Luleå University of Technology

Master Thesis

Improving Debugging For Optimized Rust Code

Niklas Lundberg
inaule-6@student.ltu.se

supervised by
Prof. Per Lindgren

September 28, 2021

Abstract

Debugging is an essential part of programming. At this moment there is no debugger that is good at debugging optimized *Rust* code. It is a problem because unoptimized *Rust* code is very slow compared to optimized. The ability of debugging optimized code is especially important for embedded systems because of the close relation to hardware. Thus a tool like a debugger is very useful because it enables the developer to see what is happening in the hardware when debugging embedded systems.

To improve debugging for optimized *Rust* code this thesis will research the available compiler settings that affect the generation of debug information. Also a debugger called ERD written in *Rust* will be presented and the goal with it is to improve debugging for embedded systems running *Rust* code. This thesis will explaining the implementation of ERD and the debug information format DWARF.

The debugger ERD will be compared to other well known *Rust* debuggers to see if it improved on the experience of debugging optimized *Rust* code on embedded systems. The comparison shows that it did improve on debugging the *Rust* enum type and temporarily optimized out values. However there is still many improvements that can be made to debugging optimized *Rust* code.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 29

CFA Canonical Frame Address. 4, 20–22

CIE Common Information Entry. 15, 20

CLI Command Line Interface. 4, 9, 26, 28

DAP Debug Adepter Protocol. 9, 26, 28, 30

DIE Debugging Information Entry. 3, 4, 14–20, 23–25

DWARF Debugging with Attributed Record Formats. 6–8, 10, 13–26, 28, 31–33, 36, 37

ELF Executable and Linkable Format. 18, 23

ERD Embedded Rust Debugger. 4, 23, 26, 33, 34, 37

FDE Frame Description Entry. 15, 20, 21

GUI Graphical User Interface. 26

MCU Microcontroller Unit. 7, 9, 26

PC Program Counter. 9, 11, 18, 21

TCP Transmission Control Protocol. 27, 28

1 Introduction

A key part of programming has always been debugging the computer program. Debugging is the process of finding and resolving errors, flaws or faults in computer programs. These errors, flaws or faults are also commonly referred to as a bug or bugs in the field of computer science. Debugging has become more difficult to do over the years because of the increasing complexity of computer programs and the hardware. Thus the importance of better debugging tools have become essential to make the process of debugging easier and more time efficient. This is especially true for debugging embedded systems because of the close relation to hardware.

One of the first types of debugging tools made, and one of the most useful is called a debugger. A debugger is a program that allows the developer to control the debugged program in some ways, for example stopping, starting and resetting. It is also able to inspect the debugged program by for example displaying the values of the variables. Debuggers are especially useful when programming embedded systems, because a debugger enables the developer to inspect what is happening in the Microcontroller Unit (MCU). Thus giving the developer a complete view of what the program is doing. However they are also a very useful tool for testing.

Debugging today works by having the compilers generate debug information when compiling the program. The debug information is then stored in a file, the file is formatted using special file format designed to be read by a debugger or another debugging tool. One of the most popular of these file formats is the format: DWARF. It is a complex format that is explained in detail in section 3.4.

The debug information stored in the DWARF file can be used to find the location of variables, in most cases the value of variables are stored in memory on the call stack. A stack is a data structure for storing information, and is usually used to store all the variables for each function that is currently being executed. Thus a debugger can use the debug information to find the location of a variable and then evaluate the value of it, which is then displayed to the user of the debugger.

The main problem with debugging optimized code is that variables are not stored on the call stack, which is in memory. Instead they are temporarily stored in registers that are faster to access but cannot hold as much information as the memory. This reduces the amount of memory needed and makes the program run faster. It also makes debugging a lot harder because the variables are only present in the register for a very short time, and thus the debugger will often not have access to that value. Then there is the problem of debugger not being able to utilize all the available debug information.

1.1 Background

In the *Rust* programming language community there is no large project focusing on creating a debugger. There is even less focus on improving the debugging of optimized *Rust* code. One of the larger project focusing on debugging is called *gimli-rs*, one of the main parts of the project is to make a *Rust* library for reading the debugging format DWARF. Then there are other projects like *probe-rs* that focus on making a library that provides different tools to interact with a range of MCUs and debug probes. One of their newest tools that has not been released yet is a debugger that uses the *gimli-rs* library to read the DWARF file.

When it comes to improving generation of debug information there is some work being done on the *LLVM* project, but there is no focus on improving debugging for optimized code.

1.2 Motivation

The main motivation is that optimized *Rust* code can be $10 - 100$ times faster than unoptimized code according to the *Rust* performance book [Net+]. That is a very large difference in speed compared to other compilers like *Clang* for example, were the optimized code is about 1-3 times faster than the unoptimized code according to the paper [RD]. Thus for a language like *C*, it is much more acceptable to not be able to debug optimized code because the difference is not that large compared to *Rust*. But in the case of *Rust*, the difference is so large that some programs are too slow to run without optimization. This is a problem because debuggers are bad at debugging optimized code, and unoptimized code is to slow to run. That is why there is a need for better debuggers that can provide a good experience for debugging optimized *Rust* code.

An argument against the need of low level debugging is that the *Rust* compiler will catch most of the errors. This is especially true regarding pointers and memory access, which are two common causes of bugs in programming languages like *C* and *C++*. Thus there is less need for a debugger that can debug *Rust* code than there are debuggers that can debug *C* and *C++* code. However when it comes to embedded applications there is still a need for low level debugging, with such a debugger.

Another motivation for creating a debugger for embedded systems is that the two supported debuggers for *Rust* by the *Rust* team are *LLVM* and *GDB*, which both requires another program to interact with the MCU. An example of such a program that is commonly used is *openocd* (openocd homepage [Rat]), which needs to be setup as a *GDB* server that the debugger connects

to. This complicates the process of debugging and makes for a bad experience, especially for new developers.

Most of the debuggers used for *Rust* code are written in other programming languages, and there is not a lot of debugging tools written in *Rust* yet. Thus one of the key motivations is to write a debugger in *Rust*, which will also lead to the debugger having all the benefit of memory safety that *Rust* provides.

1.3 Problem definition

The problem this thesis tries to tackle is the problem of improving the debugging experience of using a debugger on optimized *Rust* code for embedded systems. This problem can be divided into two parts, the first part is to improve the generation of debug information. The second part to create a debugger written in *Rust* that has a better debugging experience for optimized code than the existing debuggers.

The goal is to create a debugger which gives a better debugging experience for optimized *Rust* code on embedded systems than some of the most commonly used debuggers, such as *GDB*, *LLDB*, and to inspire further development for debugging tools in the *Rust* community.

1.4 Delimitations

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 *rustc* it is the *LLVM* library that mostly handles the debug information generation. *LLVM* is a very large project that many people are working on, and thus improving on *LLVM* is out of scope for this thesis. The same goes for improving *rustc*.

The compiler backend *LLVM* that *rustc* uses supports two debugging file formats that holds all the debug information. One of them is the format DWARF, the other one is *CodeView* which is developed by *Microsoft*. To make a debugger that supports both formats would be a lot of extra work that does not 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.

The scope of this thesis does also not include changing or adding to the DWARF format. 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 realize 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 a lot of time before the *Rust* compiler has been updated to use the new standard.

The DWARF format is very complex but provides good documentation. Thus the explanation of DWARF in section 3.4 will not go into every detail of DWARF. Instead it will focus on explaining the minimum needed to understand the implementation of the debugger. For further details we refer the reader to [Com10].

Today there are a lot of different debugging features that a debugger can have. Many of them are advanced and complicated to implement, thus it is decided to limit the amount of features to the ones that are most important. The following is the list of features the debugger is planned to have:

- Controlling the debugging target by:
  - Starting/Continuing execution.
  - Stopping/Halting execution.
  - Reset execution.
- Set and remove breakpoints.
- Virtually unwind the call stack.
- Evaluate variables.
- Find source code location of functions and variables.
- A Command Line Interface (CLI).
- Support the *Microsoft* Debug Adepter Protocol (DAP).

When debugging code on embedded systems the debugger needs to know a lot about the hardware, and some of these things differ depending on the hardware. The following are some of the things it requires:

- Number of registers.
- Which of the registers are the special ones, an example is the Program Counter (PC) register.
- The endianness of the memory.
- The machine code instruction set the MCU is using/supports.

To support all the different MCUs would be too 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 *arm Thumb mode* instruction set.

2 Related work

2.1 Debugging Optimized Code

Debugging optimized code has been a problem for a long time. There have been many scientific papers on how to solve this problem. One of the most common approaches is to make the optimizations transparent to the user. The paper [BHS92] takes this approach by providing visual aids to the user. These visual aids are annotations and highlights on both the source and machine code which helps the user understand how the source and machine code relate to each other. Some other papers that takes the same approach are [Adl96; Cop94]. The two main problems with this approach is that the debug information size get a lot larger, and the compilation times get longer as well.

Another approach is to dynamically deoptimize the subroutine that is being debugged as described in the paper [HCU92]. This gives almost the same level of debugging experience for optimized code as unoptimized code. The main problem with this approach is that it cannot debug the actual optimized code.

There are a lot more approaches for debugging optimized code, but all of them have some drawbacks. Two of the most common are that compilation times get a lot longer, and the debug information gets a lot larger. These are two drawbacks most compilers do not want.

2.2 *probe-rs* Debugger

The *probe-rs* project is currently creating a debugger in *Rust*, which is almost finished. Their debugger is made for debugging embedded systems. It uses their other tools in their library to access and control the debug target. They also use the *gimli-rs* library for reading the DWARF format.

The main difference between *probe-rs* and the debugging library *rust-debug* presented in section 4.1 of this thesis, is that *rust-debug* is designed to be platform independent. The *probe-rs* library is designed to provide tools for embedded MCUs and debug probes only. While *rust-debug* is designed to provide a layer of abstraction over the DWARF debug format, which simplifies the process of retrieving debug information from *DWARF*.

The main pros of *rust-debug* compared to *probe-rs* are:

- Platform independent.
- Less dependencies.

- Has a wider range of use cases.

The main cons of *rust-debug* compared to *probe-rs* are:

- A little bit more complex to use.
- Requires that the user provide the functionality of reading the debug targets memory and registers.
- The code is a bit more complex.

3 Theory

3.1 Registers and Memory

The values of a computer program needs to be temporary stored somewhere on the computer where they can easily be accessed while running the program. The two main ways the computer can do this is to either store values in registers or in the memory. Registers are very limited in space, and are very volatile thus they are good for storing values which are used many times in a short amount of time. Memory is much slower but has a lot more space. Memory is thus much more useful for storing values that are not needed right now but will be needed in the future.

It is the compiler that decide when and if a variable will be stored in registers or memory. The compiler decides this when compiling the computer program, which means the developer has usually very little control over where the values are stored.

Values stored in memory are either stored on the call stack or the heap. The call stack holds all the arguments and variables for all the called functions that have not finished execution. While the heap contain all the dynamically allocated values.

3.1.1 Registers

Computer registers are small memory spaces that are of fixed size. These registers can store any type of data as long as it fits within the size limit of the registers. Some of the registers are reserved for special use, one of the most important ones is the Program Counter (PC) register. This register always holds the address of the next machine code instruction that will be executed. Which of the registers that are reserved for special use is different depending on the processor.

3.1.2 Call Stack

The call stack is a stack in the memory that has the arguments and variables of all the functions that have been called, and have not finished executing. A stack is a data structure that consist of a number of elements that are stacked on top of each other and the only two operations available for a stack is push and pop. The push operations adds a new element on top of the stack, and the pop operation removes the top element of the stack. Other key characteristics are that it is only the top element that can be accessed, thus to reach the lower elements, all the above elements needs to be popped.

Stack/call frames are the elements that make up the call stack. Each stack frame has the values of the arguments and variables for one function call. Stack frames also usually contain a return address which is a pointer to a machine code instruction. This instruction is the next instruction of the previous function that called the current function. When a function has finished executing, its stack frame will be pop/removed from the stack, and the previous function will continue executing. An example of a call stack and stack frames can be seen in the figure 1.

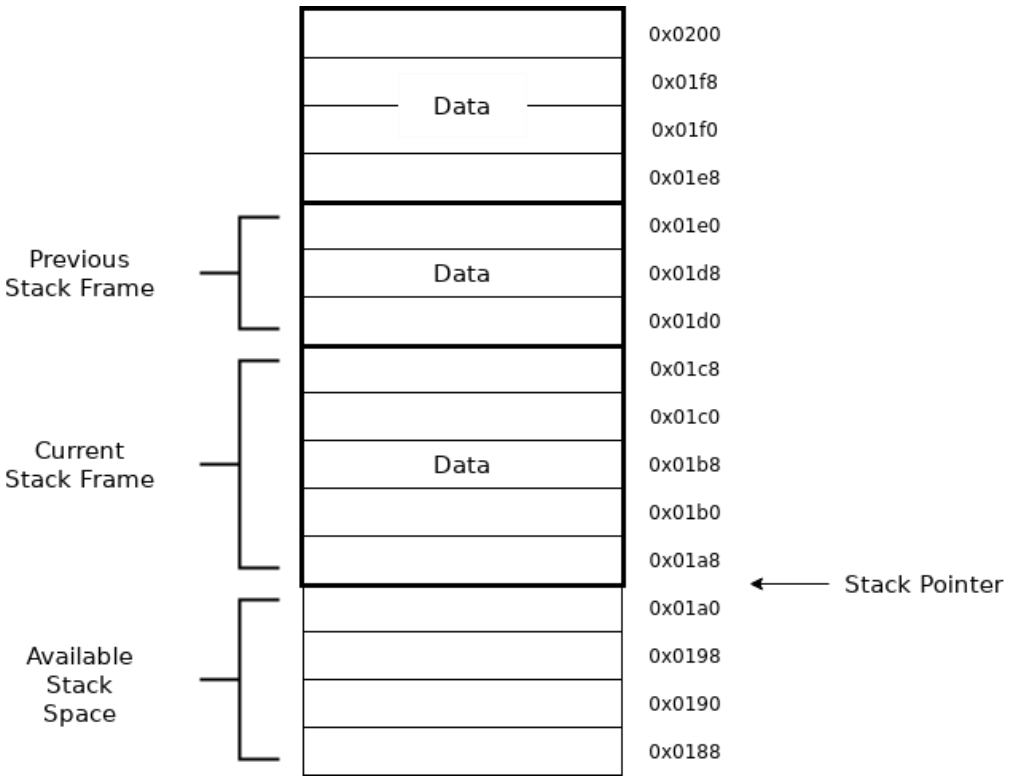

Figure 1: A visual example of how a stack and stack frames can look.

3.2 Prologue and Epilogue Code

Values in registers that need to be preserved during the execution of a subroutine will be push onto the call stack. This is done by the prologue code that is executed at the start of the subroutine. Then when the subroutine is finished executing, the stored register values are popped off the stack. This is done by the epilogue code that is at the end of the subroutine. The prologue and epilogue code is generated by the compiler.

One thing to note is that the prologue and epilogue code are not always continuous blocks of code that are in the beginning and end of a subroutine. Instead sometimes the store and read operation are moved into the subroutine. There are more of these special cases that the compiler does, and some that are hardware specific, to read more about them see [Com10] page 126-127.

3.3 Debugging

Debugging refers to the process of finding and resolving errors, flaws, or faults in computer programs. In computer science an error, flaw, or fault is often referred to as a software bug or just bug. Bugs are the cause for software behaving in a unexpected way which leads to incorrect or unexpected results. Most bugs arise from badly written code, lack of communication between the developers and lack of knowledge.

There are multiple ways to debug computer programs. One of the ways is testing, where some input is sent into the code, and then the result is compared to the expected result that is known beforehand. The amount of code being tested in a test can wary from just one function to the whole program. Another way of debugging is to do a control flow analysis to see which order the instructions, statements, or function calls are done in. There are a lot more ways to debug computer programs, but there is one that is most relevant to this thesis. This last debugging technique requires a computer program called a debugger that can inspect what is happening in the program being debugged, it can also control the execution of the debugged program.

3.3.1 Debugger

A debugger is a computer program that is used for testing and debugging other computer programs. The program that is being debugged is often referred to as the target program or just the target. The two main functionalities of a debugger is firstly the ability to control the execution of the target program. Secondly it is to translate the state of the target program into something that is more easily understandable.

Some of the most common ways a debugger can control a target program is starting, stopping, stepping, and resetting it. Starting or continuing means to continue the execution of the target program. Stopping the target program can often be done in two ways, the first is just to stop it where it is, the other way is to set a breakpoint. A breakpoint is a point in the code that if reached will stop the target program immediately. Stepping is the process of continuing the execution of the target program for only a moment, often just

until the next source code line is reached. Lastly resetting means that the target program will start execution from the start of the program.

Most debugger display the state of the target program relative to the source code. This means that if the target program has stopped, most debuggers will translate the location in the machine code it stopped on into the location of the source instruction from where the machine code instruction was generated from. They also often let the user set the breakpoint in the source code, and translate that to the closest machine code instruction. Other features debuggers have is the ability to virtually unwind the call stack, evaluate variables, and to evaluate expression. There are a lot more functionalities that a debugger can have, but these are some of the most common and used.

3.4 DWARF

This section will explain how the debug information format Debugging with Attributed Record Formats (DWARF) version 4 is structured, and how the different parts can be used to get debug information. However, it will not explain every detail about the DWARF format, because the DWARF format has a well documented specification that goes into detail how it is structured and how it functions. See [Com10] for the DWARF specification version 4.

3.4.1 Dwarf Sections

The DWARF format is divided into sections that all contain unique information with some few small exceptions. These sections use offsets from the start of other sections to point to information in the other section, most of these offsets can be found in specific DWARF attributes. The figure 2 shows all the DWARF sections and which ones point to each other.

3.4.1.1 *.debug_abbrev*

The DWARF section *.debug_abbrev* contain all of the abbreviation tables which are used to translate abbreviation codes into its official DWARF names. Some of the things these abbreviation code are used for are DIE tags and DIE attribute names. To translate an abbreviation code one has to compare each entry until the one with the matching abbreviation code is found. For further details we refer the reader to section 7.5.3 in [Com10].

3.4.1.2 *.debug_aranges*

The DWARF section *.debug_aranges* is used to lookup compilation units using machine code addresses. Each compilation unit has a range of machine code

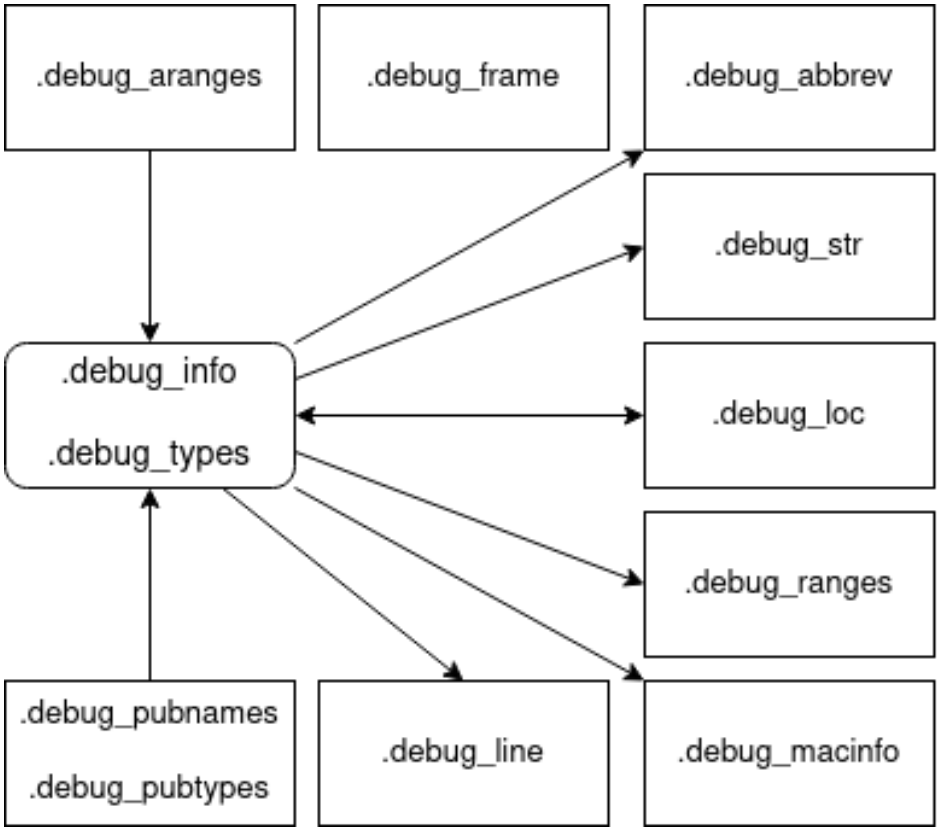

Figure 2: Diagram of all the different DWARF sections and their relations to each other.

addresses that are the addresses that the compilation unit have information on. These ranges consists of a start address followed by a length. Thus to find the compilation unit having the information about the current state. The user only needs to check if the current address is between the start address and the start address plus the length. For further details we refer the reader to section 6.1.2 in [Com10].

3.4.1.3 *.debug_frame*

In the DWARF section *.debug_frame* the information needed to virtually unwind the call stack is kept. This section is completely self-contained, and is made up of two structures called Common Information Entry (CIE) and Frame Description Entry (FDE). Virtually unwinding the call stack is complex, thus for further details we refer the reader section 3.4.5 in this document and section 6.4.1 in [Com10]

3.4.1.4 *.debug_info*

Most of the information about the source code is stored in DIEs which are low-level representation of the source code. DIEs 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 trees, which is a common data structure. Each compilation unit will have at least on of these trees made up of DIEs and each tree is stored in a DWARF unit. The trees are structured the same as the source code, which makes it easy to relate the source code to the machine code.

The section *.debug_info* consist of a number of these DWARF units, and some other debug information. This is one of the most important sections in DWARF because it is used to relate the state of the debug target and the source code, and vice versa.

3.4.1.5 *.debug_line*

The DWARF section *.debug_line* holds the needed information to find the machine addresses which is generated from a certain line and column in the source file. It is also used to store the source directory, file name, line number and column. Then the DIEs will store pointers to the source location information in the section *.debug_line* enabling the debugger to know the source location of a DIE. The section 6.2 in [Com10] explains in more detail how this information is stored in the *.debug_line* section.

3.4.1.6 *.debug_loc*

The location of the variables values are stored in location lists, each entry in the list holds a number of operations that can be used to calculate 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. These offsets are most commonly found in the attribute *DW_AT_location* which is often present in DIEs representing variables. The relation between these two sections can be seen in the figure 2.

3.4.1.7 *.debug_macinfo*

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 pointed to by DIEs in the *.debug_info* section, and those pointers can be found in the attribute *DW_AT_macinfo*. This section is a bit complex thus for further details we refer the reader to section 6.3 in [Com10].

3.4.1.8 *.debug_pubnames* and *.debug_pubtypes*

There are two sections for looking up compilation units by the name of functions, variables, types and more. The first one is *.debug_pubnames* which is for finding functions, variables and objects, and the other one is for finding types, this section is called *.debug_pubtypes*. Both of these are meant to be used for fast lookup of what unit the search information is located in. For further details we refer the reader to section 6.1.1 in [Com10].

3.4.1.9 *.debug_ranges*

DIEs that have a set of addresses that are non-contiguous will have an offset to the section *.debug_ranges* instead of having an address range. The offset points to the start of a range list that contain range entries which are used to know which of the machine code addresses the DIE is active. The DWARF section *.debug_ranges* is used for storing these lists of ranges. For further details we refer the reader to section 2.17 in [Com10].

3.4.1.10 *.debug_str*

The DWARF section *.debug_str* is used for storing all the strings that is in the debug information. An example of these strings are the names of the functions and variables. These strings are found using an offset that are located in the attribute *DW_AT_name*. The attribute is found in the function and variable DIEs, and the offset is in the form of *DW_FROM_strp*.

3.4.1.11 *.debug_type*

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

3.4.2 Dwarf Compilation Unit

The compiler when compiling a source program will mostly generate one compilation unit for each project/library. There are some cases when multiple partial compilation units will be generated instead. The compilation units are store in the DWARF section *.debug_info*. These compilation units are structured the same as the source code, which makes it easy to relate between the debug target state and the source code.

The first DIE in the tree of the compilation unit 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 as well as the directory and path of the source file.

A DIE in the tree can have multiple children. The relationship between the DIE, and the children is that all the children belong to the DIE. An example of this is if there is a function DIE, then the children of the function DIE will be DIEs that represent parameters and variables that are declared in that function. Thus if the source code has a function declared in a function then one of the children to the first functions DIE will be the second functions DIE. This makes it is easy for the debugger to know everything about a function by going through all of its children.

3.4.3 Dwarf Debugging Information Entry

One of the most important data structures in the DWARF format is the DIE. A DIE is low level representation of a small part of the source code. Some of the most common things DIEs represent are functions, variables and types. The DIEs are found in a tree structure referred to as a DIE tree. Each DIE tree will often represent a whole compile unit or a type from the source code. The ones representing compile units are found in the DWARF section *.debug_info*, while the ones representing type are found in the DWARF section *.debug_type*. The DIEs representing types are often referred to as type DIEs.

3.4.3.1 Dwarf Attribute

The information in the DIEs are stored in attributes these attributes consists of a name and a value. The name of the attribute is used to know what the value of the attribute should be used for. It is also used to differentiate the different attributes. All of the attributes names start with *DW_AT_*, and then some name that describes the attribute, an example is the name attribute *DW_AT_name*. In the DWARF file the name of the attributes will be abbreviated to their abbreviation code that can be decoded using the *.debug_abbrev* section. A DIE can only have one of the same attribute, but there is no other limit to what attributes it can have.

3.4.3.2 Example of a DIE

Looking at an example of a DIE that describes a variable. The example can be seen in figure 3 which is a screenshot of the output from the program *objdump* run on a Executable and Linkable Format (ELF) file. The first line in the figure begins with a number 8 which represents the depth in the DIE tree this DIE is located. The next number is the current lines offset into this compile unit, all the other lines in the figure also start with their offset.

Then it says "Abbrev Number: 9" on the same line, this is an abbreviation code that translates to *DW_TAG_variable*. This tag means that the DIE is representing a variable from the source code.

```
<8><241>: Abbrev Number: 9 (DW_TAG_variable)
   <242>   DW_AT_location    : 2 byte block: 7d 3c     (DW_OP_breg13 (r13): 60)
   <245>   DW_AT_name        : (indirect string, offset: 0x40466): ptr
   <249>   DW_AT_decl_file   : 1
   <24a>   DW_AT_decl_line   : 591
   <24c>   DW_AT_type        : <0x1069>
```

Figure 3: An example of a DIE representing a variable named *ptr*. This example is the output of the tool *objdump* run on a DWARF file.

The attribute *DW_AT_location* seen in figure 3 has the information of where the variable is stored on the debug target. The attribute *DW_AT_name* has an offset into the DWARF section *.debug_str* that the *objdump* tool has evaluated to "str", this is the name of the variable. Attributes *DW_AT_decl_file* and *DW_AT_decl_line* in the figure contain offsets into the section *.debug_line*. Those offsets can be evaluated to the source file path and line number that this DIE is generated from. Lastly the attribute *DW_AT_type* contain an offset into the section *.debug_types*, that points to a type DIE that has the type information for this variable.

3.4.4 Evaluate Variable

The process of evaluating the value of a variable is a bit complicated because there is a lot of variation. Thus to simplify the explanation, a simple example will be used to explain the main part of evaluating a variable.

Taking a look at the example in figure 4 there is a function/subprogram DIE with the name *my_function*(it is the DIE with the tag *DW_TAG_subprogram*). The function has a parameter called *val* which is the DIE with the tag *DW_TAG_formal_parameter*, it is a child of the function DIE. Which means that it is a parameter to the function *my_function*. It is this parameter *val* that will be used as an example of how to evaluating a variable.

A key thing to note is that the function DIE called *my_function* has two attribute called *DW_AT_low_pc* and *DW_AT_high_pc*. Those attributes describe the range of PC values in which the function is executing. There is also some other attributes in the example that will not be mentioned because they are not needed for determining the value of the attribute.

```
<2><4321>: Abbrev Number: 16 (DW_TAG_subprogram)
   <4322>   DW_AT_low_pc      : 0x8000fca
   <4326>   DW_AT_high_pc     : 0x2c
   <432a>   DW_AT_frame_base  : 1 byte block: 57       (DW_OP_reg7 (r7))
   <432c>   DW_AT_linkage_name: (indirect string, offset: 0x473b8): _ZN24nucleo_r
   <4330>   DW_AT_name        : (indirect string, offset: 0x64a52): my_function
   <4334>   DW_AT_decl_file   : 1
   <4335>   DW_AT_decl_line   : 194
   <4336>   DW_AT_type        : <0x6233>
<3><433a>: Abbrev Number: 17 (DW_TAG_formal_parameter)
   <433b>   DW_AT_location    : 2 byte block: 91 7e    (DW_OP_fbreg: -2)
   <433e>   DW_AT_name        : (indirect string, offset: 0x11d94): val
   <4342>   DW_AT_decl_file   : 1
   <4343>   DW_AT_decl_line   : 194
   <4344>   DW_AT_type        : <0x6233>
```

Figure 4: An example of a subprogram and parameter DIE. This example is the output of the program *objdump* run on a DWARF file.

3.4.4.1 Finding Raw Value Location

Examining the DIE for the argument *val* there is an attribute there called *DW_AT_location*. The value of that attribute is a number of operations, performing these operations will give the location of the variable.

In this example the operation in the *DW_AT_location* attribute in figure 4 is *DW_OP_fbreg* -2. That operation describes that the value is stored in memory at the *frame base* minus 2 (see [Com10] page 18). The *Frame base* is the address to the first variable in the functions stack frame (see [Com10] page 56).

Currently the value of the *frame base* is unknown, but the location of the *frame base* is described in the *my_function* DIE. The location of the *frame base* is also described in a number of operations. Those operations can be found under the attribute *DW_AT_frame_base*. Looking at figure 4, the *frame base* location is described with the operation *DW_OP_reg7*. The operation *DW_OP_reg7* describe that the value is located in register 7 (see [Com10] page 27). Thus register 7 needs to be read to get the value of the *frame base*.

Now knowing the value of the *frame base* the location of the parameter val can be calculated. As mentioned previously, the location of parameter *val* is the *frame base* minus 2. Thus the value of *val* can be read from the memory at address of the *frame base* minus 2. However, the value also has to be parsed into the type of *val*, see section 3.4.4.2 for how that is done.

3.4.4.2 Parsing the Raw Value

Now the first problem with parsing the value of the parameter *val* into the correct type is knowing what type the parameter has, this is where the attribute *DW_AT_type* comes in. The value of the *DW_AT_type* attribute

points to a type DIE tree, which describes the type of the DIE.

The offset to the type DIE of the parameter *val* is $0x6233$, as can be seen in figure 4. Finding that type DIE is done by going to that offset in the *.debug_types* section. The type DIE for *val* can be seen in figure 5, note that the offset of the DIEs tag is the same as 6233. That type DIE has the tag *DW_TAG_base_type* which means that it a standard type that is built into most the languages (see [Com10] page 75).

```
<1><6233>: Abbrev Number: 34 (DW_TAG_base_type)
   <6234>   DW_AT_name        : (indirect string, offset: 0x2a125): i16
   <6238>   DW_AT_encoding    : 5      (signed)
   <6239>   DW_AT_byte_size   : 2
```

Figure 5: An example of a base type DIE. This example is the output of the program *objdump* run on a DWARF file.

In this example the type DIE has three attributes which are used to describe the type. The first attribute is *DW_AT_name*, it describes the name of the type. In this case the name of the type is *i16*, which can be seen in figure 5. The next attribute is *DW_AT_encoding*, this attribute describes the encoding of the type. An encoding with the value 5 means that the type is a signed integer [Com10]. The different values for encodings are specified in the DWARF specification [Com10]. Now the last attribute is *DW_AT_byte_size*, it describes the size of the type in bytes. A byte size of 2 in this case means that the type is a 16 bit signed integer. Now that the type of *val* is known the only thing left to do is to parse the bytes of the value into a signed 16 bit integer.

3.4.5 Virtually Unwind Call Stack

Virtually unwinding the call stack is done by recursively unwinding a stack of *subroutine activations*. It is called virtual unwinding because the state of the debug target is not changed at any point during the unwinding. Every subroutine in the call stack has an activation and a stack frame. Because the activation often has the value of the stack pointer, the related stack frame is also known. Thus successfully unwinding all the *subroutine activations* will result in complete understanding of the state of the call stack.

The debug information needed to unwind activations are stored in the DWARF section *.debug_frame*. That section is made up of two data structures, one is called Frame Description Entry (FDE). A FDE contains a table used for unwinding registers and the CFA of an activation. The other data structure is called Common Information Entry (CIE), it contains information that is

shared among many FDEs. The relevant CIE and FDE to an activation can be found using the code location where it is stopped.

Unwinding the stack of activation is done by first evaluating the values of the top activation, read section 3.4.5.1 to learn how that is done. It starts with the top activation because there is too little information known of the other activations. Next step is to find the CIE and FDE that contain debug info on the next activation. When those are known the values of the next activation can be evaluated as described in section 3.4.5.1. This is then repeated for the rest of the activations.

3.4.5.1 Subroutine Activation

A *subroutine activation* contain information on a subroutine call/activation. Each *subroutine activation* contain a code location within the subroutine, it is the location where the subroutine stopped. The reason for stopping could be that a breakpoint was hit, it was interrupted by an event, or it could be a location where it made a call to the next subroutine.

The address of the stopped code location is easily found using the stack pointer of the above activation in the activation stack. That is because the return address of the above activation is the stopped code location of the current activation. The return address is almost always stored on the stack thus it can easily be read if the stack pointer is known. This works for all activations except for the top activation, where the current PC value is the stopped code location.

An activation also describes the state of some of the registers where it stopped. Those are the registers that are preserved thanks to the prologue and epilogue code of the subroutine. The rest of the registers are unknown because they have been written over, which makes them impossible to recover.

The activations is identified by there CFA value. The CFA is the value of the stack pointer in the previous stack frame. One thing to note is that the CFA is not the same value as the stack pointer when entering the current *call frame* (see [Com10] page 126).

Both the values of the CFA and the preserved register can be restored using tables located in the DWARF section *.debug_frame*. For further details, the reader is referred to section 3.4.5.2.

3.4.5.2 Unwinding CFA and Registers

The tables in the FDEs contains virtual unwinding rules for a subroutine. These virtual unwinding rules are used to restore the values of registers and the CFA.

The first column in the tables contains code addresses. Those addresses are used to identify the code location that all the virtual unwinding rules on that row applies for. Next column is special because it contains the virtual unwinding rules for CFA. The rest of the columns contain the virtual unwinding rules for registers 0 to n, where n is the last registry. Check out figure 6 for a visual of how the tables are structured.

```
LOC CFA R0 R1 ... RN
L0
L1
...
LN
```

Figure 6: This is how the table for reconstructing the CFA and registers looks like. *LOC* means that it is the column containing the code locations for 0 to N. The column with CFA has the virtual unwinding rules for CFA. The rest of the column *R0* to *RN* holds all the virtual unwinding rules for the register 0 to N.

There are a number of different virtual unwinding rules, the ones for the registers are called register rules. Some of them are easy to use such as the register rule *undefined*. This rule means that it is impossible to unwind that register. Other ones require some calculations such as the register rule *offset(N)*, where the N is a signed offset. This rule means that the register value is stored at the CFA address plus the offset N. All of the rules can be read about in the DWARF specification [Com10] on page 128.

Unwinding a register is done by first finding the correct row. That is done by finding the closest address that is less than the search one. Next step is to evaluate the new value using the register rule on the row, then go to the next row in the table, and do the same thing but with the new value. Repeat until there are no more rows. That is how to use the table to unwind a register.

4 Implementation

The implementation is separated into three different smaller projects. The first being a debug library called *rust-debug*. This library simplifies the process of retrieving debug information from the DWARF format. The second project is the debugger Embedded Rust Debugger (ERD) and the last one is a *VSCode* extension for ERD.

4.1 Debugging Library *rust-debug*

Retrieving 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 *Rust* library *gimli-rs* simplifies that problem by providing data structures and functions for reading the DWARF sections. However, the library requires a lot of knowledge about the DWARF format to use. Thus the *rust-debug* library was created to make it even easier to get the debug information. Figure 7 shows how the two libraries and the ELF file are connected.

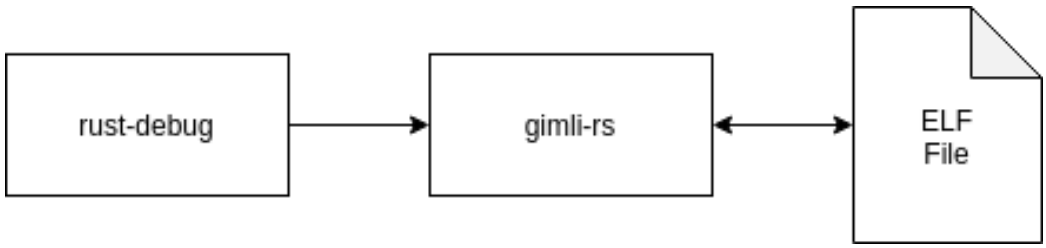

Figure 7: A diagram showing the relation between the *ELF* file and the two libraries *rust-debug* and *gimli-rs*.

Some of the main functionality that *rust-debug* provides are the following:

- Retrieving the source file location for functions, variables, types and more.
- Virtually unwinding the call stack.
- Evaluating variables.
- Translating source file line number to the nearest machine code address.
- Finding the DIE that represents a function using the name.

There is more functionalities that *rust-debug* provides but they are not noticeable. The code of this library is in the git repository [Lunc].

4.1.1 Retrieving Source File Location

Some of the DIEs in DWARF have attributes that starts with *DW_AT_decl_*. These attributes contain information on where in the source file the DIE was declared. This include file path, line number and column number. The library *rust-debug* has a function for retrieving the value of all these attributes from a given DIE.

The attribute *DW_AT_decl_file* which contain the file path of where the DIE was declared has a file index as its value. Every compilation unit has a line number information table that contain file paths which are indexed. Thus the *rust-debug* library will search for the matching index in the table to find the file path. Finding the line and column number does not require any lookup.

4.1.2 Accessing Memory And Registers

One of the requirements for evaluating the value of a variable is access to the registers and memory of the debug target. *rust-debug* does not have that functionality because it should be hardware independent as much as possible. Instead, the library uses a data structure which contain the values of the registers and memory. This keeps the library hardware independent. The data structure also has some function for reading and adding values.

The register and memory data structure is used as an argument to the functions in the library. If a value is missing, the called function will return a result that describes which value is missing. Then the user of the library can read that value from the debug target, and add it to the data structure. If there are more values missing then calling the function with the updated values will signal that additional values needs to be added. This repeats until the function give a result that says it has completed its task.

4.1.3 Evaluating Variables

The *rust-debug* library has a structure called *VariableCreator*, it takes a reference to a DWARF unit and DIE. The DIE has to be one that represents a variable. When the data structure is created, it will extract some important information about the variable from the DIE.

A constructed *VariableCreator* struct has a method for evaluating the value of the variable that requires the register and memory data structure. This method evaluates the variable as described in section 3.4.4 and the DWARF specification [Com10]. The return value of the method is an enum that is used for telling the user if a value is missing from the register and memory data structure or not. Then there exists another method for retrieving the

variable information containing the evaluated value. The variable information contains the following:

- Variables name.
- Variables type
- Variables location in the source file.
- The locations of the variables value in registers and memory.
- The evaluated value of the variable.

4.1.4 Finding a functions DIE

The *rust-debug* library has a function for finding a DIE that represents a function, using the name of the function. The function also needs a machine code address to find the correct compilation unit.

The correct compilation unit can be found using a code address. Every compilation unit has an address range that can be used to see if the machine code is from that compilation unit. Thus searching for the correct one is done by going through each unit, and checking if the address is in the range.

When the compilation unit is known, the DIE representing the subroutine can be found by searching the DIE tree of the unit. This is done by going down the path of the tree where the machine code address is in range. It returns a result when it has found the subroutine DIE with the searched name, or when there are no more DIEs to check.

4.1.5 Unwinding Call Stack

The *rust-debug* library has a data structure called *Stacktrace*, which works very similarly to the data structure describe in section 4.1.3. However, *Stacktrace* is used to virtually unwind the call stack and evaluate all the variables in it. The call stack is unwind as described in section 3.4.5 and the DWARF specification [Com10]. This results in a stack of activations, which most importantly contain restored register values and stack pointers.

All the variables in each of the activations are then evaluated using the restored register values. This is done using a data structure that works similar to the data structure describe in section 4.1.3.

The first step of evaluating the variables is finding the DIE of the subroutine that the activation is related to. This is done using the function described in section 4.1.4. After that the DIE tree is search through for variable DIEs, starting at the subroutine DIE. All the variable DIEs found are added to a

list. Each variable in the list is then evaluated as described in section 4.1.3. If there is a missing value from memory then the response from the evaluation of the variable is returned. Because the data structure works similar to the one in section 4.1.3, it can continue from were it last stopped.

4.1.6 Finding Breakpoint Location

The *rust-debug* library has a function that finds a machine code location using a source file location. This machine code location is the closest one that represents the line in the source code. The function requires a file path and line number, but it also can take a column number.

The mentioned function works by first finding out which compilation unit contains information on the inputted file path. It does this by looping through all the file entries in the line number information table for every compilation unit. Each line number information table entry have rows that each contain information on a line from the source code. Thus all the rows with the search line numbers are added to a list. The machine code address of the first element in this list is returned if no column line was inputted to the function. Otherwise, it is the one with the closest column number that is returned.

4.2 Embedded Rust Debugger

The debugger Embedded Rust Debugger (ERD) is implemented using the debugging library *rust-debug*. That library provides all of the needed functionalities for retrieving debug information from the DWARF format. The debugger also uses the *probe-rs* library for controlling the MCU. It is also used for accessing the registers and memory of the MCU.

The debugger is made up of three modules, the first one is called CLI. Its main functionality is to handle the input from the console and the output to it. The second module is called DebugAdapter, it also handles the input and output, but in the form of Debug Adepter Protocol (DAP) messages. This module is for supporting Graphical User Interfaces (GUIs) that use the DAP protocol to display debug information. The last module is called Debugger, and it runs in a separate thread from the other two modules. This is the module one that uses the *rust-debug* and *probe-rs* library to get debug information. It is also used to control the debug target. Figure 8 shows a diagram of how these modules are structured. Also all of the code for ERD can be found in this git repository [Luna].

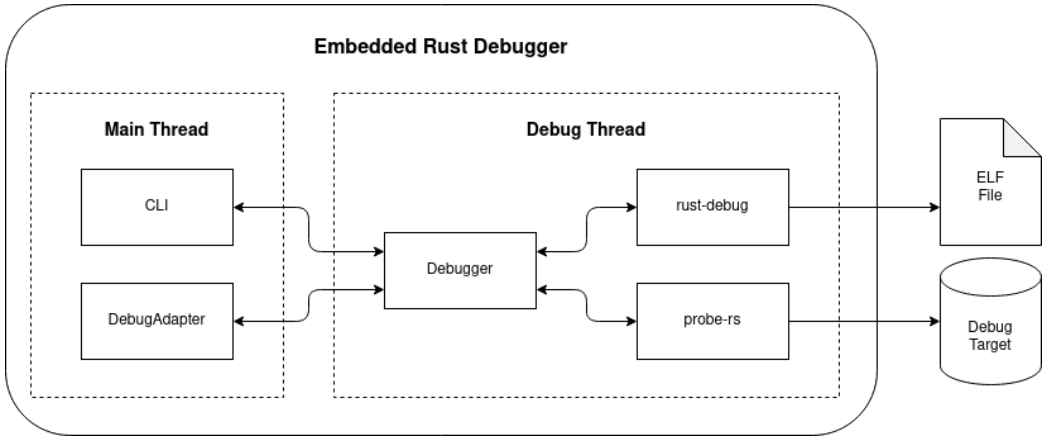

Figure 8: Diagram showing the structure of ERD.

4.2.1 The CLI module

The *CLI* is a very simple one, that has a separate thread for reading the input called input thread. The input is read from the console constantly until the user enters a line of text. Then the inputted line is sent to the main thread. After that the input thread waits for a response, the type of the response is a boolean. If the response boolean is true then the program will stop, otherwise it start reading the console for new input again. The input thread repeats this process until the main thread stops it.

When the main thread receives a message from the input thread it tries to parse it into command. The parser works by matching the first word in the input with the names of the commands. If there is a match the commands specific parser is used to parse the rest of the input. After a command is parsed, it is sent to the debug thread asynchronously. The main thread then waits for a response from the debug thread, when the response is received the main thread prints it to the console. After that it sends a response to the input thread and awaits new messages from both the input thread and the debug thread.

An example of all the communication between the user and the threads can be seen in figure 9. The example also shows what happens when an event is sent from the debug thread to the main thread.

4.2.2 The Debug Adapter module

The debug adapter is implemented as a Transmission Control Protocol (TCP) server in the main thread. When a new TCP connection is made the debug thread is started. After it has started the main thread starts listening for

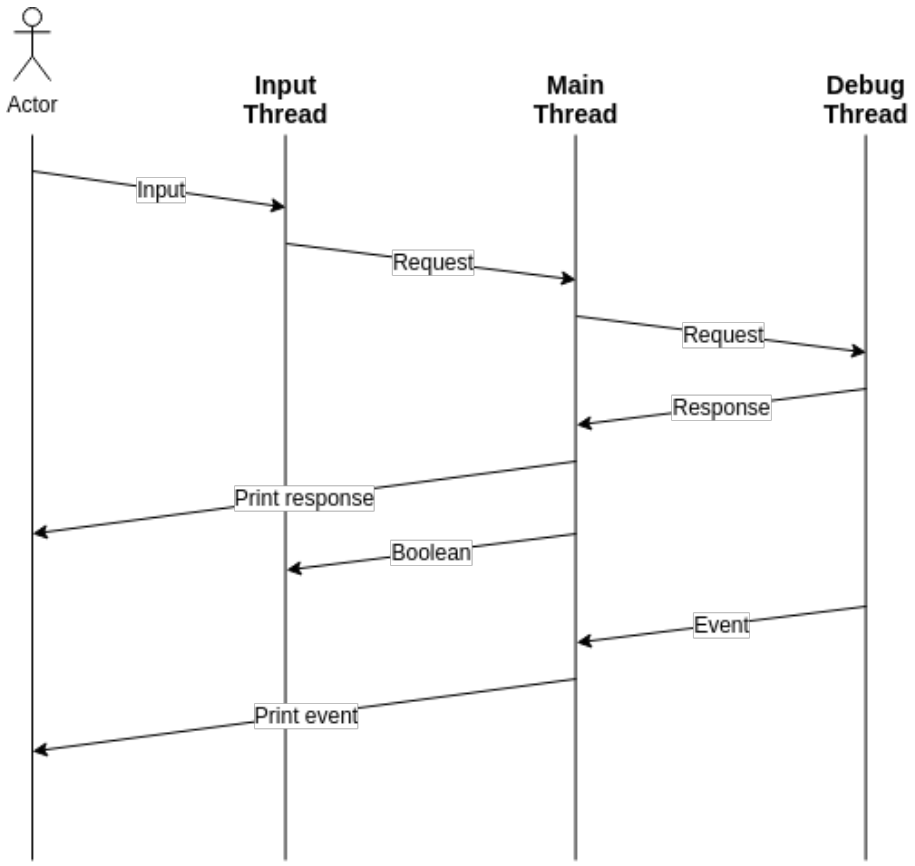

Figure 9: A diagram showing the communication between the user/actor and the three different threads.

DAP messages on the TCP connection.

The first DAP messages are for communicating the DAP functionalities that the client and the debugger has. These first DAP messages also contain some configuration for debugger module in the debug thread, those configuration are forwarded to the debug thread. After that the debug adapter will continuously pull for messages from the TCP client and the debug thread, until a message is received.

When the debug adapter receives a DAP message from the client, it is translated into one or more commands. Those commands are then sent one by one to the debugger module in the debug thread. The responses from those commands are then used to make a response to the clients DAP message. If the debug adapter gets an event from the debug thread, the debug adapter will translate it into a DAP message and send it to the client.

4.2.3 The Debugger module

The debugger module is designed as a server that listens for commands through a channel. It is running in a separate thread from the CLI and the debug adapter, which is called the debug thread. There is two states in which the debug thread can be in, that is in the attached or detached state. The two states are different in that the attached state can access the debug target and the DWARF file.

The debug thread is started by the main thread and starts in the detached state. In the detached state only the configuration and exit commands work. The other commands requires that the debug target is attached. But if one of those commands is received then the debug thread will try to enter the attached state. This can only happen if all the required configuration are set. If one of the configuration are missing then an error will be sent as a response.

When the debug thread is in the attached state it can use the *rust-debug* library to retrieve debug information. It can also use the *probe-rs* library for controlling the debug target and reading the registers and memory. The relation between the libraries can be seen in figure 10.

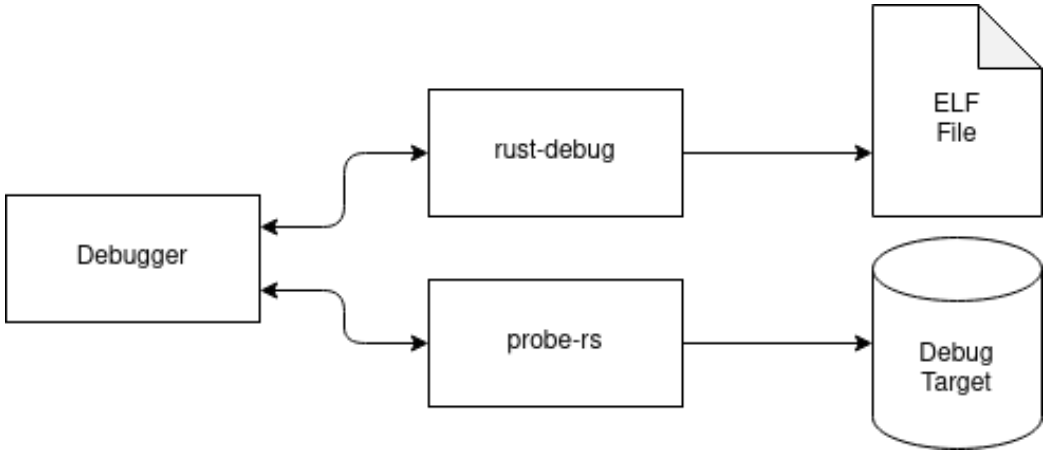

Figure 10: A diagram showing the relations between the debugger, the *ELF* file, the debug target and the two libraries *rust-debug* and *probe-rs*.

4.2.3.1 Retrieving Debug Information

The debugger module retrieves debug information by using the *rust-debug* library. This library sometimes requires a data structure that contain the values of some registers and memory addresses. Those function will return an enum that tells if a value is needed and where that value is located. Thus

the debugger module will use the *probe-rs* library to read the needed value, which is then added to the data structure. This repeats until no more values are needed, and the result of the evaluation can be retrieved. The values in the data structure are removed every time the debug target starts executing again. This is such that no old values are used, and it works because the debug target must be stopped to use related commands.

4.2.3.2 Simultaneous Handling of Request And Events

The debug thread polls the channel for incoming request and the state of the debugger. This enables the debug thread to simultaneously handle requests from the user, and events from the debug target. There is a boolean that keeps track if the debug target is running. It is used to stop the pulling of the targets state when the debug target is stopped. This is done because the debug target cannot start executing on its own.

4.2.3.3 Optimization of Repeated Variable Evaluation

To improve on the performance of the debugger it will temporarily store the values of the stack trace. This allows for fast repetitive look up of information present in the stack trace. The stored stack trace is removed any time the debug target starts executing again. This ensures that it always gives the correct values and not the old ones.

Another thing to note is that if a request is received to only evaluate one variable, the debugger will then perform a stack trace instead, or read from the temporarily stored stack trace. This simplifies the implementation a lot and also makes the subsequent evaluation request faster.

4.3 VSCode Extension

The *VSCode* extension for the debugger *Embedded Rust Debugger* is a very simple and bare bones implementation. *Microsoft* provides an Application Programming Interface (API) which can be used to start a debugging session and some trackers for logging. The implementation of the extensions uses that API to create a debugging session and a tracker that logs all the sent and received messages, it also logs all the errors. There are some configuration done to the debugging session, most of it is for the DAP messages being sent. All of the code for the extension is in the git repository [Lunb].

5 Evaluation

To evaluate the soultion to the problem(see section 1.3) there is three criterias. The first one is how much more useful debug information does the solution get from the compiler, this is to evalaute the first part of the problem. For the second part of the problem the debugger ERD needs to be compared to the aleady existing debuggers to see if any improvment is made to debugging optimized *Rust* code. The two most popular debugger used for debugging *Rust* code are *GDB* and *LLDB*. Thus it is these two debuggers that will be compared to the presented debugger. The reason being that they are the two debuggers that are supported by the *Rust* dev team according to there *rustc-dev-guide* [Teaa]. The criteria they will be compared to is how well the debuggers can retrive debug infromation from optimized code and how well they can display that information to the user. Because getting the information is important but it is pretty much useless if it is not displayed in an user friendly way.

5.1 Compiler Setting Comparison

The *Rust* compiler has some compiler options that the user can set when compiling a project that effect the code generation and the debug information. It also enables the user to access some of the compiler options for *LLVM* that is the backend of the compiler. The first part to finding if any of these compiler options effect the debug information generated in a meaningful way is to read what they do and determine if they might do that. Then the compiler options need to be tested manually to see if any of them actually have any meaningful effect, and that the option does not affect the speed of the optimised code in a negative way. This is very hard to do and takes a lot of time to do it properly, thus the testing done will be limited to checking the difference in the DWARF file and the debugging experience.

5.1.1 Settings *debuginfo* and *opt-level*

When going through all the possible *rustc* compiler options for code generation that are listed in the *rustc book* [Teab], there where two that have the most impact on the amount of debug information generated. The most important one of these is the flag named *debuginfo* which controls the amount of debug information generated. It has three options, the first is option 0 which means that no debug info will be generated. The second option is 1 which will only generate the line tables and the last is option 3 which generates all the debug information it can. Thus it is clear from the description in the *rustc book*

that this flag should always be set to 3 when debugging.

The other key compiler option is the optimization flag named *opt-level*. It controls the amount of optimization done to the code and which type of optimization, size or speed. Comparing the different optimization levels the debug information got less and less when a higher optimization level was set on a simple blink code example. The highest optimization level for speed is 3 and the result from using is that the debug information got very limited, thus making it extremely hard to debug. In the testing of the three optimization level 1, 2 and 3 it seems that optimization level 2 worked the best for debugging without making the program too slow.

5.1.2 Settings *force-frame-pointers* and *force-unwind-tables*

There are two other *rustc* compiler option flags that are important for debugging but sometimes not necessary depending on the target. The first one is called *force-frame-pointers* it forces the use of frame pointers according to the documentation [Teab]. The other one is called *force-unwind-tables* which forces the compiler to generate unwind tables in the DWARF file, according to the documentation [Teab]. These two flags set to *'yes'* can ensures that the debugger can unwind the stack and display the callstack to the user.

5.1.3 *LLVM* Settings

Looking through all the *LLVM* arguments that is available through *rustc* yielded one more argument that effects the debug information generated. The argument is named *debugify-level* and controls the kind of debug information that is added to the DWARF file. There are two options for this argument, the first is called *"location+variables"* and adds locations and variables to DWARF. The other option is called *"locations"* and only adds locations to DWARF. When testing out both of these options on optimized code there were no noticeable difference in the debugging experience and no noticeable difference in the DWARF file.

5.2 Debugger Comparison

The testing and comparing of the three different debuggers is done manually on some example code, see the git repository [HL] for the example code. The example code was many times modified to test how well the three debugger handled the different situations. This was repeated until one of the debuggers gave an unexpected result. There were two of these cases found when the code was compiled with optimization 2. To keep the comparison fair a

breakpoint was added to the same machine code instructions when comparing the debugger. This ensures that all the three debuggers stop on the same instruction.

5.2.1 Evaluation of *Rust* enums

When testing the three different debuggers it was discovered that they evaluated *Rust* enums to different values in certain situations. The first situation found where this happened, is when the value describing which variant the enum is was optimized out by the compiler. The table 5.2.1 shows a example of this situation, by showing the result of the three debuggers.

The debuggers evaluation results for variable *test_enum3*	
Source	Value
DWARF Location	(DW_OP_piece: 9; DW_OP_breg13 (r13): 156; DW_OP_piece: 3)
Rust Source Code	let mut test_enum3 = TestEnum::Struct(TestStruct { flag: true, num: 123});
Debugger (Version)	Evaluated Result
ERD	test_enum3 = TestEnum { < OptimizedOut > }
GDB (11.0.90)	(gdb) p test_enum3 $ 1 = nucleo_rtic_blinking_led::TestEnum::ITest(<optimized out>)

LLDB (13.0.0)	(nucleo_rtic_blinking_led::TestEnum) test_enum3 = {
	ITest = (0 = 0)
	UTest = (0 = 0)
	Struct = {
	0 = (flag = false, num = 0)
	}
	Non = {}
	}

Table 1: The different debuggers evaluation result for variable *test_enum3*, and the actual source code value and DWARF location.

The declared value of the variable *test_enum3* can be seen in the row

called *Rust Source Code* in table 5.2.1. Some important things to note, is that the variant of the enum is called *Struct*, the value of flag is *true* and the value of *num* is 123.

The location of the variables value can be seen in the row called *DWARF Location* in table 5.2.1, there it can be seen that the first DWARF operation is *DW_OP_piece: 9*. This operation in short means that the first 9 bytes of the variable is optimized out, because there are no other operation before it. The other two operations describe that the last 3 bytes of the value is stored in memory at the address stored in the base register 13 plus the offset 156. This all means that most of the value is optimized out, including the value describing the enum variant.

The result from evaluating *test_enum3* using the debugger ERD is that the whole enum is optimized out, as can be seen in table 5.2.1. Thus ERD gives the correct result in this case. But looking at the table it can be seen that both *GDB* and *LLDB* give different results. *GDB* give the incorrect result in that it says that the variant of the enum is *ITest* when it should be *Struct*. *LLDB* dose not give any indication of what enum variant the value is which is correct, instead it shows the value of each enum variation. But it gives the wrong value for the variant *Struct*, thus it gives a incorrect result. Thus out of the three debuggers it is ERD that gives the most correct result, in these situations.

5.2.2 Evaluation of *Rust* structs

When testing the three different debuggers it was discovered that they evaluated *Rust* structs to different values in certain situations. The table 5.2.1 shows a example of this situation, by showing the result of the three debuggers.

The declared value of the variable *test_struct* can be seen in the row called *Rust Source Code* in table 5.2.2. Some important things to note, is that the value of flag is *true* and that the value of *num* is 123.

The location of the variables value can be seen in the row called *DWARF Location* in table 5.2.2, there it can be seen that it only has two DWARF operation. The DWARF operation *DW_OP_breg13: 260* means that the value is stored in memory, at the address stored in the base register 13 plus the offset 260. And the second operation *DW_OP_piece: 8* means that the value of the previous operation has the byte size 4. The *test_struct* is larger then 4 bytes which in this case means that the attribute *flag* is optimized out.

The result from evaluating *test_struct* using the debugger ERD is that the the attribute *num* has the value 123 and the attribute *flag* is optimized out, as can be seen in table 5.2.2. Thus ERD is able to evaluate the correct value, but *GDB* gives a even better result. Because *GDB* points out that the

The debuggers evaluation results for variable *test_struct*	
Source	Value
DWARF Location	(DW_OP_breg13 (r13): 48; DW_OP_piece: 4)
Rust Source Code	let mut test_struct = TestStruct { flag: true, num: 123 };
Debugger (Version)	Evaluated Result
ERD	test_struct = TestStruct { num::123, flag::< OptimizedOut >}
GDB (11.0.90)	(gdb) p test_struct $ 1 = nucleo_rtic_blinking_led::TestStruct {flag: <synthetic pointer>, num: 123}
LLDB (13.0.0)	(nucleo_rtic_blinking_led::TestEnum) test_struct = (flag = false, num = 123)

Table 2: The different debuggers evaluation result for variable *test_struct*, and the actual source code value and DWARF location.

attribute *flag* is a synthetic pointer. Lastly the debugger *LLDB* says that the value of *flag* is *false* which is incorrect. Thus out of the three debugger *GDB* is the most descriptive and correct, in these situations.

5.2.3 Displaying Optimized Out Variables

There is one more situation where the result of the debuggers are a bit different. That situation is when a variable is optimized out, but only temporarily. The table 5.2.3 shows a example of this situation, by showing the result of the three debuggers.

The declared value of the variable *test_u16* can be seen in the row called *Rust Source Code* in table 5.2.3. The location of the variable is not describe in the table because it has none. Instead the machine code instruction ranges for the location list entries are shown, they are found in the row called *DWARF Location List Ranges*. The first column of the ranges shows the start address of the range and the second column shows the end address of the range. The first row has a larger start address because it is the first entry of the list and has special rues. This means that there is only one location list entry in this

The debuggers evaluation results for variable *test_u16*	
Source	Value
Program Counter (PC)	0x08001290
DWARF Location List Ranges	0xffffffff 0x080002f4 0x080004a4 0x080004d0
Rust Source Code	let mut test_u16: u16 = 500;
Debugger (Version)	Evaluated Result
ERD	test_u16 = <OutOfRange>
GDB (11.0.90)	(gdb) p test_u16 $ 1 = <optimized out>
LLDB (13.0.0)	(unsigned short) test_u16 = <variable not available>

Table 3: The different debuggers evaluation result for variable *test_u16*, and the actual source code value and DWARF location.

case.

The PC value in the table 5.2.3 is $0x08001290$ which a lot more then $0x080004d0$, which is the end address of the only location list entry for the variable. This means that *test_u16* will or has had a value, but correctly it dose not have one.

The result of evaluating *test_u16* using ERD is *<OutOfRange>*, as can be seen in table 5.2.3. That is a unique message that is only used when the location list entries are not in range of the current PC. The debugger *LLDB* gives the same result, except that the message is worded a bit differently. However, the result from *GDB* is not as descriptive because it uses the same message in this situation as it does when a value is completely optimized out. Thus both *LLDB* and ERD are more verbose and descriptive then *GDB* in these situations.

6 Discussion

6.1 Debug Information Generation

This thesis was not really able to improve the amount of debug information generated by *rustc* and *LLVM*. There are some options mentioned in the section 5.1 that generate debug information, but they are well known. Thus the result presented there is nothing new for most developers that program in *Rust*. To really improve on the amount of debug information generated, one has to work on the compiler *rustc* or *LLVM*.

6.2 Debug Experience For Optimized Rust Code

At the beginning of this thesis it was thought that *GDB* was not able to evaluate variables with values stored in registers. This thought came from the observation that *GDB* for the most part prints that the variables are optimized out, when debugging optimized code. Thus it was thought that those variables were completely removed from the optimized code. But as it turns out they were not, instead they were just optimized out at that particular point.

Keeping these expectation in mind the result in section 5.2 were a bit disappointing. The only improvement that could be made for this problem is that different messages are displayed for the different cases. This makes it much clearer to the user how the variable is optimized out.

Unexpectedly there was another problem with the debuggers *GDB* and *LLDB* that could be improved on. That problem is that both mentioned debuggers had problems with the evaluation of the *Rust enum* type. The reason for this problem is that many other programming language does not allow *enums* to have any value stored in them like *Rust* allows.

The *LLDB* debugger had much more problems with this then *GDB* which only gave the wrong value when the enum variant was optimized out. Thus unexpectedly the debugger in this thesis could improve on that by evaluating the correct value and by showing a correct optimization out messages, which *GDB* did not do. There is a *VSCode* extension that fixes the problem of *GDB* displaying the wrong variant of enum when the variant is optimized out. It is called *cortex-debug* and its git repository can be found here [Bal].

6.2.1 Debugging Rust Code on Embedded Systems

The experience of debugging *Rust* code on embedded systems with both *LLDB* and *GDB* is not very good. There is a lot of configurations that

has to be done and both require a program like *openocd* that handles the communication between the debugger and the target device. Using ERD made the debugging experience a lot better because it dose not need to use a program like *openocd*. The reason for it being better it that it takes fewer steps to start debugging.

6.3 Contributing to the Rust Debug Community

This problem is huge and complex so to really make a change more people have to contribute. One of the best ways of getting more people involved is by making it easier to contribute. The debugging library *rust-debug* does just that by simplifying the process of retrieving debug information from DWARF.

7 Conclusions and future work

The result from testing and comparing the different settings in the compiler is a bit disappointing. Because the settings found did not do much or were well known before, thus there was no real improvement.

But the result from comparing the debuggers was a success. The main reason for this is that the debugger ERD is able to correctly evaluate the value of variables that are of the *Rust* enum type. While both of the two supported debugger by the *Rust* team evaluated the wrong value in some special cases. The debugger ERD was also able to simplify the process of debugging embedded systems that run *Rust* code.

An unexpected result from the creation of the debugger was the debugging library *rust-debug*. This library simplifies the process of retrieving debug information from the DWARF format, which is a good contribution to the *Rust* debugging community. Overall I would say that goal of improving debugging of optimized *Rust* code was achieved, but that there is still a lot more that needs to be done.

7.1 Future Debugging Improvement

One of the main problems with debugging optimized code is that the variables are stored in registers and never pushed to the stack. This causes the variables to be overwritten when they are not needed anymore. That fact makes debugging very hard because the user has to stop the execution when the variable still exist. But if the debugger is able to get the last value of the variable and store it. The last known value of the variable could be displayed to the user if it is out of range. There is no time to make a solution for this problem, thus this will have to be left for future work.

7.2 Future Debugger Improvement

The debugger ERD only supports the most important functionalities of a debugger. Thus there is many more features that could be added. One important one is the ability to evaluate expressions which is left to be done as future work. This is a hard problem because the evolution of the expressions should work exactly the same as the *Rust* compiler. Thus it would be best if the same code could be used so that it does not need to be written twice.

References

[BHS92] G. Brooks, G.J. Hansen, and S. Simmons. "A New Approach to Debugging Optimized Code." In: *ACM SIGPLAN Notices* 27.7 (1992), pp. 1–11. ISSN: 15581160. URL: http://proxy.lib.ltu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-84976693199&lang=sv&site=eds-live&scope=site.

[HCU92] U. (1) Hölzle, C. (2) Chambers, and D. (3) Ungar. "Debugging Optimized Code with Dynamic Deoptimization." In: *ACM SIGPLAN Notices* 27.7 (1992), pp. 32–43. ISSN: 15581160. URL: http://proxy.lib.ltu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-0026993865&lang=sv&site=eds-live&scope=site.

[Cop94] M. Copperman. "Debugging Optimized Code Without Being Misled." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.3 (1994), pp. 387–427. ISSN: 15584593. URL: http://proxy.lib.ltu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-0028427062&lang=sv&site=eds-live&scope=site.

[Adl96] Ali-Reza Adl-Tabatabai. "Source-level debugging of globally optimized code". PhD thesis. Citeseer, 1996.

[Com10] DWARF Debugging Information Format Committee. *DWARF Debugging Information Format version 4*. Tech. rep. June 2010.

[Bal] Marcel Ball. *cortex-debug*. URL: https://github.com/Marus/cortex-debug. (accessed: 28.08.2021).

[HL] Mark Håkansson and Niklas Lundberg. *nucleo64-rtic-examples*. URL: https://github.com/Blinningjr/nucleo64-rtic-examples.git. (accessed: 17.08.2021).

[Luna] Niklas Lundberg. *embedded-rust-debugger*. URL: https://github.com/Blinningjr/embedded-rust-debugger.git. (accessed: 17.08.2021).

[Lunb] Niklas Lundberg. *embedded-rust-debugger-vscode*. URL: https://github.com/Blinningjr/embedded-rust-debugger-vscode.git. (accessed: 17.08.2021).

[Lunc] Niklas Lundberg. *rust-debug*. URL: https://github.com/Blinningjr/rust-debug.git. (accessed: 17.08.2021).

[Net+] Nicholas Nethercote et al. *The Rust Performance book*. URL: https://nnethercote.github.io/perf-book/build-configuration.html. (accessed: 20.08.2021).

[Rat] Dominic Rath. *Open On-Chip Debugger*. URL: https://openocd.org/. (accessed: 28.08.2021).

[RD] Gokcen Kestor Rizwan A. Ashraf Roberto Gioiosa and Ronald F. DeMara. *Exploring the Effect of Compiler Optimizations on the Reliability of HPC Applications*. Tech. rep.

[Teaa] Rust Team. *Debugging support in the Rust compiler*. URL: https://rustc-dev-guide.rust-lang.org/debugging-support-in-rustc.html. (accessed: 18.08.2021).

[Teab] Rust Team. *The rustc book: Codegen options*. URL: https://doc.rust-lang.org/rustc/codegen-options/index.html. (accessed: 19.08.2021).