:	55856 B	272 KB	20.05%																														
itcm:	64624 B	65280 B	99.00%																														
dtdcm:	0 GB	128 KB	0.00%																														

	Map file (snippet): <pre> .itcmfunc 0x0000000000000100 0xfc70 load address 0x00000000 0x0000000000000100 . = ALIGN (0x8) 0x0000000000000100 _sitcmfuncs = ABSO *(.text.hrt_absolute_time) .text.hrt_absolute_time 0x0000000000000100 0x4c platforms/nuttx/src/px 0x0000000000000100 hrt_absolute_time *(.text.s32k3xx_lpspi_send) .text.s32k3xx_lpspi_send 0x000000000000014c 0x28 NuttX/nuttx/arch/arm/s *(.text.memset) .text.memset 0x0000000000000174 0x8e NuttX/nuttx/libs/libc/ 0x0000000000000174 memset ... </pre>
Conclusion	Test successful: no linker errors, ITCM is 99% utilized and snippet of mapping report shows functions being placed at ITCM addresses (starting at 0x100 of the ITCM memory range).

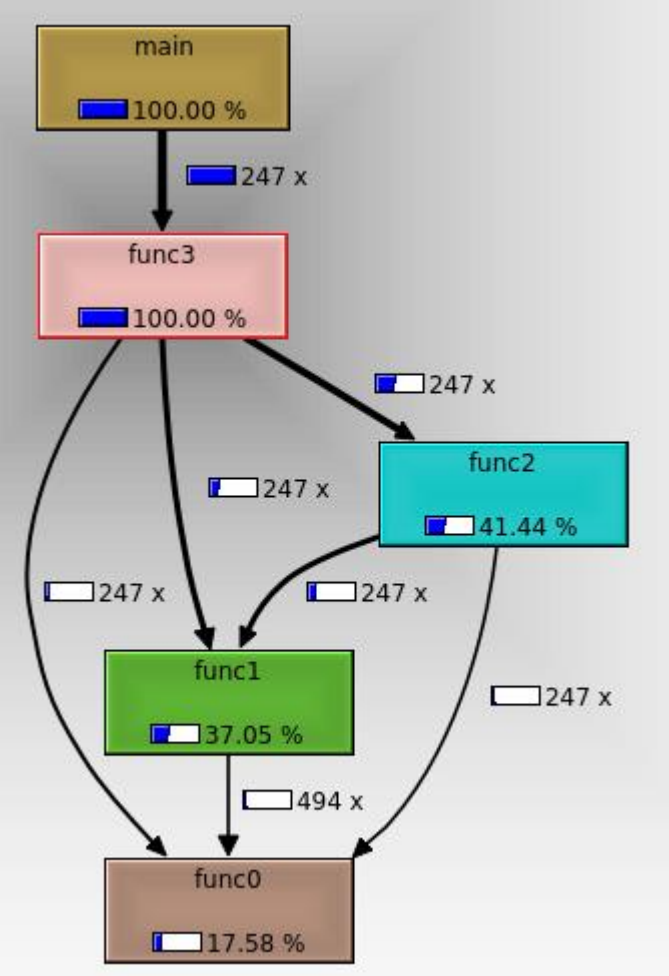
Test 3: ITCM mapping tool feature compliance

Description	This test verifies if the ITCM mapping tool correctly implements various required features.
Requirement	R.N.19, R.N.20, R.N.22, R.N.23
Test setup modifications	N/A
Passing criteria	<ul style="list-style-type: none"> • A warning is given to the user for a function that does not have a known size in GNU nm. • A warning is given to the user about a demangled function name; this function does not appear in the output file. • Functions placed on the ignore list do not appear in the output file. • The ITCM is not oversubscribed when rebuilding PX4 with the mapping file created during this testcase.

Protocol	<div><div><div>1. Execute the functional testcase for the ITCM mapping tool (Test 2: ITCM mapping tool functional compliance) and keep the orbttop JSON file and ITCM mapping tool output file generated during that process.</div><div>2. Create an ignore list for the ITCM mapping tool that lists some of the highest-load functions in the orbttop JSON file. Refer to Chapter 5 of the user guide (Appendix F) for more info.</div><div>3. Ensure there is a high-load function in the orbttop JSON for which GNU nm cannot determine the size. This can be determined by manually running GNU nm on the PX4 application binary.</div><div>4. Manually add a demangled function name among the highest-load functions in the orbttop JSON file.</div><div>5. Run the ITCM mapping tool with the modified orbttop JSON file, the PX4 application binary, and the ignore list specified as arguments.</div><div>6. Inspect the output ITCM mapping file and ensure that it meets the passing criteria.</div><div>7. Place any for which the size cannot be determined in the ignore list, run the ITCM mapping tool again and rebuild PX4 with the new ITCM mapping file. Assess whether the ITCM was oversubscribed based on the memory usage report.</div></div></div>																																
Results	<div><div><div><div>ITCM mapping tool warnings/notes shown:</div><div><div>INFO : Function hrt_absolute_time is on ignore list; skipped.</div><div>WARNING : Demangled C++ function uORB::DeviceNode::write(file*, char const*, unsigned int) skipped; the linker expects mangled function names.</div><div>INFO : Function sched_unlock is on ignore list; skipped.</div><div>INFO : Function _ZN4math17WelfordMeanVectorIfLj3EE6updateERKN6matrix6VectorIfLj3EEE is on ignore list; skipped.</div><div>INFO : Function MEM_DataCopy0_1 is on ignore list; skipped.</div><div>WARNING : Size of _do_memcpy could not be determined. ITCM could become oversubscribed.</div><div>INFO : Function MEM_DataCopy0_2 is on ignore list; skipped.</div><div>INFO : Function MEM_LongCopyJump is on ignore list; skipped.</div><div>INFO : Done. Estimated ITCM usage: 65536/65280 B</div></div></div><div><div><div>Output ITCM mapping file contains ignored or demangled functions?</div><div>No demangled or ignored functions present in output file.</div></div><div><div><div>ITCM was not oversubscribed when rebuilding PX4?</div><div><table><tr><td>Memory region</td><td>Used Size</td><td>Region Size</td><td>%age Used</td></tr><tr><td>BOOT_HEADER:</td><td>256 B</td><td>4 KB</td><td>6.25%</td></tr><tr><td>VECTORS:</td><td>992 B</td><td>4 KB</td><td>24.22%</td></tr><tr><td>flash:</td><td>1803992 B</td><td>4055039 B</td><td>44.49%</td></tr><tr><td>sram0_stdby:</td><td>0 GB</td><td>32 KB</td><td>0.00%</td></tr><tr><td>sram:</td><td>55856 B</td><td>272 KB</td><td>20.05%</td></tr><tr><td>itcm:</td><td>64832 B</td><td>65280 B</td><td>99.31%</td></tr><tr><td>dtcm:</td><td>0 GB</td><td>128 KB</td><td>0.00%</td></tr></table></div></div></div></div></div></div>	Memory region	Used Size	Region Size	%age Used	BOOT_HEADER:	256 B	4 KB	6.25%	VECTORS:	992 B	4 KB	24.22%	flash:	1803992 B	4055039 B	44.49%	sram0_stdby:	0 GB	32 KB	0.00%	sram:	55856 B	272 KB	20.05%	itcm:	64832 B	65280 B	99.31%	dtcm:	0 GB	128 KB	0.00%
Memory region	Used Size	Region Size	%age Used																														
BOOT_HEADER:	256 B	4 KB	6.25%																														
VECTORS:	992 B	4 KB	24.22%																														
flash:	1803992 B	4055039 B	44.49%																														
sram0_stdby:	0 GB	32 KB	0.00%																														
sram:	55856 B	272 KB	20.05%																														
itcm:	64832 B	65280 B	99.31%																														
dtcm:	0 GB	128 KB	0.00%																														
Conclusion	<div><div><div>Test successful: warnings/notes about demangled/ignored functions or unknown sizes are given, the output ITCM mapping file does not contain any demangled and/or ignored function names, and the ITCM mapping file does not oversubscribe the ITCM.</div></div></div>																																

Test 4: generation of call graphs with orbstat

Description	This test verifies if generation of call graphs of an application running on S32K344 with ITM instrumentation functions and orbstat is functional.
Requirement	R.N.3
Test setup modifications	N/A
Passing criteria	<ul style="list-style-type: none">The call graph generated by orbstat shows the calling relationships between functions that correspond to the source code.
Protocol	<ol style="list-style-type: none">Create an example application that has various functions which call each other in some order. Ensure at least four nested function calls occur in this application.Instrument the example application with ITM instrumentation functions by following the instructions in Chapter 6 of the user guide (Appendix F).Upload the application to S32K344 and run orbstat to generate a call graph of it.Open the call graph using e.g. KCachegrind. Assess whether the call graph matches the call structure of the functions in the application.
Results	<p>Function calls in example application:</p> <pre>void func0(void) { } void func1(void) { func0(); } void func2(void) { func0(); func1(); } void func3 (void) { func0(); func1(); func2(); }</pre>

	<p>Call graph:</p>  <pre> graph TD main["main 100.00 %"] -- "247 x" --> func3["func3 100.00 %"] func3 -- "247 x" --> func2["func2 41.44 %"] func3 -- "247 x" --> func1["func1 37.05 %"] func3 -- "247 x" --> func0["func0 17.58 %"] func2 -- "247 x" --> func1 func2 -- "247 x" --> func0 func1 -- "494 x" --> func0 </pre> <p>The call graph illustrates the execution flow of an application. It consists of five nodes: 'main' (yellow), 'func3' (pink), 'func2' (teal), 'func1' (green), and 'func0' (brown). Each node displays a progress bar and a percentage value. The edges represent function calls, with labels indicating the number of calls: '247 x' for most calls and '494 x' for the call from 'func1' to 'func0'.</p>
<p>Conclusion</p>	<p>The call graph fully corresponds to the function calls shown in the example application. With this, a correct call graph of the application has been generated and orbstat can be declared functional.</p>

Test 5: pyOCD debugging

Description	This test verifies if basic debugging of the M7 in S32K344 through pyOCD is functional.
Requirement	R.N.17
Test setup modifications	N/A
Passing criteria	<ul style="list-style-type: none"> Single-stepping three instructions with GDB advances the core execution by three instructions. After a breakpoint has been set in GDB and the application is run, the application breaks at the correct location.
Protocol	<ol style="list-style-type: none"> Flash the PX4 application to S32K344 and ensure the application is running. Start a pyOCD GDB server: <code>pyocd gdb -t S32K344</code> Start GDB and configure the environment: <code>set architecture armv7e-m</code> <code>file <path_to_px4_elf_file></code> Connect GDB to S32K344: <code>target extended-remote localhost:3333</code> Disassemble the current location of the program and take note of the current instruction being executed: <code>disass</code> Single-step the CPU by three instructions and take note of the instruction being executed now: <code>si 3</code> <code>disass</code> Set a breakpoint at the entry of function <code>nxsem_post</code> and continue execution: <code>b nxsem_post</code> <code>c</code> Once the CPU halts, take note of the current location. Assess whether this matches the start of <code>nxsem_post</code>.
Results	<p>Disassembly at moment of connection:</p> <pre> 0x0069a1b0 <+0>: push {r4, r5, r6, lr} => 0x0069a1b2 <+2>: mov r5, r0 0x0069a1b4 <+4>: mov r1, lr 0x0069a1b6 <+6>: ldr r0, [pc, #76] 0x0069a1b8 <+8>: mov r4, lr </pre> <p>Disassembly after single-stepping three instructions:</p> <pre> 0x0069a1b0 <+0>: push {r4, r5, r6, lr} 0x0069a1b2 <+2>: mov r5, r0 0x0069a1b4 <+4>: mov r1, lr 0x0069a1b6 <+6>: ldr r0, [pc, #76] => 0x0069a1b8 <+8>: mov r4, lr </pre>

	<p>CPU location after encountering nxsem_post breakpoint:</p> <pre>(gdb) b nxsem_post Breakpoint 1 at 0x405f80: file semaphore/sem_post.c, line 78. (gdb) c Continuing. Breakpoint 1, nxsem_post (sem=sem@entry=0x2001fe1c) at semaphore/sem_post.c:78 78 if (sem != NULL) (gdb) disass Dump of assembler code for function nxsem_post: => 0x00405f80 <+0>: push {r4, r5, r6, lr} 0x00405f82 <+2>: mov r5, r0</pre>
Conclusion	<p>The CPU moved three instructions in response to the command <code>si 3</code>. The CPU also stopped at the first instruction of <code>nxsem_post()</code> when setting a breakpoint at this function. With this, debugging of S32K344 through pyOCD is deemed functional.</p>

Test 6: pyOCD application flashing

Description	This test verifies if application flashing through pyOCD is functional on S32K344.
Requirement	R.N.18
Test setup modifications	N/A
Passing criteria	<ul style="list-style-type: none"> No application executes on S32K344 after erasing the on-chip flashes with pyOCD (i.e. no console output is shown on the NuttX console UART). After flashing PX4 to S32K344 that had its flash erase earlier, pyOCD reports no sectors as having been skipped during flash programming. The application runs as expected after uploading it using pyOCD.
Protocol	<ol style="list-style-type: none"> Erase all on-chip flash of S32K344 using pyOCD: pyocd cmd -t S32K344 erase Reset the S32K344 using the reset button on the MR-CANHUBK3. Take note of the output shown on the MR-CANHUBK3 NuttX console UART (if any). Upload PX4 to S32K344 using pyOCD: pyocd flash <path_to_px4_elf_file> -t S32K344 Reset the S32K344 using the reset button. Assess whether PX4 runs as expected.
Results	<p>Application runs after erasing S32K344 flash and reset? No; no output on UART shown.</p> <p>pyOCD flash results after flashing PX4 again:</p> <pre>[=====] 100% 0055339 I Erased 3227648 bytes (394 sectors), programmed 3227648 bytes (394 pages), skipped 0 bytes (0 pages) at 57.38 kB/s [loader]</pre> <p>PX4 behaves as expected after flashing and reset? Yes; NuttX and PX4 start up and the NuttX shell on UART functions as expected.</p>
Conclusion	Test successful: results align with acceptance criteria.

Appendix B: pyOCD target configuration for S32K344

To allow debugging of the S32K344 using the ORBTrace Mini, a target configuration for the pyOCD debugging toolchain has been written. This target configuration is listed below. The flash algorithms were omitted for brevity; these can be manually generated using the S32K344 CMSIS-Pack and the guide provided by (Reed, Adding a new built-in target, 2020).

```
from ...coresight.coresight_target import CoreSightTarget
from ...core.memory_map import (FlashRegion, RamRegion, MemoryMap)

# AP IDs:
# [1]    APB_AP
# [4]    CM7_0_AHB_AP
# [6]    MDM_AP
# [7]    SDA_AP
APB_AP_ID      = 1
CM7_0_AHB_AP_ID = 4
MDM_AP_ID      = 6
SDA_AP_ID      = 7
AP_ID_LIST     = [APB_AP_ID, CM7_0_AHB_AP_ID, MDM_AP_ID, SDA_AP_ID]

# SDA_AP registers:
# [0x80]    Debug Enable Control (DBGENCTRL)
SDA_AP_DBGENCTRL_ADDR = 0x80

# SDA_AP DBGENCTRL bit fields:
# [31:30]    reserved
# [29]       Core Non-Invasive Debug Enable (CNIDEN)
# [28]       Core Debug Enable (CDBGEN)
# [27:8]     reserved
# [7]        Global Secure Privileged Non-Invasive Debug Enable (GSPNIDEN)
# [6]        Global Secure Privileged Debug Enable (GSPIDEN)
# [5]        Global Non-Invasive Debug Enable (GNIDEN)
# [4]        Global Debug Enable (GDBGEN)
# [3:0]     reserved
SDA_AP_CNIDEN_MASK      = 0x20000000
SDA_AP_CNIDEN_SHIFT     = 29
SDA_AP_CDBGEN_MASK      = 0x10000000
SDA_AP_CDBGEN_SHIFT     = 28
SDA_AP_GSPNIDEN_MASK    = 0x80
SDA_AP_GSPNIDEN_SHIFT   = 7
SDA_AP_GSPIDEN_MASK     = 0x40
SDA_AP_GSPIDEN_SHIFT    = 6
SDA_AP_GNIDEN_MASK      = 0x20
SDA_AP_GNIDEN_SHIFT     = 5
SDA_AP_GDBGEN_MASK      = 0x10
SDA_AP_GDBGEN_SHIFT     = 4
SDA_AP_EN_ALL           = (SDA_AP_CNIDEN_MASK | SDA_AP_CDBGEN_MASK |
                           SDA_AP_GSPNIDEN_MASK | SDA_AP_GSPIDEN_MASK |
                           SDA_AP_GNIDEN_MASK | SDA_AP_GDBGEN_MASK)
```

```

FLASH_ALGO_CODE = {
...
}

FLASH_ALGO_DATA = {
...
}

class S32K344(CoreSightTarget):

    VENDOR = "NXP"

    MEMORY_MAP = MemoryMap(
        FlashRegion(name="pflash", start=0x00400000, end=0x7ffffff,
            blocksize=0x2000, is_boot_memory=True, algo=FLASH_ALGO_CODE),
        FlashRegion(name="dflash", start=0x10000000, end=0x1001ffff,
            blocksize=0x2000, algo=FLASH_ALGO_DATA),
        RamRegion(name="itcm", start=0x00000000, length=0x10000), # 64 KB
        RamRegion(name="dtcm", start=0x20000000, length=0x20000), # 128 KB
        RamRegion(name="sram", start=0x20400000, length=0x50000), # 320 KB
    )

    def __init__(self, session):
        super(S32K344, self).__init__(session, self.MEMORY_MAP)

    def create_init_sequence(self):
        seq = super(S32K344, self).create_init_sequence()

        seq.wrap_task('discovery',
            lambda seq: seq
                # Normally the discovery sequence will scan for APs and then
                # add those found to a list. Unfortunately, the S32K344 freaks
                # out when you scan for nonexistent APs, so the list of APs are
                # provided statically here.
                .replace_task('find_aps', self.create_s32k344_aps)

                # Debug needs to be enabled in the SDA_AP before pyOCD can probe
                # for components.
                .insert_before('find_components',
                    ('enable_debug', self.enable_s32k344_debug))
        )

        return seq

    def create_s32k344_aps(self):
        self.dp.valid_aps = AP_ID_LIST

    def enable_s32k344_debug(self):
        self.dp.aps[SDA_AP_ID].write_reg(SDA_AP_DBGENCTRL_ADDR, SDA_AP_EN_ALL)

```

Appendix C: GDB files for setting up tracing on S32K344

This appendix documents the full GDB initialization scripts/commands for which the designs were described in chapter 4.

Trace component setup commands

A GDB command file, containing a collection of GDB commands for setting up various trace components, is provided by the Orbuculum project (Marples, Palsson, Girault, & Izptr, 2022). Compatibility for S32K344 has been added to this file by modifying existing commands for e.g. DWT and ITM setup and adding a new command (shown below) that configures the trace bus of the S32K344 based on the design in Section 4.2. The S32K344-specific changes are intended to be upstreamed back into the Orbuculum project in the future.

```
define enableS32K344SW0
    set $CPU = $CPU_S32K344
    _setAddressesS32K344

    # Enable access to SW0 TPIU, all funnels, and all ETFs.
    set *($TPIU_SW0_BASE + 0xfb0) = 0xc5acce55
    set *($FUNNEL_0_BASE + 0xfb0) = 0xc5acce55
    set *($FUNNEL_1_BASE + 0xfb0) = 0xc5acce55
    set *($FUNNEL_2_BASE + 0xfb0) = 0xc5acce55
    set *($ETF_CM7_CLUSTER_ETMI_BASE + 0xfb0) = 0xc5acce55
    set *($ETF_CM7_CLUSTER_ETMD_BASE + 0xfb0) = 0xc5acce55
    set *($ETF_SHARED_SYSTEM_BASE + 0xfb0) = 0xc5acce55

    # Enable all inputs of all funnels to ensure that all trace sources
    # can pass. Enabling all inputs it probably not optimal, so this
    # might be changed in the future.
    set *($FUNNEL_0_BASE) |= 0xff
    set *($FUNNEL_1_BASE) |= 0xff
    set *($FUNNEL_2_BASE) |= 0xff

    # Configure all ETFs in hardware FIFO mode. This configuration is
    # based on section 2.2.2 from ARM DDI0461B.
    set *($ETF_CM7_CLUSTER_ETMI_BASE + 0x28) = 0x2
    set *($ETF_CM7_CLUSTER_ETMI_BASE + 0x304) = 0x3
    set *($ETF_CM7_CLUSTER_ETMI_BASE + 0x20) = 0x1

    set *($ETF_CM7_CLUSTER_ETMD_BASE + 0x28) = 0x2
    set *($ETF_CM7_CLUSTER_ETMD_BASE + 0x304) = 0x3
    set *($ETF_CM7_CLUSTER_ETMD_BASE + 0x20) = 0x1

    set *($ETF_SHARED_SYSTEM_BASE + 0x28) = 0x2
    set *($ETF_SHARED_SYSTEM_BASE + 0x304) = 0x3
    set *($ETF_SHARED_SYSTEM_BASE + 0x20) = 0x1

    # Disable parallel trace TPIU backpressure (we only use SW0) through
    # MDM_AP register. Note: this monitor command probably only works on pyOCD.
    monitor wap 6 0x4 0x06100000
```

```

# Configure I/O pad PTA10 for SW0: output mode, high slew rate, etc.
set *(0x40290268) = 0x00200127

# Overwrite TPIUBASE with TPIU_SW0_BASE of S32K344 for use in subsequent
# function calls.
set $TPIUBASE = $TPIU_SW0_BASE
end

```

GDB initialization script to set up tracing for orbttop

The GDB initialization script shown below can be used when profiling an application with orbttop. The script first calls the `enableS32K344SW0` and `prepareSW0` commands to set up the trace bus and SW0 TPIU, followed by command calls to set up the DWT for periodic PC sampling and to let the ITM pass through hardware trace messages from DWT.

```

# Example .gdbinit that configures the S32K344 tracing components for
# use with orbttop. The ORBTRace Mini is also configured.

set mem inaccessible-by-default off
set architecture armv7e-m
file <path to elf file>
!killall pyocd
!pyocd gdb -t S32K344 &
target extended-remote localhost:3333

# Import trace config commands.
source <path to gdbtrace s32k344.init>

# Configure SW0 I/O pad and trace bus components.
enableS32K344SW0

# Configure SW0 TPIU in Manchester mode at specified speed.
prepareSW0 160000000 32000000 0 1

# Enable DWT PC sampling and configure the desired sampling rate using
# the POSTCNT counter.
dwtSamplePC 1
dwtPostTap 1
dwtPostInit 1
dwtPostReset 7
dwtCycEna 1

# Enable ITM DWT message passthrough.
ITMId 1
ITMTXEna 1
ITMEna 1

# Configure ORBTRace Mini to receive SW0 in Manchester mode.
!orbttrace -T m

```

GDB initialization script to set up tracing for orbstat

The GDB initialization script shown below can be used when profiling and application with orbstat. The script has a similar structure to the GDB initialization script for orbttop, but configures the ITM to only enable its software channels and doesn't enable any hardware tracing features of the DWT.

```
# Example .gdbinit that configures the S32K344 tracing components for
# use with orbstat. The ORBTRace Mini is also configured.

set mem inaccessible-by-default off
set architecture armv7e-m
file <path_to_elf_file>
!killall pyocd
!pyocd gdb -t S32K344 &
target extended-remote localhost:3333

# Import trace config commands.
source <path_to_gdbtrace_s32k344.init>

# Configure SWO I/O pad and trace bus components.
enableS32K344SWO

# Configure SWO TPIU in Manchester mode at specified speed.
prepareSWO 16000000 3200000 0 1

# Enable DWT cycle counter; this is used by the ITM instrumentation
# functions.
dwtCycEna 1

# Enable ITM and all of its software channels.
ITMId 1
ITMTXEna 1
ITMEna 1
ITMTER 0 0xFFFFFFFF
ITMTPR 0xFFFFFFFF

# Configure ORBTRace Mini to receive SWO in Manchester mode.
!orbtrace -T m
```

Appendix D: ITCM mapping tool source code

The source code for the ITCM mapping tool discussed in Section 4.4 is listed below.

```
ITCM_SIZE = 65280
INVALID_FUNCTION_NAMES = ("** Sleeping **")
INVALID_SECTION_CHARS = ":"

import sys
import json
import subprocess
import logging

# Create a list of dictionaries, with each dictionary storing the size and
# name of each function/symbol.
def gen_functions_list(elf_file):
    nm_result = subprocess.run(["arm-none-eabi-nm", "-S", "--size-sort", elf_file],
                               capture_output=True,
                               text=True,
                               check=True)

    functions_list = []

    for l in nm_result.stdout.split("\n"):
        l_list = l.split(" ")

        # Only parse lines of the right symbol type (must be T or t) and
        # if they have the right number of fields.
        if (" T " in l or " t " in l or " W " in l) and len(l_list) == 4:
            functions_list.append({"name" : l_list[3].strip("\n"),
                                   "size" : int(l_list[1], 16)})

    return functions_list

# Return the size of the specified function in functions_list.
def get_function_size(functions_list, name):
    for function in functions_list:
        if function["name"] == name:
            return function["size"]

    # No function with that name found, so return None as size.
    return None

if __name__ == "__main__":
    logging.basicConfig(format='%(levelname)-8s: %(message)s', level=logging.INFO)

    # Check input args.
    if (len(sys.argv) - 1 < 3 or len(sys.argv[1]) == 0 or
        len(sys.argv[2]) == 0 or len(sys.argv[3]) == 0):
        logging.error("Invalid arguments. Should be: <output_filename> <elf_filename> " +
                      "<orbtcp_json_filename> [ignorelist_filename]")
```

```

    exit()

out_filename = sys.argv[1]
elf_filename = sys.argv[2]
orbtob_json_filename = sys.argv[3]
try:
    ignorelist_filename = sys.argv[4]
except IndexError:
    ignorelist_filename = None

# Create list of functions and their sizes in the program for which we're
# generating an ITCM mapping.
functions_list = gen_functions_list(elf_filename)

# Add contents of ignore list to a local list.
if ignorelist_filename != None:
    with open(ignorelist_filename, "r") as f_ignorelist:
        ignorelist = f_ignorelist.read().split("\n")
else:
    ignorelist = []

# Go over each function in orbtob json and add each to the ld file until
# the ITCM is full.
f_ld = open(out_filename, "w")
orbtob_json = json.load(open(orbtob_json_filename, "r"))
i = 0
mapped_size = 0

while mapped_size < ITCM_SIZE:
    function_name = orbtob_json["toptable"][i]["function"]

    # Increment for next iteration; i is not referenced for the rest
    # of this loop iteration.
    i += 1

    # Skip if section name is invalid.
    if function_name in INVALID_FUNCTION_NAMES:
        continue

    # Skip if function is in ignore list.
    if function_name in ignorelist:
        logging.info("Function {} is on ignore list; skipped."
                    .format(function_name))
        continue

    # Skip if invalid chars found in section name.
    if any((c in INVALID_SECTION_CHARS) for c in function_name):
        logging.warning("Demangled C++ function {} ".format(function_name) +
                        "skipped; the linker expects mangled function names.")
        continue

```



```

# No issues with function name found; add function to ld file and
# update mapped size.
f_ld.write("*(.text.{})\r\n".format(function_name))

size = get_function_size(functions_list, function_name)

if size == None:
    logging.warning("Size of {} could not be determined. "
                    .format(function_name) +
                    "ITCM could become oversubscribed.")
else:
    mapped_size += size

# Done.
logging.info("Done. Estimated ITCM usage: {}/{ B".format(mapped_size, ITCM_SIZE))
f_ld.close()

```

Appendix E: orbstat ITM instrumentation functions

The source for the example ITM instrumentation functions discussed in Section 4.5 is listed below.

```
// C source containing the instrumentation functions that send the data
// expected by orbstat using an ITM software channel.

#include <stdint.h>

// Truncated typedefs for access to necessary ITM and DWT registers
typedef struct
{
    volatile    uint32_t PORT[32u];
                uint32_t _reserved[864u];
    volatile    uint32_t TER;
} ITM_t;

typedef struct
{
    uint32_t _reserved;
    volatile    uint32_t CYCCNT;
} DWT_t;

#define ITM ((ITM_t *) 0xe0000000u)
#define DWT ((DWT_t *) 0xe0001000u)

#define ITM_ORBSTAT_CHANNEL 1          // Orbstat data transmission channel
#define FN_MASK              0x03ffffff
#define FN_ENTRY_VAL        0x40000000 // Value to indicate function entry
#define FN_EXIT_VAL         0x50000000 // Value to indicate function exit

__attribute__((no_instrument_function, optimize("Os")))
void __cyg_profile_func_enter (void *this_fn, void *call_site)
{
    // Exit if ITM channel is not enabled.
    if (!(ITM->TER & (1 << ITM_ORBSTAT_CHANNEL)))
        return;

    // NOTE: disable interrupts here.

    // Send current CYCCNT value, with the upper bits masked. Insert a value
    // into the upper bits to signify function entry or exit.
    while (ITM->PORT[ITM_ORBSTAT_CHANNEL] == 0);
    ITM->PORT[ITM_ORBSTAT_CHANNEL] = (DWT->CYCCNT & FN_MASK) | FN_ENTRY_VAL;

    // Send values of call_site and this_fn.
    while (ITM->PORT[ITM_ORBSTAT_CHANNEL] == 0);
    ITM->PORT[ITM_ORBSTAT_CHANNEL] = (uint32_t)call_site & 0xFFFFFFFF;
    while (ITM->PORT[ITM_ORBSTAT_CHANNEL] == 0);
    ITM->PORT[ITM_ORBSTAT_CHANNEL] = (uint32_t)this_fn & 0xFFFFFFFF;
```

```

    // NOTE: re-enable interrupts here.
}

__attribute__((no_instrument_function, optimize("Os")))
void __cyg_profile_func_exit (void *this_fn, void *call_site)
{
    // Exit if ITM channel is not enabled.
    if (!(ITM->TER & (1 << ITM_ORBSTAT_CHANNEL)))
        return;

    // NOTE: disable interrupts here.

    // Send current CYCCNT value, with the upper bits masked. Insert a value
    // into the upper bits to signify function entry or exit.
    while (ITM->PORT[ITM_ORBSTAT_CHANNEL] == 0);
    ITM->PORT[ITM_ORBSTAT_CHANNEL] = (DWT->CYCCNT & FN_MASK) | FN_EXIT_VAL;

    // Send values of call_site and this_fn.
    while (ITM->PORT[ITM_ORBSTAT_CHANNEL] == 0);
    ITM->PORT[ITM_ORBSTAT_CHANNEL] = (uint32_t)call_site & 0xFFFFFFF;
    while (ITM->PORT[ITM_ORBSTAT_CHANNEL] == 0);
    ITM->PORT[ITM_ORBSTAT_CHANNEL] = (uint32_t)this_fn & 0xFFFFFFF;

    // NOTE: re-enable interrupts here.
}

```

Appendix F: S32K344 tracing tools user guide (external)

This external appendix is the user guide that documents all tracing tools. Refer to the file named “*Appendix F - S32K344 tracing tools user guide_Graduation report_Ian Baak_2023-05-22_v1.1.pdf*” to view this user guide.