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

November 29, 2022

Abstract

Debugging is an essential part of programming. No debugger is good at debugging optimized *Rust* code, which is a problem because unoptimized *Rust* code is slow compared to optimized. The ability to debug optimized code is important for embedded systems because of the close relation to hardware. Therefore, a tool like a debugger is handy because it enables the developer to see what is happening in the hardware when debugging embedded systems.

This thesis presents a debugger named ERDB and a debugging library named *rust-debug*. The goal of the library is to make it easier to create debugging tools by simplifying the process of retrieving debug information from the DWARF format. The goal of ERDB is to improve debugging for optimized *Rust* code.

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

ERDB will be compared to other debuggers to see if it has achieved its goal. The result highlight that the problem is in the compiler not generating the needed debug information.

Contents

List of Figures

Chapter 1

Introduction

A crucial 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 challenging because of the increasing complexity of computer programs and hardware. Therefore, better debugging tools have become essential to make the debugging process more accessible and time efficient. It is especially true for debugging embedded systems because of their close relation to hardware.

One of the first debugging tools made, and one of the most valuable types, is a debugger. A debugger is a program that allows the developer to control the debugged program in some ways, for example, continuing, halting, and resetting. It can also 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). It gives the developer a complete view of what the program is doing. Debuggers are also valuable tools for testing.

Debugging today works by having the compilers generate debug information when compiling the program. The debug information is stored in a file in a unique format designed to be read by a debugger or other debugging tools. One of the most popular formats is named DWARF, and it is a complex format that section 3.4 explains.

The debug information stored in the DWARF file can be used to locate the values of variables that, 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 contains the data type information of the variable. The debug information is used to interpret the raw value of a variable found in memory into a typed

value.

Variables stored in registers are a significant problem with debugging optimized code. Registers are faster to access than other memory and have a much lower capacity. Therefore, storing variables there often increases the speed of the programs, but the variables are not persistent because they get overwritten. It makes debugging a lot harder because the variables are very short-lived compared to variables located on the heap or call stack. We refer the reader to section 3.1 for more detailed information about registers, the call stack, and the heap.

1.1 Background

In the *Rust* programming language community, very few projects focus on debugging embedded systems and even less on improving the debugging of optimized *Rust* code. One of the larger projects focusing on debugging is named *gimli-rs*. One of the main goals of the projects 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 various MCUs and debugging probes. One of their newest tool, which is in early development, is a debugger that also uses the *gimli-rs* library to read the DWARF file.

There is some work done on the *LLVM* project to improve the generation of debugging information. However, there is no clear 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+]. Compared to other languages, for example, *C*, the optimized code can be about $1 - 3$ times faster than the unoptimized code, as seen in the paper [RD17]. Therefore, for a language like *C*, it is much more acceptable not to be able to debug optimized code because the difference is not that large compared to *Rust* code. However, in the case of *Rust*, the difference is significant. Therefore, it 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 to debug embedded systems is that the

Rust compiler will catch most of the errors. It is especially true regarding pointers and memory access, which are the two most common causes of bugs in programming languages like *C* and *C++*. Therefore, 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 that *Memory-mapped I/O* is used to perform *input/output (I/O)* between the *CPU* and peripheral devices. Also, a debugger is an excellent tool for showing these peripherals' state, making the development and testing much more straightforward. Another reason is that hardware bugs are much easier to find if it is possible to inspect the state of a device quickly.

Another motivation for creating a debugger for embedded systems is that the *Rust* team's two supported *Rust* debuggers are *LLDB* (part of the *LLVM* project) and *GDB*, which at the time of writing, both require another program to interact with the MCU. An example of such a program is *openocd* (openocd homepage [Rat]), which works as a *GDB* stub to which the debugger connects. It complicates the process of debugging, especially for new developers.

Most debuggers for *Rust* code are written in other programming languages, and there are not many debugging tools written in *Rust* yet. Therefore, one of the critical motivations for this thesis is to write a debugger in *Rust*, which will also lead to the debugger having all the benefits of memory safety and the excellent ecosystem *Rust* provides.

1.3 Problem definition

This thesis's main problem is improving the experience of debugging optimized *Rust* code on embedded systems. It will be achieved by creating a new debugger and debugging library solving the three following problems:

- Two of the most popular debuggers, *GDB* and *LLDB*, require another program to debug embedded systems. This problem makes both of them harder to use and is a bad user experience.
- Both *GDB* and *LLDB* often display that variables are optimized out when debugging optimized code, even with the less aggressive optimization options. Their messages are very vague and can make it impossible to understand what is happening in the program.
- Retrieving debugging information from the *DWARF* format is challenging to do. It makes creating tools that require debug information difficult and time-consuming to implement.

This thesis aims to create a new debugger and library that addresses the abovementioned problems. The debugger should also support the most common debugging features listed in section 1.4.

1.4 Delimitations

Currently, there are a lot of different debugging features that other debuggers have. Many of them are advanced and complicated to implement. Therefore, the number of features is limited to the most important ones, limiting the scope of this thesis. The following is the list of features the debugger is required to have:

- Controlling the debugging target by:
  - Continuing execution.
  - Halting execution.
  - Reset execution.
  - Stepping
- Set and remove breakpoints.
- Virtually unwind the call stack.
- Evaluating variables and their 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 that the debugger knows specific information about the target system, e.g., the architecture. Supporting every MCU out there would be too big of a scope for this thesis. Therefore, the debugger must only work with the target system *Nucleo-64, STM32F401RET6 64 pin*.

The compiler backend *LLVM* that *rustc* uses support two debugging file formats, according to their documentation [Teab]. One is the format DWARF, and the other is *CodeView* which *Microsoft* develops. To make a debugger that supports both formats would require much extra work that does not contribute to solving the main problems of this thesis. Therefore, it

has been decided only to support the DWARF format because it has good documentation.

This thesis's scope does 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; therefore, this thesis cannot include such a change or addition. Another reason is that it would take some time before the Rust compiler implements the new version.

The DWARF format is very complex but well-documented. Due to this, the description 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. As a result, 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. Other papers that take the same approach are [Adl96; Cop94]. The two main problems with this approach are that the debug information size gets larger, and the compilation times gets longer.

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 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 other tools in its library to access and control the debug target and uses 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, is that *rust-debug* is designed to be platform and architecture independent. The *probe-rs* library is designed to provide tools only for embedded MCUs and debug probes. 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 the *probe-rs* debugger is not able to evaluate some of the more complex data structures in DWARF, but *rust-debug* can evaluate them.

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

- Platform and architecture independence.
- Has a wider range of use cases.
- Fewer 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.
- Increased code complexity.

Chapter 3

Theory

3.1 Registers and Memory

The two main places a computer stores values are in registers or 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. Therefore, registers are suitable for storing values used many times in a concise amount of time. Memory is much slower but has a lot more space. Therefore, it is better for storing long-lived values.

The compiler decides when and where values are stored, and it decides this during the compilation of the program. Therefore, the developer has little control over where the values are stored.

Values located in memory are either stored on the call stack or the heap. The call stack holds all the values of arguments and variables for all the called functions that have not finished execution. In contrast, the heap contains 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 data if it fits within the size limit. Some registers are reserved for a particular use. The Program Counter (PC) register is among the most important and contains the succeeding machine code instructions to be executed. The registers reserved for a particular use differ depending on the processor architecture.

3.1.2 Call Stack

The call stack consists of each function call that has not terminated. A stack is a data structure consisting of several elements stacked on top of each other, and the only two operations available for a stack are 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. Another critical characteristic is that only the top element can be accessed; therefore, all the above elements need to be popped to reach the lower elements.

Stack/call frames are the elements that make up the call stack. Each stack frame contains the values of the arguments and variables for a function call, and stack frames usually contain a return address. When a function is finished, the return address is written to the PC register, making the program jump to the return address. The return address usually points back to the previous function. Also, when a function has finished executing, its stack frame will be popped/removed from the stack. See figure 3.1 for an example of a call stack and stack frames. The values starting with *0x* to the right in the figure are hexadecimal addresses.

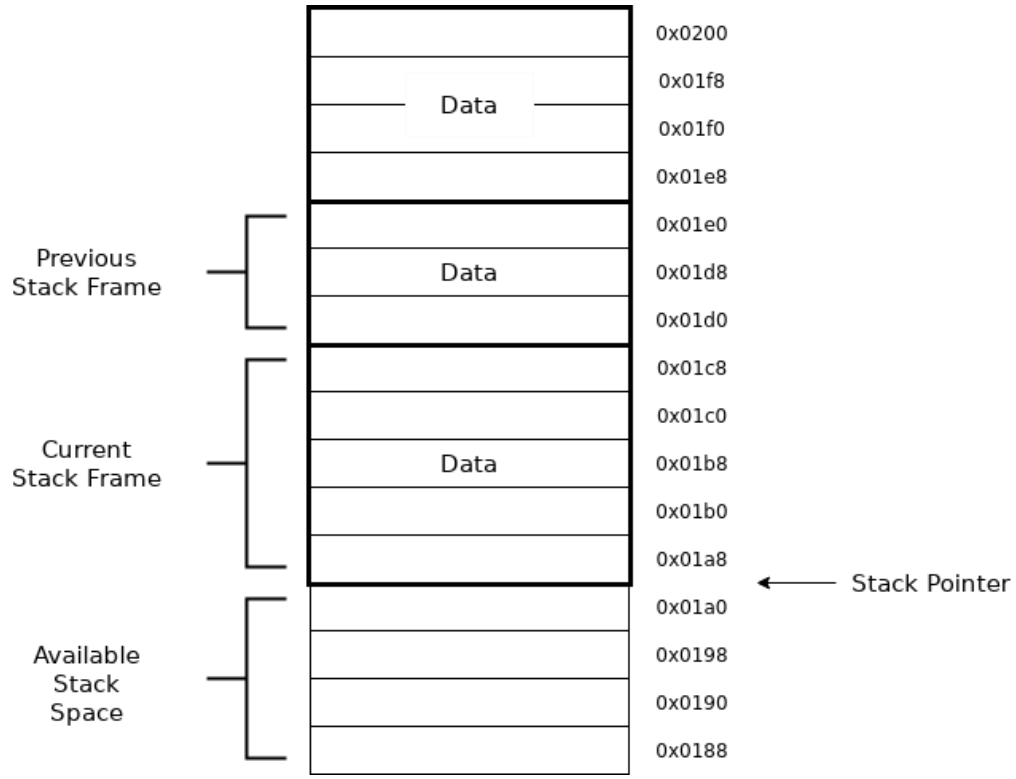

Figure 3.1: A visual example of how a stack and stack frames can look.

3.2 Prologue and Epilogue Code

Some registers need to be preserved during the execution of a subroutine. The prologue code pushes those registers onto the call stack at the start of the subroutine. At the end of the subroutine, the epilogue code pops the stored registers from the stack. It is the compiler that generates the prologue and epilogue code.

Note that the prologue and epilogue code is only sometimes continuous blocks of code at the beginning and end of a subroutine. Read [Com10] pages 126-127 to learn more about the prologue an epilogue.

3.3 Debugging

Debugging refers to the process of finding and resolving errors, flaws, or faults in computer programs. A bug is an error, flaw, or fault in computer science. Bugs are the cause of software behaving unexpectedly. Most bugs arise from poorly 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, which is done by sending some input to the code and comparing the result to the expected result. The amount of code tested in a test can vary from one function to the whole program. Another debugging method is a control flow analysis, which analyzes the order in which the instructions, statements, or function calls are. There are many more methods to debug computer programs, but they all try to achieve the same thing. That is, to give the programmer a deeper understanding of what is happening.

There is no better tool for understanding a program than a debugger. That is because a debugger can display the state of a program, and it has control of its execution. A debugger enables the inspection of every part of a program, which is especially true for modern debuggers with much more advanced features.

3.3.1 Debugger

A *debugger* is a computer program used to test and debug other computer programs. The two main functionalities of a debugger are, 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 are by continuing, halting, stepping, and resetting the execution. Continuing the execution means that the target program continues the execution from the current stopped location. Halting the target program can often be done in two ways, the first is to stop the execution where it currently is, and 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. Breakpoints are very useful for inspecting specific points in the code, while the other way of halting is used more when the location is unknown. Stepping in the process of continuing the target program execution for only a moment, often just until the following source code line is reached, there are many stepping variants. Lastly, resetting means that the target program will start execution from the beginning.

Most debuggers display the state of the target program relative to the source code. Therefore, if the target program has halted, the closest source code location will be shown as the halt location. 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, variables (type and value), and evaluate an expression. There are a lot more functionalities that a debugger can have, but these are some of the most common ones.

3.4 DWARF

This section will explain how the debug information format Debugging with Attributed Record Formats (DWARF) version 4 is structured and what functionality the different sections have. However, it will not explain every detail about the DWARF format because the DWARF specification already does that. Instead, this section will focus on the type of information stored in the DWARF format and how to use it. For more details, see the DWARF specification [Com10].

3.4.1 Dwarf Sections

The DWARF format consists of sections containing different information. Pointers consisting of different offsets connect some sections. Many DWARF attributes contain these offsets. See section 3.4.3.1 for information about attributes. Figure 3.2 shows the DWARF sections and which ones point to each other. All DWARF sections are stored in an ELF file, a binary file format divided into different sections. The ELF format contains a table describing what each of its sections contains and where they begin and end.

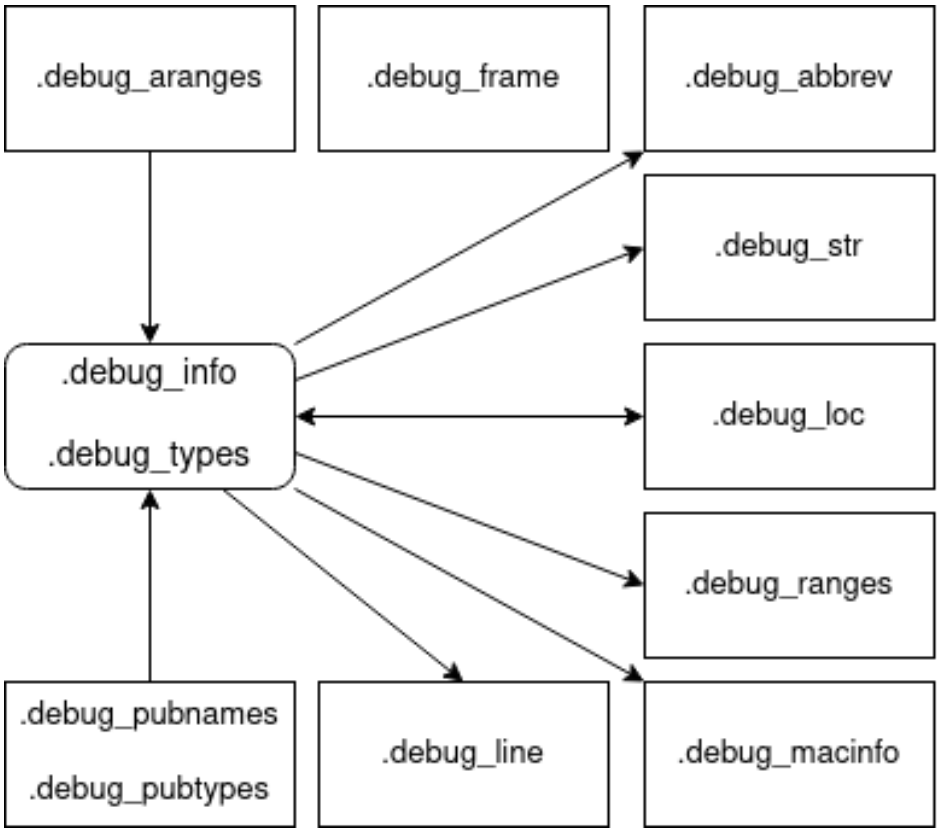

Figure 3.2: Diagram of the different DWARF sections and their relations to each other.

3.4.1.1 *.debug_abbrev*

The DWARF section *.debug_abbrev* contains all the abbreviation tables used to translate abbreviation codes into their official DWARF names. Abbreviation codes are used in Debugging Information Entry (DIE) tags, DIE attribute names, and more. An abbreviation code is translated by looping through the entries in one of the abbreviation tables until it finds the matching one. For further details, we refer the reader to section 7.5.3 in [Com10].

3.4.1.2 *.debug_aranges*

All the information needed to look up compilation units using machine code addresses is in the DWARF section *.debug_aranges*. Compilation units contain debugging information for a range of machine code addresses. Each compilation unit has a start address followed by a length. Therefore, to find the correct compilation unit, 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 unwind the call stack virtually is kept. It consists of two structures named Common Information Entry (CIE) and Frame Description Entry (FDE) and is entirely self-contained. Virtually unwinding the call stack is complex. Therefore, for further details, we refer the reader to 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 in DIEs which are a low-level representation of the source code. DIEs have a tag that describes what it represents. An example tag is *DW_TAG_variable*, meaning 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* consists of several DWARF units and other debug information. Therefore, this is one of the most valuable sections in DWARF because it contains the relation between 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 generated from a specific line and column in the source file. It contains the source directory, file name, line number, and column number. Pointers to this information are in some DIEs. Therefore, enabling the debugger to get the source location of a DIE. Section 6.2 in [Com10] explains more about the information in the *.debug_line* section.

3.4.1.6 *.debug_loc*

The location of the variables is in location lists, which are in the *.debug_loc* section. Each location list entry holds several operations that, when performed, will yield the location of a value. Some DIEs in the *.debug_info* section have attributes that point to entries in the location lists. The most common of these attributes is named *DW_AT_location*, which is often present in DIEs

representing variables. See figure 3.2 for a graph showing the relationship between these two sections.

3.4.1.7 *.debug_macinfo*

The *.debug_macinfo* section contains all the macro information stored in entries representing the macro after the compiler has expanded it. Some DIEs in the DWARF section *.debug_info* point to these entries, and those pointers are in the DIEs attribute *DW_AT_macinfo*. For further information, 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 is *.debug_pubnames*, which is for finding functions, variables, and objects, and the other is for finding types. This section is named *.debug_pubtypes*. For further details, we refer the reader to section 6.1.1 in [Com10].

3.4.1.9 *.debug_ranges*

DIEs with a set of non-contiguous addresses will have an offset to the section *.debug_ranges* instead of an address range. The offset points to the start of a range list containing range entries, which are for knowing which of the machine code addresses the DIE is active. The DWARF section *.debug_ranges* is 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 for storing all the debug information strings. An example of these strings is the names of the functions and variables. DIEs representing variables and functions will have an offset that points to these strings in the attribute *DW_AT_name*.

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 compilation 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 mainly generate one compilation unit for each project/library. There are some cases when it generates multiple partial compilation units instead. The compilation units are 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 many valuable debug information about the source file, one being the compiler used and its version. It also says which programming language the source file is in as well as the directory and path of the source file.

3.4.3 Dwarf Debugging Information Entry

One of the most important data structures in the DWARF format is the DIE. A DIE is a low-level representation of a small part of the source code. The most common source code objects that the DIEs represent are functions, variables, and types. The DIEs are in a tree structure called a DIE tree. Each DIE tree will often represent a whole compile unit or all types used in the source code. The ones representing compile units are found in the DWARF section *.debug_info*, while the ones representing type are in the DWARF section *.debug_type*. The DIEs representing types are named type DIEs.

3.4.3.1 Dwarf Attribute

All the information stored in DIEs is in unique attributes. These attributes consist of a name and a value. The attribute's name is unique and describes what type of information the attribute's value has. All the attribute 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 is abbreviated to its abbreviation code which can be decoded using the *.debug_abbrev* section.

3.4.3.2 Example of a DIE

Figure 3.3 shows an example of a DIE. The figure is a screenshot of the output from the program *objdump* run on an ELF file. The first line in the figure begins with the number 8, representing the depth in the DIE tree where this DIE is located. The following number is the current offset into this compile unit. All the other lines in the figure also start with their offset.

Then comes *"Abbrev Number: 9"* on the same line. It is an abbreviation code that translates to *DW_TAG_variable*. This tag means that the DIE represents 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 3.3: An example of a DIE representing a variable named *ptr*. This example is the output of the tool *objdump* run on an ELF file.

The attribute *DW_AT_location* (seen in figure 3.3) has information about 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", which is the variable's name. Attributes *DW_AT_decl_file* and *DW_AT_decl_line* in the figure contain offsets into the section *.debug_line*. Those offsets can be used to find the source file path and the line number from which the DIE was generated. Lastly, the attribute *DW_AT_type* contains an offset into the section *.debug_types*, which points to a type DIE that has the type information for this variable.

3.4.4 Evaluate Variable

Evaluating a variable's value is complicated because there are many data types and combinations of data types. Therefore, the following example is presented to simplify the explanation.

In 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 named *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*. This parameter *val* will be used as an example of evaluating a variable.

Take note that the function DIE named *my_function* has two attributes named *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. Other attributes in the example will not be mentioned because they are not needed to determine the attribute's value.

```
<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 an ELF file.

3.4.4.1 Finding Raw Value Location

Examining the DIE for the argument *val*, there is an attribute named *DW_AT_location*. The value of that attribute is several operations, and 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 several operations, and those operations are under the attribute *DW_AT_frame_base*. Figure 3.4 describes the *frame base* location with the operation *DW_OP_reg7*. The operation *DW_OP_reg7* describes that the value is located in register 7 (see [Com10] page 27). Therefore register 7 needs to be read to get the value of the *frame base*.

Now that the value of the *frame base* is known, we can proceed to calculate the location of the parameter *val*. As mentioned previously, the location of parameter *val* is the *frame base* minus 2. Therefore, the value of *val* can be read from memory at the address of the *frame base* minus 2. However, the value must also be parsed into the type of *val*. See section 3.4.4.2 for how to parse the value into the correct type.

3.4.4.2 Parsing the Raw Value

The first problem with parsing the value of the parameter *val* into the correct type is to know what type the parameter has, and 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 shown 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 $0x6233$. That type DIE has the tag *DW_TAG_base_type*, which means it is a standard type built into most 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 ELF file.

In this example, the type DIE has three attributes that are used to describe the type. The first attribute is *DW_AT_name*, which describes the type's name. In this case, the name of the type is *i16*, which can be seen in figure 3.5. The following attribute is *DW_AT_encoding*, which describes the type's encoding. 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]. The last attribute is *DW_AT_byte_size*, which describes the type size in bytes. In this case, a byte size of 2 means that the type is a 16-bit signed integer. The last step is to parse the bytes of the value into a signed 16-bit integer.

3.4.5 Virtually Unwind the Call Stack

The call stack is virtually unwinding by recursively unwinding a stack of *subroutine activations*. It is named 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 stack pointer value, the related stack frame is also known. Therefore, successfully unwinding all the *subroutine activations* will result in a 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 named Frame Description Entry (FDE). An FDE contains a table used for unwinding registers and the CFA of an activation. The other data structure is named 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 PC value.

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 too little information about the other activations is known. The next step is to find the CIE and FDE that contain debug info on the subsequent activation. When those are known, the values of the subsequent activation can be evaluated as described in section 3.4.5.1. The unwinding is then repeated for the rest of the activations.

3.4.5.1 Subroutine Activation

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

The address of the stopped code location is 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. Therefore, it can be read if the stack pointer is known. It is true 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 registers 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 their 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).

The CFA and the preserved register values can be restored using tables 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 contain 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 identify the code location to which all the row's virtual unwinding rules apply. The next column is unique and contains the CFA's virtual unwinding rules. The remaining 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 several virtual unwinding rules. The ones for the registers are named register rules. Some of them are easy to use, such as the register rule *undefined*, and this rule means that it is impossible to unwind that register. Other rules require 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 an address that can be calculated by adding the CFA address and 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 row which contains the address that is closest but not greater than the current address. The 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

Three projects comprise the implementation, the first being a debug library named *rust-debug*. Its purpose is to simplify retrieving debug information from the DWARF format. The second project is the debugger ERDB which implements the Debug Adapter Protocol (DAP) protocol. The last project is a *VSCode* extension that launches the ERDB debugger and connects 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 must 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 know the DWARF format, the target system, and the programming language/compiler. The *rust-debug* library is 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 solutions for are the following:

- Retrieve the source code file location where 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.

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 where any data was declared is stored in three attributes on the DIE representing that data. One attribute contains indirect information about which file the data was declared, another contains the line number, and the last contains the column number.

The attribute contains the file path and name information named *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 line number information table, and each compilation unit has one such table. File index 0 is reserved and used when the source file is unknown.

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

Rust-debug provides a function that performs the file index lookup and gets the line and column numbers. The function requires that a DIE and related compilation unit is given.

4.1.2 Accessing Memory And Registers

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

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

All the functionalities in *rust-debug* that require access to the registers require a struct named *Registers*. This struct contains a *HashMap* with the register numbers and their corresponding values, and it also contains the register number of some special registers, like the PC register. Therefore, it is up to the user to fill in the correct register numbers and values. An example of implementing this can be found in the git repository for ERDB [Luna].

The functionalities that require access to the debug target's memory require a struct that implements the *rust-debug* trait *MemoryAccess*. This trait is straightforward and only has one function: reading various bytes at a given address. Users can create a struct that implements that function almost however they want. An example of implementing this using the *probe-rs* library can be found in the git repository for ERDB [Luna].

4.1.3 Evaluating Variables

To make evaluating a variable's value and retrieving other information about the variable easy, *rust-debug* provides a struct named *Variable*. That struct only has one function named *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. The structure is similar to how the information is stored in the DWARF format. However, the value of each branch is evaluated into a primary value type, such as a signed integer. Therefore, it removes the most challenging 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 to understand how variables are structured 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, which is the furthest down the DIE tree, in the branch where the current address is in range. Therefore, 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 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. Each *CallFrame* struct

contains information about each of the stack frames. The essential information the struct contains is the unwound register values, but it also contains other helpful information, such as the stack frame's start and end memory addresses. 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. It includes the subroutine's name and the declaration location, which is found using the function described in section 4.1.1. It also includes the variable information retrieved using the function described 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 has rows containing information on a line from the source code. 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 into the function. Otherwise, the one with the closest column number is returned.

4.2 Embedded Rust Debugger

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

The debugger consists of three modules. The first one is named CLI. Its primary function is to handle the input and output from the terminal. The second module is named *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 are for supporting the DAP protocol. The last module

is named *Debugger*. It is for getting debug information and controlling the debug target.

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

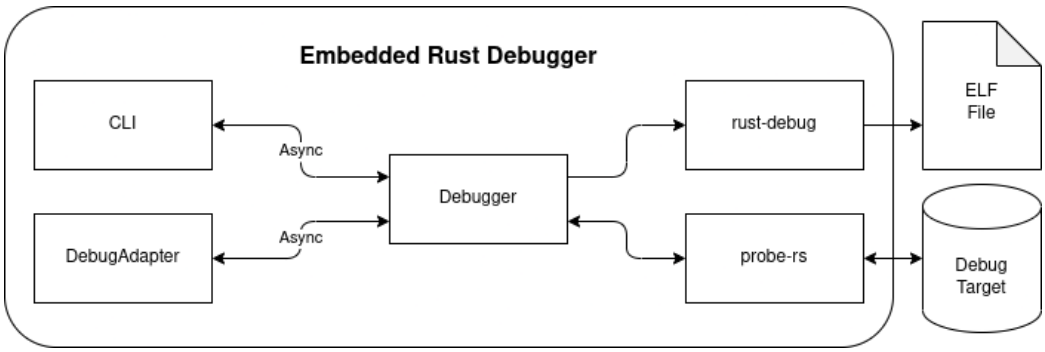

Figure 4.1: Diagram showing the structure of ERDB.

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 from the terminal into a request that the *Debugger* module can understand. The second is to convert the responses from the Debugger module to text and output that text to the terminal.

Reading the input from the terminal is done asynchronously, which enables other tasks to be done while the CLI waits 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 matching the first word in the string to a command. Also, each command has a parser that is then used to parse the rest of the string. The resulting *DebugRequest* enum is then returned to the main loop and sent to the *Debugger* module.

After the *Debugger* module has done its work, the result is printed to the terminal. It 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 variant has a unique message printed to the terminal, and the messages often contain data from the *DebugResponse* enum. A helpful message is printed to the terminal if the input text cannot be parsed.

There is also a *DebugEvent* enum, that is printed 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 task's primary 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 requests, which are forwarded to the *debugger* module. It also handles translating the responses and events from the *debugger* module.

The DAP protocol communicates through Transmission Control Protocol (TCP), requiring the *debug adapter* module to have a TCP server. All the TCP communication is handled asynchronously, and it is to enable the other tasks to run simultaneously. Currently, only one TCP connection can be open at a time, and it is because the rest of the system still needs to support multiple simultaneous debugging sessions.

4.2.3 The Debugger Module

The *debugger* module primarily consists of one struct named *DebugHandler*. Its primary job is to get the debug information from the DWARF format, and another is to keep track of the debug targets state. This struct only has two methods for accomplishing its jobs, one for polling the state of the debug target. The other method, named *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 perform some task related to the given argument, which often involves 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 requires values from the debug target, such as the values stored in registers and memory addresses, and 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 100th millisecond to detect if the debug target has continued or stopped the program execution. Other tasks can be handled in the time between the polls, such as parsing the input from the CLI or doing a stack trace. It is possible because it is an asynchronous function that sleeps for 100 milliseconds and then polls the debug target state, and that function repeats if the debug target is being debugged. Therefore, the debugger can simultaneously handle both inputs from the user and state changes in the debug target. A state change in the debug target is an event in ERDB.

4.2.3.3 Optimization of Repeated Variable Evaluation

All the variables and stack frames are temporarily stored, improving the debugger's performance. It allows for fast repetitive lookup of debugging information and simplifies the implementation. All the temporarily stored information is removed when the state of the debug target changes, and this is to ensure that the correct information is returned.

Note that if a request is received to evaluate one variable, the debugger will perform a whole stack trace instead.

4.3 VSCode Extension

Creating a *VSCode* extension is simple because *Microsoft* has a tool named *vscode-generator-code* [Micb] to generate an empty extension. The provided documentations also explain 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 are more difficult because a debug adapter needs to translate the DAP protocol messages to a command or multiple commands that the debugger understands. The difficulty depends on the debugger implementation. The other type of debug extension is a wrapper for a debugger that already implements the DAP protocol. These extensions are elementary to create because they only need to start the debugger and connect to it using the DAP protocol. Read *Microsoft's* documentation to learn more about *VSCode* debug extension [Mica].

The debugger Embedded Rust Debugger implements the DAP protocol over a TCP server. The *VSCode* extension is a wrapper that starts the debugger TCP server and then connects to it. Connecting to the debugger server over TCP is easy because *Microsoft* has a library named *VSCode* that

does it. The extension also captures the debugger logs 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 named *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 getting debug information from the DWARF format. It requires many lines of code and knowledge about the DWARF format to get the wanted debug information. One way to measure if *rust-debug* makes it easier is to compare the number of code lines required to create a debugger with and without *rust-debug*.

Unfortunately, no other debugger has a similar feature set as ERDB. Also, it would be unfair to compare ERDB with a debugger like *GDB*, which has many 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 was counted using the tool *tokei* [XAM]. *Rust-debug* contains 3626 lines of *Rust* code, and the debugger module contains 1358 lines. Also, the whole debugger ERDB contains 3068 lines of *Rust* code.

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

5.2 Evaluating *ERDB*

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

ERDB only supports hardware breakpoints, limiting how many breakpoints can be set. The number of breakpoints can be increased significantly

by implementing software 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, and it would have been beneficial if it supported stepping one source code instruction.

One of the main goals of ERDB is 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, significantly improving 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 debuggers used for debugging *Rust* code are *GDB* and *LLDB*. They are also the two debuggers supported by the *Rust* development team, according to the *rustc-dev-guide* [Teaa].

The testing and comparison of the three different debuggers are made manually on some example code. See the git repository [HL] for the example code. The example code was modified to test how well the three debuggers handled different situations.

A breakpoint was added to the same machine code instructions to keep the comparison fair when comparing the debugger. It ensures that all three debuggers stop on the same instruction. Also, the latest released version of each compiler was used to keep the comparison fair. 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 most

significant is that *LLDB* cannot evaluate *Rust* enums, which both *GDB* and *ERDB* can.

Another difference was that both *LLDB* and *GDB* evaluated an *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. All the debuggers returned that the raw bytes were $0x33332341$ in hexadecimal, and 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 slightly 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. Therefore, it showed the wrong values and did not have this problem when debugging the same code but unoptimized.

Compared to the other debuggers, ERDB had no problems evaluating variables. However, it sometimes does not give the correct stack trace when there are inline functions, and this problem has only occurred on optimized code.

All three debuggers were equally bad regarding the main problem of debugging optimized code: the problem of many variables being optimized out. That makes it very hard to understand what is happening in the program and, therefore, difficult to debug.

Chapter 6

Discussion

6.1 Usefulness of *rust-debug*

The *rust-debug* library makes it much easier to get information from DWARF. One reason is that it requires much less knowledge about DWARF. Also, the documentation for *rust-debug* is much less text than the specification for the DWARF format. Therefore, reading and understanding take less time, making it easier to use.

The amount of code needed to get the debug information is much less using *rust-debug*, as section 5.1 shows. Using the number of code lines to measure complexity is not the most accurate method.

6.2 Debug Experience For Optimized Rust Code

At the beginning of this thesis, it was thought that *GDB* could not evaluate variables located in registers. This thought came from the observation that *GDB* mainly prints that the variables are *optimized out* when debugging optimized code. Therefore, it was thought that those variables were not wholly removed from the optimized code. However, *GDB* was right in that they were wholly optimized out.

With 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* and *LLDB* have other minor problems. Such as, *GDB* parsed 64-bit integers as 32-bit. Those minor problems might be specific to the debugger's Command Line Interface (CLI) and therefore not present in other releases of the debuggers. However, the problems show that

ERDB has some slight advantages.

6.2.1 Debugging Rust Code on Embedded Systems

The experience of debugging *Rust* code on embedded systems with both *LLDB* and *GDB* could be better. It requires much configuring and a program like *openocd*, which 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 variables could sometimes live longer on the debugged target than described in DWARF. It means that some variables can potentially be read when DWARF says they have no location.

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

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

The documentation about the three significant factors does not mention how the ranges could be set too tight. Therefore, to find out if this can be improved, there has to be a good understanding of *LLVM*, which is out of the scope of this thesis.

The documentation also mentions that *LLVM* ignores some location changes in some situations. One situation they mention is that location changes are ignored for a function's prologue and epilogue code. However, 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 could have been a lot better. Because ERDB could not improve on the main problem with debugging optimized code, i.e., variables are often displayed as optimized out, which makes it impossible for the user to understand what is happening in the program.

The debugging library *rust-debug* successfully simplifies retrieving debug information from DWARF. It is a highly versatile library because it has very few dependencies, whereas *gimli-rs* is one of the dependencies. Also, it does not block the user from using *gimli-rs* directly.

Overall, the goal of this thesis was achieved. However, many developments still need to be done to the compiler and debugging tools.

7.1 Potential Future Debugging Improvement

The ranges for the location information seem to be set a bit too 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. Three significant factors affect the variable location correctness, as mentioned in section 6.3. A 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. Therefore, there are many more features that could be added. One important one, left for future work is the different stepping variations. The ability to step one source code instruction and step in and out of functions is

valuable when debugging because it makes it easy to traverse the code while debugging.

One of the main problems with debugging optimized code is that the variables are stored in registers and never pushed to the stack. It causes the variables to be overwritten when they are not needed anymore. That fact makes debugging very hard because the user must halt the execution when the variable still exists. However, if the debugger could get the last value of the variable and store it, the last known value of the variable could then 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).