Luleå University of Technology

Master Thesis

Improving Debugging For Optimized Rust Code On Embedded Systems

Niklas Lundberg
inaule-6@student.ltu.se

supervised by
Prof. Per Lindgren

August 22, 2022

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 presents a debugger called ERDB. And a debugging library called *rust-debug*. The goal of the library is to make it easier to make debugging tools, by making it easier to get debug information from the Debugging with Attributed Record Formats (DWARF) format.

This thesis will explain the implementation of ERDB and *rust-debug*, it will also explain how the debug information format DWARF works.

The debugger ERDB will be compared to other debuggers to see if it achieved its goal. Comparing the debuggers showed that the biggest problem with debugging optimized code is that there is not enough debug information produced by the compiler.

Contents

List of Figures

Chapter 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 stored in a file, the information is stored in a special format designed to be read by a debugger or other debugging tools. One of the most popular of these formats is called DWARF, it is a complex format that is explained in section 3.4.

The debug information stored in the DWARF file can be used to find the location of the variables values. Which in most cases are stored on the call stack in memory, section 3.1.2 explains what a call stack is. The debug information also contain the data type information of the variable, it is used to interpret the raw value of variable found in memory into a typed value.

The main problem with debugging optimized code is that values of variables are sometimes stored in registers. Registers are faster to access then other memory and has a much lower capacity. Thus, storing values there often increases the speed of the programs, but the values are not persistent, because they get overwritten. This makes debugging a lot harder because the values are very short-lived compared to values that are stored on the call stack. Which often have the same life span as the stack frame. I refer the reader to section 3.1 for more detailed information about registers and memory.

1.1 Background

In the *Rust* programming language community there was no project focusing on creating a debugger for embedded systems when this thesis was started. There was 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 goals 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 for interact with a range of MCUs and debugging probes. One of their newest tools that is in early development is a debugger that also 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 in the *LLVM* project.

1.2 Motivation

A motivation for this thesis 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 languages like *C* for example, were the optimized code can be about 1-3 times faster than the unoptimized code as can be seen in the paper [RD17]. 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* code. But in the case of *Rust*, the difference is so large that some programs are too slow to run on MCUs without optimization. This is a problem because debuggers are bad at debugging optimized code as explained in the introduction, and unoptimized code is too slow to run. That is why there is a need for better

debuggers that can provide a good experience when debugging optimized *Rust* code.

An argument against the need of debugging embedded systems 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++*. Because most *C* and *C++* compilers have very few restrictions on how pointers can be used, and the value pointed to thus can be removed while the pointer still exists. 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 debugging with such a debugger. One reason is because *Memory-mapped I/O* is used to preform input/output (*I/O*) between the *CPU* and peripheral devices. And a debugger is very good tool for showing the state of these peripheral, which makes the development and testing much easier. Another reason is that hardware bugs are much easier to find if it is possible to inspect the state of a device easily.

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 at the time of writhing this 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 needlessly complex experience, especially for new developers.

Many of the debuggers used for debugging *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.

Currently, there knowledge needed to write debugging tools that utilizes the information stored in the DWARF format is very high. Thus, another motivation is to make an abstraction layer that reduced the amount of knowledge needed to some degree. This will be achieved by creating a debugging library, which hopefully will help others get information from the DWARF format and help them understand it better.

1.3 Problem definition

The main problem of this thesis is to improve the experience of debugging optimized *Rust* code on embedded systems. This will be achieved by creating a new debugger and debugging library that aims to solve three problems.

One of the problems is that two of the most popular debuggers, *GDB* and *LLDB*, requires another program to debug embedded systems. This problem makes both of them harder to use and is overall a bad user experience.

Another problem is that both *GDB* and *LLDB* often only display that variables have been optimized out. This happens very often when debugging optimized code, even with the less aggressive optimization options. Also, the message display by the debuggers should contain more information about why the variable was optimized out.

The last problem is that retrieving debugging information form the DWARF format is difficult to do. This makes creating tools that require debug information difficult and time-consuming to implemented, when they don't need to be.

Thus, the goals for this thesis is to create a new debugger and library that fix the problems mentioned above. The debugger should also support the most common debugging features, they are listed in section 1.4.

1.4 Delimitations

Currently, there are a lot of different debugging features that other debugger 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, to keep the scope of this thesis reasonable. The following is the list of features the debugger is required to have:

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

Debugging embedded systems requires the debugger knows certain information about the target system, like the architecture. Supporting every MCU out there would be to big of a scope for this thesis. Thus, the debugger is only required to work with the target system that is available. Which is a *Nucleo-64, STM32F401RET6 64 pin*.

The compiler backend *LLVM* that *rustc* uses supports two debugging file formats, according to their documentation [Teab]. 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 would be 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 theory to understand the implementation of the debugger. For further details we refer the reader to [Com10].

Chapter 2

Related work

2.1 Debugging Optimized Code

Debugging optimized code has been a problem for a long time. Thus, there are 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 take 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 does not debug the actual optimized code, which can cause all sorts of issues.

There are a lot more approaches for debugging optimized code, but all of them have some drawbacks. Two of the most common drawbacks is that compilation times get a lot longer, and the debug information gets a lot larger. These are two drawbacks that most compilers try to avoid.

2.2 The *probe-rs* Debugger

The *probe-rs* project is currently creating a debugger in *Rust*, for 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 and architecture 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. Another difference is that there debugger is not able to evaluate some of the more complex data structures in DWARF, but *rust-debug* is able to evaluate them.

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

- Platform and architecture independence.
- Has a wider range of use cases.
- Less dependencies.

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

- More complex to use.
- Requires that the user provide the functionality of reading the debug target's memory and registers.
- The code is a bit more complex.

Chapter 3

Theory

3.1 Registers and Memory

The values of a computer program need 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. Volatile in this case means that new values get written to the register often. Thus, registers are good for storing values which are used many times in a very 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 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 consists 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 will add an 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 contain the values of the arguments and variables for a function call, stack frames also contain a return address usually. When a function is finished the return address is written to the PC register, this makes the program jump to the return address. The return address usually points back to the previous function, which called the function. Also, when a function has finished executing, its stack frame will be popped/removed from the stack. An example of a call stack and stack frames can be seen in figure 3.1. The values to the right in the figure are addresses, and they start with *0x* because they are in hexadecimal.

3.2 Prologue and Epilogue Code

Values in registers that need to be preserved during the execution of a subroutine will be pushed 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.

Note 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 learn more about them see [Com10] page 126-127.

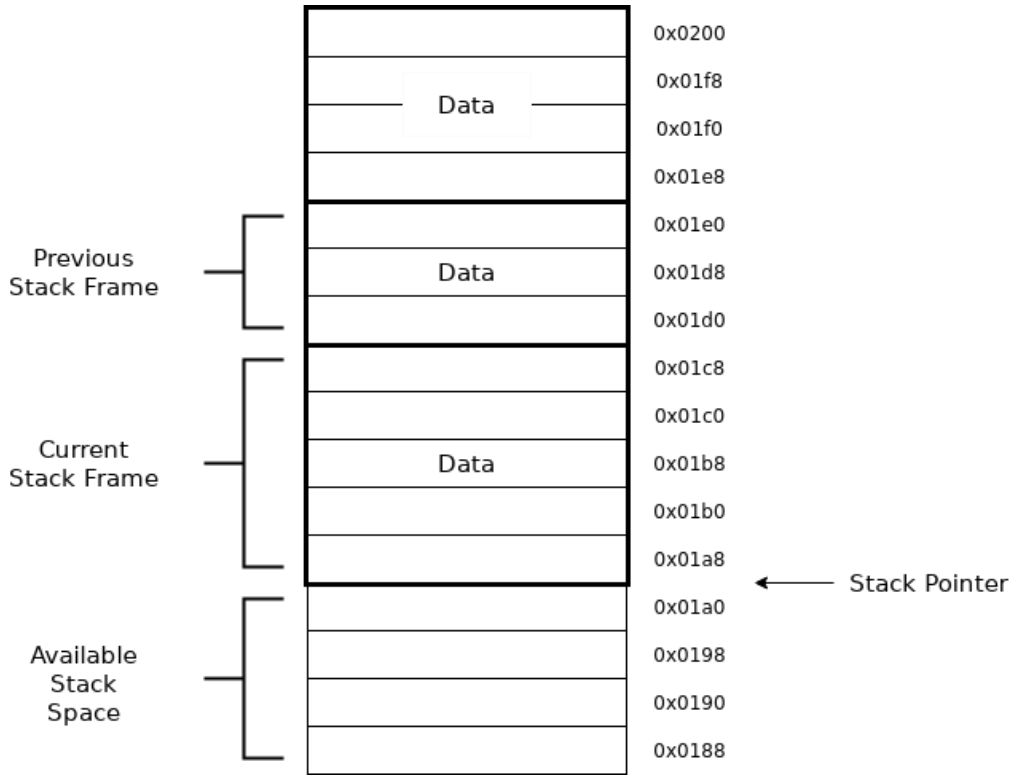

Figure 3.1: A visual example of how a stack and stack frames can look.

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 bug. Bugs are the cause for software behaving in an unexpected way. Most bugs arise from badly written code, lack of communication between the developers and lack of programming knowledge.

There are multiple methods to debug computer programs. One of the most common methods is testing, where some input is sent into the code, and then the result is compared to the expected result. The amount of code being tested in a test can vary from just one function to the whole program. Another method 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 methods to debug computer programs, but they all try to achieve the same thing. And that is to give the programmer a deeper understanding of what is happening.

There is no better tool for understanding a program then a debugger.

That is because a debugger can display the state of a program, and it gives the program control of the execution. Thus, a debugger enables a program to inspect every part of a program, this is especially true for modern debuggers, which have a lot more advance features.

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. The two main functionalities of a debugger, is firstly the ability to control the execution of the target program. Secondly it is the ability to inspect the state of the target program.

Some of the most common ways a debugger can control a target program is by starting, stopping, stepping, and resetting the execution. Starting the execution means that the target program continues the execution from the current stopped location, this also called continuing. Stopping the target program or halting it, can often be done in two ways, the first is to stop the execution where it currently 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. Breakpoint are very useful for inspecting certain point in the code, while the other way of stopping is more used when the stopping location is unknown. 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, but there are many variants of stepping. Lastly, resetting means that the target program will start execution from the beginning.

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 most debuggers have is the ability to show a stack trace, variable values and to evaluate expression. There are a lot more functionalities that a debugger can have, but these are some of the most common.

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 specification already does that. Instead, this section will focus on giving the reader an understanding of what type of information is stored in the DWARF format. And, to give some simple examples of how that information can be used to get the value of a variable for example. Thus, we refer the reader to the DWARF specification [Com10] for more details.

3.4.1 Dwarf Sections

The DWARF format is divided into sections that all have different information and purpose. 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 3.4.3.1. The figure 3.2 shows all the DWARF sections and which ones point to each other. All the DWARF sections are stored in a Executable and Linkable Format (ELF) file, which is a binary file format that is divided into different sections. The ELF format contain a table describe what each of its sections contain and where they begin and end.

3.4.1.1 *.debug_abbrev*

The DWARF section *.debug_abbrev* contain all the abbreviation tables which are used to translate abbreviation codes into its official DWARF names. These abbreviation codes are used for DIE tags, DIE attribute names and more. 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 addresses that are the addresses that the compilation unit has information on. These ranges consist 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

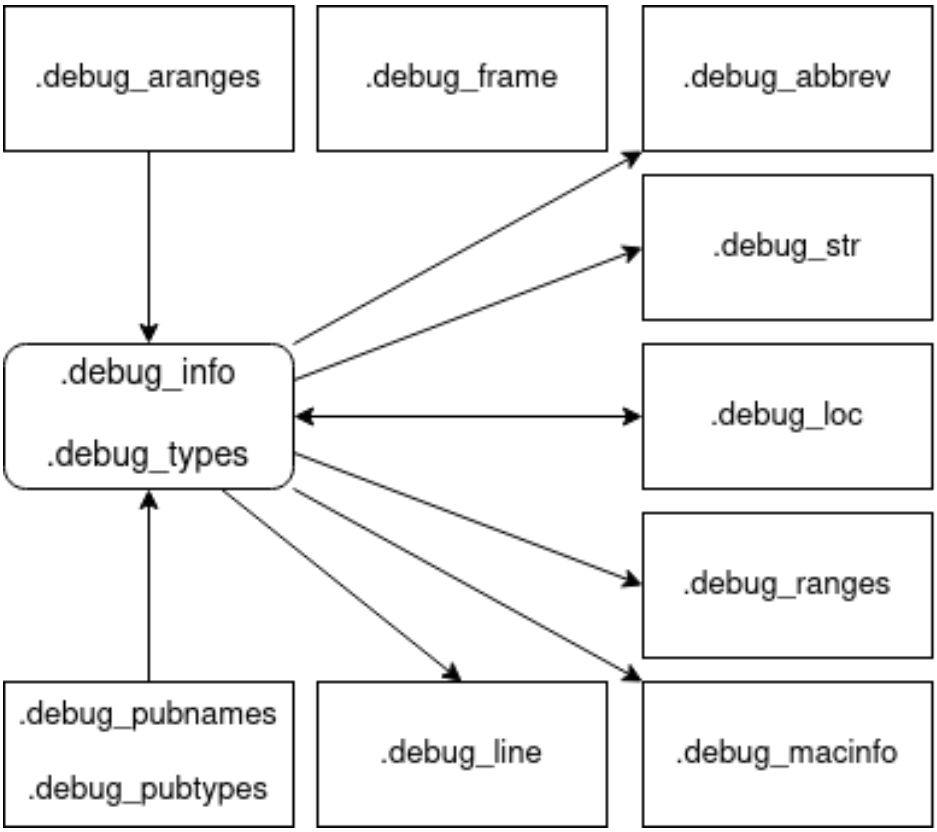

Figure 3.2: Diagram of all the different DWARF sections and their relations to each other.

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, each one of the trees is a DWARF compilation unit or a partial one. 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. Thus,

this is one of the most useful 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 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 figure 3.2.

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 too complex, 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

When compiling a source program, the compiler 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 parent DIE, and the children is that all the children belong to the parent 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 source code objects the 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

All the information stored in DIEs, are stored in unique 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 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.

3.4.3.2 Example of a DIE

An example of a DIE can be seen in figure 3.3, the figure is a screenshot of the output from the program *objdump* run on a 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.

The attribute *DW_AT_location* seen in figure 3.3 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*.

```
<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 3.3: An example of a DIE representing a variable named *ptr*. This example is the output of the tool *objdump* run on a DWARF file.

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 3.4 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 evaluate a variable.

```
<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 3.4: An example of a subprogram and parameter DIE. This example is the output of the program *objdump* run on a DWARF file.

Take note to the fact that the function DIE called *my_function* has two attributes called *DW_AT_low_pc* and *DW_AT_high_pc*. Those attributes

describe the range of Program Counter (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.

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 3.4 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 3.4, 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 to know 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 3.4. 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 3.5, 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 is 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 3.5: 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 3.5. 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 encoding 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 last step left to do is to parse the bytes of the value into a signed 16 bit integer.

3.4.5 Virtually Unwind the 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, see 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 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 are identified by there CFA value. The CFA is the value of the stack pointer in the previous stack frame. Note 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. See figure 3.6 for a visual of how the tables are structured.

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

Figure 3.6: 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 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 repeat but with the new value. Repeat until there are no more rows. That is how to use the table to unwind a register.

Chapter 4

Implementation

There are three different project that make up the implementation. The first being a debug library called *rust-debug*, which purpose is to simplify the process of retrieving debug information from the DWARF format. The second project is the debugger ERDB, that implements the Debug Adapter Protocol (DAP) protocol. And the last one is a *VSCode* extension that launches the ERDB debugger and connect to it.

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 the user to be knowledgeable about the DWARF format, the target system and the programming language/compiler used. Thus, the *rust-debug* library was created to make it possible to get information from the DWARF format for someone without that knowledge.

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

- Retrieving the source code file location for were functions, variables, types and more are declared.
- 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 are more functionalities that *rust-debug* provides, but they are less noteworthy. The code for this library is in the *GitHub* repository [Lunc].

4.1.1 Retrieving Source Code File Location

In the *DWARF* format, the source code location of where any type of data was declared, is stored in three attributes on the DIE that represents that data. One of the attributes contain indirect information about which file the data was declared, another contain the line number and the last one contain the column number.

The attribute contains the file path and name information is called *DW_AT_decl_file*. It does not contain the file path and name directly, instead it contains a file index. This file index is used to look up the file path and name in a table called, *line number information table*. There is one such table in each compilation unit, and the file index 0 is special. File index 0 is reserved for when a source file can not be specified.

The source code line and column number can be read directly from the attributes *DW_AT_decl_line* and *DW_AT_decl_column* respectively. Thus, they are much easier to retrieve.

rust-debug provides a function that performs the lookup file index lookup, and it also gets the line and column numbers. The function requires that a DIE and related compilation unit is given. But, the library has other function for retrieving the required information.

4.1.2 Accessing Memory And Registers

Some functionalities in the DWARF format requires access to register and memory values. One of them, is the evaluation of variables values.

Accessing the debug targets registers and memory is a different problem from reading and using the DWARF format. Thus, *rust-debug* lets the user of the library solve that problem. This has the benefit that any existing solutions can be used, like *probe-rs*. But it also has one downside, *rust-debug* is a little harder to use because of extra work the user has to do.

All the functionalities in *rust-debug* that requires access to the registers requires a struct from the library called *Registers*. This struct contains a *HashMap* with the register numbers and there corresponding values. It also contains the register number of some special registers, like the PC register. Thus, it is up to the user that the correct register numbers and values are

stored on the struct. An example on how this can be implemented can be found in the git repository for ERDB [Luna].

The functionalities that require access to the memory of the debug target, requires a struct that implements the *rust-debug* trait *MemoryAccess*. This trait is very simple and only has one function, which is for reading a varied amount of bytes at a given address. Thus, the user can create a struct that implements that function almost however they want. An example on how this can be implemented using the *probe-rs* library can be found in the git repository for ERDB [Luna].

4.1.3 Evaluating Variables

To make evaluating the value of a variable and retrieving other information about the variable easy, *rust-debug* provides a struct called *Variable*. That struct only has one function called *get_variable*, which takes a DIE and returns a *rust-debug Variable* struct. The *rust-debug Variable* struct may contain the following information:

- Variable name.
- Variable value, type and location on the debug target. All this information is stored in a tree structure.
- Variable declaration location, the file, line and column.

All the information in the *Variable* struct may not be present, because not all DIEs contain the information the struct may have.

The evaluated value of a variable is represented in a tree structure which is very similar to how the information is stored in the DWARF format. But, the value of each branch is evaluated into a value of one of the basic types, such as a signed integer. Thus, it removes the hardest part of evaluating a variable, without losing the flexibility of the DWARF format. The only downside of this solution is that it requires the user understand how variables are structure in the DWARF format.

4.1.4 Finding the current function

When debugging, knowing the current function that is being executed is very useful. Finding that function can be done by finding the function DIE that is the furthest down the DIE tree, in the branch where the current address is in range. Thus, the function DIE can easily be found by tracking all the function DIEs while going down the DIE tree. *rust-debug* makes this even easier by providing a function that does exactly this.

4.1.5 Unwinding the Call Stack

The *rust-debug* library has a function for virtually unwinding the current call stack, and it does it in two separate steps. Unwinding the call stack to get all the register values for each stack frame, is the first step. The second step is to find the subroutine and to evaluate all the variables for each stack frame.

Unwinding the call stack to get the register values is done by a separate function, which returns a *Vec* of *CallFrame* structs. Where each *CallFrame* struct contain information about each of the stack frames. The most important information the struct contains is the unwound register values, but it also contains other useful information, such as the start and end memory address of the stack frame. All the information in the struct is retrieved using the method described in section 3.4.5.

In the second step, the function loops through all the *CallFrame* structs and tries to retrieve all the information it can about the stack frame. This includes the name of the subroutine and the declaration location, which is found using the function describe in section 4.1.1. It also includes the variable information that retrieved using the function describe in section 4.1.3.

4.1.6 Finding Breakpoint Location

The *rust-debug* library has a function that finds a machine code address using a source code location. This machine code location is the closest one that represents the given 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 (ERDB) is implemented using the debugging library *rust-debug*, which requires the *gimli-rs* library. *rust-debug* provides all the needed functionalities for retrieving debug information from the DWARF format. The debugger also uses the *probe-rs* library for controlling

the debug target, it is also used for accessing the registers and memory of the debug target.

The debugger is made up of three modules, the first one is called CLI. Its main functionality is to handle the input and output from the terminal. The second module is called *DebugAdapter*. It also handles the input and output, but in the form of DAP messages. These DAP messages are sent through a TCP connection, and is for supporting the DAP protocol. The last module is called *Debugger*, it is for getting debug information and controlling the debug target.

The debugger is implemented asynchronously, because all the modules handle input, either from the user or from the debug target. Thus, the main function of the debugger waits for input from any of the modules, and then handles them. Figure 4.1 shows a diagram of how these modules are structured.

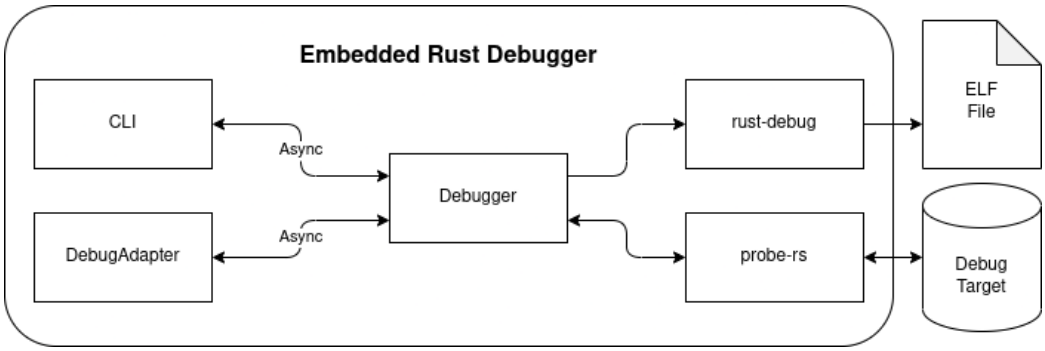

Figure 4.1: Diagram showing the structure of ERDB.

The code for ERDB can be found in this git repository [Luna].

4.2.1 The CLI Module

The CLI module has two jobs, the first is to read and parse the input into a request that the *Debugger* module can understand from the terminal. And the second is to convert the responses it gets from the *Debugger* module to text, and output that text to the terminal.

Reading the input from the terminal is done asynchronously, thus other tasks can be done while the CLI wait for the user input. When a new line has been entered, the CLI module will try to parse the string into a *DebugRequest* enum. The string is parsed by first matching the first word in the string to a command. Each command has a parser that is then used to parse the rest of the string. The resulting *DebugRequest* enum is then return to the main function, and then sent to the *Debugger* module.

After the *Debugger* module has done its work, the result it outputted to the terminal. This is done by sending the *DebugResponse* enum, that the *Debugger* module returned, to a function that has a match for each of the enum variants. Each of the variants have a unique message that is outputted to the terminal, and the messages often contain data from the *DebugResponse* enum. If the input text can not be parsed, a helpful message is printed to the terminal.

There is also a *DebugEvent* enum, that is outputted to the terminal in the same way the *DebugResponse* is. The *DebugEvent* enum is created by another task, which polls the debug target state. That tasks main job is to report if the debug target has halted the program that is being debugged.

4.2.2 The Debug Adapter module

All the DAP communication is handled in the debug adapter module. The main job of this module is to translate DAP messages into request, which are forwarded to the debugger module. It also handles the translation of the responses and events from the debugger module.

The DAP protocol communicates through Transmission Control Protocol (TCP), thus the debug adapter module has a TCP server. All the TCP communication is handled asynchronously, this is to enable the other tasks to run simultaneously. Currently, only one TCP connection can be open at a time. That is because the rest of the system does not yet support multiple simultaneous debugging session.

4.2.3 The Debugger Module

The debugger module mainly consist of one struct called *DebugHandler*, its main job is to get the debug information from the DWARF format. But it also keeps track of the debug targets state. The struct only has two methods for accomplishing this, one for them is for polling the state of the debug target. And the other method called *handle_request* takes a *DebugRequest* as an argument, and calls a function based on the argument. The method also returns a *DebugResponse* with the result.

The general flow when the *handle_request* method is called is as follows. First the *DebugRequest* enum is matched, and a check is done to see that all the required information is present. Then, the debugger module will preform some task to related to the given argument. Which often involve using the *probe-rs* library to access the debug target, and/or the *rust-debug* for retrieving debug information. Lastly the result is returned using the enum *DebugResponse*.

4.2.3.1 Retrieving Debug Information

The debugger module retrieves debug information by using the *rust-debug* library. That library sometimes require values from the debug target, such as the values stored in some registers and memory addresses. Those values are retrieved from the debug target using *probe-rs*.

4.2.3.2 Simultaneous Handling of Requests And Events

The state of the debug target is polled every 100 millisecond, to detect if the debug target has continued of stopped the program execution. In the time between the polls other tasks can be handled, such as parsing the input from the CLI or doing a stack trace. This is possible because there is an asynchronous function which sleeps for 100 millisecond, afterwards it polls the debug target state. That function repeats as long as the debug target is being debugged. Thus, the debugger can simultaneously handle both input from the user and state changes in the debug target. A state change in the debug target is called an event in ERDB.

4.2.3.3 Optimization of Repeated Variable Evaluation

To improve on the performance of the debugger, all the variables and stack frames are temporarily stored. This allows for fast repetitive lookup of debug information. All of this temporarily stored information is removed when the state of the debug target changes. This is to ensure that the correct information is returned.

Note that if a request is received to evaluate one variable, the debugger will then perform a whole stack trace instead. This simplifies the implementation a lot, it also makes the subsequent evaluation requests faster. Because, the value does not need to be evaluated, instead it can be taken from temporarily stored information.

4.3 VSCode Extension

Creating a *VSCode* extension is simple because *Microsoft* has a tool called *vscode-generator-code* [Micb], that will generate a empty extension. They also have a lot of documentation on how to get started with creating an extension.

There are two different types of debug extensions, the first is an extension that implements a debug adapter for a specific debugger. These often require a lot of work because a debug adapter needs to translate the DAP protocol messages to command that the debugger understands. Thus, depending on

the debugger it can require a lot of work to get this working well. The other type of debug extension is just a wrapper for the debugger that already implements the DAP protocol. These extensions are very simple to create, because they only need to start the debugger and connect to it using the *dap* protocol. To learn more about *VSCode* debug extension, I refer the reader to *Microsofts* documentations [Mica] for further reading.

The debugger *Embedded Rust Debugger* implements the DAP protocol over a TCP server. Thus, the *VSCode* extension is just a wrapper that starts the debugger TCP server and then connects to it. Connecting to the debugger server over TCP is very easy to do, because *Microsoft* has a library called *VSCode* that does that for you. The extension also captures the logs of the debugger and outputs them to the user using the *VSCode* library.

There are also some configurations that the user of the ERDB can set in the *launch.json* file. The available configurations are defined in a *json* file called *package.json* in the extension project [Lunb].

Chapter 5

Evaluation

5.1 Evaluating *rust-debug*

The debugging library *rust-debug* aims to solve the problem of it being difficult to get debug information from the DWARF format. Where the difficult part is that it requires many lines of code and knowledge about the DWARF format, to get the debug information wanted. Thus, one way to measure if *rust-debug* makes it easier to retrieve debug information, is to compare the amount of code lines it takes to create a debugger with and without using *rust-debug*.

Unfortunately there is no other debugger that has the exact same features as ERDB. And it would be unfair to compare ERDB with a debugger like *GDB*, which has a lot more features.

Instead, the number of lines in *rust-debug* will be compared against the debugger module in ERDB. The number of *Rust* code lines will be counted using the tool *tokei* [XAM]. *rust-debug* contains 3550 lines of *Rust* code, and the debugger module contains 1310 lines. Also, the whole debugger ERDB contains 2834 lines of *Rust* code.

The amount of *Rust* code in *rust-debug* is more than double that of the debugger module in ERDB. Thus, *rust-debug* has made it significantly easier to get some debug information.

5.2 Evaluating *ERDB*

One of the requirements set in the section 1.3 was that ERDB needs to implement some of the most common debugging features. A list of these features is can be found in section 1.4. ERDB implements all of those mentioned features, but there are two of them that is not fully supported.

Currently, ERDB only supports hardware breakpoints, thus there is a

limit to how many breakpoints can be set. To have more breakpoints the debugger needs to support software breakpoints, which will make it possible to have a lot more breakpoints. The other feature that is not fully supported are the different stepping variants. Stepping one machine code instruction is currently the only supported stepping function. But, it would have been very useful if it supported stepping one source code instruction.

One of the main goals of ERDB was to improve the user experience, by mainly removing the hassle of using an external program to debug embedded systems. ERDB has achieved this goal by using the *probe-rs* library. *probe-rs* allows ERDB to access the debug target without starting another program, which greatly improves the user experience. There is also a *vscode* extension for ERDB that makes the user experience as good as other debuggers.

5.3 Debugger Comparison

The debugger ERDB needs to be compared to the already existing debuggers to see if any improvement is made to debugging optimized *Rust* code. The two most popular debugger used for debugging *Rust* code are *GDB* and *LLDB*. They are also the two debuggers that are supported by the *Rust* development team according to there *rustc-dev-guide* [Teaa].

The testing and comparison 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 different situations. This was repeated until one of the debuggers gave an unexpected result.

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. Also, the latest released version of each compiler was used to keep the comparison fair. At the time of writing the latest versions of debuggers are:

- *GDB* 12.1
- *LLDB* 14.0.6
- ERDB 0.2.0

5.3.1 Unoptimized Code Comparison

All the debuggers were first tested on unoptimized *Rust* code, to see that the debuggers were correctly installed and configured. While testing them there

were some differences between the debuggers that are interesting. The largest one being that *LLDB* is not able to evaluate *Rust enums*, which both *GDB* and ERDB are capable of.

Another difference found is that both *LLDB* and *GDB* evaluated a *f32* to 10.1999998, and ERDB evaluated it to 10.2. In the *Rust* source code the *f32* was assigned the value of 10.2, and all the debuggers returned that the raw bytes were $0x33332341$ in hexadecimal. Reading those raw bytes as a 32 bit float using little endian, gives 10.2.

5.3.2 Optimized Code Comparison

Debugging the code with optimization 2, *LLDB* differed a bit from the other two debuggers. It is still not able to evaluate *Rust enums*. Also, it wrote all 8 bit integers in hexadecimal format, but the values were correct.

GDB differed in that it evaluated all the 64 bit integers as 32 bit integers, thus it showed the wrong values. It did not have this problem when debugging the same code but unoptimized.

Compared to the other debuggers ERDB did not have any problems with evaluating variables. But, it sometimes does not give the correct stack trace when there are inline functions. This problem has only occurs on optimized code.

All the three debuggers where equally bad when it comes to the main problem of debugging optimized code. Which is the problem of many variables being optimized out, which makes debugging optimized code very hard.

Chapter 6

Discussion

6.1 Usefulness of *rust-debug*

Using the *rust-debug* library makes it a lot easier to get debug information form DWARF. One reason, is that it takes a lot less knowledge about DWARF to retrieve debug information. The documentation for *rust-debug* is a lot less text than the specification for the DWARF format. Which makes it a lot faster and easier to get debug information using *rust-debug*.

Also, the amount of code needed to get the debug information is a lot less using *rust-debug*, as the result of section 5.1 show. Using the amount of code lines to measure complexity is not the best method. But, when the difference is so large it is hard to deny that the completely has reduced.

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 not completely removed from the optimized code. But it turns out that *GDB* was right in that they were completely optimized out.

Keeping these expectations in mind, the result in section 5.3 is disappointing. Because, the goal was to create a debugger that could improve on this problem.

Unexpectedly, both *GDB LLDB* has other smaller problems. Such as *GDB* parsed 64 bit integers as 32 bit. Those small problems might be specific

to the debuggers Command Line Interface (CLI), and thus not present in other releases of the debuggers. But, the problems show that *erdb* has some small advantages.

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 ERDB is easier, because it removes the step of starting a separate program.

6.3 Accuracy of the DWARF Location Ranges

While working on *rust-debug* and ERDB it was discovered that values can sometimes live longer on the debugged target then described in DWARF. This means that some values can potentially be read when DWARF says that they have no location.

The documentation for *LLVM* [Teab] mentions three major factors that effect the variable location fidelity, they are:

1. Instruction Selection
2. Register allocation
3. Block layout

The documentation about the three major factors does not mention a lot about how the ranges could be set to tight. Thus, to be able to find if this can be improved on, one has to have a good understanding of *LLVM*, which is out of scope for this thesis.

The documentation also mention that some location changes are ignored by *LLVM* in some situations. One situation they mention, is that location changes is ignored for the prologue and epilogue code of a function. But, this should not be a problem, because most debuggers will step over the prologue and epilogue code.

Chapter 7

Conclusions and future work

The result from comparing the debuggers was a bit disappointing. Because, ERDB was not able to improve on the main problem with debugging optimized code. Which is that many times the debugger gives no useful information about the state. Often because the variables are optimized out, thus making it difficult for the user to understand the state of the program.

The debugging library *rust-debug* is successful in simplifying the process of retrieving debug information from DWARF. It is a extremely versatile library because it has very few requirements, where *gimli-rs* is one of the requirements. Also, it does not block the user from using *gimli-rs* directly.

Overall I would say that the goal of this thesis was achieved, but that there is still a lot that needs to be done. Both to the development of debugging tools, and the compiler.

7.1 Potential Future Debugging Improvement

The ranges for the location information seam to be set a bit to tightly by *LLVM* in some cases, meaning that the value still is in the debugged target for some time after the end address of the given address range. There are three major factors that affect the variable location correctness, they are mentioned in section 6.3. Thus, one potential improvement to debugging would be to improve how the variable location ranges are set.

7.2 Future Debugger Improvement

The debugger ERDB 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.

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 halt 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 instead.

Acronyms

CFA Canonical Frame Address. 9, 32–34

CIE Common Information Entry. 26, 32

CLI Command Line Interface. 17, 39, 41

DAP Debug Adapter Protocol. 17, 39, 41, 43

DIE Debugging Information Entry. 9, 26–32, 36–38

DWARF Debugging with Attributed Record Formats. 3, 7, 9, 13, 14, 16, 20, 24–30, 32–38, 41, 46, 47, 49, 54, 55, 57

ELF Executable and Linkable Format. 25, 29, 35

ERDB Embedded Rust Debugger. 3, 35, 38, 39, 45, 47–49, 51, 54, 57, 58

FDE Frame Description Entry. 26, 32, 33

GUI Graphical User Interface. 39

MCU Microcontroller Unit. 13–15, 17, 20, 39

PC Program Counter. 21, 30, 33, 50, 51

TCP Transmission Control Protocol. 41, 43

Bibliography

[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.

[RD17] 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. 2017, pp. 1274–1283. DOI: 10.1109/IPDPSW.2017.7.

[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).

[Mica] Microsoft. *Debugger Extension*. URL: https://code.visualstudio.com/api/extension-guides/debugger-extension. (accessed: 18.07.2022).

[Micb] Microsoft. *generator-code*. URL: https://github.com/Microsoft/vscode-generator-code. (accessed: 18.07.2022).

[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).

[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] The LLVM Team. *Source Level Debugging with LLVM*. URL: https://llvm.org/docs/SourceLevelDebugging.html. (accessed: 28.09.2021).

[XAM] XAMPPRocky. *Tokei*. URL: https://github.com/XAMPPRocky/tokei. (accessed: 29.07.2022).