Master Thesis

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

July 8, 2021

1 Abstract

TODO

Contents

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 *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 *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

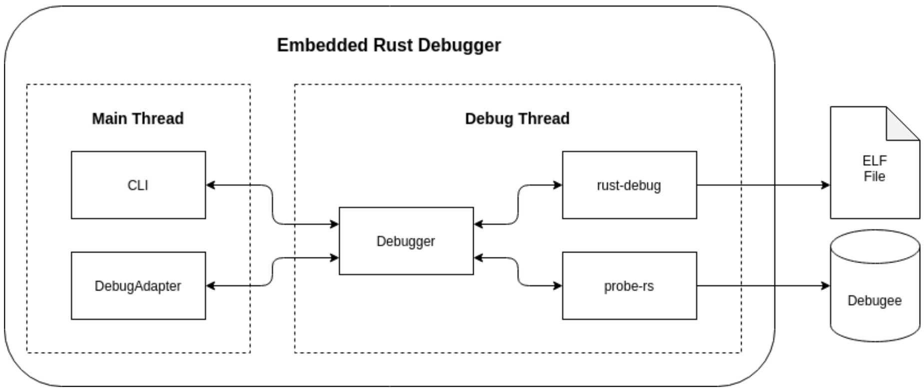
TODO

4 Theory

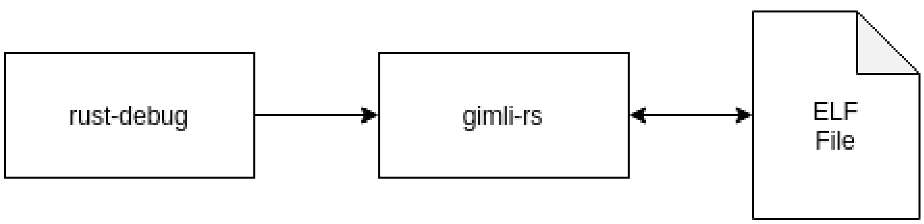
TODO

5 Implementation

TODO



5.1 Debug Library

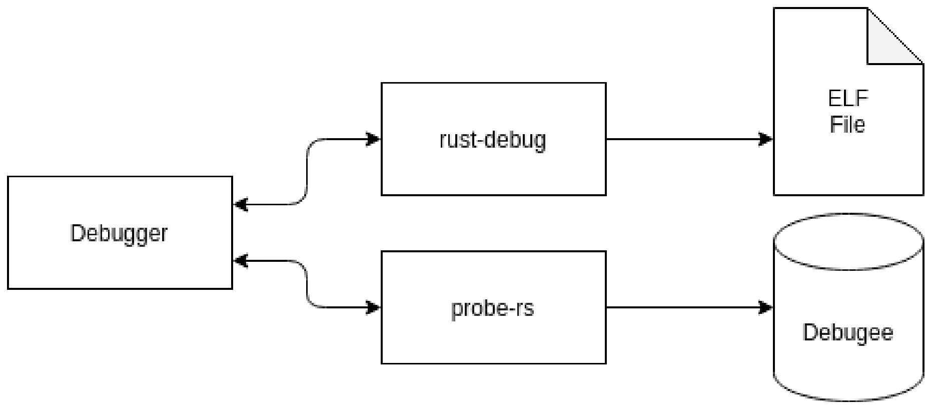


5.2 Debugger

TODO

5.3 CLI

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

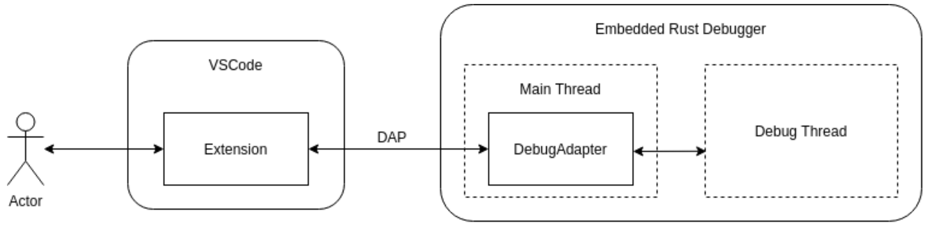


thread it prints the result to the user and sends a boolean back to the thread that reads the input. The boolean tells the thread if it should continue reading inputs or if it should stop. The main tread constantly awaits a request form the input thread or a response or event from the debug thread. It 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