



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



*Institut für
Technische Informatik und
Kommunikationsnetze*

Distributed Debugging for FlockLab 2

Semester Thesis

Lukas Daschinger

ldaschinger@student.ethz.ch

Computer Engineering and Networks Laboratory
Department of Information Technology and Electrical Engineering
ETH Zürich

Supervisors:

Roman Trüb

Reto Da Forno

Prof. Dr. Lothar Thiele

July 2, 2020

Abstract

In order to develop wireless sensor network (WSN) applications on embedded systems, a testbed infrastructure is required. Some well known examples of such testbeds are Flocklab, Motelab or Twist. Among those, FlockLab offers the most advanced services like GPIO actuation and tracing, highly accurate timing information and the possibility to profile and control power usage. The newer version FlockLab 2 released in March 2020 improves FlockLab in many ways including more accurate power profiling, GPS-based time synchronization and native SWD debugging. Previously debugging could only be done by using the GPIO pins or `printf` statements sent over the serial port. These options either limit the complexity of the output or require software instrumentation. To mitigate these limitations, alternative debugging options based on SEGGER J-Link are explored and analyzed in this thesis. The most suitable solution is implemented and integrated in the FlockLab 2 testbed. The result is a debugging feature that lets the user non-intrusively trace up to four variables and take PC snapshots on specified events. The debugger is capable of tracing variables with up to 77 kHz and the accuracy has been measured to be as good as 1 ms for a 30 min test duration. The timestamps in the trace are all taken with reference to the global GPS-synchronised time. This allows to compare the data traces from different observers making the feature a truly distributed debugger.

Contents

Abstract	i
1 Introduction	2
1.1 Related Work	3
1.1.1 CD-TRS	3
1.1.2 HATBED	3
1.1.3 Minerva	4
2 Background	5
2.1 Hardware Assisted Tracing on ARM	5
2.2 The FlockLab 2 Architecture	6
2.3 The DPP2 LoRa ComBoard	7
2.4 Clock Configuration on the DPP2 LoRa ComBoard	7
3 Exploration of Debugging Features	9
3.1 The FlockLab 2 Debugging Architecture	9
3.1.1 Debugging Features of the Cortex-M4 Processor	10
3.1.2 Debugging Software	12
3.2 Debugging Features of a Single Observer and Target	14
3.2.1 Breakpoints and Watchpoints	14
3.2.2 Read out and write into Memory	14
3.2.3 Data Trace	16
3.2.4 Printf SWIT	17
3.2.5 SEGGER's Real Time Transfer (RTT)	17
3.2.6 Instruction Trace	18
3.3 Distributed Debugging Features	18
3.3.1 Halt and start execution synchronously on whole testbed	18
3.3.2 Distributed Assertions/Watchpoints	19

CONTENTS	iii
3.3.3 Global Data Tracing	19
3.3.4 Selection of a Distributed Debugging Feature	19
4 Implementation	21
4.1 Target Configuration	21
4.2 SWO Buffer Reading	22
4.3 DWT Packet Parser	24
4.4 Time Correction with Regression	24
4.5 Implementation on the FlockLab 2 Architecture	26
5 Performance Measurement	31
5.1 Influence of target clock source on the timestamp accuracy . . .	31
5.1.1 Methodology	31
5.1.2 Results	32
5.1.3 Discussion	32
5.2 Accuracy of the Corrected Timestamps	36
5.2.1 Methodology	36
5.2.2 Results	37
5.2.3 Conclusions	37
5.3 Maximal Tracing Frequency	39
5.3.1 Methodology	39
5.3.2 Results	40
5.3.3 Discussion	41
5.4 CPU Usage	41
5.4.1 Methodology	41
5.4.2 Results	42
5.4.3 Discussion	42
6 Future work and conclusions	44
6.0.1 Future work	44
6.0.2 Conclusions	45
Appendices	46

CONTENTS	iv
A Schedule	47
B Original Assignment	49
C How To	56
C.1 Software installation	56
C.2 Observer	56
C.2.1 Observer setup	56
C.2.2 Observer use	57
C.2.3 J-Link software installation	57
C.2.4 Pylink library installation	58
C.2.5 GDB multiarch installation	60
C.3 Software Use	60
C.3.1 Pylink examples	60
C.3.2 J-Link GDB server	60
C.3.3 JLinkExe	62
C.3.4 JLinkSWOViewer	62

Acronyms

BBG	BeagleBone Green.	7, 10
DLL	Dynamic Link Library.	10
DPP2	Dual Processor Platform 2.	7
DWT	Data Watchpoint and Trace.	5
DWT_COMPn	Comparator register.	21
DWT_FUNCTIONn	Comparator Function register.	21
ETB	Embedded Trace Buffer: A small amount of internal RAM used as an ETM trace buffer.	12
ETM	Embedded Trace Macrocell.	5
ITM	Internal Trace Macrocell.	5
ITM_TCR	Trace Control Register.	21
MCU	Microcontroller unit.	9
NRZ	Non-return-to-zero.	23
PC	Program Counter.	3, 5, 6, 11, 12
SWD	Serial Wire Debug.	9
SWIT	Software Instrumentation.	5
SWO	Serial Wire Output.	10
SWV	Serial Wire Viewer.	11
TPIU	Trace Port Interface Unit.	22
WSN	Wireless Sensor Network.	6

Introduction

To develop the protocols and applications running on wireless embedded devices, a testing and debugging infrastructure is required. Testbeds provide the necessary functionalities and let the developers closely observe what happens on every node in a network. As of today many such testbeds exist. Some well known examples are FlockLab [1], Motelab[2] or Twist [3]. These testbeds all offer many services but often have limited debugging options. In the case of FlockLab for example the tracing features are limited to `printf` debugging over the serial port, setting digital GPIO pins and power tracing.

`Printf`-style debugging is a very flexible method because the user can extract arbitrary information at virtually any point in the program execution. The drawback of this feature is that it is intrusive, meaning it requires software instrumentation which introduces significant delays in the execution of a program. This can lead to Heisenbugs. These are bugs appearing only when debugging the program [4]. Therefore this method is not suitable for time sensitive debugging.

GPIO instrumentation is much more performant in terms of timing overhead. To set a GPIO pin only a few microprocessor instructions are required which results in a delay of about 0.05 μ s [5]. The drawback of this method is that the user can only extract limited information from a GPIO pin. The pin is either set or not set (binary information). Another drawback of this debugging method is that the code needs to be recompiled after changing the observed variables.

Finally, power tracing can also help in debugging but is limited in the way that a user cannot obtain information about the internal state of the microcontroller.

In the past several solutions to mitigate these drawbacks like CD-TRS [6], HATBED [7] or MINERVA [5] have been presented. These tools are all based on the hardware debugging features built into modern MCUs. The Cortex-M4 used on the targets of FlockLab 2 for example offers features like breakpoints, watchpoints or instruction and data tracing. In order to access these functionalities a debug probe is required. FlockLab 2 includes a SEGGER OB on-board probe that will be used to debug the target processors. In this

work the DPP2 LoRa ComBoard ¹ [8] will be used to implement and test the debugging feature. This is a single representative platform and the same debugging concept applies to all ARM Cortex-M processors.

As part of this semester thesis the powerful debugging features presented in the HATBED and MINERVA papers will be integrated on an observer of the FlockLab 2 testbed. Using the CoreSight [9] features available on ARM microprocessors, it will be possible to do flexible non-intrusive tracing on the targets. The proposed solution will further improve the services of FlockLab 2.

1.1 Related Work

In this section several debugging solutions for WSN testbeds are presented with a focus on debugging capabilities.

1.1.1 CD-TRS

CD-TRS stands for Cross-device Testing and Reporting System for Large-scale Real-Time Wireless Networks [6]. The debugging service of this system is based on a logging approach: previously defined events and device status are sent over the J-Link log interface as streams during runtime. Logs from several nodes are sent to the analyzer simultaneously. The analyzer will parse the streams and convert them back into events that are then sorted by the Absolute Slot Number (ASN). CD-TRS also offers automatic comparison of the analyzer output with a simulation result based on a specified topology and device tasks.

1.1.2 HATBED

HATBED stand for Hardware Assisted Testbed for Non-invasive Profiling of IoT Devices [7]. HATBED offers three debugging services:

- Network-Wide Remote Debugging: This feature offers non-intrusive tracing and global breakpoints and assertions. This is achieved by using remote Serial Wire Debugging (SWD).
- Flexible Software Tracing: This is similar to `printf` debugging but uses the ITM message output instead of UART. This solution greatly reduces the timing overhead of the UART protocol.
- Non-invasive Software Profiling: This feature uses DWT unit to enable the user to set watchpoints in the code. This makes it possible to capture the changes of important variables or to sample to Program Counter (PC) and detect when a specific function is executed.

¹<https://gitlab.ethz.ch/tec/public/dpp/-/wikis/>

HATBED uses OpenOCD and a FT2232HL IC which offer similar debugging capabilities as J-Link. For remote debugging OpenOCD builds a local server and then GDB running on the observer can connect to it.

1.1.3 Minerva

Minerva is a testbed developed specifically for distributed debugging and therefore offers the most advanced features compared to the other platforms [5]. Software and hardware in Minerva are the same as on HATBED. Namely an FT2232HL IC and OpenOCD is used to create a remote debug target to which GDB can connect to. Minerva offers the following debugging features:

- **Tracing of the internal state:** Minerva implements a polling system where the observer periodically polls a specified memory location in the target (Opal platform) and informs the controller (central server) whenever the value changes. This tracing with JTAG does not cause any timing overhead compared to GPIO or `printf`.
- **Ability to halt the whole testbed:** For this functionality the controller sends a halting command over UDP to the observers. The observers will then make the target enter debugging mode which will stop its processor clock. Due to jitter of around 100 ms in the arrival of the UDP packets this can result in inconsistencies for timing-sensitive applications.
- **Snapshots of memory regions:** This feature enables the user to read out the SRAM through the JTAG port even when the microprocessor is running.
- **Real-time assertions:** Minerva offers distributed assertions. These are conditions on global variables that need to be true on all nodes. In case the assertion fails, the user can specify the program to be halted and enter debug mode or to continue running and set a flag in memory based on the assertion.

Most features of the Minerva testbed are not useful for time-critical, low-level code which was also stated by the authors.

Background

In this chapter, first the basic debugging mechanism on ARM platforms are described in section 2.1. Then the FlockLab 2 architecture and the DPP2 LoRa ComBoard are introduced in sections 2.2 and 2.3. Finally, section 2.4 lists the possible clock sources that can be used on the ComBoard.

2.1 Hardware Assisted Tracing on ARM

Hardware assisted tracing is a functionality that lets a user observe the program execution without impacting the performance of the CPU. The tracing is done by special hardware modules on the microprocessor such that no CPU cycles are spent on debugging. For ARM processors this technology is called CoreSight.

ARM Cortex-M controllers include three Trace Units to enable hardware assisted tracing: A Data Watchpoint and Trace (DWT), an Internal Trace Macrocell (ITM) and an Embedded Trace Macrocell (ETM) all shown in Figure 2.1

The DWT offers features like data tracing and PC sampling. Data tracing is a feature that allows the user to observe the value of a global variable during program execution. When PC sampling is enabled, the current value of the PC is sent to the debug port of the microprocessor with a predefined frequency. The implementation of these functionalities is based on debug events such as exception events and data watchpoint events. Upon an event a data packet is generated in the DWT and sent to the ITM where it is timestamped. The ITM timestamps the said packets and also handles Software Instrumentation (SWIT) events which are custom `printf`-style statements specified in the code. The ETM provides cycle accurate instruction trace that can be used to analyze the execution history or to do code coverage analysis [10]. Instruction trace is a trace that will include every single assembly instruction that is executed on the microprocessor.

Apart from the `printf`-style debugging with the ITM, the described features are all so-called non-intrusive methods. This means, that the microprocessor does not use any additional CPU cycles for debugging.

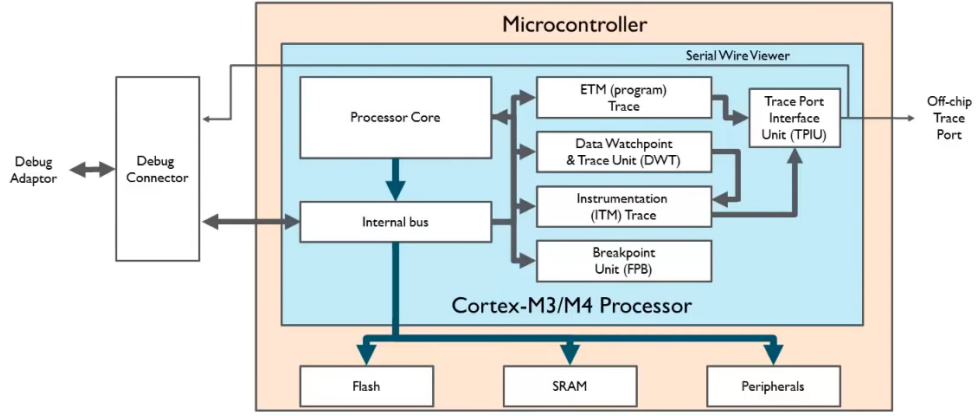


Figure 2.1: The hardware debugging modules built into a Cortex-M4 processor [12]

The mentioned hardware modules are also used by debugging software like GDB. GDB stand for GNU Debugger and is a widely used debugger for Linux systems. It allows to start and stop execution, place breakpoints and watchpoints or inspect the execution trace. Breakpoints can be further categorized in hardware and software breakpoints. Hardware breakpoints do not change the content of the memory on the target but passively listen to the internal bus. Whereas when a software breakpoints is set, the instruction where it is placed on is replaced with 0xffff. This will cause an exception that the debugger can catch and then halt the execution. For hardware breakpoints comparators built into the DWT will compare the breakpoint address with the PC address and stop the execution when they match. This limits the number of breakpoints to the number of comparators. The implementation of watchpoints is very similar but here a variable or memory location is compared to the current variable value instead of the PC [11].

Breakpoints and watchpoints are intrusive debugging methods since the execution is halted once the condition is true.

The debugging hardware of the FlockLab 2 architecture is described in more detail in section 3.1

2.2 The FlockLab 2 Architecture

FlockLab 2 is the successor of FlockLab, a Wireless Sensor Network (WSN) testbed developed by the Computer Engineering and Networks Laboratory. It is based on three Layers: a system layer, an observer layer and a server.

The system layer describes the nodes of the WSN, also referred to as tar-

gets. The user can upload code to the nodes and observe power usage, GPIO pins and the serial output during the execution.

The monitoring is done by the observer layer consisting of more powerful nodes based on the BeagleBone Green (BBG) platform. The observers use GPS-based time synchronization such that the GPIO tracing and power profiling is comparable between nodes. Most importantly, every node features a SEGGER J-Link OB debug probe allowing to use most of the Cortex-M4 debugging features. Section 3.1) elaborates on the exact debugging hardware available on FlockLab 2.

The server is the third layer of the FlockLab architecture. It schedules and starts the tests, the user uploaded. After the test, the server collects the tracing data from the observers and makes it available to the user.

2.3 The DPP2 LoRa ComBoard

The Dual Processor Platform 2 (DPP2) is an architecture template that allows to separate communication tasks from sensing, actuation and data processing [8]. This is achieved by mapping the different tasks to two microcontrollers, both optimized for the respective task. The communication between the two processor happens over the stateful interconnect BOLT [13].

For this work the "DevBoard", A low-power development board based on a MSP432, is used for the processing task ¹. The communication task is done on a low-power communication board based on a SX126x ².

2.4 Clock Configuration on the DPP2 LoRa ComBoard

Since debugging is often a very time sensitive task, it is important to understand the different clock sources the target may use. In the following all the possible clock configurations for the DPP2 LoRa ComBoard are listed. The list includes internal clocks of the STM32L433CC but also external oscillators which are fitted on the ComBoard and can be used with STM32L433CC.

- **12 MHz high-speed external crystal oscillator (HSE):** This source is often just referred to as "crystal" or "crystal oscillator". It is not built into the STM32L433CC but is situated on the ComBoard. It is the most

¹<https://gitlab.ethz.ch/tec/public/dpp/-/wikis/Application/DevBoard>

²<https://gitlab.ethz.ch/tec/public/dpp/-/wikis/Communication/DPP2CC430>

accurate clock with an advertised ppm of ± 15 at 25 degrees Celsius and ± 25 in the operating range [14].

- **Multi-speed internal RC oscillator (MSI):** this oscillator is built into the STM32L433CC. It supports speeds from 100 kHz to 48 MHz. This is an RC oscillator (Resistor, Capacitor) which means it is fast to startup, cheap but also less accurate. The drift over temperature is -3.5% to 3%, drift over voltage is -1.6% to 1% [15].
- **16 MHz High-speed internal RC oscillator (HSI16):** This clock source runs at 16 MHz and has the following specifications: The drift over temperature is -1% to 1%, drift over voltage is -0.1% to 0.05% [15].
- **32 kHz internal low-power RC oscillator (LSI):** This clock source runs at 32 kHz and is made for low power use. The datasheet does not contain any information about its accuracy.
- **32 kHz external low-speed crystal oscillator (LSE):** The LSE has a ppm of ± 20 for a temperature of 25 ± 3 degrees Celsius [16].

In this work the HSE crystal and the MSI are used and tested as clock sources. They allow to test the debugging feature with an example of a high-accuracy and a low-accuracy clock. In general, the timestamps on debugging events suffer from less drift over time when crystals are used instead of RC oscillators.

Exploration of Debugging Features

This chapter describes debugging architectures and their features. The first section describes the debugging hardware on FlockLab 2 and possible debugging architecture. The following sections 3.1.1 and 3.1.2 describe the components of the debugging architecture in more detail. Finally sections 3.2 and 3.3 list features that can be implemented using a combination of different debugging components.

3.1 The FlockLab 2 Debugging Architecture

A debugging system consists of three modules. A target, a debug probe and an observer as shown in Figure 3.1.

The target is the Microcontroller unit (MCU) or platform we would like to extract information from for debugging. On FlockLab 2 the target platforms are the DPP2 SX1262 LoRa ComBoard and the nRF52840 Dongle. Both use a Cortex-M4 processor. The target offers debugging features like breakpoints that can be accessed with the debug probe. Section 3.1.1 describes these in more detail.

The debug probe is connected to the target's processor and handles the trace streams and configures the CoreSight modules for debugging. The observers on FlockLab 2 use a SEGGER J-Link OB on-board debug probe. The communication between the target and probe happens over Serial Wire Debug (SWD). SWD is a hardware interface that allows to directly access the MCU's memory bus making it useful for debugging. It is the alternative debug interface to JTAG and is developed and used by ARM. Compared to JTAG, SWD only requires two pins instead of four. One pin is used for a clock signal that can

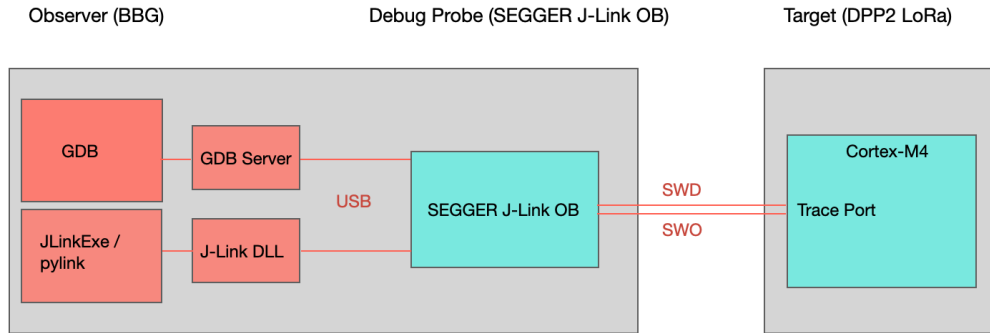


Figure 3.1: Possible debugging architectures. Software and protocols are shown in red, hardware in cyan

be input or output from the device (SWCLK) and the other one serves as a bidirectional data line (SWDIO). In addition to SWD, the observer hardware also features a Serial Wire Output (SWO) pin. It can be used for unidirectional data transfer from `printf` statements or variable tracing.

The observer is a more powerful node based on the BBG [17]. There are various software solutions and libraries to access a debug probe from the observer. For example J-Link specific software like `JLinkExe` or `JLinkGDBServer`, GDB or the Python library `pylink`. When GDB is used, it connects to a local server set up on the observer that communicates with the debug probe. The other tools use a Dynamic Link Library (DLL) offered by J-Link. This library offers functions to control the target and for example set breakpoints or start tracing the execution.

In order to use the tracing features provided by the ETM, a trace probe like J-Trace would be required. (FlockLab 2 only features a J-Link OB debug probe).

3.1.1 Debugging Features of the Cortex-M4 Processor

This section lists all the available debugging features of the Cortex-M4 processor and limitations of the available debug probe. An overview of the features is shown in Figure 3.2.

Most debugging features of the Cortex-M processor make use of the SWO pin. On the FlockLab 2 PCB the J-Link OB module is connected to the target over an SWD interface including the SWO line which makes the use of these features possible. Note that when using the DPP2 DevBoard [18] and a J-Link mini debug for testing the SWO pin is not connected.

The SEGGER J-Link OB debug probe supports the hardware tracing

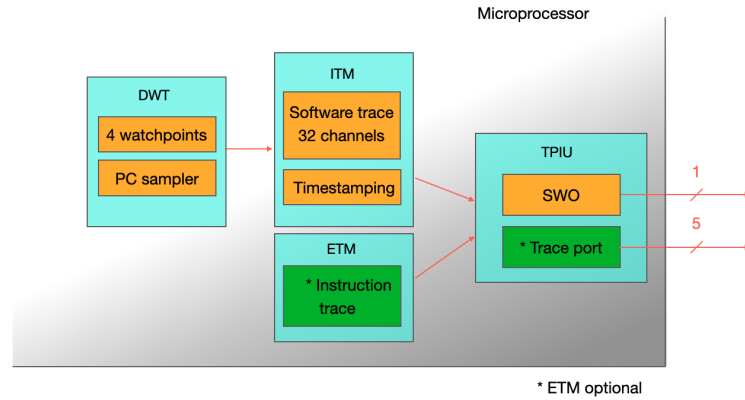


Figure 3.2: Debugging Functionalities of the ARM Cortex-M processors (based on [19])

features of the DWT and the software trace capabilities of the ITM. Solely the ETM instruction tracing features cannot be used with this debug probe. To process the trace stream generated by the ETM a SEGGER J-Trace probe would be required. However, it is possible to read the last few executed instruction from the ETB once the execution is halted.

In the following the debugging features based on ITM, DWT and ETM module of the Cortex-M4 processor are listed:

- Serial Wire Viewer (SWV) features using the ITM (Software trace):
The ITM offers a feature that uses port 0 of the ITM to send `printf`-style messages over the SWO pin. In order to use it we need to overwrite the `_write` function in software code running on the MCU which is debugged such that the `printf` function uses the ITM instead of UART. The other ports can be used to transmit further information.
A disadvantage of this method is that it is not hardware based i.e. we still need to call a function to print over ITM which puts a small load on the CPU.
- SWV features using the DWT (Hardware trace):
The DWT offers several features for tracing variables and the execution:
It is possible to trace the following events and exceptions: Cycles per instruction, sleep cycles, folded instructions, exception overhead, load store unit cycles and trace exceptions. This functionality can be used to measure the execution time of a function by checking how many cycles have passed from start until function termination. However this costs CPU cycles since the register with the number of cycles stored must be read.
A second function is PC sampling with different resolutions of cycles per

sample.

The most powerful feature are the four comparators. They can be configured as a hardware watchpoint, an ETM trigger, a PC sampler and event trigger or a data address sampler event trigger. This allows to track a variable or address and store the value with or without the current PC. If the PC is also stored, it is possible to find the function that wrote to a specific memory address given only the write event at the address.

- **ETM tracing:** The ETMv3 module on Cortex-M4 processors allows to record a complete instruction trace. This data is then either streamed to a PC with a J-Ttrace module (not available) or saved in the Embedded Trace Buffer: A small amount of internal RAM used as an ETM trace buffer (ETB). Once the system is halted, this trace buffer can be read out to backtrace any operation. Due to the limited memory of an ETB, SEGGER recommends to use a streaming capable probe like the J-Trace. However it is possible to use the ETB without access to a J-Trace module which might enable some features for the debugging service.

3.1.2 Debugging Software

This section describes a selection of debugging software that can be used on the BeagleBone single-board computer with the J-Link OB debug probe. Note that not all software can be used to access all debugging features of the Cortex-M core. Table 3.2 provides more information on possible combinations.

The installation and use of these tools is described in the appendix C.

- **J-Link Commander (JLinkExe):** This is a very basic command line tool supporting J-Link probes only. Apart from simple commands like memory dump, halt, step and go the J-Link Commander can also be used to read DWT and ITM packets sent over the SWO pin.
 - + The tool can receive DWT and ITM packets
 - It is challenging to automate interaction with the tool since it has a command line interface that is not optimized for automation
 - The tool cannot parse all DWT packets
 - The tool cannot be extended or customized since the source code is unavailable
- **JLinkGDBServerCLExe with gdb-multiarch:** The JLinkGDBServerCLExe tool sets up a GDB server making it possible for GDB to communicate with the target over a J-Link debug probe. The server runs on the observer and the GDB process can run on the user's computer or even remotely. To communicate with the J-link GDB server, gdb-multiarch must be used.

In order to automate the interaction with GDB, the machine interface interpreter (or GDB/MI) can be used. This command interpreter is independent of the GDB-multiarch tool and can be selected with the option `--interpreter` when launching GDB.

The possible commands include the basic GDB command set plus the SEGGER-specific GDB protocol extensions. The latter include options for simple tracing and SWO reading to the standard GDB commands.

Furthermore, a `.gdbinit` file can be used to automate the setup of the GDB client.

- + It is possible to use the standard GDB environment
 - + An automatic interaction is possible but limited
 - Server and client communication adds overhead and delays
 - It is not possible to parse DWT packets with the tool
 - The tool cannot be extended or customized since the source code of `JLinkGDBServerCLExe` and `gdb-multiarch` is not public
- **Pylink:** This is a Python library that provides Python functions using the DLL from J-Link. The functions allow to use all debugging features of the target if the required hardware is present.
 - + `pylink` is very flexible and configurable and the source code is available
 - + The interaction with J-Link can be integrated in a python script
 - + It works with Python ≥ 2.7
 - It is less popular than GDB
 - It is not an official SEGGER product
 - **STM32CubeIDE:** The STM32CubeIDE cannot be used on the BBG board but is listed here since it is functionalities inspired the development of the distributed debugging tool. The STM32CubeIDE allows to set hardware breakpoints and watchpoints, trace variables, read ITM packets or trace the PC value. The IDE starts a J-Link GDB server and then communicates with this server over a client also launched by the IDE. The most powerful feature is the data tracing of global variables. The IDE does this with a plugin that parses the received DWT packets and then plots the variable value over time.
 - + The tool is easy to use
 - It can not be used on a BBG
 - It can not extended or customized since source code unavailable

3.2 Debugging Features of a Single Observer and Target

The features listed in this section are combinations of the debugging features of the Cortex-M4, the debug probe and a software solution.

They are based on a single observer and section 3.3 lists possible **distributed** debugging features using multiple observers.

An overview of the features can be found in Table 3.1. The table does not list the STM32CubeIDE as a software solution when it could be used, since it is not possible to run the IDE on the BBG board. However, it can be used on a computer to set breakpoints and watchpoints, to configure and record a data trace and for PC sampling.

3.2.1 Breakpoints and Watchpoints

As described in section 2 it is beneficial to use hardware breakpoints instead of software breakpoints since the first method is non-intrusive. Watchpoints are always using the hardware mechanism. However, hardware breakpoints are based on the four comparators in a Cortex-M4 processor which limits the maximum number to four. The comparators, located in the DWT unit, will contain an address that is compared to the current PC value and stop the execution on a match. Hardware breakpoints can be set using the `hardware_breakpoint_set` in `pylink` or `hbreak` in GDB.

Once a watchpoint or breakpoint is hit, the MCU is halted which makes it possible to read or write values into the memory or write values to the flash.

It is possible to restart the microprocessor immediately after stopping and reading values from the memory. To automate this the `.gdbinit` file can be used with the "action" command or the `restart` function in `pylink`.

- + It is possible to define different conditions on every node
- + It is possible to read all register and memory once halted
- It is not possible to halt the whole testbed synchronously
- Once halted it is impossible to start the whole testbed synchronously again
- It is challenging to make the feature distributed

3.2.2 Read out and write into Memory

This is only possible when the microprocessor is halted as stated above. To read or write into memory and write into flash the functions `memory_write32` and

Table 3.1: Summary of the available debugging features

Feature	Required hardware***	Available software	Halt required	Software instrumentation	Time impact	Collected values	Register configuration	Used in implementation
Breakpoints and watchpoints	ITM, DWT, J-Link	GDB**, pylink	Yes	No	n/a	PC, variables, registers	No	No
Read out memory	J-Link	GDB**, pylink, JLinkExe	Yes	No	3.75us*	PC, variables, registers	No	No
Data trace	ITM, DWT, SWO, J-Link	pylink, JLinkExe	No	No	None	PC, 4 global variables	Yes	Yes
Printf SWIT	ITM, SWO, J-Link	pylink, JLinkExe	No	Yes	20.7us	PC, variables	No	No
RTT	J-Link	Telnet, J-Link RTT tools	No	Yes	n/a	PC, variables, registers	No	No
Instruction trace	ETM, ETB, JTrace	GDB**, pylink	No	No	None	PC, instructions	Yes	No

* This is the time impact when using 2 GDB clients as described in 3.2.2

** GDB means we use the JLinkGDBServerCLExe in combination with gdb-multiarch

*** SWD is required for all features except for printf SWIT

`flash.write32` can be used in `pylink`. In GDB the command to read or write is `memU32`. To read out the memory with the least impact on execution time, two clients are connected to the server. Then one client will issue a `continue` command and the other one will issue a `memU32` command to read out or write into the memory. This will stop the running target for a minimal amount of time only. Another option is to use GDB trace points which implement the same functionality of halting, reading and continuing.

- + It is possible to see the state of all registers and memory or manipulate memory
- It is only possible to use this feature when the target is halted,
- 300 cycles are required to stop read and start again
- When using two GDB clients we do not know the state of the program when reading out the memory

3.2.3 Data Trace

The data trace is also based on the functions of the four comparators in the DWT unit. To use the data tracing feature the DWT and ITM configuration registers have to be set to the correct value beforehand. This can be done in `pylink`, `JLinkExe` or GDB. Once this is done, the DWT will automatically output a DWT data packet if a specified memory address is accessed (read or write). The ITM will then put a timestamp on the packet and send it to the J-Link module over the SWO line.

The user can specify if a packet should be generated on a read or write access or for both. The packet can also be configured to include the data value, the PC value when the variable was accessed or both. The timestamps can also be turned off and a prescaler of 1, 4, 16 or 64 for the timestamps can be chosen.

The DWT data packets will be encoded according to the DWT packet protocol specified in the ARM-v7-M Architecture Reference Manual [20] and must be parsed by a tool on the observer. An example of a data trace in STM32CubeIDE can be seen in Figure 3.3.

- + This feature is non-intrusive
- + It is possible to trace four data values with PC sample and get cycle accurate timestamps
- It is complex to configure the microprocessor for data tracing
- It is necessary to parse DWT packets and the accurate timestamping of the arriving packets is challenging

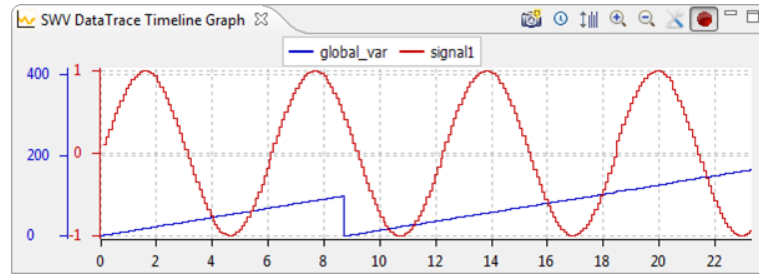


Figure 3.3: A data trace in STM32CubeIDE [21]

3.2.4 Printf SWIT

This feature uses the ITM software tracing capabilities described in the first point in section 3.1.1. The packets coming from the ITM have to be parsed on the observer by J-Link Software or a script using `pylink`.

- + This feature is easy to use and very flexible
- When sending a `printf` statement sending 9 bytes an additional delay of approximately 2200 cycles is introduced (at 80 MHz this is equal to 20.7 μ s). This is still better than sending it over UART requiring approximately 42000 cycles for the same number of bytes (at 80 MHz this is equal to 522 μ s)

3.2.5 SEGGER's Real Time Transfer (RTT)

RTT is a J-Link specific technology that allows to output information from the target microcontroller as well as sending input to the application at a very high speed.

Using RTT reduces the time taken for `printf` to a minimum. It uses background memory access which allows the data and memory to be accessed simultaneously in the background while the CPU is executing its normal instructions. This means that the microprocessor does not lose any cycles transmitting data to the debug probe. However, the software must be instrumented with functions like `SEGGER_RTT_Write` in order to write data into the output buffer which makes RTT an intrusive debugging method. To receive RTT packets on the host it is possible to create a connection to `localhost:19021` with Telnet when connected to J-Link.

- + This feature is very flexible debugging with `printf`-style messages
- It is slightly intrusive because data must be written into the output buffer

3.2.6 Instruction Trace

Instruction trace is a feature that allows the user to record a very detailed trace including all the executed assembly instructions. To make use of this feature a SEGGER J-Trace probe would be required (not available on FlockLab 2 observers).

Instruction trace should not be confused with tracepoints in GDB. The latter feature allows the user to set tracepoints at different locations in the code where the microprocessor will be stopped, some registers read out, and then restarted. To access the instruction trace features either `pylink` or the GDB extension commands from SEGGER together with GDB can be used. Before program execution tracing has to be started and when the microprocessor is stopped, the trace buffer can be read out.

- + This features allows to record a complete execution trace
- It generates a lot of data
- A J-Trace probe is required to get the full instruction trace stream. Reading the ETB will only show the last few instructions executed previously to the halt.

3.3 Distributed Debugging Features

This section describes possible **distributed** debugging features based on the debugging features listed in 3.2. In subsection 3.3.4 the suitability of these features for distributed debugging on FlockLab 2 is discussed.

3.3.1 Halt and start execution synchronously on whole testbed

This feature would allow a user to specify that for example after 60 seconds the execution is stopped on all nodes. Once the program is halted all registers and the memory can be read out on all nodes. Then all the targets are restarted simultaneously.

Another possibility would be that the user can specify in real time when the testbed should be halted and restarted (step through execution).

- + It is possible to read register state and memory content
- This feature is intrusive
- There is some time jitter when halting and restarting the target.
- A user must be present for the real-time option of the feature

3.3.2 Distributed Assertions/Watchpoints

This feature would allow to set an assertion at a specific location in the code that has to be true for a set of nodes or all nodes when the line is executed. Once the assertion fails, all targets would be halted and register and memory content extracted.

- + This feature is very flexible and powerful
- Communication over server is required to inform all nodes
- The delay between the assertion fail and when all nodes are stopped is several hundred microseconds long.

3.3.3 Global Data Tracing

For this feature every node configures the target for data tracing and then collects and possibly parses the DWT data packets. Besides the variable value these packets will also include the address of the instruction that changed the value and a timestamp.

After the test completed, the collected information from every node is sent to the server where the logs could be combined into a single log based on the timestamps.

- + This feature is non-intrusive
- + It is possible to find the function that modified a variable
- Synchronisation of time in local data traces necessary
- Only four global variables can be traced per node
- It is not possible to trace variables that change with a high rate (every cycle) when the PC is also traced. This is because the SWO bandwidth is not high enough to send all the data produced by the DWT to the debug probe.

3.3.4 Selection of a Distributed Debugging Feature

In order to implement the first debugging feature, a mechanism to start and stop all nodes simultaneously would be required. However, it is challenging to implement such a mechanism since even if the stop command is issued by all observers simultaneously, the targets will stop the execution at different times. This is due to the high level nature of Python and scheduling conflicts in the Linux user space.

Distributed assertions or watchpoints are also a powerful feature but not suitable to use with time synchronised applications. This is because it is not possible to stop all targets immediately once an assertion fails. The information about a possible failure has to be sent to the server and then back to all nodes which takes several hundred milliseconds.

The last option is to trace variables globally. This feature is non-intrusive and does not pose the same synchronisation problem as the previously mentioned solutions. It also offers a flexible way of debugging and it is feasible to implement it on the FlockLab 2 hardware. Therefore this option was chosen to implement and test in this semester thesis.

Implementation

As described in section 3.3.3 global data tracing is the most suitable among the possible features and was therefore selected for the implementation on FlockLab 2. Regarding the software, `pylink` was the only tool capable of data tracing that was flexible enough and could be integrated in the FlockLab 2 architecture. This is why the feature was implemented as Python functions using the `pylink` module. The setup of observer and target is illustrated in Figure 4.1.

The data tracing can be split into four basic steps which are all implemented in a separate functions:

1. Configure the target processor for the desired data tracing.
2. Read out the data from the SWO buffer on the debug probe while the target executes the code. This function will be referred to as SWO reader.
3. Parse the SWO data packets.
4. Remove the time jitter in the python timestamps by calculating a regression with the local timestamps from the target.

These functions are described in more detail in sections 4.1, 4.2, 4.3 and 4.4. Section 4.5 describes how the three modules are integrated in the FlockLab 2 architecture.

4.1 Target Configuration

In order for the Cortex-M4 processor to generate DWT data packets, several CoreSight registers must be configured before starting the execution. This is done by using the `pylink` function `memory.write`. The most important register are the Trace Control Register (ITM_TCR), the Comparator register (DWT_COMPn) and the Comparator Function register (DWT_FUNCTIONn). The exact description can be found in the ARMv7-M Architecture Manual, the

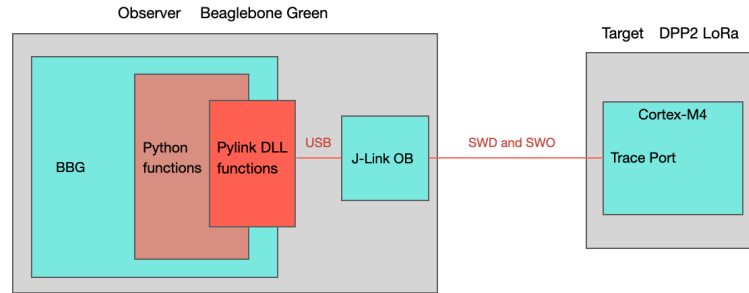


Figure 4.1: Setup for configuration and SWO reading

following explains just the settings of interest for the presented data tracing feature.

ITM_TCR: Bits 0 to 3 in this register are all set to 1 in order to enable the ITM, local timestamping, synchronisation packets and the Trace Port Interface Unit (TPIU). Bits 8 and 9 can be set to 0, 1, 2 or 3 to set a prescalers of 1, 4, 16 or 64 for the local timestamp clock.

DWT_COMPn: The Comparator register provides provides the reference value to be used by the comparator n (0 to 3). In the case of data tracing the function will write the variable's address into this register. For hardware breakpoints a PC value will be written into the DWT_COMPn register.

DWT_FUNCTIONn: The Comparator Function register controls the operation of comparator n. In the case of data tracing seven different functions summarized in Table 4.1 can be chosen.

The settings read, write and read/write define upon which event a data packet is generated. For example when read/write is chosen, the DWT unit generates a packet when the traced variable is accessed with a read or a write operation. The options Data+PC, PC and Data describe what the generated packet should contain. For example when choosing Data+PC, the DWT unit will include the current value of the PC and the traced variable in the generated packet.

4.2 SWO Buffer Reading

After configuring the debugging hardware, the microprocessor is started and the SWO buffer on the debug probe is periodically checked for packets.

Those packets are first written into the the DWT output buffer by the DWT and ITM unit and then sent to the J-Link debug probe at SWO frequency. On the J-Link debug probe they are stored in another larger buffer with a default size of 4 MB. The possible SWO frequencies are 4000000, 2000000, 1945946,

Table 4.1: Possible configuration of the Comparator Function register

Traced values	read	write	read/write
Data + PC	0xe	0xf	0x3
PC	n/a	n/a	0x1
Data	0xc	0xd	0x2

1333333, 1309091, 1014085, 1000000, 972973, 960000 and 808989 Hertz. For the implementation a default frequency of 4 MHz was chosen.

For the communication over SWO either Manchester or Non-return-to-zero (NRZ) encoding can be used. Manchester has less overhead but is not supported by the J-Link debug probe which is why NRZ has to be used [22].

To read from the probe buffer `pylink` provides the two functions `swo_num_bytes` and `swo_read` which return the number of bytes in the buffer and read a specified number of bytes from it, respectively. Section 5.4 describes the use of these two functions in more detail. After every read from the buffer, a timestamp is taken in Python with the function `time`. Since the BBG system time is synchronised on all observers, timestamps from different observer nodes can be compared with each other.

The DWT packets are saved in a file together with the timestamp taken in Python for parsing in the next step after the execution. A typical structure of this file can be seen in Figure 4.2. All the values are written in base 10. The first line starts with a data packet (header 143), which is followed by a local timestamp (header 192) and another data packet and timestamp. The second line is the global timestamp taken in Python. Because the SWO output is a stream of bytes it can happen, that a global timestamp appears in the middle of a packet.

The third line is a special local timestamp packet since it does not come after a data packet. It was generated because of an overflow of the local timestamp counter. The overflow happens at a value of 1999999 cycles which corresponds to roughly 25ms without a prescaler and microcontroller clock frequency of 80 MHz. The remaining lines are similar to the first four.

Besides the timestamp counter, also the DWT output buffer can overflow. This happens when the comparators generate too many packets. The microprocessor will then send a packet with a value 112 to indicate such an overflow. In the case of such a high load it can also happen that the local timestamp packet is sent with a delay to the corresponding data packet. In this case the timestamp header has a value of 208, 224 or 240 depending on the exact situation. More


```

143 13 0 0 0 192 132 232 106 143 14 0 0 0 192 17
1589056542.1242049
192 255 136 122
1589056543.7323976
143 15 0 0 0 192 131 232 106 143 16 0 0 0 192 18
1589056545.136611
192 255 136 122
1589056546.7426412

```

Figure 4.2: The typical structure of the file written by the SWO reading function

details of the buffer overflow limits are described in section 5.3.

Unfortunately, the J-Link module does not generate an interrupt when receiving a DWT packet, therefore the buffer needs to be periodically polled for SWO outputs. This polling frequency can not be arbitrarily high since this would result in 100% CPU usage on the BBG. On the other hand it can not be too low since this would strongly reduce the timestamp precision. Section 5.4 provides a detailed analysis of this trade-off.

4.3 DWT Packet Parser

The parser function reads the content of the file created by the SWO reader and parse the contents. The parser is based on the "Debug ITM and DWT Packet Protocol" specified in the ARMv7-M Architecture Reference Guide [20].

The current implementation supports overflow packets, local timestamp packets, instrumentation packets and hardware source packets from the DWT. The hardware source packets can contain the variable value or the PC value from any of the four comparators (see Figure 4.3). A PC value or variable of type integer always use all four payload bytes. It has not been tested if smaller variables use fewer payload bytes.

The parser takes the data value and PC if available and writes these values together with the corresponding global and local timestamp and comparator ID in a new line in a pandas dataframe (Python). It also indicates if the operation causing the packet generation was a read or a write access (operation) of the variable. Once all packets are parsed, the dataframe is converted into a CSV file that will be used by the next function in the workflow. An example of an output from tracing 3 variables without PC is shown in Figure 4.4.

4.4 Time Correction with Regression

As described in section 4.2, the read function measures the arrival time of packets on the observer by taking a timestamp with the Python function `time`. This timestamp will be referred to as "global timestamp" in the following text. It is useful when comparing the arrival time with timestamps of other observers,

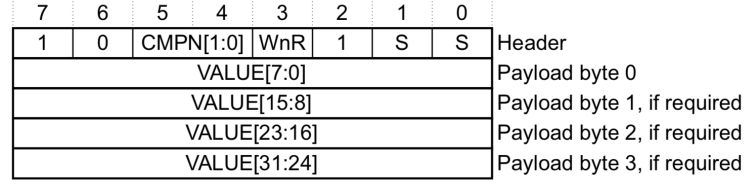


Figure 4.3: Structure of a DWT data packet

global_ts	comparator	data	PC	operation	local_ts
1588843135.83062	0	11		1	55
1588843135.83062	1	309		1	55
1588843135.83062	2	30		1	55
1588843135.83062	2	33		1	58
1588843137.6488	0	12		1	250032
1588843137.6488	2	36		1	58

Figure 4.4: Structure of the CSV file the parser outputs

but it suffers from time jitter. This jitter has two sources: the first one is due to the high level nature of Python and scheduling conflicts in the Linux user space, which leads to unpredictable delays and execution times. The second jitter is due to the delay in the loop. It could happen that a packet becomes available right after the buffer was polled, which means the packet will only be read and timestamped in the next loop iteration after the delay. In order to eliminate the jitter we can use so called local timestamps. These local timestamps are generated by the ITM unit and are sent to the observer over SWO together with every data packet. Their value indicates the difference in cycles since the last timestamp/data packet was sent.

Given the local and global timestamps it is possible to calculate a regression and then project the global timestamps onto it. This removes the time jitter, assuming that jitter is distributed symmetrically around the true value and drift does not change rapidly over time. The regression will have the global timestamps on the y-axis and the summed up value of the local timestamps on the x-axis. This procedure is visualized in Figure 4.5.

This method works mostly well when using the external crystal as a clock source on the target but there are challenges (described in section 5.1) when using the MSI clock. In order to solve these problems two additions were made to the regression calculation:

Outlier correction: An outlier is a data point that differs strongly from the rest of the data. In this case an outlier has a global timestamp that deviates significantly from the expected value due to jitter in Python (see Figure 4.6). The deviation is always positive since an outlier occurs when the timestamp

on the observer was taken much too late. This results in the global timestamp being much larger than the local timestamp which means that the data point lies far above the regression.

Especially when a limited amount of data points is used for the regression calculation, an outlier strongly influences the parameters of the regression resulting in errors in the timestamp correction. To avoid this, outliers must be detected and the regression calculated without these values. After the regression was calculated, also the global timestamps of the outlier point are projected onto it to correct them. The effect of outliers on the regression can be seen in Figure 4.7. The Figure shows a zoom in on the deviation from the regression when outliers are included for the calculation (left) and when they are excluded (right). We can see that when the outliers are included the regression will have an unwanted slope and will not fit the data well.

Local regressions: Due to the drift of the MSI oscillator, it is not desirable to calculate a regression over all data points but rather calculate regressions over smaller intervals (reasons described in 5.1). For this the data points are split up in smaller sets of configurable length and a regression is calculated based on the points in these sets. Then the points in a set are projected onto the corresponding regression. Figure 4.8 gives an illustrative example of how local regressions are calculated with an interval size four.

A drawback of this method is that it may correct timestamps in a way such that the order of events is not preserved. That is, an event B might have a lower timestamp than event A even though it happened after event A. This can happen when event B and event A lie on different regressions. This could be avoided by imposing additional conditions on the calculation of regressions. In Figure 4.8 such a case is illustrative with the data points labeled with A and B. In order to have enough data points for local regressions, the local timestamp prescaler was set to 16. This results in a counter overflow every 0.4 s at 80 MHz clock speed which guarantees at least 150 data points every minute. This rate is sufficient to create regressions small enough to account for the speed of drift change of the MSI clock source.

4.5 Implementation on the FlockLab 2 Architecture

After testing the data trace feature on the observer, the implementation was integrated in the FlockLab 2 architecture. This allows to use the data tracing feature with tests running on FlockLab 2. Figure 4.9 illustrates how the data tracing is integrated. In order to use data tracing with a test run on FlockLab 2, the user has to specify which configuration he or she would like to use in the XML test configuration file. This is done by defining a `<debugConf>` block with the configuration elements shown in Table 4.2 and 4.3. In the `<mode>` element a

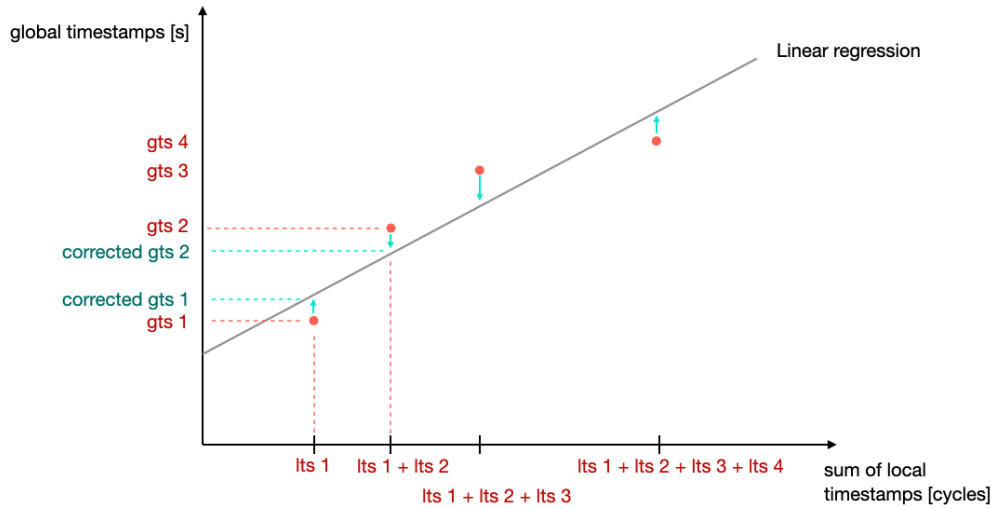


Figure 4.5: The global timestamps are corrected by projecting them onto the linear regression. gts and lts stand for global and local timestamp, respectively

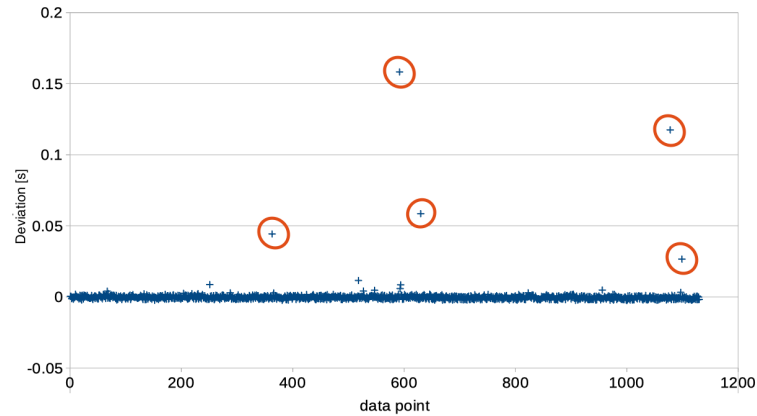


Figure 4.6: Deviation from regression for a 20 min test with the external crystal oscillator

combination of R, W, RW and PC separated by space can be specified. Specifying only R, W or RW will trace the variable upon a read, write or for both operations. When PC is specified in addition, also the PC value is listed in the trace for the specified operation. When only PC tracing is chosen the data tracing is turned off and only the PC value is saved upon read and write operations on the variable. An example of a possible configuration is given in Figure 4.10.

The parser on the FlockLab 2 server will then extract the configuration values from the XML. These values are passed to the observer which will feed them to the configuration function. Before the test is started, the observer calls the

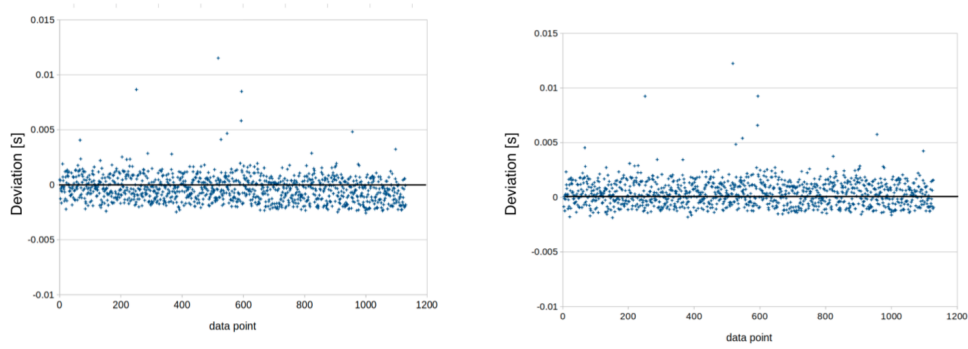


Figure 4.7: Deviation from the regression with outliers (left) and without (right). In the left figure we can see that, towards the end, the deviation tends to be negative instead of being randomly distributed around zero as in the right figure.

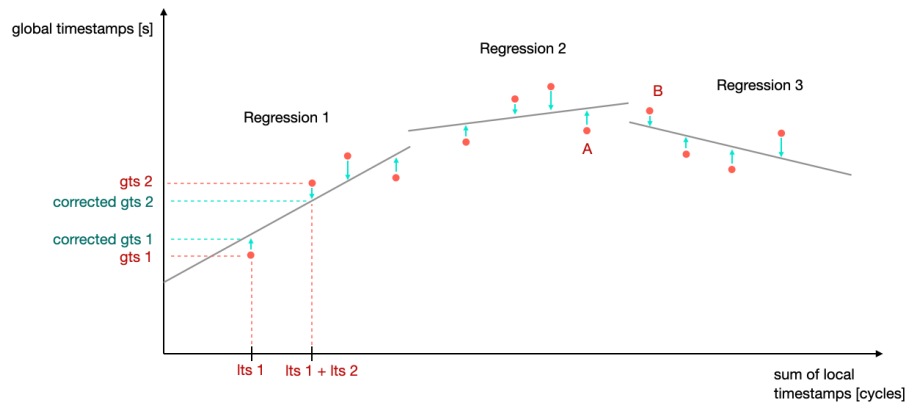


Figure 4.8: Correction of global timestamps by using local regressions. The data points labeled A and B illustrate the problem of mixing the order of events.

SWO-read function which will run as a daemon process during the whole test. After the end of the test, the log file is sent back to the server where it is parsed. In a last step the global timestamps are corrected on the server with the function described in 4.4. The CSV file with the timestamps and variable values can then be downloaded by the user with the other test results from the FlockLab 2 server.

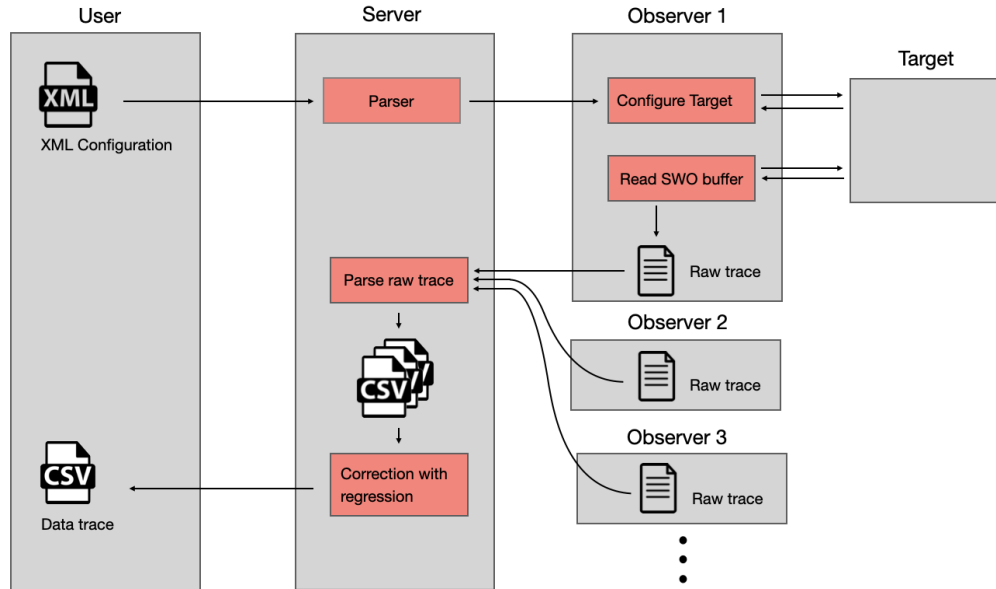


Figure 4.9: Complete workflow on FlockLab 2

Table 4.2: Configuration elements of the `<debugConf>` block. (* for mandatory elements)

Element	Description	Format
<code><obsIds> *</code>	One or more observer IDs for which this service should be used	ID(s) of the observer(s) to use
<code><dataTraceConfig> *</code>	1-4 blocks containing configuration for a variable	See Table 4.3

Table 4.3: Configuration elements of the `<dataTraceConfig>` block. (* for mandatory elements)

Element	Description	Format
<code><variable> *</code>	The name of the variable to trace	Name of the variable
<code><mode> *</code>	The desired access modes of R and W, and whether the PC should be traced	The desired mode separated by a space: R, W or RW and PC if PC tracing is chosen additionally. Only specifying PC sets the mode to RW and traces only the PC

```

44     <debugConf>
45         <obsIds>12</obsIds>
46         <dataTraceConf>
47             <variable>counter</variable>
48             <mode>W</mode>
49         </dataTraceConf>
50         <dataTraceConf>
51             <variable>status</variable>
52             <mode>PC</mode>
53         </dataTraceConf>
54     </debugConf>

```

Figure 4.10: A Possible configuration to trace the variable "counter" upon a write operation. Additionally, upon a read or write operation on the variable "status", the current PC value is recorded.

Performance Measurement

In this chapter the performance of the data trace debugging service is evaluated. The metrics include accuracy of the timestamps on data packets, CPU usage and maximal supported frequency of variable updates. Every section is divided into the subsections Methodology, Results and Discussion.

5.1 Influence of target clock source on the timestamp accuracy

As described in section 4.4 it is possible to calculate a regression based on local timestamps from the target and global timestamps taken on the observer in order to eliminate jitter in the global timestamps. In this part it is evaluated how well this method works when using the MSI and external crystal on the target. (For information on these clock sources refer to section 2.4).

5.1.1 Methodology

In order to determine how well the regression fits the recorded values, several tests with duration between 30 and 60 minutes and different clock sources were conducted on a FlockLab 2 observer. For all tests a single variable that was increased with varying random frequency was used. The prescaler for local timestamps was set to 64 (average error of 64 cycles). The settings for the delay in the read loop and the clock source were varied for different tests. Namely, delays of 2, 10 and 100 ms were chosen in combination with the use of the MSI and the external crystal oscillator.

To analyze the regression the recorded trace from the experiment was parsed and then a regression was calculated in LibreOffice Calc. The datapoints are the global timestamps (y-axis) and local timestamps (x-axis) reported by the parser. Then the residuals were determined and plotted.

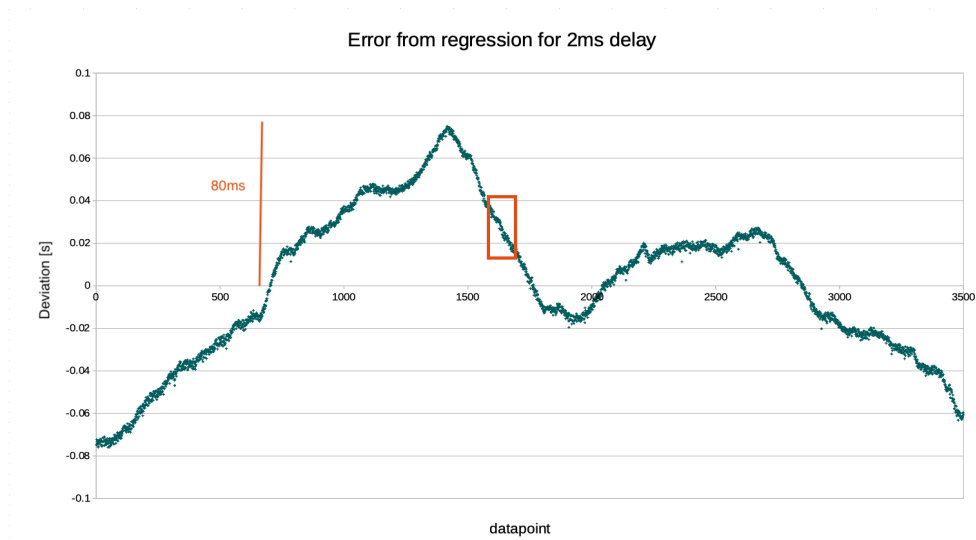


Figure 5.1: Deviation from the regression for 2 ms delay in read loop and a 60 min test (MSI as clock source)

5.1.2 Results

The results for a delay of 2, 10 and 100 ms when using the MSI are depicted in Figures 5.1, 5.2 and 5.3, respectively. The details marked with an orange rectangle are shown in Figure 5.4.

The results from tests with the crystal are shown in Figure 5.5. From the MSI test results we can clearly see that there is a varying drift during the test. It causes the residuals to be as high as 130 ms. This drift is independent of the delay in the SWO read loop. When look at the details of the traces in Figure 5.4 another distribution, depending on the delay, can be seen. This smaller jitter strongly depends on the chosen delay. The maximal difference between two residuals is always very close to the delay.

5.1.3 Discussion

The results let us conclude that the MSI clock used for this test has a strong drift. The smaller jitter is most probably due to the jitter of the SWO read loop and the timestamping. This claim is supported by the results from tests with the crystal. We can see that there the stronger, global drift is eliminated and the jitter remains the same.

Given these results we can conclude that it does not make sense to calculate a global regression when the MSI is used. The result would be that the global timestamps are corrected by up to 130 ms when projecting them onto the regres-

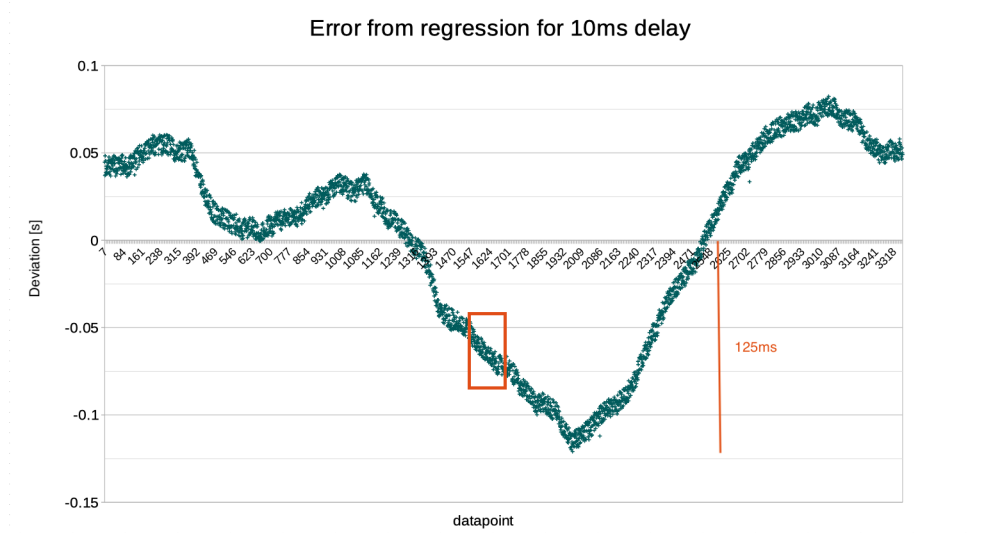


Figure 5.2: Deviation from the regression for 10 ms delay in read loop and a 60 min test (MSI as clock source)



Figure 5.3: Deviation from the regression for 100 ms delay in read loop and a 45 min test (MSI as clock source)

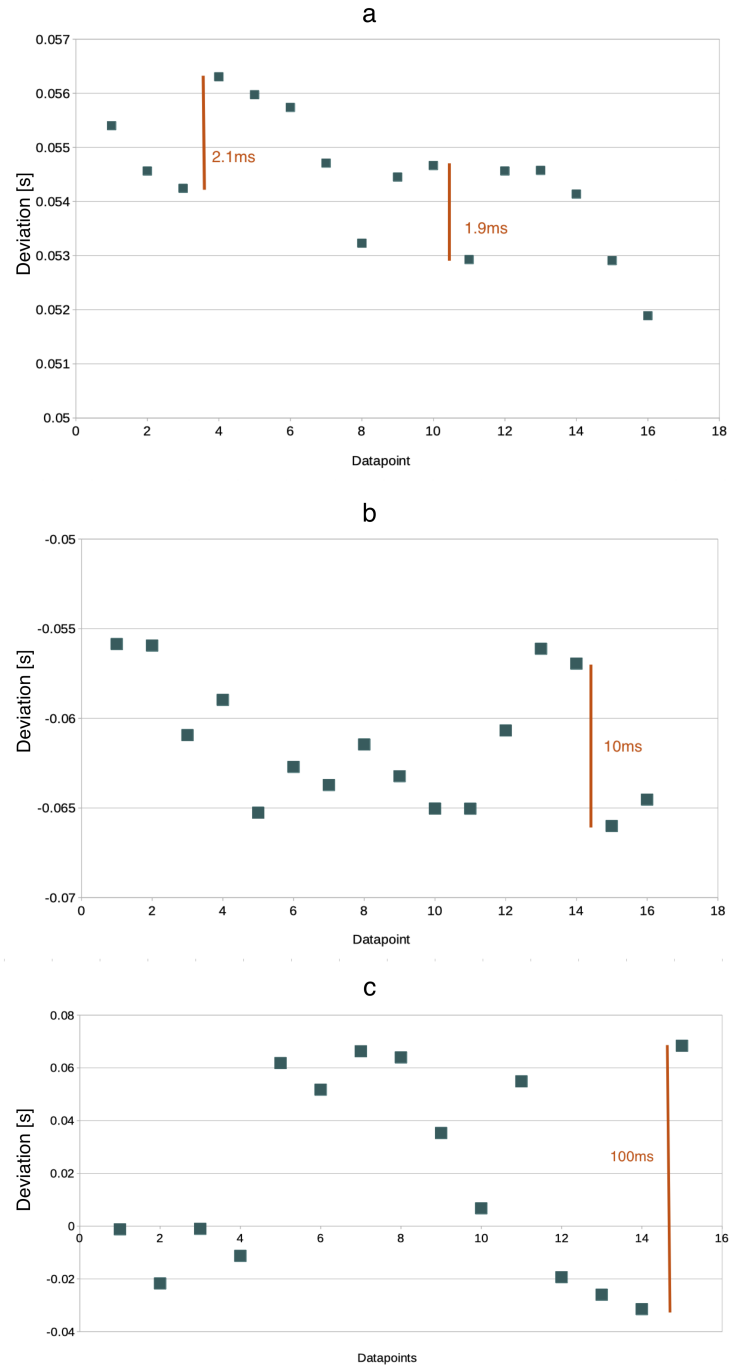


Figure 5.4: A zoom-in on the Figures of the deviation from the regression for a delay of 2 ms (a), 10 ms (b) and 100 ms (c)

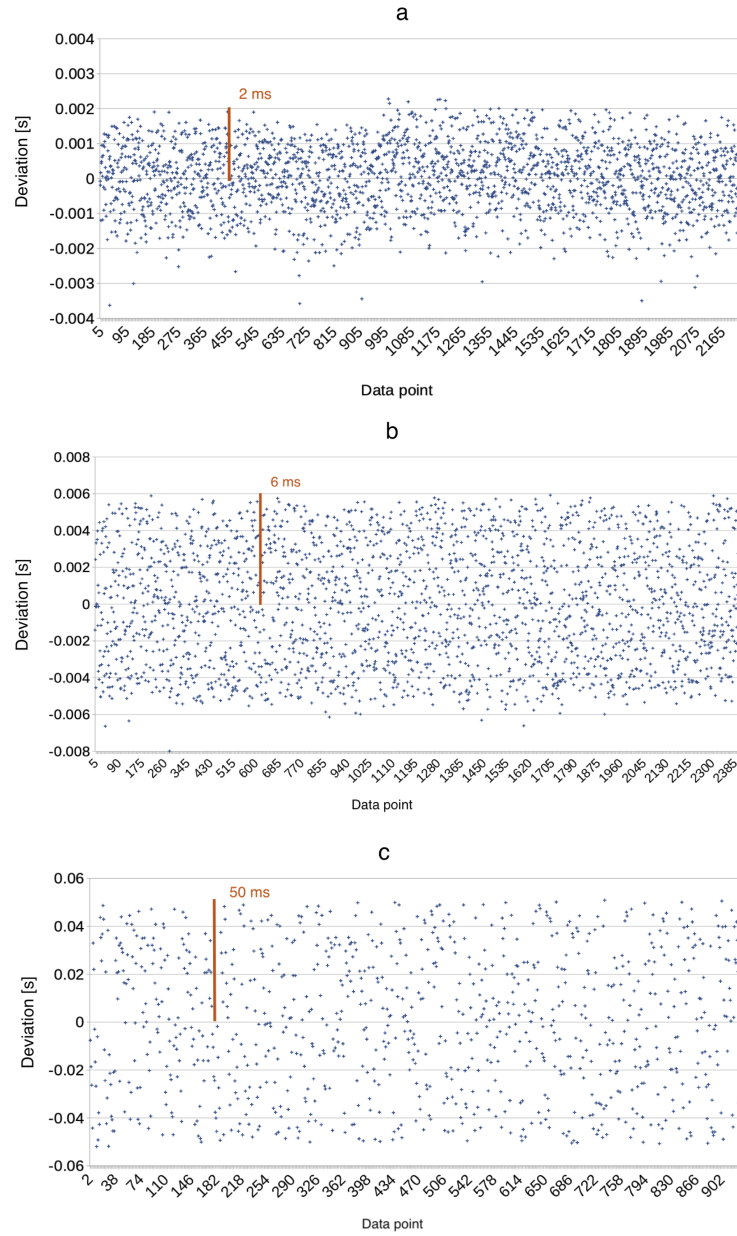


Figure 5.5: Deviation when using the crystal and a delay of 2 ms (a), 10 ms (b) and 100 ms (c)

sion. This projection would correct the MSI drift but not the jitter. In order to only remove the jitter from the global timestamps we would have to calculate a local regression based on a subset of datapoints. The performance of this method is determined in the next section 5.2.

5.2 Accuracy of the Corrected Timestamps

5.2.1 Methodology

In this test, the accuracy of the corrected timestamps of variable changes was measured. For this a GPIO pin was toggled right before an incrementation of the traced variable (see Figure 5.6). This generated a GPIO trace that was recorded and timestamped on the FlockLab 2 observer. These timestamps served as ground truth for the timestamps on the data packets from the debugging. Theoretically the timestamps should differ less than a microsecond since in the program code running on the microcontroller the GPIO event and variable incrementation happens within less than 0.25 microseconds.

In order to have realistic test conditions, a random delay in a range of 10 ms to 1.2 s between variable increases was used in the code running on the target. For the tests the MSI and crystal were used in combination with delays of 2, 10 and 100 ms in the read loop. When the MSI served as a clock source several local regressions were calculated in addition to the outlier correction. When using the crystal oscillator only outlier correction was done.

In order to understand how the regression corrects the timestamps, the difference between uncorrected and corrected timestamps and the difference between uncorrected timestamps and the GPIO pin timestamps was plotted. It is expected that the first difference is similar to the latter difference. This would mean that the correction of the timestamps will eliminate the error between uncorrected and GPIO timestamps.

```
91 while (1)
92 {
93     LL_GPIO_TogglePin(FLOCKLAB_GPIO_PORT, FLOCKLAB_INT1_PIN);
94     counter++;
95
96     random_divider = rand()%20+2; //goes from 2 to 22
97     delay(SystemCoreClock / random_divider);
98 }
```

Figure 5.6: The loop used to compare the timestamps reported by the debugger and the real time of change in variable value

5.2.2 Results

Crystal Oscillator as a Clock Source:

In Figure 5.7 it is visible that the two differences are very similar apart from an offset and a drift. This means that in the corrected timestamps we expect offset and a drift when comparing them to the actual time of the GPIO event. This assumption is confirmed when we look at the difference between corrected timestamps and GPIO event (Figure 5.8). This Figure also shows that the drift is similar for different delays (2 ms left, 100 ms right), meaning that it is possible to use a higher delay with lower CPU usage on the BBG and still get a good timing accuracy. The drift is 1 ms and 2 ms over a test duration of 20 and 30 minutes, respectively. The offset of the error is 5.2 and 3.2 ms for read loop delays of 2 ms and 100 ms, respectively.

MSI Oscillator as Clock Source:

The error between corrected timestamps and the GPIO timestamp for various regression interval sizes can be seen in Figure 5.9. Note that the scale of the y-axis varies between the subfigures. The results show errors of 150 ms, 20 ms and 14 ms for different interval sizes. The average error without correction is at around 12 ms but outliers are present.

5.2.3 Conclusions

Crystal Oscillator as a Clock Source:

The **drift** in error is most probably due to the slight drift of the crystal oscillator (1 ms on 20 min). It is possible to reduce the drift by approximately 20% when using local regressions with outlier correction. On the other hand, local regressions could lead to the timestamps not being in order anymore. Therefore it might be better to accept a slightly bigger drift but have consistent timestamps.

The **offset** most probably originates from the communication delay between target and debug probe and observer. The higher offset for a higher loop delay can be explained by the fact that the buffer contains more data after a longer delay and therefore more time is required to transfer the content. Additional tests are required to determine the exact dependence of the offset on the read loop delay, packet frequency and test duration.

MSI Oscillator as Clock Source:

It is clear that by using smaller intervals, the error can be reduced by an order of magnitude. The drawback of this method is that it may lead to order inversion for timestamps of adjacent regression intervals. However, this could be avoided by imposing boundary conditions on the calculation of regressions. Another approach would be to just use the uncorrected timestamps. The problem with this

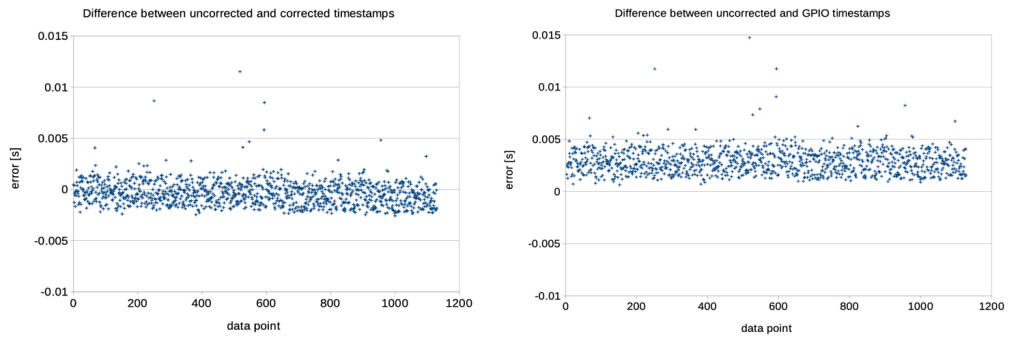


Figure 5.7: Difference between uncorrected and corrected (left) and difference between uncorrected and GPIO timestamps (right)

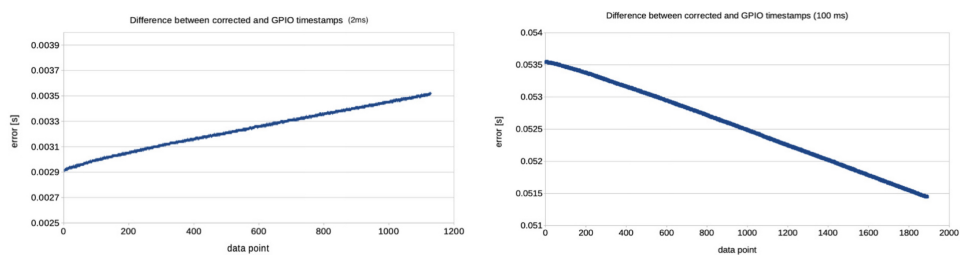


Figure 5.8: Difference between corrected and GPIO timestamps for delays of 2 ms (left) and 100 ms (right)

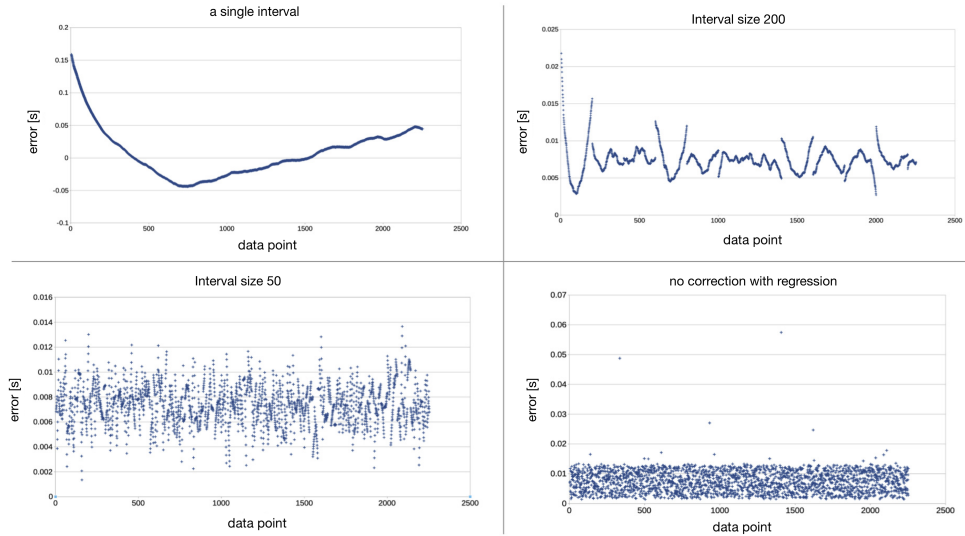


Figure 5.9: The error between corrected and GPIO timestamp for interval size of all values, 200, 50 and without correction

method is that outliers are not corrected and that for higher loop delays of for example 100 ms the error will increase to approximately 100 ms (here 10 ms were used which is why the error is approximately 10 ms). To conclude, either further conditions on the regression have to be introduced or errors in the order of the loop delay have to be accepted when using the MSI as a clock source.

5.3 Maximal Tracing Frequency

5.3.1 Methodology

In this section it was tested how fast a variable can be updated repeatedly without losing packets. Basically, the SWO frequency determines at which speed packets from the DWT unit can be sent to the J-Link debug probe. When they arrive faster at the DWT output buffer, it will overflow and packets will get lost. In order to predict and understand the results we must have a model of the flow of DWT data packets:

The source of the packets are the four comparators in the DWT unit that send

the generated packets to the DWT output buffer. Buffer overflows can either occur due to one comparator generating too many packets or all of them combined, it does not make a difference. Based on the conducted experiments it is assumed that when the buffer is full and another packet arrives, the last packet in the buffer is overwritten and an overflow packet (0x70) is generated to indicate the it.

In the test the maximal variable update frequency for the three SWO frequencies 4MHz, 2MHz and 1MHz were experimentally determined and compared with the theoretical maximal speed. For the theoretical maximal speed calculation the typical size of packets was first observed in a series of tests. These showed that there is always three data packets, each of size 5 bytes, and then a timestamp packet of size 2 bytes. This adds up to a total of 136 bits. Between the data packets there would be a period that we denote as δ , so a total of 136 bits during 3δ . Given the SWO frequency in bits per second we can determine the minimal period δ necessary between the packets:

$$\delta \geq \frac{136}{3 * f_{SWO}} \quad (1)$$

For the experimental determination of the limit, a loop with 10000 increases of a traced variable was implemented in C and flashed on the target. The period between the loop iteration was first set to a value of 1 ms and then decreased in every iteration until an overflow occurred. To detect overflows the log file was scanned for overflow packets with the parser script.

5.3.2 Results

In Table 5.1 the theoretical value calculated with formula (1) and the experimentally determined minimal δ are listed.

Table 5.1: Minimal necessary period between data packets

SWO frequency [Hz]	4000000	2000000	1000000
Minimum period [cycles] (experimentally)	1024	2019	3998
Minimum period [us] (experimentally)	12.94	25.24	49.98
Minimum period [us] (theoretically)	11.33	22.66	45.33

5.3.3 Discussion

The results of the test show that the achieved minimal period between variable updates is close to the theoretical limit. This confirms the model for buffer overflows. The difference between the theoretical value and the result can be explained by the overhead of the NRZ communication over SWO. The results also show that it is best to use a high SWO frequency if possible in order to loose as few packets as possible.

The maximal frequency for variable updates can be read from the table and is approximately 77kHz. (no PC tracing and $f_{SWO}=4\text{MHz}$)

5.4 CPU Usage

In order to read the traced data from the J-Link buffer, the observer must regularly check if there are new values in the buffer and read them out. This is implemented in form of an infinite loop in the `swo_read_buffer` function. However, we cannot just continuously read from the buffer since this would result in 100% CPU usage. Therefore we introduce a configurable delay with `time.sleep` after every time the buffer is checked. In addition, the read loop is optimized such that the CPU is used as little as possible.

5.4.1 Methodology

For the following test, delays of 2 ms, 10 ms and 100 ms and an efficient and expensive loop design (Figure 5.11) were implemented. In the expensive loop, `swo_read` is always called which costs 1.4 ms on average. In the efficient loop, it is first checked if there even is something to read with `swo_num_bytes` which costs 0.3 ms, but in most cases saves the 1.4 ms from `swo_read`.

It is expected that the efficient loop can reduce CPU usage significantly. But it is also probable that the J-Link connection will put a relatively high load on the CPU for a high SWO frequency of 4 MHz. The CPU usage was determined with the Linux command line tool `top`. The setup is a typical application scenario: We trace the two out of four possible variables that both change value approximately every 1.8s (14 bytes every 1.8s). The DWT is configured to only generate a packet upon a read operation on the variables. The prescaler for the local timestamps was set to 64. This setting will be referred to as average load. The CPU usage was also tested with maximal load. For this scenario the variables are updated every 25ms resulting in a data rate that uses the whole SWO bandwidth of 4 MHz.

```

while running:
    global_time = time.time()
    num_bytes = jlink.swo_num_bytes()
    if num_bytes:
        data = jlink.swo_read(0, num_bytes, remove=True)
        file.write(' '.join(str(x) for x in data) + "\n"
                  + str(global_time) + "\n")
    time.sleep(loop_delay_in_s) # time in seconds to sleep.

```

Figure 5.10: efficient loop

```

while running:
    global_time = time.time()
    data = jlink.swo_read(0, 50, remove=True)
    if data:
        file.write(' '.join(str(x) for x in data) + "\n"
                  + str(global_time) + "\n")
    time.sleep(loop_delay_in_s) # time in seconds to sleep.

```

Figure 5.11: expensive loop

5.4.2 Results

The results of the test can be seen in Table 5.2. The settings SWO 4 MHz and 2 MHz describe the SWO frequency used in the respective test. Avg Load and Max Load refer to the average and maximal load describe in the Methodology section.

Table 5.2: CPU usage for different settings

Delay in read loop	2ms	10ms	100ms
Expensive loop, SWO 4 MHz, Avg Load	64.2%	43.3%	35%
Efficient loop, SWO 4 MHz, Avg Load	56.3%	39.1%	33.1%
Efficient loop, SWO 1 MHz, Avg Load	38.2%	16.1%	8.9%
Efficient loop, SWO 4 MHz, Max Load	79.4%	68.7%	67.4%

5.4.3 Discussion

The results show that using a more efficient loop can save up to 10% CPU usage under average load. However, the usage is still relatively high for any delay. This might be due to the J-Link connection using CPU cycles also for other tasks than reading the buffer. Another observation is that a higher SWO frequency uses substantially more CPU power which confirm that the usage is mostly due to the J-Link connection.

At first it might seem that using a lower SWO frequency is the best solution but as documented in section 5.3, a lower SWO frequency also reduces the maximal tracing frequency.

Given that the CPU usage is lower for a higher delay, it is also desirable to choose a delay of 10 or 100 ms in the read loop. As seen in section 5.2 the delay only

has a minimal impact on the accuracy of the corrected timestamps.

To conclude, we can say that choosing a delay of 100 ms and an SWO frequency of 4 MHz is the best option to have a low CPU usage and a high tracing frequency.

Future work and conclusions

6.0.1 Future work

This section lists all the limitations of the implemented data tracing feature and proposes future improvements. Most limitations are directly related to the limitations of the debugging features of the Cortex-M microprocessor. A note is added if it is possible to improve a limitation by software.

- The maximal number of global variables that can be traced simultaneously is four.
- The maximal frequency for variable updates of all four comparators combined is 77 kHz
- The CPU usage is approximately 30% when using the crystal and 40% when using the MSI as clock source
- The timestamp accuracy when using the crystal has been measured to be below 1 ms for a test duration of 20 minutes.
- The timestamp accuracy when using the MSI has been measured to be below 10 ms
- When using the MSI and running test with a duration of more than 3 minutes it is necessary to do local regressions. The use of a global regression will introduce errors of up to 130 ms in the global timestamp values.
- Only the clock sources MSI and external crystal were tested so far. The two other clock sources might require different regression calculations.
- The parsing for large debug trace files needs be done on the server since it is a computationally intensive task.

6.0.2 Conclusions

In this work, first various debugging solutions including different features of the Cortex-M and different debugging software has been tested. All the advantages and drawbacks of the tested features are documented in a comprehensive overview.

Based on the results of this first part of this work, a solution using `pylink` and the data trace functionality of CoreSight was selected to be implemented on FlockLab 2. The feature is based on four Python functions that configure the target, read data from the SWO line and then parse and correct the timestamps of the output. The result is a data tracing service allowing the user to trace the value of up to 4 variables. Using the PC tracing it is possible to also trace the PC value when one of these 4 variables is accessed with a read, write or for both operations.

The timestamps are taken on the observer which are time synchronised. Therefore a distributed debugging is possible by directly comparing traces from different observers. A time correction scheme based on local timestamps is applied such that negative effects of jitter are mitigated.

The evaluation has shown that variables and PC values can be traced with a maximum speed of 77kHz.

When using external crystal oscillator, it was observed that the timestamp accuracy can be as good as 1 ms for a 30 min test duration and a CPU usage of 30% on the observer.

In conclusion, the implemented data tracing feature is a powerful, completely non-intrusive tool. A first version of a data tracing service has been integrated and tested on the FlockLab 2 architecture.

Appendices

APPENDIX A

Schedule

- **Concept:** Search and read related work and related technical documents. Get to know the FlockLab 2 platform (FlockLab Wiki). Then also analyze the current usage and protocols tested on FlockLab in order to adapt the debugging feature to the user's needs. In parallel start writing the "Related work" part in the report. Based on the supported features of the J-Link OB debugger and the needs of the users of FlockLab, generate a list of desired distributed debugging features/services for FlockLab 2.
- **Testing features:** Test the most important debug service proposals from the list of proposals with a Laptop, a J-Link EDU mini debugger and a single device-under-test. Note down important observations and limitations. Based on this, review the desired features and refine one of them for implementation.
- **Implementation:** Implement debugging for a single device on FlockLab first (remote or local debugging on the observer). Then implement the distributed service on FlockLab. Learn how users specify test configurations over the XML file and how it can be adapted to be used for debugging as well. Then also think about a effective way how to record a trace or debugging response that can be included with the other measurements.
- **Testing Performance:** With the provided FlockLab 2 observer hardware, determine the performance limits of the implementation. This can be for example timing overhead or power overhead caused by debugging. Also write a demo program including bugs to best show the use of the feature.
- **Report:** The report will allow a follow-up project to build upon the work, understand the design decisions taken as well as to recreate the experimental results.

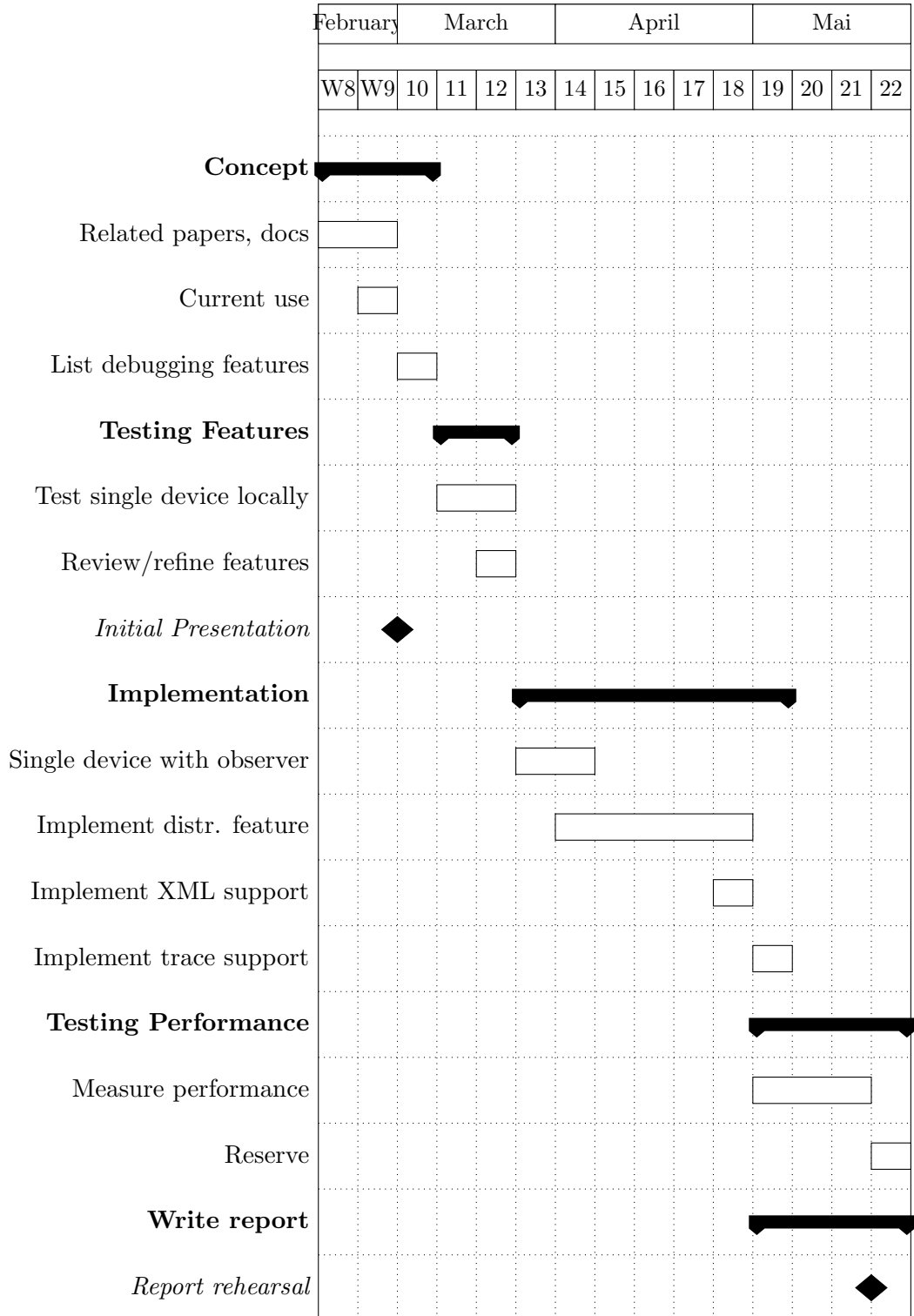


Figure A.1: Schedule for the semester thesis

APPENDIX B

Original Assignment



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Institut für
Technische Informatik und
Kommunikationsnetze

Semester Thesis at the
Department of Information Technology and
Electrical Engineering

for

Lukas Daschinger

Distributed Debugging for FlockLab 2

Advisors: Roman Trüb
Reto Da Forno

Professor: Prof. Dr. Lothar Thiele

Handout Date: 17.02.2020
Official Start Date: 21.02.2020
Due Date: 29.05.2020

1 Project Description

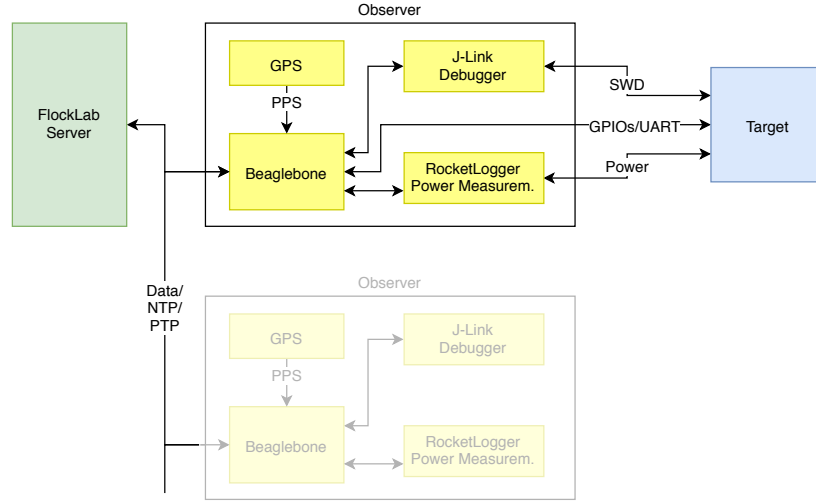


Figure 1: FlockLab 2 Architecture.

Since 2012, the Computer Engineering Group (TEC) operates the FlockLab testbed [7] for developing and evaluating wireless sensor network protocols. A testbed helps to reduce the effort of repeatedly deploying test networks when developing protocols for wireless sensor networks. Furthermore, such a testbed improves the reproducibility of experiments and allows to share infrastructure.

Currently, we are in the process of extending the existing short baseline distances in FlockLab by adding additional nodes on rooftop locations and with significantly larger spacing. The existing implementation of FlockLab is based on hardware components which are no longer in production. Therefore, we developed the next generation of the FlockLab, FlockLab 2. The new architecture is depicted in Figure 1. Among other improvements, FlockLab 2 incorporates a Linux platform with more performance (Beaglebone Green), more precise power tracing based on the RocketLogger [8], as well as state-of-the-art debugging support (J-Link OB).

The J-Link debugger allows to influence the program flow by setting breakpoints and to read out internal state of the microcontroller. Some information can be traced in real-time without halting the processor. In a typical FlockLab test, the devices-under-test (DUTs) interact with each other via wireless communication links. Therefore, it is usually not enough to debug a single node of the network in case a software bug has been observed. It would be useful to debug multiple nodes simultaneously. However, a standardized way/tool to achieve such a distributed debugging does not exist (yet).

2 Project Goals

The goals of this project are:

- Exploration of different debug features of the J-Link OB used on FlockLab 2 in combination with SWD and the ARM Cortex-M architecture.
- Implementation and characterization of at least one debug service for the distributed application scenario of FlockLab 2.

3 Project Tasks

- Formulate a time schedule and milestones for the project. Discuss and approve this time schedule with your supervisors.
- Search and read related work and related technical documents. Some pointers are provided as a starting point:
 - **Related Scientific Work:** [9, 6]
 - **Related Technical Documents:** [3, 2, 1, 5, 4]
 - **Important Keywords:** ARM Serial Wire Debug (SWD), ARM SWO trace port, System Trace Macrocell (STM), Embedded Trace Macrocells (ETM), Instrumentation Trace Macrocells (ITM), ARM Cortex-M, Segger J-Link.
- Based on the supported features of the J-Link OB debugger, generate a list of desired distributed debugging features/services for FlockLab 2 which make use of the J-Link OB in combination with SWD debugging of a ARM Cortex-M4 microcontroller. Examples are:
 - Stop execution when meeting condition X (e.g. a breakpoint) and log the internal processor state
 - Continuously log the value-changes of variable Y
 - etc.
- Test the most important debug service proposals from the list of proposals with a Laptop, a J-Link EDU mini debugger and a single device-under-test. Note down important observations and limitations.
- Select a service proposal from the list of proposals and implement it in a way that it works in the distributed case of Flocklab 2. The interface for the service is based on the existing interface of FlockLab: The user configures the debug service before starting the FlockLab test via the FlockLab test configuration (xml file) and obtains the corresponding results together with the other test results after completion of the FlockLab test. Test the implementation with a sample program which contains artificial bugs.
- Optional: If time allows, more than one service proposal can be implemented.

- With the provided FlockLab 2 observer hardware, determine the performance limits of your implementation.
- Document the project with a written report. As a guideline, your documentation should be as thorough to allow a follow-up project to build upon your work, understand your design decisions taken as well as recreate the experimental results.

4 Project Organization

Deliverables

- Time schedule (2 weeks after start date)
- Initial presentation (3 min)
- Final presentation (15 min)
- Weekly report, which includes: current progress, problems encountered and next steps.
- Code of implementation including documentation
- Final report, which includes: introduction, analysis of related work, documentation of decisions, evaluation, description and HowTo guide of the developed software.

Offers

- The supervisors offer the student the opportunity to do a rehearsal of the initial and the final presentation. The supervisors offer to give feedback how to improve the presentations.
- The supervisors offer to proof-read a draft of the final report. The draft is not required to be complete. The draft should be handed in no later than 1 week before the due date of the thesis.

General Requirements

- The project progress shall be regularly monitored using the time schedule. Unforeseen problems may require adjustments to the planned schedule. Discuss such issues openly and timely with your supervisors.
- Use the work environment and IT infrastructure provided with care. The general rules of ETH Zurich (BOT) apply. In case of problems, contact your supervisors.
- Code versioning is **mandatory** throughout the thesis and the student is responsible for regularly pushing her/his contributions to the repository.

Weekly Meeting

- At the beginning of the thesis, a time slot for the weekly meeting will be agreed on. The weekly meeting is used to discuss the project's progress based on a schedule defined at the beginning of the project.
- The weekly report should be provided at latest at 23:59 on the day before the weekly meeting.

Initial Presentation

Prepare the initial presentation which should include:

- Short introduction (e.g. name, origin, previous studies, current study status, why you are interested in this topic)
- Short description of the project (What do we do, why do we do it and why is it hard)
- Project goals (What do we want to achieve)
- Intended way to reach goals (How do we want to achieve it)

The presentation must not exceed 3 minutes (approx. 3 slides with content). A date for the presentation will be assigned during the project.

Final Presentation

Prepare the final presentation which needs to include:

- Short project description and project goals
- Detailed presentation of the work done
- Detailed presentation of the results
- Conclusion and outlook on possible future work

The presentation must not exceed 15 minutes. A date for the presentation will be assigned during the project.

Handing In

- Hand in a single PDF file of your project report via email. In addition, hand in the signed declaration of originality on paper. A hard-copy of the report is not required.
- Clean up your digital data in a clear and documented structure using the provided GitLab repository. In the end, all digital data should be contained in the student's GitLab repository for the thesis. This includes: developed software, measurements, presentations, final report, etc. An exception are large amounts of measurement data which is stored separately (ask your supervisors!).

References

- [1] ARM® Cortex®-M4 Processor, Technical Reference Manual. https://static.docs.arm.com/100166/0001/arm_cortexm4_processor_trm_100166_0001_00_en.pdf.
- [2] CoreSight Technical Introduction. http://infocenter.arm.com/help/topic/com.arm.doc.epm039795/coresight_technical_introduction_EPM_039795.pdf.
- [3] J-Link / J-TraceUser Guide. https://www.segger.com/downloads/jlink/UM08001_JLink.pdf.
- [4] PyLink (Python Package for J-Link). <https://github.com/Square/pylink>.
- [5] STM32 microcontroller debug toolbox, Appliation note AN4989. https://www.st.com/content/ccc/resource/technical/document/application_note/group0/3d/a5/0e/30/76/51/45/58/DM00354244/files/DM00354244.pdf/jcr:content/translations/en.DM00354244.pdf.
- [6] T. Gong, H. Ji, S. Han, T. Zhang, C. Gu, X. S. Hu, and M. Nixon. Demo abstract: A cross-device testing and reporting system for large-scale real-time wireless networks. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 157–158. IEEE, 2017.
- [7] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel. FlockLab: A Testbed for Distributed, Synchronized Tracing and Profiling of Wireless Embedded Systems. In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks, IPSN '13*, pages 153–166, New York, NY, USA, 2013. ACM.
- [8] L. Sigrist, A. Gomez, R. Lim, S. Lippuner, M. Leubin, and L. Thiele. Measurement and validation of energy harvesting iot devices. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE 2017)*, Lausanne, Switzerland, Mar 2017.
- [9] L. Yi, J. Ma, and T. Zhang. Hatbed: a distributed hardware assisted testbed for non-invasive profiling of iot devices. In *Proceedings of the 2nd Workshop on Benchmarking Cyber-Physical Systems and Internet of Things*, pages 13–17, 2019.

APPENDIX C

How To

This is a guide on how to install the required tools and use the developed distributed debugging software on FlockLab 2.

C.1 Software installation

In order to use the distributed debugging tool you first need to install the J-Link software and the `pylink` library. The respective installation procedure is documented in C.2.3 and C.2.4.

Section C.2.1 describes all the steps required to setup an observer for data tracing. Section C.2.2 explains how to use the observer for development.

C.2 Observer

C.2.1 Observer setup

1. Connect to the observer by first connecting to the ETHZ network over the VPN and then using `ssh` to connect to the observer. (Your public key must be configured on the observer)

```
$ ssh -p 2322 flocklab@fl-12.ethz.ch
```

2. Install the J-Link software and `pylink` as described in sections C.2.3 and C.2.4.

3. If parsing is done on the observer install `pandas`:

```
$ pip3 install pandas
```

C.2.2 Observer use

To enable the observer and multiplexer and select the LoRa board (assumed to be connected to target slot 1) use the `tg_ctrl.py` script:

```
$ tg_ctrl.py -e
$ tg_ctrl.py -m 1
$ tg_ctrl.py -s 1
```

To copy a file to the remote location first close all open connections then from folder with file run:

```
$ scp -v -P 2322 BlinkLED.elf flocklab@fl-12.ethz.ch:/
  home/flocklab/ldaschinger
```

To copy files from the remote to local, first disconnect and then run:

```
$ scp -v -P 2322 flocklab@fl-12.ethz.ch:/home/flocklab
  /ldaschinger/swo_read_log /home/user/Desktop
```

To flash an image to the target use the command:

```
$ tg_prog.py --image=[image file] --target=dpp2lora
```

Another option is to use "loadfile" in `JLinkExe`. This however only accepts *.hex files. `JRunExe` does not download the program correctly. It somehow stores the global variable somewhere else. The command would be:

```
$ JRunExe -device STM32L433CC -if swd -speed 4000 <elf
  file to flash>
```

To manually debug an application `JLinkExe` can be used:

```
$ JLinkExe -device STM32L433CC -if SWD -speed 4000 -
  autoconnect 1
```

By typing "?" inside `JLinkExe` the available commands are shown. For example enter "h" and then "s" to halt and step through the execution.

The section C.3 describes how other SEGGER software and GDB can be used.

The serial number of the J-Link module can be found with the following line after enabling the `pylink` command with export:

```
$ pylink emulator -l
```

C.2.3 J-Link software installation

On a computer with a J-Link edu mini debug probe:

In this case download the "J-Link Software and Documentation pack for Linux"

from SEGGER.

On the BBG

In this case download the ARM specific "J-Link Software and Documentation pack for Linux ARM systems" from SEGGER.

C.2.4 Pylink library installation

On the BBG:

1. Install the J-Link software packets as described in C.2.3
2. Install pylink at `./home/flocklab/.local/lib/python3.5/site-packages/pylink` by running:


```
$ pip3 install pylink-square
```
3. Now the DLL library of the J-Link Software package needs to be moved to a place where pylink can find it.
From the folder where the J-Link software is installed (`./opt/JLink.Linux.V662b.arm/libjlinkarm.so`):

```
$ sudo cp libjlinkarm.so /usr/local/lib/
```

4. To be able to use the pylink command from the terminal during this session do:

```
$ export PATH=HOME/.local/bin:$PATH
```

5. Then modify the library.py file as described in point 6. and 7. of the pylink installation on a computer.

On a computer with a J-Link edu mini debug probe:

1. Install the J-Link software packets as described in C.2.3
2. If you want to use pylink with Python 3.7 install the following modules:

```
$ install python 3.7
$ sudo apt-get install python3.7-dev
```

3. install pip and pylink:

```
$ sudo apt install python-pip
$ pip3 install pylink-square
```

It is also possible to use pylink with Python 2.7. For this use pip instead of pip3 to install the package

4. Now the DLL library of the J-Link Software package needs to be moved to a place where pylink can find it.

From the folder where the J-Link software is installed do:

```
$ sudo cp libjlinkarm.so /usr/local/lib/
```

5. Then to be able to use the pylink command in the terminal do:

```
$ export PATH=$HOME/.local/bin:$PATH
```

6. The library includes some errors. We have to change some code in the library.py file to make it work properly.[23] This might however remove the ability to debug several targets from one device running pylink. The file is usually located at /home/user/.local/lib/python3.7/site-packages/pylink/library.py

On line 85 change (version 0.6.1 of pylink)

```
JLINK_SDK_NAME = 'libjlinkarm'
```

to

```
JLINK_SDK_NAME = 'jlinkarm'
```

Furthermore the library might not find the path to the DLL. In this case it is possible to hardcode the path.

Replace the line:

```
self._path = path or self._path
```

with

```
self._path = '/usr/local/lib/libjlinkarm.so'
```

Then try if the library can find the DLL and works properly by connecting the board and J-Link edu mini and running:

```
$ pylink emulator -l
```

7. Only do this part if the library still does not work since it might break the module.

Replace the lines 340 to 353 (version 0.6.1 of pylink) in library.py with

```
self._temp = None
self._lib = ctypes.cdll.LoadLibrary(self._path)
```

Troubleshooting:

If you are unable to use "pip install pylink-square" try to install the -dev version of Python:

```
$ sudo apt-get install python3.7-dev
```

C.2.5 GDB multiarch installation

In order to use GDB with a microcontroller the gdb-multiarch version is required. To install it run:

```
$ sudo apt-get update -y
$ sudo apt-get install -y gdb-multiarch
```

Then launch this version by typing "gdb-multiarch" instead of just "gdb".

C.3 Software Use

In general always try 4000 and 4000000 in the applications as the communication speed when the speed is 4000kHz

Also check if the application sets a default speed that is most probably wrong or if need to specify 4000 or 4000000.

C.3.1 Pylink examples

The "code" folder in the workspace of the project repository includes some example scripts showing how to use pylink to set breakpoints, watchpoints and read data from the SWO line (breakpoints.py, watchpoints.py and SWO.py). The usage of these scripts on the observer is as follows: (The serial number of the J-Link module can be found with the command described in section C.2.2)

```
$ python3.5 script.py <device address> <commands depending on script>
```

C.3.2 J-Link GDB server

The J-Link GDB server application **JLinkGDBServerCLExe** will create a GDB server to which GDB clients can connect to and use the J-Link debugging functionalities. The command line options for the application are specified in chapter 4.1 and 4.5 in the "J-Link and J-Trace" manual.

To start server:

```
$ ./JLinkGDBServerCLExe -device STM32L433CC -if swd
-timeout 5000
```

Then start a client from the directory where the *.elf file is located:

```
$ gdb-multiarch --interpreter=mi
```

In the GDB client issue the following commands or use a gdbinit file:

```
(gdb) file <filename>.elf
(gdb) target remote localhost:2331
(gdb) monitor reset
(gdb) monitor device = <device name>
(gdb) load
```

The JLinkGDBServer supports SEGGER-specific GDB protocol extensions like tracing and reading the SWO pin. These commands can be found in [24] These commands need to be sent over maintenance commands:

```
(gdb) maint packet <SEGGER extension command>
```

Breakpoints and watchpoints:

For watchpoints the variable needs to be global since otherwise it will be on stack and cannot be watched.

Setting a watchpoint is done with watch, rwatch or awatch for a halt when the variable is written, read or for both. The usage is as follows:

```
(gdb) watch [-l|-location] expr [thread threadnum] [
      mask maskvalue]
```

If you want to watch an address you need to reference and dereference it since GDB does not watch constants like addresses directly:

```
(gdb) watch *(int *) 0x600850
```

Reading memory

In order to read from memory in the least intrusive way, launch the execution from one client and use the GDB protocol commands from a second client to read out memory. The command memU32 can also be used to write into memory. For this specify a value as a second argument after the address.

```
(gdb) memU32 <address> [= <value>]
```

The video *JLinkGDBServerCLExe_memory_readout_mem32.mp4* in the video collection shows how two GDB clients can be connected to one server.

SWO tracing with extension commands

Note that the target needs to be configured first, before it will generate DWT and ITM packets. After configuration these commands can be used to receive data DWT packets in GDB:

```
(gdb) maint packet qSeggerSTRACE:GetSpeedInfo:0
(gdb) maint packet qSeggerSWO:start:0x0:0xfa0
```

Let the application run for some time then:

```
(gdb) maint packet qSeggerSWO:stop
(gdb) maint packet qSeggerSWO:read:0x4
```

GDB tracepoints Note that this does requires a J-Trace probe. GDB tracepoints are an intrusive method for debugging.

```
(gdb) (f)trace printf
(gdb) (f)trace *0x2117c4.
(gdb) actions
(gdb) collect $regs,$locals
(gdb) end
(gdb) tstart
(gdb) continue
(gdb) tstop
(gdb) tfind start
(gdb) tdump
```

C.3.3 JLinkExe

To manually debug an application **JLinkExe** can be used:

```
$ JLinkExe -device STM32L433CC -if SWD -speed 4000 -
  autoconnect 1
```

By typing "?" inside **JLinkExe** the available commands are shown. For example enter "h" and then "s" to halt and step through the execution.

C.3.4 JLinkSWOViewer

Use the **JLinkSWOViewer** to receive and parse ITM **printf** messages on port 0:

```
$ ./JLinkSWOViewerCLExe -device STM32L433CC -itmport
  2331 -itm mask 0x1
```

Bibliography

- [1] Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. Flocklab: a testbed for distributed, synchronized tracing and profiling of wireless embedded systems. pages 153–166, 04 2013. doi: 10.1145/2461381.2461402.
- [2] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. volume 2005, pages 483 – 488, 05 2005. doi: 10.1109/IPSN.2005.1440979.
- [3] Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. Twist: A scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. *REALMAN 2006 - Proceedings of Second International Workshop on Multi-hop Ad Hoc Networks: from Theory to Reality*, 2006, 04 2006. doi: 10.1145/1132983.1132995.
- [4] Jing Yang, Mary Soffa, Leo Selavo, and Kamin Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. pages 189–203, 01 2007. doi: 10.1145/1322263.1322282.
- [5] Philipp Sommer and Branislav Kusy. Minerva: distributed tracing and debugging in wireless sensor networks. 11 2013. doi: 10.1145/2517351.2517355.
- [6] T. Gong, H. Ji, S. Han, T. Zhang, C. Gu, X. S. Hu, and M. Nixon. Demo abstract: A cross-device testing and reporting system for large-scale real-time wireless networks. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 157–158, April 2017. doi: 10.1109/RTAS.2017.21.
- [7] Yi Li, Junyan Ma, and Te Zhang. Hatbed: a distributed hardware assisted testbed for non-invasive profiling of iot devices. In *CPS-IoTBench '19*, 2019.
- [8] Jan Beutel, Roman Trüb, Reto Da Forno, Markus Wegmann, Tonio Gsell, Romain Jacob, Michael Keller, Felix Sutton, and Lothar Thiele. The dual processor platform architecture: Demo abstract. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, IPSN '19, pages 335–336, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6284-9. doi: 10.1145/3302506.3312481. URL <http://doi.acm.org/10.1145/3302506.3312481>.
- [9] *CoreSight Technical Introduction*. ARM, 2013.

- [10] Mathieu Poirier. Hardware assisted debugging on arm with coresight and opencsd, 2017. URL https://elinux.org/images/b/b3/Hardware_Assisted_Tracing_on_ARM.pdf.
- [11] Erich Styger. Software and hardware breakpoints, 2012. URL <https://mcuoneclipse.com/2012/07/29/software-and-hardware-breakpoints/>.
- [12] ARM. Tutorial on arm cortex-m series - debug and trace, 2013. URL <https://www.youtube.com/watch?v=Mm-zBVhEnFc>.
- [13] Felix Sutton, Marco Zimmerling, Reto Da Forno, Roman Lim, Tonio Gsell, Georgia Giannopoulou, Federico Ferrari, Jan Beutel, and Lothar Thiele. Bolt: A stateful processor interconnect. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems, SenSys '15*, page 267–280, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336314. doi: 10.1145/2809695.2809706. URL <https://doi.org/10.1145/2809695.2809706>.
- [14] *Crystal Unit NX3225SA*. Nihon Dempa Kogyo Co., 2003.
- [15] *Ultra-low-power Arm Cortex-M4 32-bit MCU+FPU*. STMicroelectronics, 5 2018. Rev. 5.
- [16] *Crystal Unit NX3215SA*. Nihon Dempa Kogyo Co., 2011.
- [17] *BeagleBone Green System Reference Manual*. beagleboard.com, 7 2015. Rev. 1.
- [18] TIK ETHZ Computer Engineering Group. Dpp devboard, 2019. URL <https://gitlab.ethz.ch/tec/public/dpp/-/wikis/Application/DevBoard#dpp-devboard>.
- [19] Mitch C. Etm and itm (swo) trace in cortex-m3 and cortex-m4, 2018. URL https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2018/09/12/etm_and_itm_swo_tr-VB56.
- [20] *ARM-v7-M Architecture Reference Manual*. ARM, 5 2014. Rev. 3.
- [21] Magnus Unemyr. Cortex-m debugging: Oscilloscope style graphical data plot in real-time, 2015. URL <http://blog.atollic.com/cortex-m-debugging-oscilloscope-style-graphical-data-plot-in-real-time>.
- [22] SEGGER. J-link interface description, 2020. URL <https://www.segger.com/products/debug-probes/j-link/technology/interface-description/>.
- [23] Scott Wohler. Jlink instantiation looking for dll, 2018. URL <https://github.com/square/pylink/issues/33>.

- [24] *J-Link GDB Server - GDB protocol extension*. SEGGER, 3 2018. Rev. 0.