Master Thesis

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

July 15, 2021

1 Abstract

TODO

Contents

Glossary

debugee The program or machine that is being debugged. 12, 15, 16

GDB The GNU Project Debugger. 9

LLDB A debugger made using libarises from the LLVM project. 9

LLVM The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.. 9

Acronyms

DAP Debug Adepter Protocol. 8, 12, 16–18

GUI Graphical User Interface. 8, 17, 18

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 to problems is to create a debugger that gives a better debugging experience for optimized rust code on embedded systems then 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 lager project that many people are working on and thus it would be to 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 holds 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 a opensource community around it.

The scope of this thesis also dose 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 release 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 functionality 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 endianity 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 form 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

On of the most well known and used debuggers is GDB which supports a lot of different programming languages including *Rust* [GNU]. 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 make in this thesis will be compared to GDB to see if it actually made any improvement on debugging optimized *Rust* code.

One of the problem 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 reason why GDB is not a very good debugger for debugging optimized *Rust* code.

Another problem with GDB on optimized code is that if a variables value is only stored in a register then the time the value is know 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 to 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

The LLDB project is a debugger that uses many of the libraries from the LLVM project. This debugger has the benefit of using the same libraries that the *Rust* compiler uses. It is also one of the most commonly used debugger for debugging *Rust* code. There is a wrapper made for LLDB to give better printing for *Rust* types made by the *Rust* team called *rust-lldb*. But LLDB does not list *Rust* as a support language on there website [Tea].

3.4 probe-rs

TODO

3.5 gimli-rs

TODO

3.6 DWARF

TODO

4 Theory

TODO

4.1 DWARF

The *DWARF* fromat is very complex which makes it hard to work with. This section will go through how the fromat is structured and how the diffrent parts can be used to get debug information. But because the *DWARF* format has a great specification that goes into detail how it is structed and work, this section will only go through the format in a higher level. Skiping much of the detail that is not needed to understand how the implementation of the debugger works, checkout the specification [Com] for more information.

4.1.1 Dwarf Sections

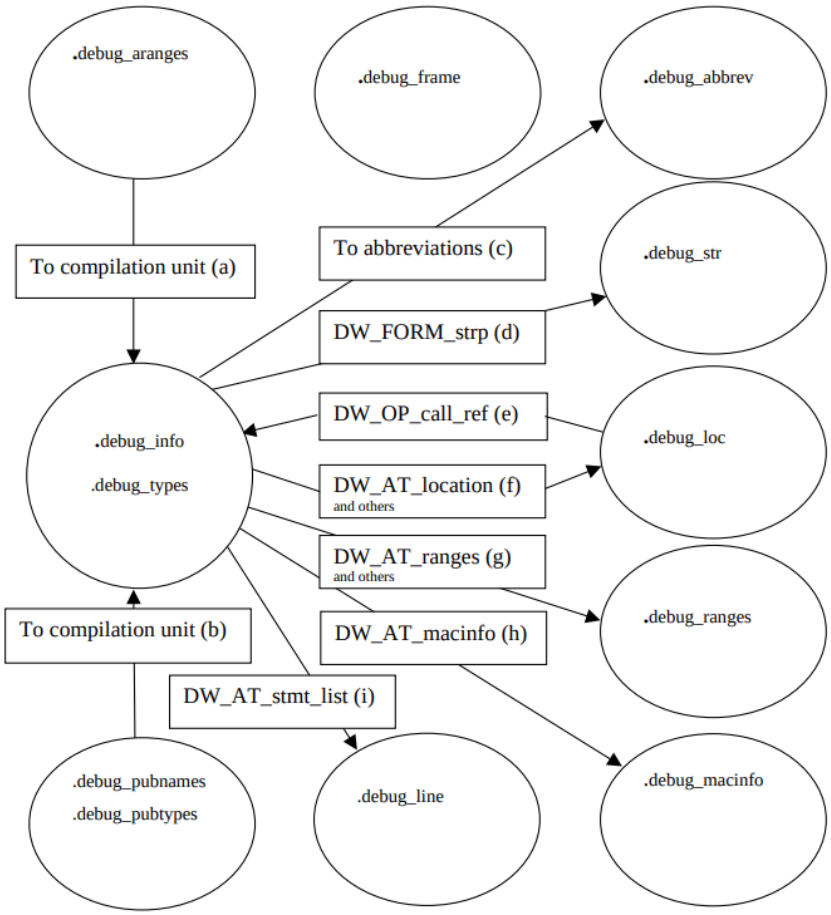
The *DWARF* format is divided into sections in the object file which all contain specific information, these sections use offsets to point to information in another section, see figure 4.1.1. All of the sections can be different from depending on *DWARF* versions and some doesn't exist in the older versions. Thus these explanations only apply to *DWARF* version 4 and some of the older versions, checkout Appendix F in [Com] for more information.

Going in alphabetical order the first *DWARF* section is called *.debug_abbrev*. This section contain all of the abbreviation tables which can be used to find a specific die using the abbreviation. These table entries contain information about the die tag, attributes and if it has children. The library *gimli-rs* simplifies the process of using this table and thus removes the need to know the detail of how to read it, but checkout section 7.5.3 in [Com] to know more.

The *DWARF* section *debug_aranges* is used to lookup which machine address corresponseds to which compilations unit. This address information is stored in ranges where a compilation unit can have multiple ranges. These ranges consists of a start address followed by lenght. Thus to lookup the user only needs to check if the search addres is between the start addres and the start address plus the lengh. To read more about this section checkout section 6.1.2 in [Com].

In the *DWARF* section *.debug_frame* the infromation needed to virtualy unwind the call stack is kept. Unwinding the call stack is complex and is hardware specific but the *gimli-rs* library simplifies the process a lot. This section is completly self-contained and is made up of two structures called Common Information Entry (CIE) and Frame Description Entry (FDE). To learn more of this section checkout section 6.4.1 in [Com]

Infomation about the source code are store in Debugging Information Entrys (DIEs) which are low-level representation of the source code. These have a tag that describes what it represents, an example tag is *DW_TAG_variable* which means that the DIE represents a variable from the source code. All DIEs are stored in tree structure that represents a compilation unit or a partial one. These tree structures are structure like the source program and makes it possible to



relate the source code to the machine code. The section *.debug_info* contains a number of units that all have one of these DIE trees and some more important debug information. Thus this is one of the most important sections in *DWARF* because it is used to understand the relation ship between the source code and the macine code.

The *DWARF* section *.debug_line* holds the needed informtion to find the machine addresses that is generated from a certain line and column in the source file. It is also used to store the source director, file name, line number and column. Then the DIEs will store pointers to the source location information in the section *.debug_line* enabeling the debuger to know the source location of a DIE. The section 6.2 in [Com] explains in more detail how this informatiobn is stored in the *.debug_line* section.

The location of the variables values are stored in location lists, each entry in the list holds a number of operation that can be used to calcualte the location

of the value. All of the location lists are stored in the section *.debug_loc* and are pointed to by DIEs in the *.debug_info* section. The pointers are most commenly found in the attribute *DW_AT_location* which most dies representing variables have. Thre realation between these sections can be seen in the figure 4.1.1.

In the section *.debug_macinfo* the macro information is stored, it is stored in entries that each represents the macro after it has been expanded by the compiler. These entries are also pointer to by DIEs in the *.debug_info* section and those pointers can be found in the attribute *DW_AT_macinfo*. This section is a little complex thus to learn more about it read section 6.3 in [Com].

There are two sections for looking up compilation units by name. The first one is *.debug_pubnames* which is for finding functions and objects. And the other one is for finding types, this section is called *.debug_pubtypes*. Both of these work in the same way and are farily simple to understand, checkout section 6.1.1 in [**fig:debugsections**] for more information.

DIEs that have a set of addresses that are non-contiguous will have offset in to the section *.debug_ranges* instead of having a address range. This offset points to the start of a rangelist that contain range entries which are used to know for which addresses the DIE is used in the program. Checkout section 2.17 in [Com] to learn more about code addresses and ranges.

The section *.debug_str* is used to holds the strings that some of the variable are. Then the DIEs somtimes point to the string in this section as the value of the DIE. The DIEs point to the value using a offset that can be found in *DW_FROM_strp*.

The last section is *.debug_type* it is very similar to section *.debug_info* in that it is also made up of units with each a tree of DIEs. The diffrence is that the DIEs are a low-level representation of the types in the source code insead of a representation of the source program.

4.1.2 Dwarf Compilation Unit

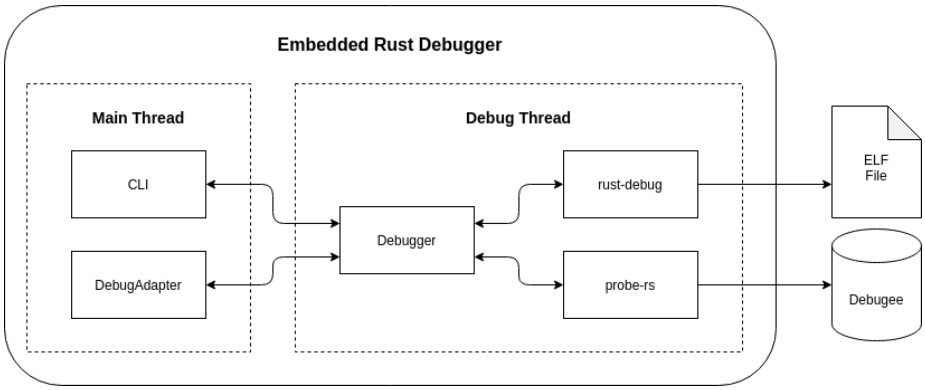
The compiler when compiling a source program will most often generate one compilation unit for each source file, there are some cases when partial compilation units will be made. These compilation units holds most of the debug information generated from the source file. This naturally creates that the debug information in most of the different *DWARF* sections are divided into these compilation units. It also makes it easy for the debugger to find information because the debug information is structured similar to the source code. The main use of these compilation units in a debugger is to first find which unit the program is currently in. Then it is easy to search through the DIE tree in the *.debug_info* section to find the information needed. Finding the right compilation unit can take time but there is the *DWARF* sections *.debug_aranges*, *.debug_pubnames* and *.debug_pubtypes* that provides a fast way of finding it.

The fist DIE in the tree from the section *.debug_info* will have the tag *DW_TAG_compile_unit*. This DIE has a lot of useful debug information about the source file, one being the compiler used and the version of it. It also says which programming language the source file is written in and directory and path

of the source file. The tree is structured in a way that the children of a DIE representing a function will all be decleard inside of the function in the source file. Thus it is easy for the debugger to know everything about the function by going through all of its children.

5 Implementation

The implementations of the debugger is separade into three diffrent smaller projects, the fist being a debug library that simplifies the process of retriving infomation from the *DWARF* format. To learn more about how the debug library project is implemented checkout the subsection 5.1. The second project is to create a debugger using the debug library and the last project is to make a debug adapter extension for *VSCode*. The structure of the debugger can be seen in the figure 5, as can be seen in the figure the debugger uses the debug library *rust-debug* to read the *ELF* file and the probe-rs library for interfacing with the microcontroller being debugged. It can also be noted that the debugger is devided into two separate thread, one for handeling the users input and displaying information. An another one for interacting with the debugee and the *ELF* file. These are not the only threads but they are the most important ones and to simplify this general overviw it can be seen as only two threads. The *Main Thread* is the thread that can eiter be started as a cli that handles the users commands form the command line or as a debug adapter server that handles DAP commands over a *TCP* connection. The main purpos of this thread is to handle input from the user and to display or forwad inormation from the *Debug Thread*. The purpose of the debug thread is to handle requests from the *Main Thread* and then send back a response, it also cheacks the state of the debugee and sends events to the *Main Thread* if the state of the debugee changes. These two threads comunicates using a asyncronus channel that both sides polls to check for any new messages.

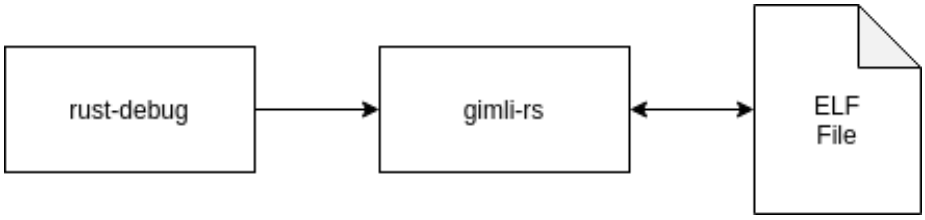


5.1 Debug Library

Retriving the debug infromation from the *DWARF* secionts 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 efficent so that the binaryfile dosen't get to large. This feature of the *DWARF* fomat has a side effect in that it makes

it much more complicated for retriving the infromation wanted. Lukily there exist 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 knowlage about the *DWARF* format. Thus the *rust-debug* library was made to simplify this problem even more so that almost no knowlage of the *DWARF* format is needed.

The libarary *rust-debug* is built apone the *gimli-rs* libarary and dosen't restrict the user from accesing the functionality of that libarary. It uses the *gimli-rs* library for paring the *DWARF* sections into more workable datastructures, the figure 5.1 shows how they are connected. The goal of the desing of the library is that a user should be able to just call a function for retriving debug information such as a stacktrace and at the same time be able to use the *gimli-rs* functionality to retrive the same iformation if wanted. The library is has the features to retive the call stack, stackframes, variables, sourceinfomaiton and a evalution function for evaluating the values of variables.



Some of the dies have attributes that starts with *DW_AT_decl_*, these attrbitues describe which source file and where in it this dies was decleard. The library has a function for retriving the value of all these attributes and it returns as *SourceInformation* struct. This struct contain the directory, file name, line number and column number where the die was decleard in the source files. The line and column number is very simple to retrive 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 pressent in the die then the result of this funciton will be that that entry has a *None* value.

As mention before one of the requirements for evaluating the value of a variable is access to the registers and memory of the debugee. This libarary does not have that functionality because the problem of accesing memory and register is out of scope for this library. The reason being that it would limit the number of systmes it could be used for too the systmes it support and also the goal of this library is to make it simpler to get debug information from *DWARF*, and not to proveide an interface to the systmes memory and register. That lead to the motivaiton of creating a datastructure that holds

these needed values of the system. The implimitation is just a struct with two hashmaps, one for the memory values and one for the register values. It also has some metods for retriving the stored values. This struct is then used as an argument to the functions in the library and if a value that is not pressent in the datastucure is needed the result of the fuction call will be a request ot add the to the datastracure and call the function again. This has a negative effect in that some of the calucations will be repeated mutiple times but it is not very notesable.

The libarary has a structur called *VariableCreator* which takes a refrense to the *DWARF* unit and die of the variable. Then there is a evalution method that takse the memory and registers data structure and proforms caluclation to evalue and retrive variable information. This method return an enum that eiter tell the user what memory or reigster valuse are needed, or that the it is done and the method *get_variable* can be used to get the vairable. The variable structure continas the name, value, type, sourcelocaiton and the location of where the value was evaluated from.

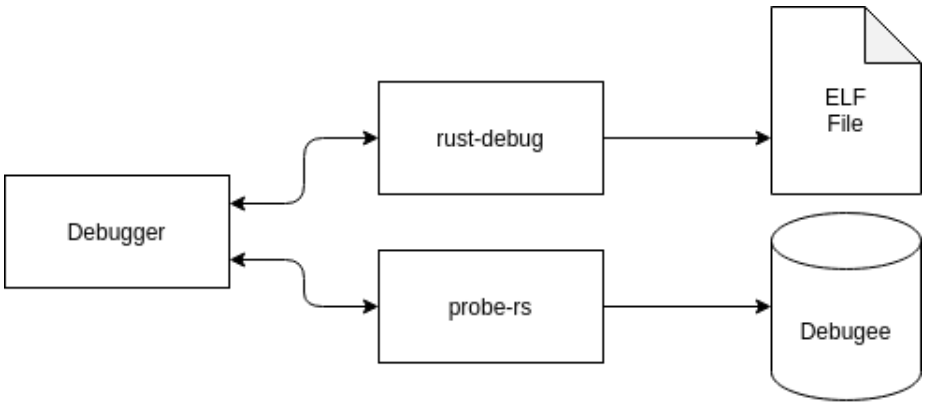
Using the call stack result each call frame can be used to create a stackframe. These stackframes are like the callframes execpt that they have more information like the function name of the frame, where the function was decleard and the value of all the variables. The way this library constructs a *StackFrame* struct works the same way as when creating a variable. Meaning that there is a helper struct for creating the stackframe which requroes the memory and register struct as an argument. This helper struct when created will find the name of the funciton for this frame and where it was decleared, then it will go thorugh 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 echec 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 alrady evaluated dosne't need to be evaluated again. The evaluating of the variables is done in the same way as descirbe above the only diffrense 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 retrive using the method *get_stack_frame* from the stack frame helper struct.

There is also a function for finding out what machine code addres a certain line in a source file correspondse to. It works by first finding out which compilation unit that the source file corresponse to by looping trough 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 atleas one instrcution that corresponse to that line. When there are multiple match the funciton will take the one that has the nearest column value to the soruce 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 debugee and is called DebugHandler. The second state is when it is attach to the debugee which means that this state is where debugging can happen, it is called Debugger. Going back to the fist state, its purpus is to await instructions to attach to the debugee and to recive configureation required for that to happen. These configurations that the debuger requires is a path to the elf file, a path to the work directory of the code that should be debuged and lastly the type of chip. When all these are configure 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 debuging 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 seprate 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 retriving information form the *DWRAF* 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 requiese that it needs some specifig value the *Debugger Rust* struct can use *probe-rs* to read those values. Then those values can be sent in as a argument ot the *rust-debug* library. This repeats until the *rust-debug* library returns the requested value or an error.



To simontainisly handle incoming requiest form the user and events that happens in the debugee, the debugger polls the channel for incoming request and the state of the debugge. To reduece the amount of polling of the state of the debugee the debugger has a boolean that keeps track of the state of the debugee. This boolean is to track if the debugee is running or is stopped. And

because events can only ocur if the debugee is running the polling only needs to be done when it thinks the debugee is running. Thus this is how the debugger can handle request from the user and near simontainisly handle events that happen in the debugee.

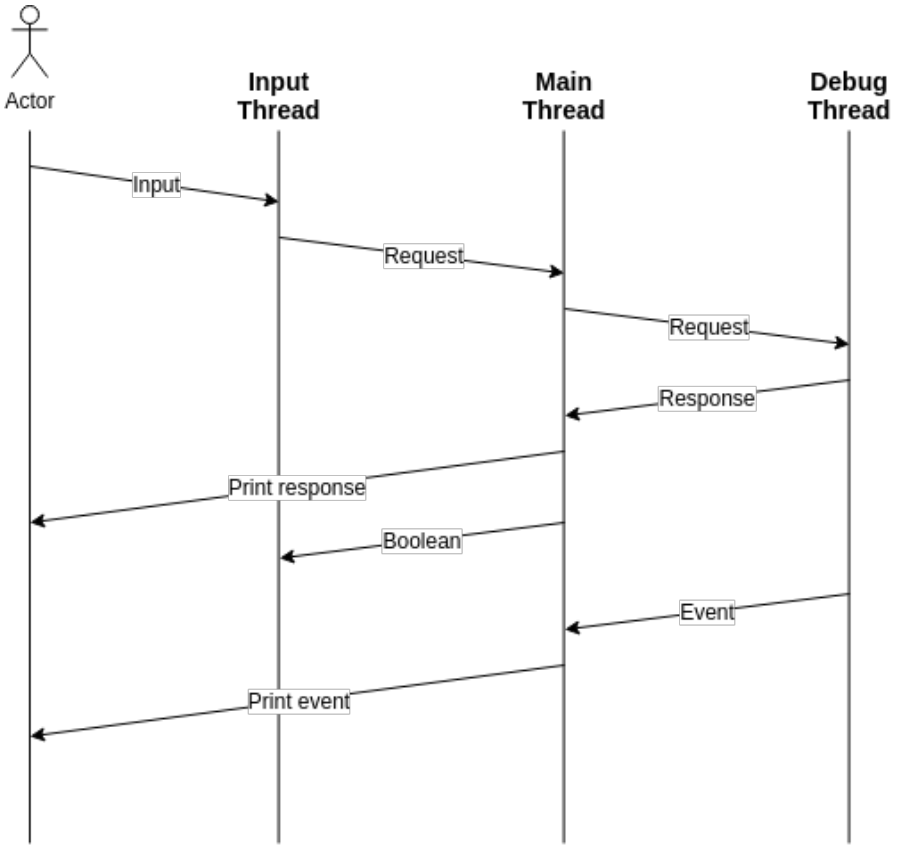
To improve of the preformance of the debugger the debugger stores the value of the stackframes every time they are callcualted. This alows for fast repetetiv look up of stackframe information and variables. The stored stackframe valuse are only stored when the debugee is stopped and are removed when the debugee 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 requeste the debugger will retive all the stackframes instad and then search for the requested variable. This simplifies the implementation a lot and will also make the debugger faster when repeted request 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 parses the input into a request for the debugger. It parses the input by first comparing the fist 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 mach 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 tread constantly awaits a request form 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 event is displays it to the user and continues as usual.

5.4 Debug Adapter

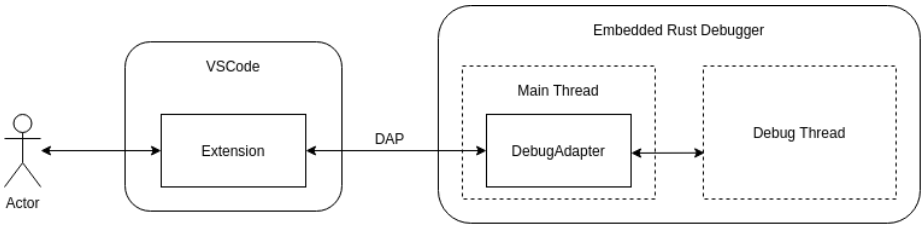
The debug adapter is implemanted as a *TCP* server that listens for new connection on a specified port. Then when a new connection is made it comunicates with the client using the *Microsoft Debug Adapter Protocol*. Looking at figure 5.4 show the flow of comunication between the diffrent processes form the user to the debugger. When a user interacts whith the debugger extansion 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 comunicating 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 debuged. 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 comunicate asyncronosly using chanels. The chanels 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 loging 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

[Com] DWARF Standards Committee. *DWARF Homepage*. URL: `http://dwarfstd.org/Home.php`. (accessed: 26.01.2021).

[GNU] GNU. *GDB: The GNU Project Debugger*. URL: `https://www.gnu.org/software/gdb/`. (accessed: 26.01.2021).

[Tea] The LLDB Team. *The LLDB Debugger*. URL: `https://lldb.llvm.org/`. (accessed: 13.07.2021).

[Yat] Yatekii. *probe-rs Homepage*. URL: `https://probe.rs/`. (accessed: 26.01.2021).