Master Thesis

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

August 26, 2021

1 Abstract

TODO

Contents

2 Introduction

Ever since the first programming language was made debugging has been a key part of the programming process. 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 more important to make the process of debugging easier and more time efficient.

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 a very useful tool for debugging but it is also a very useful tool for testing.

Debugging today works by having the compilers generate debug information when compiling the program. The debug information is then stored in a file, the file is formatted using special file format designed to be read by a debugger or a another debugging tool. One of the most popular of these file formats is the format Debugging with Attributed Record Formats (DWARF). It is a complex format that is explained in detail in section 4.4.

The debug information stored in the DWARF file can be used to find the location of variables, in most cases the value of variables are stored in memory on the call stack. A stack is a data structure for storing information and is usually used to store all the variables for each function that is currently being executed(see section 4.1 for more information). Thus a debugger can use the debug information to find the location of a variable and then evaluate the value of it, which is then displayed to the user of the debugger.

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

2.1 Background

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

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

2.2 Motivation

The main motivation is that optimized *Rust* code can be $10 - 100$ times faster then unoptimized code [Net+]. That is a very large difference in speed compared to other compilers like *Clang* for example, were the optimized code is about 1-3 times faster then the unoptimized code [RD]. Thus for language like *C* it is much more acceptable to not be able to debug optimized code because the different isn't that large compared to *Rust*. But in the case of *Rust* the difference is so large that some programs are too slow to run without optimization. This causes the problem that the code cannot be debugged because debuggers don't work well on optimized code and unoptimized code is too slow to run. Because of that there is a need for better debuggers that can provide a good experience debugging optimized *Rust* code.

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

Another motivation for creating a debugger for embedded systems is that the two supported debuggers for *Rust* by the *Rust* team are LLVM and GDB, which both requires another program to interact with the MCU. An example of such a program that is commonly used is *openocd*, which needs to be setup as a GDB server that the debugger connects to. This complicates the process of debugging and makes for a bad experience, especially for new developers.

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

2.3 Problem definition

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

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

2.4 Delimitations

One of the main problems for getting a debugger to work for optimized code is getting the compiler to generate all the debug information needed. In the case of the *Rust* compiler rustc it is the LLVM library that mostly handles the debug information generation. LLVM is a very lager project that many people are working on and thus improving on LLVM is out of scope for this thesis, the same goes for improving rustc.

The compiler backend LLVM that rustc uses supports two debugging file formats that holds all the debug information. One of them is the format DWARF, the other one is *CodeView* which is developed by *Microsoft*. To make a debugger that supports both formats would be a lot of extra work that doesn't contribute to solving the main problem of this thesis. Thus it has been decided to only support the DWARF format because it has very good documentation.

The scope of this thesis also dose 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 release in the span of this thesis, it would take a lot of time before the *Rust* compiler had been updated to use the new standard.

The DWARF format is very complex and is very well explain in there documentation. Thus the explanation of DWARF in section 4.4 will not go into every detail of DWARF. Instead it will focus on explaining the minimum needed to understand the implementation of the debugger. Checkout the document [Com10] for more information on DWARF.

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

- Controlling the debugging target by:
  - Starting/Continuing execution.
  - Stopping/Halting execution.
  - Reset execution.
- Set and remove breakpoints.
- Virtually unwind the call stack.
- Evaluate variables.

- Find source code location of functions and variables.
- A Command Line Interface (CLI).
- Support the *Microsoft* Debug Adepter Protocol (DAP).

When debugging code on embedded systems the debugger needs to know a lot about the hardware and some of these things differ depending on the hardware. Here is a list of some of those things:

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

To support all the different MCUs would be to much work for this thesis. Thus the debugger is limited to work with the *Nucleo-64 STM32F401* card because it is the one that was available. And it will only support the *arm Thumb mode* instruction set..

2.5 Thesis structure

TODO

3 Related work

TODO

4 Theory

4.1 Registers and Memory

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

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

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

4.1.1 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 withing the size limit of the registers. Some of the registers are reserved for special use, one of the most important ones is the Program Counter (PC) register. This register always holds the address of the next machine code instruction that will be executed. Which of the registers that are reserved for special use is different depending on the processor.

4.1.2 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 are not finished executing. A stack is a data structure that consist of a number of elements that are stacked on top of each other and the only two operations available for a stack is push and pop. The push operations adds a new element on top of the stack and the pop operation removes the top element of the stack. Other key characteristics are that it is only the top element that can be access thus too reach the lower elements all the above elements needs to be popped.

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

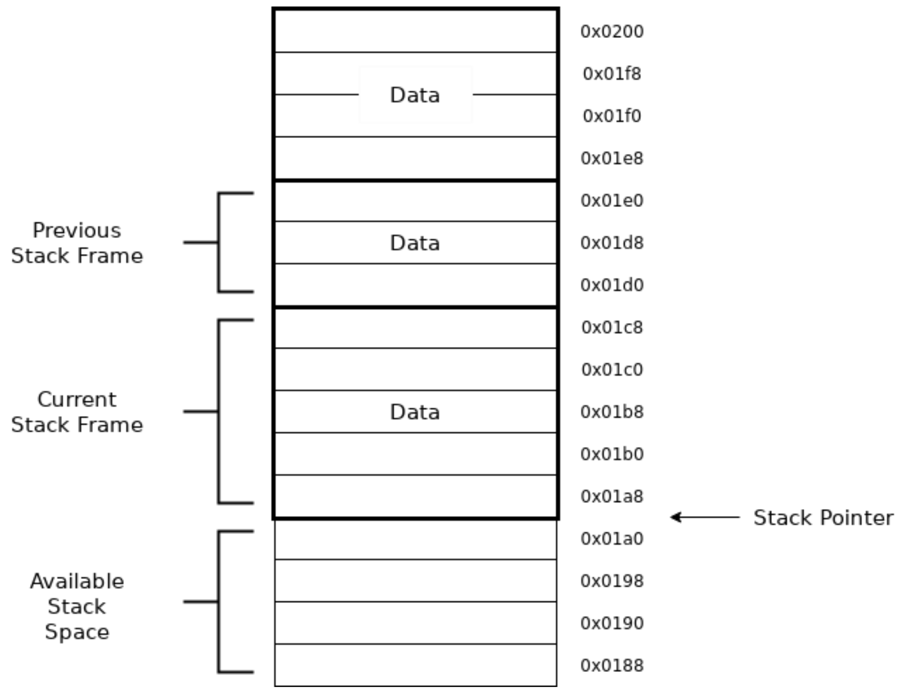

Figure 1: An visual example of how a stack and stack frames can look.

4.1.3 Heap

TODO

4.2 Prologue and Epilogue Code

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

One thing to note is that the prologue and epilogue code are not always continues 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 more of these special cases that the compiler does and some that are hardware specific, to read more about them see [Com10] page 126-127.

4.3 Debugging

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

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

4.3.1 Debugger

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

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

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

4.4 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. But it will not explain every detail about the DWARF format, because the DWARF format has a great specification that goes into detail how it is structured and work. See [Com10] for the DWARF specification version 4.

4.4.1 Dwarf Sections

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

4.4.1.1 *.debug_abbrev*

The DWARF section *.debug_abbrev* contain all of the abbreviation tables which are used to translate abbreviation codes into its official DWARF names. Some of the things these abbreviation code are used for are DIE tags and DIE attribute names. To translate a abbreviation code one has to go through each entry in the table until the one with the same abbreviation code is found. Checkout section 7.5.3 in [Com10] to learn more.

4.4.1.2 *.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 have information on. These ranges consists of a start address followed by a length. Thus to find the compilation unit having the information about the current state. The user only needs to check if the current address is between the start address and the start address plus the length. To read more about this section checkout section 6.1.2 in [Com10].

4.4.1.3 *.debug_frame*

In the DWARF section *.debug_frame* the information needed to virtually unwind the call stack is kept. This section is completely self-contained and is made up of two structures called Common Information Entry (CIE) and Frame Description Entry (FDE). Virtually unwinding the call stack is complex, thus to learn more about that checkout section 4.4.5 and section 6.4.1 in [Com10]

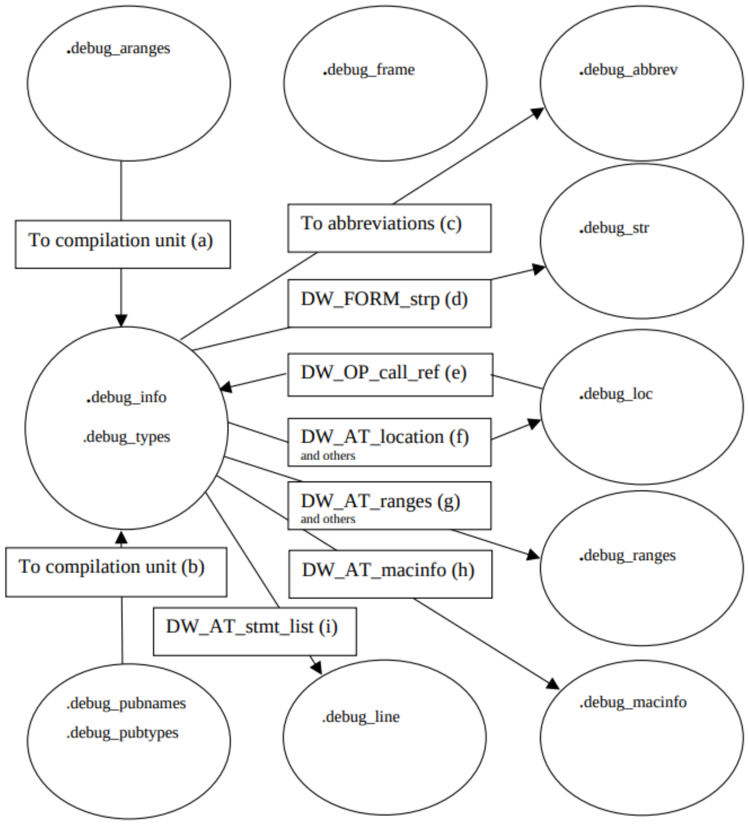

Figure 2: Diagram of all the different DWARF sections and there relations to each other. This diagram is taken directly from the DWARF specification [Com10].

4.4.1.4 *.debug_info*

Most of the information about the source code are store in DIEs which are low-level representation of the source code. DIEs have a tag that describes what it represents, an example tag is *DW_TAG_variable* which means that the DIE represents a variable from the source code. All DIEs are stored in trees, which is a common data structure. Each compilation unit will have at least on of these trees made up of DIEs and each tree is stored in a DWARF unit. The trees are structured the same as the source code, which makes it easy to relate the source code to the machine code.

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

4.4.1.5 *.debug_line*

The DWARF section *.debug_line* holds the needed information to find the machine addresses that is generated from a certain line and column in the source file. It is also used to store the source 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.

4.4.1.6 *.debug_loc*

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

4.4.1.7 *.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 pointer to by DIEs in the *.debug_info* section and those pointers can be found in the attribute *DW_AT_macinfo*. This section is a little bit complex thus to learn more about it read section 6.3 in [Com10].

4.4.1.8 *.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. Checkout section 6.1.1 in [Com10] for more information.

4.4.1.9 *.debug_ranges*

DIEs that have a set of addresses that are non-contiguous will have offset in to the section *.debug_ranges* instead of having a address range. This offset points to the start of a range list that contain range entries which are used to know for which addresses the DIE is used in the program. The DWARF section *.debug_ranges* is used for storing these list of ranges. Checkout section 2.17 in [Com10] to learn more about code addresses and ranges.

4.4.1.10 *.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 string are found using the offset 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*.

4.4.1.11 *.debug_type*

The DWARF section *.debug_type* is very 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.

4.4.2 Dwarf Compilation Unit

The compiler when compiling a source program will most often 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 sources 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 and also the directory and path of the source file.

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

4.4.3 Dwarf Debugging Information Entry

One of the most important data structures in the DWARF format is the DIE. A DIE is low level representation of a small part of the source code. Some of the most common things DIEs represent are functions, variables and types. The DIEs are found in a tree structured referred to a as a DIE tree. Each DIE tree will most often represents 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 a type DIEs.

4.4.3.1 Dwarf Attribute

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

4.4.3.2 Example of a DIE

Lets look at an example of a DIE that describes a variable, the example can be seen in figure 3 which is a screen shoot of the output from the program *objdump* run on a Executable and Linkable Format (ELF) file. The first line in the figure begins with a number 8 which represents the depth in the DIE tree this DIE is located. The next number is the current lines offset into this compile unit, all the other lines in the figure also start with there offset. Then it says "Abbrev Number: 9" on the same line, this is a abbreviation code that translates to *DW_TAG_variable*. This tag means that the DIE is representing a variable from the source code.

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

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

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

4.4.4 Evaluate Variable

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

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

```
<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_rtic_blinking_led7my_function17hefe1787a0f0f5f97E
   <4330>   DW_AT_name        : (indirect string, offset: 0x64a52): my_function
   <4334>   DW_AT_decl_file   : 1
   <4335>   DW_AT_decl_line   : 194
   <4336>   DW_AT_type        : <0x6233>
<3><433a>: Abbrev Number: 17 (DW_TAG_formal_parameter)
   <433b>   DW_AT_location    : 2 byte block: 91 7e    (DW_OP_fbreg: -2)
   <433e>   DW_AT_name        : (indirect string, offset: 0x11d94): val
   <4342>   DW_AT_decl_file   : 1
   <4343>   DW_AT_decl_line   : 194
   <4344>   DW_AT_type        : <0x6233>
```

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

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

4.4.4.1 Finding Raw Value Location

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

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

Currently the value of the *frame base* is unknown, but the location of the *frame base* is describe in the *my_function* DIE. The location of the *frame base* is also describe in a number of operation. Those operations can be found under the attribute *DW_AT_frame_base*. Looking at figure 4 the *frame base* location is describe with the operation *DW_OP_reg7*. The operation *DW_OP_reg7* describe 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 mention 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. But the value also has to be parsed into the type of *val*, see section 4.4.4.2 for how that is done.

4.4.4.2 Parsing the Raw Value

Now the fist problem with parsing the value of the parameter *val* into the correct type is knowing what type the parameter has, this is where the attribute

DW_AT_type comes in. The value of the *DW_AT_type* attribute points to a type DIE tree, which describes the type of the DIE.

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

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

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

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

4.4.5 Virtually Unwind Call Stack

Virtually unwinding the call stack is done by recursively unwinding a stack of *subroutine activations*. It is called virtual unwinding because the state of the debugged target is not changed at any point during the unwinding. Every subroutine in the call stack has a activation and a stack frame. And 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 a activation. The other data structure is called Common Information Entry (CIE), it contain information that is shared among many FDEs. The relevant CIE and FDE to a activation can be found using the code location where it is stopped.

Unwinding the stack of activation is done by first evaluating the values of the top activation, read section 4.4.5.1 to learn how that is done. It starts with the top activation because there is to 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 describe in section 4.4.5.1. This is then repeated for the rest of activation.

4.4.5.1 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 a event or it could be location were 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. Because the return address of the above activation is almost always stored on the stack. This works for all activations except for the top activation, were stopped code location it is the current PC value.

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

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

Both the values of the CFA and the preserved register can be restored using tables located in the DWARF section *.debug_frame*. Checkout section 4.4.5.2 to learn how that is done.

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

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

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

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

Unwind a register is done by first finding the correct row. That is done by finding the closes address that is less then the search one. Next step is to evaluate the new value using the register rule on the row. Then go to the next row in the table and do the same thing but with the new value. Repeat until there are now more rows. That is how to use the table to unwind a register.

5 Implementation

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

5.1 Debug Library

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. But the library requires a lot of knowledge about the DWARF format to use. Thus the *rust-debug* library was created to make it even easier to get the debug information. Figure 7 shows how the two libraries and the ELF file are connected.

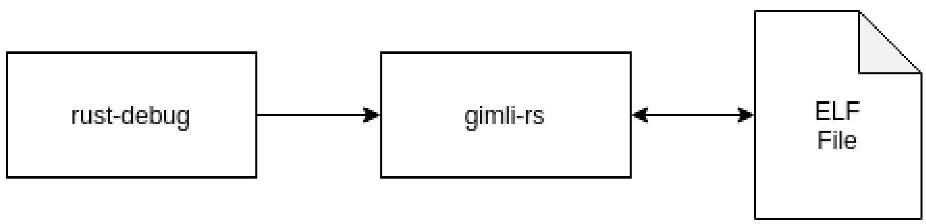

Figure 7: A diagram showing the relation between the *ELF* file and the two libraries *rust-debug* and *gimli-rs*.

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

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

There is more functionalities that *rust-debug* provides but they are not noticeable.

5.1.1 Retrieving Source File Location

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

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

5.1.2 Accessing Memory And Registers

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

The register and memory data structure is used as an argument to the functions in the library. If a value is missing the called function will return a values that says which value is missing. Then the user of the library can read that value from the debugged target and add it to the data structure. Now the function can be called again and there are no more missing values it will return the requested value. Otherwise the user has to repeat the same process.

5.1.3 Evaluating Variables

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

A constructed *VariableCreator* struct has a method for evaluating the value of the variable that requires the register and memory data structure. This method evaluates the variable as describe in section 4.4.4 and the DWARF specification [Com10]. The return value of the method is a enum that is used for telling the user if a value is missing from the register and memory data structure or not. Then there is another method for retrieving the variable information containing the evaluated value. The variable information contains the following:

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

5.1.4 Finding a functions DIE

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

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

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

5.1.5 Unwinding Call Stack

The *rust-debug* library has a data structure called *Stacktrace*, it works very similarly to the data structure describe in section 5.1.3. But *Stacktrace* is used to virtually unwind the call stack and evaluate all the variables in it. The call stack is unwind as described in section 4.4.5 and the DWARF specification [Com10]. This results in a stack of activations, which most importantly contain restored register values and stack pointers.

All the variables in each of the activations are then evaluated using the resorted register values. This is done using a data structure that works similar to the data structure describe in section 5.1.3.

The first step of evaluating the variables is finding the DIE of the subroutine that the activation is related to. This is done using the function describe in section 5.1.4. After that the DIE tree is search through for variable DIEs, starting at the subroutine DIE. All the variable DIEs found are added to a list. Each variable in the list is then evaluated as described in section 5.1.3. If there is a missing value from memory then the response from the evaluation of the variable is returned. Because the data structure works similar to the one is section 5.1.3 it can continue from were it last stopped.

5.1.6 Finding Breakpoint Location

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

The mention function works by first finding out which compilation unit contains information on the inputted file path. It those 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.

5.2 Debugger

The acutal debugger called *Embedded Rust Debugger*(the code for the debugger can be found in the git repository [Luna]) consists mainly of two main threads called the *main thead* and the *debug thread*. The *main thead* is the thread that handels the input from the user from the console or trough Debug Adepter Protocol (DAP) and the *debug thread* handles the reading of DWARF information and connection to the debug target. There is also another thread in the debugger called *input thread* that is only activated if the user is using the console, its jobb is to read the input and send it to the *main thread*.

5.2.1 The Debug Thread

The debug thread has two main state that it changes between, the first state is when it is not attached to any debug target and is called DebugHandler. The second state is when it is attach to the debug target which means that this state is where debugging can happen, it is called Debugger.

Going back to the fist state, its purpose is to await instructions to attach to the debug target and to receive configuration required for that to happen. These configurations that the debugger requires is a path to the elf file, a path to the work directory of the code that should be debugged and lastly the type of chip. When all these are configure the attach command can be used to attach to the micro controller, all the other commands that require that the chip is attach can also be used to attach.

The debugger uses the library *probe-rs* [Yat] to attach to the micro controller and to interact with it. Thus a lot of useful debugging features like stopping, continuing and setting hardware breakpoints is already given by the *probe-rs* library. The other features supported by the debugger uses the library *rust-debug* together with the *probe-rs* library. But the two library are separate so they never interact with each other, the figure 8 show how all of these parts interact. The *rust-debug* library as mention above and seen in the figure 8 is a library for retrieving information from the *DWRAF* sections in the ELF file. To get some of the information from the library values in the debug targets memory and/or registers are needed. Thus when calling the *rust-debug* library it can sometimes give a response that says it requires some value from a registry or a memory address, the debugger then uses *probe-rs* to get that value. Then the read values are sent in to the same *rust-debug* library function as before by storing them in the memory struct that the library uses to read values from the target. This repeats until the *rust-debug* library returns the requested value or an error if something has gone wrong with reading the DWARF format.

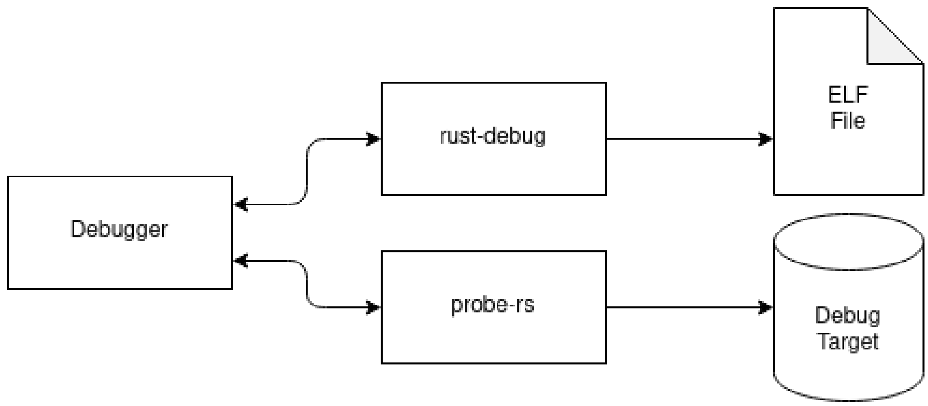

Figure 8: A diagram showing the relations between the debugger, the *ELF* file, the debug target and the two libraries *rust-debug* and *probe-rs*.

5.2.2 Simultaneous Handling Of Request And Events

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

5.2.3 Optimization Of Repeated Variable Evaluation

To improve on the performance of the debugger the value of the stack frames are stored every time they are calculated. This allows for fast repetitive look up of information that is stored in the stack frames. The stored stack frames are removed any time the debugged target starts executing again, thus it will not give old values.

5.2.4 Command Line Interface

The *CLI* is a very simple one, that has a separate thread for reading the input called input thread. The input is read form the console constantly until the user hits enter. Then the inputted line is sent to the main thread. After that the thread waits for a response, the response is a boolean. if the boolean is true then the program will stop, otherwise it start reading the console input again. The thread repeats until it is stopped.

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

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

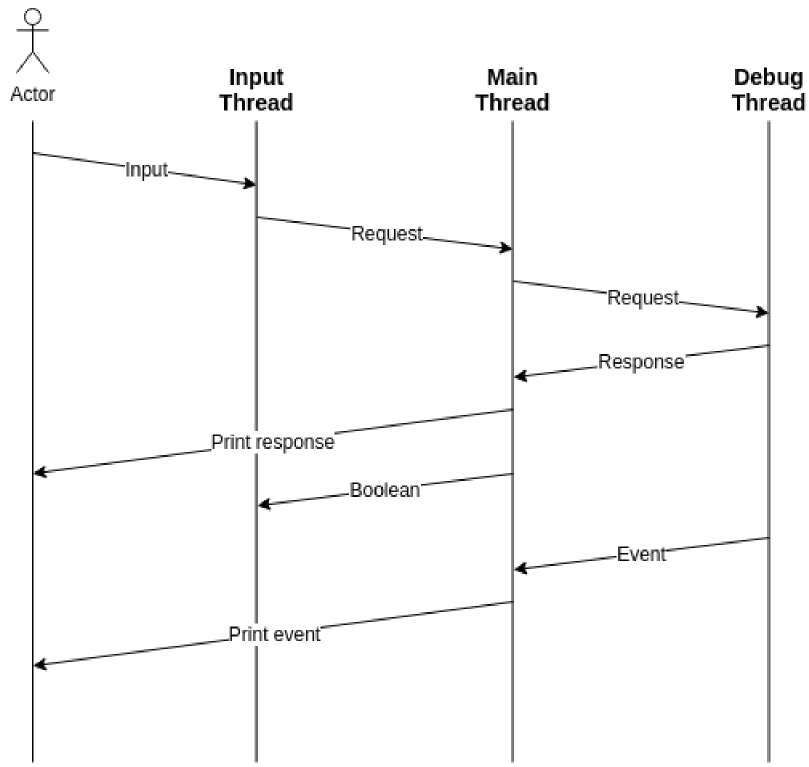

Figure 9: A diagram showing the communication between the user/actor and the three different threads.

5.2.5 Debug Adapter

The debug adapter is implemented as a Transmission Control Protocol (TCP) server in the main thread. It starts a debug thread when a new TCP connection is made. After that it starts listening for DAP messages on the TCP connection.

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

When the debug adapter receives a DAP message from the server it translates it to one or more commands for the debug thread. Those commands are then sent one by one to the debug thread. The responses from those commands are then used to make the response to the client. If the main thread gets a event from the debug thread it will translate it into a DAP message and send it to the client.

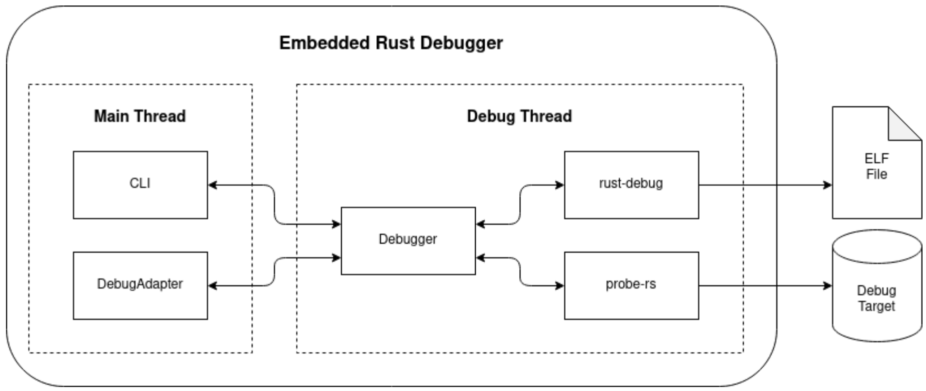

Figure 10: Diagram showing all the moduals of emmbedded rust debugger and there relations to each other.

5.3 VSCode Extension

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

6 Evaluation

To evaluate the soultion to the problem(see section 2.3) there is three important criterias. The first one is how much more useful debug infroatmion does the solution get from the compiler, this is to evalaute the first part of the problem. For the secound part the debugger pressented in this paper needs to be compared to the aleady existing debuggers to see if any improvment is made to debugging optimized *Rust* code. The two most popular debugger used for debugging *Rust* code are *GDB* and *LLDB*, thus it is these two debuggers that will be compared to the presented debugger. The reason being that they are the two debuggers that are supported by the *Rust* dev team acording to there *rustc-dev-guide* [Teaa]. The cirteria they will be compared to is how well the debuggers can retrive debug infromation from optimized code and how well they can display that information to the user. Because getting the information is important but it is pretty much useless if it is not displayid in a user friendly way.

6.1 Compiler settings comparison

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

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

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

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

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

6.2 Debugger Comparison

The testing and comparing of the three diffrent debugger is done manualy 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 handeld the different situations. This was repeatd untill there was any differens in the result between the three debuggers. There were two of these cases found when the code was compiled with optimiation 2. Also to keep it fair in these cases a software breakpoint was added to the code, this ensures that all the three debuggers stopes on the same machine code instuction. This is important because if they are not stoped on the same machine code instuction then the comparison is unfair because some of the information can have been optimized out for one of the debugger making it look worse then the other ones.

The GDB debugger gave a wrong answer when debugging the value of a enum named *test_enum3*, the value that GDB can be seen in figure 11. The expected value is *TODO* which is not the same as what GDB gave.

```
(gdb) p test_enum3
$ 1 = nucleo_rtic_blinking_led::TestEnum::ITest(<optimized out>)
```

Figure 11: GDB debugging result from evaluating variable *test_enum3* when stopped at the software brekpoint in the example code.

Doing the same using the debugger presented in this thesis shows that it also is not the same value as expected, the result can be seen in figure 12. The printing look a bit diffrent but the result from the debugger presented in this thesis tells the user that the enum *test_enum3* is a enum of type *TestEnum* where the acatual variant has been optimized out. While GDB also tells that *test_enum3* is of type *TestEnum* and that it is of the enum variant *ITest* where the value is optimized out. As can be seen GDB gives the wrong answere because it says that *test_enum3* is the enum variant *TestEnum::ITest* which is wrong. While the debugger presesneted in this thesis says that the value has been optimized out which is a more correct anwser.

```
Some("test_enum1") = "TestEnum { ITest::ITest::ITest { __0::20 } }"
Some("test_enum2") = "TestEnum { Non::Non::Non {  } }"
Some("test_struct") = "TestStruct { num::123, flag::< OptimizedOut > }"
Some("test_enum3") = "TestEnum { < OptimizedOut }
```

Figure 12: Debugger presented in this thesis debugging result from evaluating some enum variables when stopped at the software brekpoint in the example code.

Doing the same using LLDB give almost the smae result as the debugger presented in this thesis and that result can be seen in figure 13. LLDB dosnet print that the variant of the enum has been optimized out which makes the reason why it is not printed ambigius.

```
(nucleo_rtic_blinking_led::TestEnum) test_enum1 = {}
(nucleo_rtic_blinking_led::TestEnum) test_enum2 = {}
(nucleo_rtic_blinking_led::TestEnum) test_struct = (flag = false, num = 123)
(nucleo_rtic_blinking_led::TestEnum) test_enum3 = {}
```

Figure 13: LLDB debugging result from evaluating some enum variables when stopped at the software brekpoint in the example code.

Now when inspecting what is stored in the DWARF format is shows that the variant of the enum is optimized out but not the two other values that make up the value stored in the variant. The fact that the value that indicate with variant is optimized out makes it impossible for any debugger to evalute the value stored in the enum. This is because the encoding of the bytes is unknown and because the number of bytes to read from the stack is unknown.

Looking back at figure 13 there are three other enums there where two of them dosen't have a value as well, they are named *test_enum1* and *test_enum2*. Those two enums should have a value but for some reason LLDB is not able to evaluate them, but looking at figure 12 shows the values they should have. Also looking at figure 11 shows the same result as in figure 12 thus both GDB and the debugger presented in this thesis is able to evaluate the correct value. Going back again to figure 13 which shows that the value of the attribute *flag* is equal to *false*, but looking at figure 11 and 13 shows that the value of *flag* is equal to *true*. The correct value when looking at the original source code is that the attribute *flag* should be equal to *true*, thus meaning that the result from LLDB is incorrect.

Another problem found is with values that are temporarly not present in any register or the stack which means that it is temporarly optimized out or out of range. An example of this is the value of the variable *test_u16* which is a unsinged 16 bit integer that is temporarly optimized out when stoped at the software breakpoint in the example. When evaluating this value GDB prints that the value is optimized out which can be seen in figure 14, this is the same output it gives for a value that is totaly optmized out(example of this is the value of *test_struct* shown in figure 11).

```
(gdb) p test_u16
$1 = <optimized out>
```

Figure 14: GDB debugging result from evaluating variable *test_u16* when stopped at the software brekpoint in the example code.

Doing this with LLDB gives the result ¡ *variable is not available* ¿ which can be seen in figure 15.

```
(unsigned short) test_u16 = <variable not available>
```

Figure 15: LLDB debugging result from evaluating variable *test_u16* when stopped at the software brekpoint in the example code.

Lasy compring this to the output of the debugger presented in this thesis which give the value ¡*OutOfRange*¿(see figure 16). The result from both LLDB and the debugger presented in this thesis are uniques and only happen in these situations thus making it easier for the user to understand that the value is temporary optimized out then the reuslt from GDB. This is because the resutl that GDB generates is used in multiple situations thus making it unclear if the variable is totaly optimized out or just that is temprarly optimized out.

```
>> variable test_u16
test_u16 = <OutOfRange>
        line: 69
        file: src/main.rs
        directory: /home/niklas/Desktop/exjobb/nucleo64-rtic-examples
Location: []
>>
```

Figure 16: Debugger presented in this thesis debugging result from evaluating variable *test_u16* when stopped at the software brekpoint in the example code.

7 Discussion

TODO

7.1 Debug Information Generation

This thesis was not really able to improve the amount of debug information generated by rustc and LLVM. There are some options mention in the section 6.1 that generate debug information, but they are well known and a must to debug *Rust* code. Thus the result presented there is noting new for most developers that program in *Rust*. To really improve on the amount of debug information generated one has to work on the compiler or LLVM and make the optimization passes keep more information.

7.2 Debug Experience For Optimized Rust Code

At the beginning of this thesis it was thought that the GDB debugger was not able to evaluate variables where the value is located in registers. This thought came from the observation that GDB for the most time prints that almost all variables are optimized out when debugging optimized code. Thus it was thought that those variables were completely removed from the optimized code. But as it turns out they weren't, instead they were just optimized out at that pedicular point. Keeping these expectation in mind the result in section 6.2 were a bit disappointing. The only improvement that could be made for this problem is that different messages are displayed for the different situations. This helps the user understand more what is happening in the code because they can now know if a variable is completely optimized out of the code or if it is just temporally optimized out.

Unexpectedly there was another problem with both debugger GDB and LLDB that could be improved on. That problem is that both mentioned debuggers had problems with the evaluation of the *enum* type in *Rust*(see section 6.2 for the specific problems they have). The reason for this problem is that many other programming language doesn't allow *enums* to have any value stored in them like *Rust* allows. The LLDB debugger had much more problems with this then GDB which only gave the wrong value when the enum variant was optimized out. Thus unexpectedly the debugger in this thesis could improve on that by evaluating the correct value. And in the case of the value of the enum variant being optimized out it could display that it was optimized out, thus not giving the wrong value as GDB does.

7.2.1 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 configuring that has to be done and both require a program like *openocd* that handles the communication between the debugger and the target device. Removing this need of using a program like *openocd* made the debugging experience a lot better using the debugger presented in this thesis. The reason being that it takes fewer steps to start debugging.

7.2.2 Contributing to the Rust Debug Community

As mention before this problem is huge and complex thus to really make a change in debugging optimized *Rust* code more people have to contribute. One of the best ways of getting more people involved is by making it easier to contribute. The debugging library *rust-debug* does just that by simplifying the process of retrieving debug information from dwarf. If more developers start using *rust-debug* will force the library to also become better and intern also force *gimli-rs*, rustc, LLVM and DWARF to become better.

8 Conclusions and future work

The problem of improving debugging for optimized *Rust* code is a very complex problem and a very large one. To make it easier to solve this problem, the problem was divided into two parts and had a lot delimitations. The first part of generating more debug information was not very successful, because all of the options found to generate more debug information are well known. Thus none of them are really an improvement because most *Rust* developers already use them. But the other part of creating a debugger that improve on optimized *Rust* code on embedded systems was successful. The main reason for this is that the debugger in this thesis is able to correcly evalute the value of variables that are of the enum type. Were Both of the two supported debugger by the *Rust* team does sometimes evalute variables of the enum type into the incorrect value. It was also able to simplify the process of debugging embeded systems that run *Rust* code. Then there is also the fact that this thesis lead to the creation of a debugging library called *rust-debug* which simplifies the process of retriving debug information from the DWARF format. Thus overall I would say that goal of improving debugging of optimized *Rust* code was achived, but that there is still a lot that needs to be done.

8.1 Future Debugging Improvement

One of the main problems with debugging optimized code is that the variables are stored in registers and never pushed to the stack. Thus after the variables are done with they are often overwritten and this makes it impossible for the debugger to know the value they had. This makes debugging very hard because the user has to find the correct machine code locations where the value is in a registry, which is not easy. But if the debugger is able to get the last value of the variable somehow it could display it as the last known value of the variable. It could maybe be done by storing the value of the variable before it get overwritten, but that is a hard problem in it self. There is no time to make a solution for this problem, thus this will have to be left for future work.

8.2 Future Debugger Improvement

The debugger presented in this thesis only supports most important functionalities of a debugger. Thus there is many more features that could be added. One important one is the ability to evaluate expression which is left to be done as future work. This is a really 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 doesn't need to be written twice.

8.3 Future Debugger Library Improvement

Currently the debugging library *rust-debug* is not able to display a pointer and the value it points to. When a user wants to evaluate the value of a variable that is a pointer such as a string, they most often want the value of the string and not the pointer to the string. This problem is a little hard because sometimes the user want the value of the pointer. Implementing a solution for this is left as future work.

Glossary

GDB The GNU Project Debugger. 6, 7, 31–34

LLDB A debugger made using libraries from the LLVM project. 7, 32–34

LLVM The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.. 6, 7, 30, 31, 34, 35

rustc The name of the *rust* compiler is rustc.. 7, 30, 31, 34, 35

tree A tree is a data structure that consist of nodes with children. The first node is called root and the tree cannot have any circular paths.. 14, 16, 19

Acronyms

CFA Canonical Frame Address. 3, 19–21

CIE Common Information Entry. 13, 19, 20

CLI Command Line Interface. 8, 22

DAP Debug Adepter Protocol. 4, 8, 22, 25, 27–29

DIE Debugging Information Entry. 3, 13–19

DWARF Debugging with Attributed Record Formats. 5–7, 13–20, 22–26, 30–32, 35, 36

ELF Executable and Linkable Format. 17, 22, 23, 26

FDE Frame Description Entry. 13, 19, 20

GUI Graphical User Interface. 28, 29

MCU Microcontroller Unit. 6, 8

PC Program Counter. 8, 10, 18, 20

TCP Transmission Control Protocol. 22, 27, 29

References

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

[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).

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

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

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

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

[Yat] Yatekii. *probe-rs Homepage*. URL: `https://probe.rs/`. (accessed: 17.08.2021).