

Master Thesis

Niklas Lundberg, inaule-6@student.ltu.se

July 13, 2021



1 Abstract

TODO

Contents

1	Abstract	2
	Glossary	4
	Acronyms	4
2	Introduction	5
2.1	Background	5
2.2	Motivation	5
2.3	Problem definition	6
2.4	Equality and ethics	7
2.5	Sustainability	7
2.6	Delimitations	7
2.7	Thesis structure	8
3	Related work	9
3.1	llvm	9
3.2	GDB	9
3.3	LLDB	9
3.4	probe-rs	9
3.5	gimli-rs	9
3.6	DWARF	9
4	Theory	10
5	Implementation	11
5.1	Debug Library	11
5.2	Debugger	13
5.3	Command Line Interface	14
5.4	Debug Adapter	15
5.5	VSCoDe Extension	16
6	Evaluation	18
7	Discussion	19
8	Conclusions and future work	20

Glossary

debugee The program or machine that is being debugged. 13, 14

GDB The GNU Project Debugger. 9

Acronyms

DAP Debug Adepter Protocol. 8, 16, 17

GUI Graphical User Interface. 8, 16

2 Introduction

Debugging Rust code on embedded system today is not a very good experience for many reasons. One of the big problems is debugging optimized code is often near impossible. That is because the compilers today are very good at inlining the code and removing unused code. This changes the program so drastically in many cases that when trying to debug the code all the original variable can be optimized out. Thus the user doesn't get any useful information from the debugger about the program which makes is near impossible to debug. One of the big reasons why variable gets optimized out is because unoptimized code always push the value of the variable to memory which then can be read by the debugger at anytime. This is not done for optimized code because speed is prioritized and storing the value of a variable that will not be used on the stack is costly. Thus the time that most variable exist is very short in optimized code which results in that the debugger saying that a variable is optimized out.

2.1 Background

TODO

2.2 Motivation

The main motivation is that optimized *Rust* code can be up to 100 times faster then unoptimised code. That is a very large difference in speed compared to other languages like *C* for example, were the optimized code is about 2-4 times faster then the unoptimized code. Thus for language like *C* it is much more acceptable to not be able to debug optimized code because the different isn't that large compared to *Rust*. But in the case of *Rust* code the different is so large that some programs are to slow to run without optimization. This causes the problem that the code can't be debugged because debuggers don't work well on optimized code and unoptimized code is to slow to run. Because of that there is a need for debuggers that can give enough information to debug optimized *Rust* code. Then there is also the argument that relying only on testing the optimized code instead of going through and checking that is is correct is bad. The reason being that there can be extremely many paths that needs to be tested and it is sometimes not feasible to test all of them. In these cases it is less costly to verify that the code is correct then to test every path.

Another large motivation for this thesis is that the most common way of debugging *Rust* code on embedded systems is complicated for a beginner. One of the reason being that it requires two programs being *openocd* and *gdb*, it also requires configuration files which takes some time to understand and configure. This is not very accessible for people that have no experience with these programs and is unnecessarily complicated. The ideal solution for accessibility would be to have a single program instead of two and that it requires less configuring, thus making it as easy as possible for a new person to debug there code.

Most of the debuggers used for *Rust* code are written in other programming languages and there isn't a lot of debugging tools written in *Rust* yet. Thus one of the motivation for the thesis debugger to be written in *Rust* is to contribute with a example of a debugger written in *Rust* to the *Rust* community. This also relates back to improving debugging for optimized *Rust* code because that is a large and hard problem that requires a lot of work to solve and to maintain the solution. One of the most realistic way that will happen is if the *Rust* community around debugging grows and more people contribute with there solutions and ideas.

Another way this thesis contributes to the *Rust* community is by making a library that simplifies the process of retrieving information form the debug information. This is important because it makes retrieving debug information simpler for new developers to start contributing to the *Rust* debug community. Which will hopefully lead to better debugging for optimized *Rust* code.

There is also a economical reason wanting to have debuggers that work well for optimized code. As mentioned before testing all the paths in a program is sometimes not feasible because of the amount of work needed. But verifying that the program is correct and that the implementation is correct is another way to ensure that the program works as intended. It is even sometimes the preferred solution because then the program is proven to work correctly. And a debugger is one of the vital tools needed for confirming that a implementation is correct. Verifying the correctness of a program and the implementation can also be cheaper then testing in those cases where the programs has extremely many paths. Thus improving the tools needed to verify the correctness of the implementation can intern reduce the cost of verifying that the program works as intended. And it can even reduce the amount of money witch companies spend on testing there code. The amount of money spent on testing of code each year is about xxx and thus the potential savings on testing us huge.

2.3 Problem definition

There are two main problem that this thesis tries to tackle to improve the experience of debugging optimized code for embedded systems. The fist problem is about the generation of the debug information, if more debug information can be generated then there is more information the debugger can retrieve an show the user. This is also were the problem starts with debugging optimized code, because debuggers needs the debug information to understand the relation between the source code and the machine code. Thus it is very important that the compiler generates as much debug information as possible, because there is nothing that can be done later to get more information. The first problem then is to look at the different options that can be set in the *llvm* compiler to improve the generations of debug information without impacting the optimisation of the code to much. Speed of the resulting code is still a big priority.

The second problem is looking at the available debug information that the *llvm* compiler generates for optimized code and creating a debugger that utilises that information to the fullest. This problem has two parts to it, the first being

retrieving the needed information from the debug information. This will be the hardest part and is the most important for improving the debugging for optimized code. The second part is to display the debug information to the user in a user friendly way.

The goal of solving these two problems is to create a debugger that gives a better debugging experience for optimized rust code on embedded systems than some of the most commonly used debugger, such as *gdb* and *lldb*. And to inspire further development for debugging tools in the rust community.

2.4 Equality and ethics

TODO

2.5 Sustainability

TODO

2.6 Delimitations

As mentioned one of the main problems for getting a debugger to work for optimized code is getting the compiler to generate all the debug information needed. In the case of the *Rust* compiler it is the *LLVM* library that handles the debug information generation. *LLVM* is a very large project that many people are working on and thus it would be too much work for this thesis to try and improve the debug information generation. Thus this thesis is limited in solving this problem by the current functionality of *LLVM*.

The compiler backend *LLVM* that the *Rust* compiler uses supports two debugging file formats that hold all the debug information. One of them is the *DWARF* format that has been around for a long time and is supported by many compilers and debuggers. The other one is *CodeView* format which is developed by *Microsoft* and has also been around for a long time. To make a debugger that supports both formats would be a lot of extra work that doesn't contribute to solving the main problem of this thesis. Thus it has been decided to only support the *DWARF* format because it has good documentation and an open-source community around it.

The scope of this thesis also does not include changing or adding to the *DWARF* format standard. The main reason is that it takes years for a new version of the standard to be released and thus there is not enough time for this thesis to see and realise that change or addition. Another reason is that even if a new version of the *DWARF* format could be released in the span of this thesis, it would take years before the *Rust* compiler had been updated to use the new standard. Currently the newest *DWARF* version is 5 but the *Rust* compiler still uses the *DWARF* 4 format.

Many of the debuggers today have a lot of functionality to help the user understand what is happening in the program that they are debugging. An example of these functionalities are debuggers that support the ability to go backwards

in the program. Functionalities like this are useful but not contributing much to the main problem of debugging optimized code. Which is that most of the source code variables get optimized away and thus making it extremely hard to understand what is happening in the code. So to keep this thesis focused on the main problem the feature the debugger will have is restricted to evaluating stack frames and the variables present in each frame, the ability to add and remove breakpoints. And the ability to control the program by stopping, continuing and stepping an instruction.

When debugging code on embedded systems the debugger needs to know a lot about the hardware the code is running on. It has to know the size of the memory and the number of registers, it also has to know which are the special registers such as the program counter register and more. There is also the endianness of the values store on in memory and registers so the debugger can convert it to the correct type. Then there is also the type of machine code the processes uses and the instruction mode use. To support all the different microcontroller would be to much work for this thesis. Thus the debugger is limited to work with the *Nucleo-64 STM32F401* card because it is the one that is available. And it will only support the instruction set *Thumb mode* made by *arm*. The debugger will be design to work with other similar microcontroller but to test and grantee that it will work with them is to much work for this thesis.

Another part of this thesis is the interaction between the user and the debugger. Existing debugger like *gdb* both have a *CLI* and a Graphical User Interface (GUI), thus it is up to the user which one they want to use. From a usability perspective the debugger in this thesis should also have both of the option for the user to choose from. A *CLI* is not that much work to implement but a GUI takes a lot of work to implement. Luckily *Microsoft* has made a protocol for debuggers that specifies an adapter that handles the communication between the GUI and the debugger. This protocol is called Debug Adepter Protocol (DAP) and is used by *VSCode*. Thus the scope of the debugger will include implementing the DAP protocol and an extension for *VSCode*.

The *DWARF* format is very extensive and supports a lot of different program languages, the specifications for the different languages are a little different from each other. Because this thesis is about *Rust* code the thesis will only go into detail in how to read the *DWARF* format for *Rust*. The specification for the *DWARF* format is also very good at explaining how the information is structured. Thus a this thesis will not go into all the detail in how to read the *DWARF* format instead it will focus on explaining how the information in the *DWARF* format can be used in combination to get important information that the user wants.

2.7 Thesis structure

TODO

3 Related work

3.1 llvm

TODO

3.2 GDB

One of the most well known and used debuggers is GDB which supports a lot of different programming languages including *Rust* [GDB]. It is also one of the most common debuggers used for debugging embedded *Rust* code together with the tool *openocd*. Thus it is for this reason that the debugger made in this thesis will be compared to GDB to see if it actually made any improvement on debugging optimized *Rust* code.

One of the problems found with running GDB on optimized code is that if the enumerator variant value is optimized out by the compiler but not the value contained in the variant. Then GDB will default to choosing the 0 variant of the enumerator and thus display the wrong value because it is optimized out thus the correct value could be any of the enumerator variants. This is one of the reasons why GDB is not a very good debugger for debugging optimized *Rust* code.

Another problem with GDB on optimized code is that if a variable's value is only stored in a register then the time the value is known is very short because the register will be written over very soon after it is not needed. In these cases GDB will say that the variable is optimized out when the value is not present, but this result can make the user think that the value is totally optimized away and never present in the program, which is not true. The lack of information in these cases makes debugging very hard because the user has no way of knowing if optimized out means that a variable is optimized out completely or if it doesn't have a value at this precise time.

3.3 LLDB

TODO

3.4 probe-rs

TODO

3.5 gimli-rs

TODO

3.6 DWARF

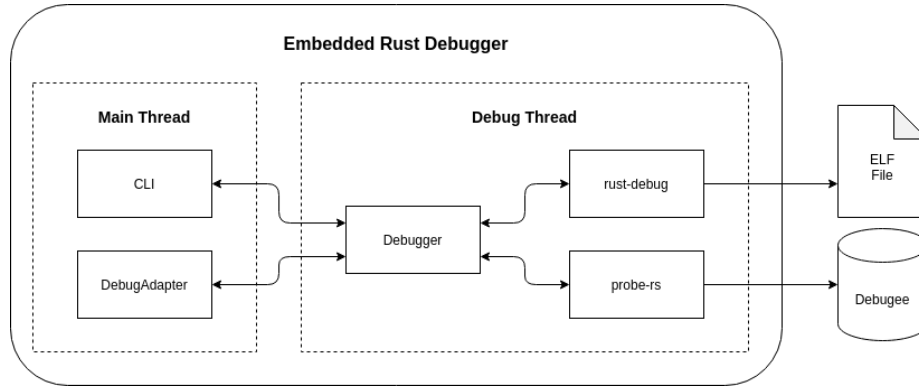
TODO

4 Theory

TODO

5 Implementation

TODO

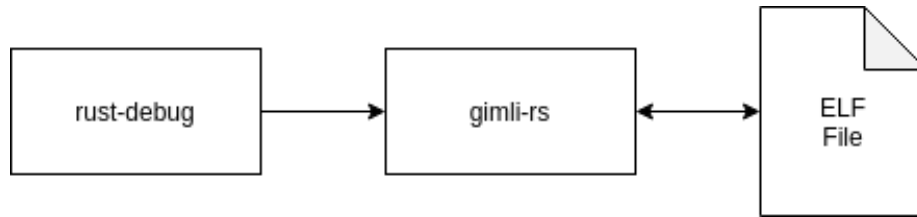


5.1 Debug Library

Retrieving the debug information from the *DWARF* sections in the *ELF* file is one of the main problems that needs to be solved when creating a debugger. The *DWARF* format is made to be space efficient so that the binary file doesn't get too large. This feature of the *DWARF* format has a side effect in that it makes it much more complicated for retrieving the information wanted. Luckily there exists a library called *gimli-rs* that simplifies reading the *DWARF* format. But the library still is very complicated to use because it required a lot of knowledge about the *DWARF* format. Thus the *rust-debug* library was made to simplify this problem even more so that almost no knowledge of the *DWARF* format is needed.

The library *rust-debug* is built upon the *gimli-rs* library and doesn't restrict the user from accessing the functionality of that library. It uses the *gimli-rs* library for parsing the *DWARF* sections into more workable data structures, the figure 5.1 shows how they are connected. The goal of the design of the library is that a user should be able to just call a function for retrieving debug information such as a stacktrace and at the same time be able to use the *gimli-rs* functionality to retrieve the same information if wanted. The library has the features to retrieve the call stack, stackframes, variables, source information and an evaluation function for evaluating the values of variables.

Some of the dies have attributes that starts with *DW_AT_decl*, these attributes describe which source file and where in it this die was declared. The library has a function for retrieving the value of all these attributes and it returns as *SourceInformation* struct. This struct contains the directory, file name, line number and column number where the die was declared in the source files. The line and column number is very simple to retrieve because it is the same as the



value of the attributes *DW_AT_decl_line* and *DW_AT_decl_column*. It is much more complicated to get the directory and file name because the value of the attributes *DW_AT_decl_file* is a file index into the current units line program. The line program indexes all the file entries and can thus be used to find the directory and file name from the file index. If any of these attributes are not present in the die then the result of this function will be that that entry has a *None* value.

As mentioned before one of the requirements for evaluating the value of a variable is access to the registers and memory of the debuggee. This library does not have that functionality because the problem of accessing memory and register is out of scope for this library. The reason being that it would limit the number of systems it could be used for to the systems it supports and also the goal of this library is to make it simpler to get debug information from *DWARF*, and not to provide an interface to the system's memory and register. That led to the motivation of creating a data structure that holds these needed values of the system. The implementation is just a struct with two hashmaps, one for the memory values and one for the register values. It also has some methods for retrieving the stored values. This struct is then used as an argument to the functions in the library and if a value that is not present in the data structure is needed the result of the function call will be a request to add it to the data structure and call the function again. This has a negative effect in that some of the calculations will be repeated multiple times but it is not very noticeable.

The library has a structure called *VariableCreator* which takes a reference to the *DWARF* unit and die of the variable. Then there is an evaluation method that takes the memory and registers data structure and performs calculation to evaluate and retrieve variable information. This method returns an enum that either tells the user what memory or register values are needed, or that it is done and the method *get_variable* can be used to get the variable. The variable structure contains the name, value, type, source location and the location of where the value was evaluated from.

Using the call stack result each call frame can be used to create a stack frame. These stack frames are like the call frames except that they have more information like the function name of the frame, where the function was declared and the value of all the variables. The way this library constructs a *StackFrame* struct works the same way as when creating a variable. Meaning that there is a helper

struct for creating the stackframe which requires the memory and register struct as an argument. This helper struct when created will find the name of the function for this frame and where it was declared, then it will go through all dies that belong to that function to find all the variable dies and store it in a list. This list of variable dies is then used to evaluate all variables and each entry is removed when evaluated and the result stored in a variables attribute. Thus if one of the variables requires a value from memory that is not present in the memory and register struct then all the variables already evaluated don't need to be evaluated again. The evaluating of the variables is done in the same way as describe above the only difference is that the registers values is set to be the ones evaluated in the callframe. Thus adding the value of a register to the memory and register struct won't do anything when evaluating a stackframe. Then lastly the stackframe can be retrieve using the method *get_stack_frame* from the stack frame helper struct.

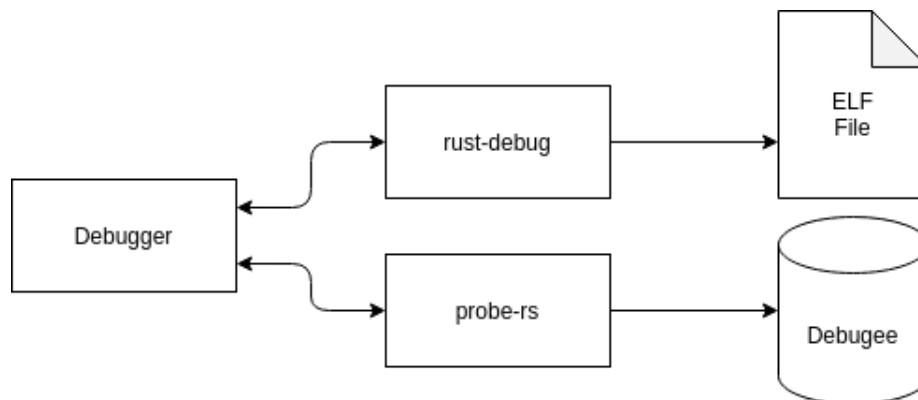
There is also a function for finding out what machine code address a certain line in a source file correspond to. It works by first finding out which compilation unit that the source file correspond to by looping through all the file entries in all of the compilation units. Then it loops through all the rows in that file entry and adds all the rows that match the source line number. If the vector is empty after that then there is no machine code instruction that matches that source line otherwise it is not empty there is at least one instruction that correspond to that line. When there are multiple match the function will take the one that has the nearest column value to the source lines column value.

5.2 Debugger

The debug thread has two main state that it changes between, the first state is when it is not attached to any debuggee and is called DebugHandler. The second state is when it is attach to the debuggee which means that this state is where debugging can happen, it is called Debugger. Going back to the first state, its purpose is to await instructions to attach to the debuggee and to receive configuration required for that to happen. These configurations that the debugger requires is a path to the elf file, a path to the work directory of the code that should be debugged and lastly the type of chip. When all these are configured the attach command can be used to attach to the microcontroller, all the other commands that require that the chip is attach can also be used to attach.

The debugger uses the library *probe-rs* [Yat] to attach to the microcontroller and to interact with it. Thus a lot of useful debugging features like stopping, continuing and setting hardware breakpoints is already given by the *probe-rs* library. The other features supported by the debugger uses the library *rust-debug* together with the *probe-rs* library. But the two library are separate so they never interact with each other, the figure 5.2 show how all of these parts interact. The *rust-debug* library as mention above and seen in the figure 5.2 is a library for retrieving information from the *DWARF* sections in the *ELF* file. To get some of the information from the library values in the debugees memory and/or registers are needed. Thus when the *rust-debug* library gives a response

to a request that it needs some specific value the *Debugger Rust* struct can use *probe-rs* to read those values. Then those values can be sent in as an argument of the *rust-debug* library. This repeats until the *rust-debug* library returns the requested value or an error.



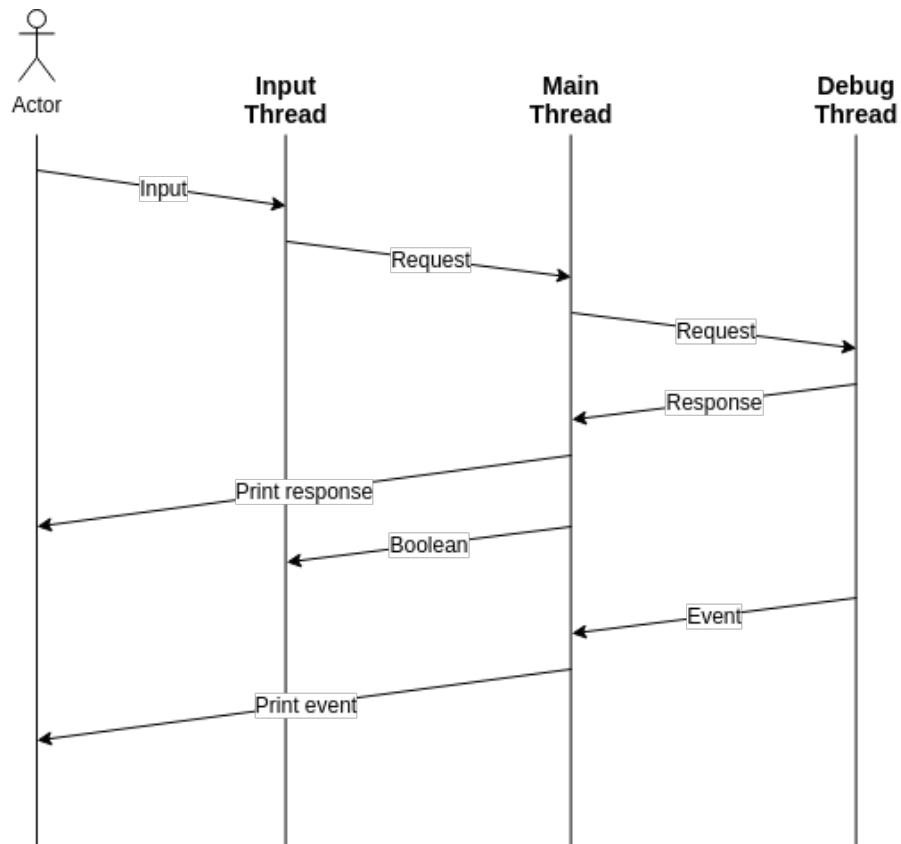
To simultaneously handle incoming requests from the user and events that happen in the debuggee, the debugger polls the channel for incoming requests and the state of the debuggee. To reduce the amount of polling of the state of the debuggee, the debugger has a boolean that keeps track of the state of the debuggee. This boolean is to track if the debuggee is running or is stopped. And because events can only occur if the debuggee is running, the polling only needs to be done when it thinks the debuggee is running. Thus this is how the debugger can handle requests from the user and simultaneously handle events that happen in the debuggee.

To improve the performance of the debugger, the debugger stores the value of the stackframes every time they are calculated. This allows for fast repetitive lookups of stackframe information and variables. The stored stackframe values are only stored when the debuggee is stopped and are removed when the debuggee starts. Thus the wrong values will never be shown to the user. Also another feature of the debugger is that if the value of a variable is requested, the debugger will retrieve all the stackframes instead and then search for the requested variable. This simplifies the implementation a lot and will also make the debugger faster when repeated requests are made.

5.3 Command Line Interface

The *CLI* is very simple and works by having a thread that constantly waits for an input from the command line. When an input is given to the thread, it tries to parse the input into a request for the debugger. It parses the input by first comparing the first word of the input to all the commands; if it matches a command, then the rest of the input is parsed by using the specific parser for that command. If the input does not match any of the commands, then an

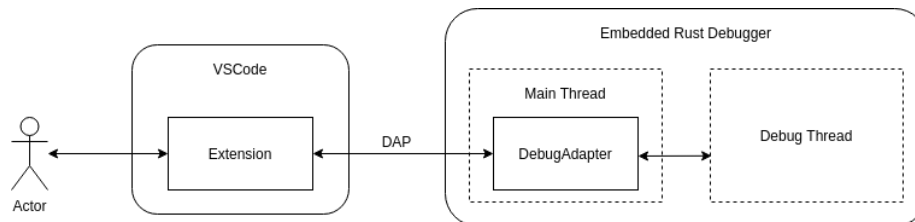
error is printed to the user. When the input has been parsed into a request it is then sent through a channel to the main thread which forwards it to the debug thread. Then when the main thread gets a response back from the debug thread it prints the result to the user and sends a boolean back to the thread that reads the input. The boolean tells the thread if it should continue reading inputs or if it should stop. The main thread constantly awaits a request from the input thread or a response or event from the debug thread. It does this by constantly polling the two channels. If the main thread receives a event from the debug thread it displays it to the user and continues as usual.



5.4 Debug Adapter

The debug adapter is implemented as a *TCP* server that listens for new connections on a specified port. Then when a new connection is made it communicates with the client using the *Microsoft Debug Adapter Protocol*. Looking at figure 5.4 show the flow of communication between the different processes from the user to the debugger. When a user interacts with the debugger extension in *VSCode*,

it will intern send a DAP message to the debug adapter server over the *TCP* connection. The debug adapter will then process the message and translate it to commands that the debugger in the debug thread can understand and send them through a channel to the debug thread. The debug thread will then intern procces the commands and send responsese back to the debug adapter which in tern can translate the reponse and forward it back to the *VSCode* extension. That is the normal flow of comunication but there is another case that can happen. Which is when a event happens on the debugee, this event could be that the debugee has stoped for some reason. In this case the flow of comunication starts at the debug thread which can be seen in figure 5.4. The debug thread then sends the event to the debug adapter wich in tern sends it to the *VSCode* extension, where it is then shown to the user by *VSCode*.



It is required the the first few DAP messages sent to the debug adapter is for configuring the debug adapter by communicating the supported capabilities of the debugger and *VSCode*. This debugger dosen't support any of the optional capabilities that are defined in the DAP protocol and will thus not show up in *VSCode*. After everyting is configured then the debugger will get a requires to flash the connected debugee with the program that is going to be debugged. Now the debug adapter is started and will work as a middle man between the debugger and the GUI.

The debug adapter and the debugger is running on separate threads and thus communicate asynchronously using channels. The chanel uses a diffrent set of commands then that is defined in DAP which means that the debug adapter translates these commands and forwards them. This means that a single DAP message can result in multiple commands being sent from the debug adapter to the debugger.

Because the debug adapter can get messages from both the GUI and the deugger at any time it uses continues polling on the *TCP* connection and the channel. This enables the debug adapter to forwad messages sent by both *VSCode* and the debugger.

5.5 VSCode Extension

The *VSCode* extension for the debugger *Embedded Rust Debugger* is a very simple and bare bones implementation. *Microsoft* provides *API* which can be used to starting a debugging session and trackers for logging what is happening

in the debugging session. The implementation of the extensions uses the *API* to create a tracker that logs all the sent and received messages, it also logs all the errors. It also creates a session that tries to connect to a DAP server on a configurable port or uses a already existing session. Then there is some extra arguments added to the configuration DAP message sent to the debug adapter.

6 Evaluation

TODO

7 Discussion

TODO

8 Conclusions and future work

TODO

References

- [GDB] GDB. *GDB Homepage*. URL: <https://www.gnu.org/software/gdb/>. (accessed: 26.01.2021).
- [Yat] Yatekii. *probe-rs Homepage*. URL: <https://probe.rs/>. (accessed: 26.01.2021).