Master Thesis

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

August 17, 2021

1 Abstract

TODO

Contents

1 Abstract **2**

Glossary **5**

Acronyms **5**

2 Introduction **6**
2.1 Background 6
2.2 Motivation 6
2.3 Problem definition 7
2.4 Delimitations 8
2.5 Thesis structure 9

3 Related work **10**
3.1 llvm 10
3.2 GDB 10
3.3 LLDB 10
3.4 probe-rs 10
3.5 gimli-rs 10
3.6 DWARF 11

4 Theory **12**
4.1 Registers and Memory 12
4.1.1 Call Stack 12
4.1.2 Stack Frame/Call Frame 12
4.2 DWARF 13
4.2.1 Dwarf Sections 13
4.2.2 Dwarf Compilation Unit 16
4.2.3 Dwarf Debugging Information Entry 16
4.2.4 Dwarf Attribute 17
4.2.5 Evaluate Variable 17
4.2.6 Virtually Unwind Call Stack 19

5 Implementation **22**
5.1 Debug Library 22
5.2 Debugger 25
5.3 Command Line Interface 26
5.4 Debug Adapter 26
5.5 VSCode Extension 28

6 Evaluation **29**
6.1 Compiler settings comparison 29
6.2 Debugger Comparison 29

7 Discussion **33**

8 Conclusions and future work **34**

Glossary

Debugee The program or machine that is being debugged. 17, 22, 25, 26

GDB The GNU Project Debugger. 10, 29–31

LLDB A debugger made using libraries from the LLVM project. 10, 29–31

LLVM The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.. 10

Tree A tree is a data structure that consist of nodes with children. The first node is called root and the tree cant have any circular paths.. 15, 16

Acronyms

CFA Canonical Frame Address. 19–21

CIE Common Information Entry. 14, 20, 21

DAP Debug Adepter Protocol. 9, 22, 26–28

DIE Debugging Information Entry. 15–19

DWARF Debugging with Attributed Record Formats. 8, 9, 13–24, 29, 30

ELF Executable and Linkable Format. 16, 22, 25

FDE Frame Description Entry. 14, 20, 21

GUI Graphical User Interface. 9, 28

pc Program Counter. 12, 19

TCP Transmission Control Protocol. 22, 26, 28

2 Introduction

Debugging Rust code on embedded system today is not a very good experience for many reasons. One of the big problems is debugging optimized code is often near impossible. That is because the compilers today are very good at inlining the code and removing unused code. This changes the program so drastically in many cases that when trying to debug the code all the original variable can be optimized out. Thus the user doesn't get any useful information from the debugger about the program which makes is near impossible to debug. One of the big reasons why variable gets optimized out is because unoptimized code always push the value of the variable to memory which then can be read by the debugger at anytime. This is not done for optimized code because speed is prioritized and storing the value of a variable that will not be used on the stack is costly.Thus the time that most variable exist is very short in optimized code which results in that the debugger saying that a variable is optimized out.

2.1 Background

TODO

2.2 Motivation

The main motivation is that optimized *Rust* code can be up to 100 times faster then unoptimised code. That is a very large difference in speed compared to other languages like *C* for example, were the optimized code is about 2-4 times faster then the unoptimized code. Thus for language like *C* it is much more acceptable to not be able to debug optimized code because the different isn't that large compared to *Rust*. But in the case of *Rust* code the different is so large that some programs are to slow to run without optimization. This causes the problem that the code can't be debugged because debuggers don't work well on optimized code and unoptimized code is to slow to run. Because of that there is a need for debuggers that can give enough information to debug optimized *Rust* code. Then there is also the argument that relying only on testing the optimized code instead of going through and checking that is is correct is bad. The reason being that there can be extremely many paths that needs to be tested and it is sometimes not feasible to test all of them. In these cases it is less costly to verify that the code is correct then to test every path.

Another large motivation for this thesis is that the most common way of debugging *Rust* code on embedded systems is complicated for a beginner. One of the reason being that it requires two programs being *openocd* and *gdb*, it also requires configuration files which takes some time to understand and configure. This is not very accessible for people that have no experience with these programs and is unnecessarily complicated. The ideal solution for accessibility would be to have a single program instead of two and that it requires less configuring, thus making it as easy as possible for a new person to debug there code.

Most of the debuggers used for *Rust* code are written in other programming languages and there isn't a lot of debugging tools written in *Rust* yet. Thus one of the motivation for the thesis debugger to be written in *Rust* is to contribute with a example of a debugger written in *Rust* to the *Rust* community. This also relates back to improving debugging for optimized *Rust* code because that is a large and hard problem that requires a lot of work to solve and to maintain the solution. One of the most realistic way that will happen is if the *Rust* community around debugging grows and more people contribute with there solutions and ideas.

Another way this thesis contributes to the *Rust* community is by making a library that simplifies the process of retrieving information from the debug information. This is important because it makes retrieving debug information simpler for new developers to start contributing to the *Rust* debug community. Which will hopefully lead to better debugging for optimized *Rust* code.

There is also a economical reason wanting to have debuggers that work well for optimized code. As mentioned before testing all the paths in a program is sometimes not feasible because of the amount of work needed. But verifying that the program is correct and that the implementation is correct is another way to ensure that the program works as intended. It is even sometimes the preferred solution because then the program is proven to work correctly. And a debugger is one of the vital tools needed for confirming that a implementation is correct. Verifying the correctness of a program and the implementation can also be cheaper then testing in those cases where the programs has extremely many paths. Thus improving the tools needed to verify the correctness of the implementation can intern reduce the cost of verifying that the program works as intended. And it can even reduce the amount of money witch companies spend on testing there code. The amount of money spent on testing of code each year is about xxx and thus the potential savings on testing us huge.

2.3 Problem definition

There are two main problem that this thesis tries to tackle to improve the experience of debugging optimized code for embedded systems. The fist problem is about the generation of the debug information, if more debug information can be generated then there is more information the debugger can retrieve an show the user. This is also were the problem starts with debugging optimized code, because debuggers needs the debug information to understand the relation between the source code and the machine code. Thus it is very important that the compiler generates as much debug information as possible, because there is nothing that can be done later to get more information. The first problem then is to look at the different options that can be set in the *llvm* compiler to improve the generations of debug information without impacting the optimization of the code to much. Speed of the resulting code is still a big priority.

The second problem is looking at the available debug information that the *llvm* compiler generates for optimized code and creating a debugger that utilizes that information to the fullest. This problem has two parts to it, the first being

retrieving the needed information from the debug information. This will be the hardest part and is the most important for improving the debugging for optimized code. The second part is to display the debug information to the user in a user friendly way.

The goal of solving these to problems is to create a debugger that gives a better debugging experience for optimized rust code on embedded systems then some of the most commonly used debugger, such as *gdb* and *lldb*. And to inspire further development for debugging tools in the rust community.

2.4 Delimitations

As mentioned one of the main problems for getting a debugger to work for optimized code is getting the compiler to generate all the debug information needed. In the case of the *Rust* compiler it is the *LLVM* library that handles the debug information generation. *LLVM* is a very lager project that many people are working on and thus it would be to much work for this thesis to try and improve the debug information generation. Thus this thesis is limited in solving this problem by the current functionality of *LLVM*.

The compiler backend *LLVM* that the *Rust* compiler uses supports two debugging file formats that holds all the debug information. One of them is the Debugging with Attributed Record Formats (DWARF) format that has been around for a long time and is supported by many compilers and debuggers. The other one is *CodeView* format which is developed by *Microsoft* and has also been around for a long time. To make a debugger that supports both formats would be a lot of extra work that doesn't contribute to solving the main problem of this thesis. Thus it has been decided to only support the DWARF format because it has good documentation and a opensource community around it.

The scope of this thesis also dose not include changing or adding to the DWARF format standard. The main reason is that it takes years for a new version of the standard to be released and thus there is not enough time for this thesis to see and 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 years before the *Rust* compiler had been updated to use the new standard. Currently the newest DWARF version is 5 but the *Rust* compiler still uses the DWARF 4 format.

Many of the debuggers today have a lot of functionality to help the user understand what is happening in the program that they are debugging. An example of these functionality are debuggers that support the ability to go backwards in the program. Functionalities like this are useful but not contributing much to the main problem of debugging optimized code. Which is that most of the source code variables get optimized away and thus making it extremely hard to understand what is happening in the code. So to keep this thesis focused on the main problem the feature the debugger will have is restricted to evaluating stack frames and the variables present in each frame, the ability to add and remove breakpoints. And the ability to control the program by stopping, continuing and stepping an instruction.

When debugging code on embedded systems the debugger needs to know a lot about the hardware the code is running on. It has to know the size of the memory and the number of registers, it also has to know which are the special registers such as the program counter register and more. There is also the endianity of the values store on in memory and registers so the debugger can convert it to the correct type. Then there is also the type of machine code the processes uses and the instruction mode use. To support all the different micro controller would be to much work for this thesis. Thus the debugger is limited to work with the *Nucleo-64 STM32F401* card because it is the one that is available. And it will only support the instruction set *Thumb mode* made by *arm*. The debugger will be design to work with other similar micro controller but to test and grantee that it will work with them is to much work for this thesis.

Another part of this thesis is the interaction between the user and the debugger. Existing debugger like *gdb* both have a *CLI* and a Graphical User Interface (GUI), thus it is up to the user which one they want to use. From a usability perspective the debugger in this thesis should also have both of the option for the user to choose from. A *CLI* is not that much work to implement but a GUI takes a lot of work to implement. Luckily *Microsoft* has made a protocol for debuggers that specifies an adapter that handles the communication between the GUI and the debugger. This protocol is called Debug Adepter Protocol (DAP) and is used by *VSCode*. Thus the scope of the debugger will include implementing the DAP protocol and an extension for *VSCode*.

The DWARF format is very extensive and supports a lot of different program languages, the specifications for the different languages are a little different from each other. Because this thesis is about *Rust* code the thesis will only go into detail in how to read the DWARF format for *Rust*. The specification for the DWARF format is also very good at explaining how the information is structured. Thus a this thesis will not go into all the detail in how to read the DWARF format instead it will focus on explaining how the information in the DWARF format can be used in combination to get important information that the user wants.

2.5 Thesis structure

TODO

3 Related work

3.1 llvm

TODO

3.2 GDB

On of the most well known and used debuggers is GDB which supports a lot of different programming languages including *Rust* [GNU]. It is also one of the most common debuggers used for debugging embedded *Rust* code together with the tool *openocd*. Thus it is for this reason that the debugger make in this thesis will be compared to GDB to see if it actually made any improvement on debugging optimized *Rust* code.

One of the problem found with running GDB on optimized code is that if the enumerator variant value is optimized out by the compiler but not the value contained in the variant. Then GDB will default to choosing the 0 variant of the enumerator and thus display the wrong value because it is optimized out thus the correct value could be any of the enumerator variants. This is one of the reason why GDB is not a very good debugger for debugging optimized *Rust* code.

Another problem with GDB on optimized code is that if a variables value is only stored in a register then the time the value is know is very short because the register will be written over very soon after it is not needed. In these cases GDB will say that the variable is optimized out when the value is not present, but this result can make the user to think that the value is totally optimized away and never present in the program, which is not true. The lack of information in these cases makes debugging very hard because the user has no way of knowing if optimized out means that a variable is optimized out completely or if it doesn't have a value at this precise time.

3.3 LLDB

The LLDB project is a debugger that uses many of the libraries from the LLVM project. This debugger has the benefit of using the same libraries that the *Rust* compiler uses. It is also one of the most commonly used debugger for debugging *Rust* code. There is a wrapper made for LLDB to give better printing for *Rust* types made by the *Rust* team called *rust-lldb*. But LLDB does not list *Rust* as a support language on there website [Tea].

3.4 probe-rs

TODO

3.5 gimli-rs

TODO

3.6 DWARF

TODO

4 Theory

TODO

4.1 Registers and Memory

The values of a program needs to be temporary stored some were on the computer were it can easily be access while running the program. The two ways the computer can do this is to either store it in a register or in the memory. Registers are very limited in space and are very volatile thus it is good for storing values that are needed often in a short amount of time. Memory is slower but has a lot more space and is thus much more useful for storing values that will be needed in a while. The compiler will compile the program to store the different variables and values in the registers and memory in a structured way so it is easily managed.

There is no real structure for the registers except that certain registers have a specific purpose, which can be the Program Counter (pc) for example. A pc is a register that keeps track of which machine code instruction the program is currently at. There are other important registers like the stack pointer register and more. On the other hand the memory all structure and one of the structures is a stack that consist of stack frames/call frames, the stack is called *call stack*.

4.1.1 Call Stack

The Call stack is a stack in the memory that holds stack frames which intern contain information about the values of variables and other values. 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 to reach the lower elements all the above elements needs to be popped. The call stack is used to keep track of all the stack frames which holds almost all the values of the running program, the values are also present in the registers and in the heap which is also located in the memory. An example of a call stack can be seen in the figure 1.

4.1.2 Stack Frame/Call Frame

A stack frame or call frame is a element of the call stack that contain variable values for a scope of the running program. The scope is usually a function that has been called that have its own arguments and variables that are needed to be stored in memory for later use. Stack frames also contain a return address which is pointer to the machine code that the program should jump to after the computation in the current scope is done. When the computation is done the stack frame will be popped from the call stack and the previous stack frame will become the current stack frame, check out figure 1 for an visual example.

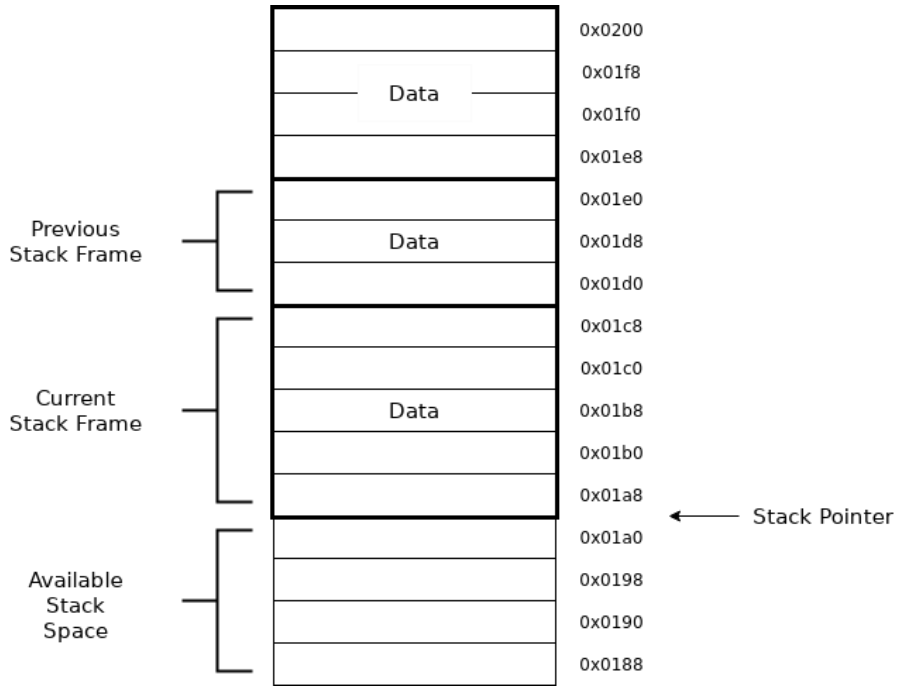

Figure 1: An visual example of how a stack and stack frames can look.

4.2 DWARF

The DWARF fromat is very complex which makes it hard to work with. This section will go through how the fromat is structured and how the diffrent parts can be used to get debug information. But because the DWARF format has a great specification that goes into detail how it is structed and work, this section will only go through the format in a higher level. Skiping much of the detail that is not needed to understand how the implementation of the debugger works, checkout the specification [Com10] for more information.

4.2.1 Dwarf Sections

The DWARF format is divided into sections in the object file which all contain specific information, these sections use offsets to point to information in another section, see figure 2. All of the sections can be different from depending on DWARF versions and some doesn't exist in the older versions. Thus these explanations only apply to DWARF version 4 and some of the older versions, checkout Appendix F in [Com10] for more information.

Going in alphabetical order the first DWARF section is called *.debug_abbrev*. This section contain all of the abbreviation tables which can be used to find a specific die using the abbreviation. These table entries contain information

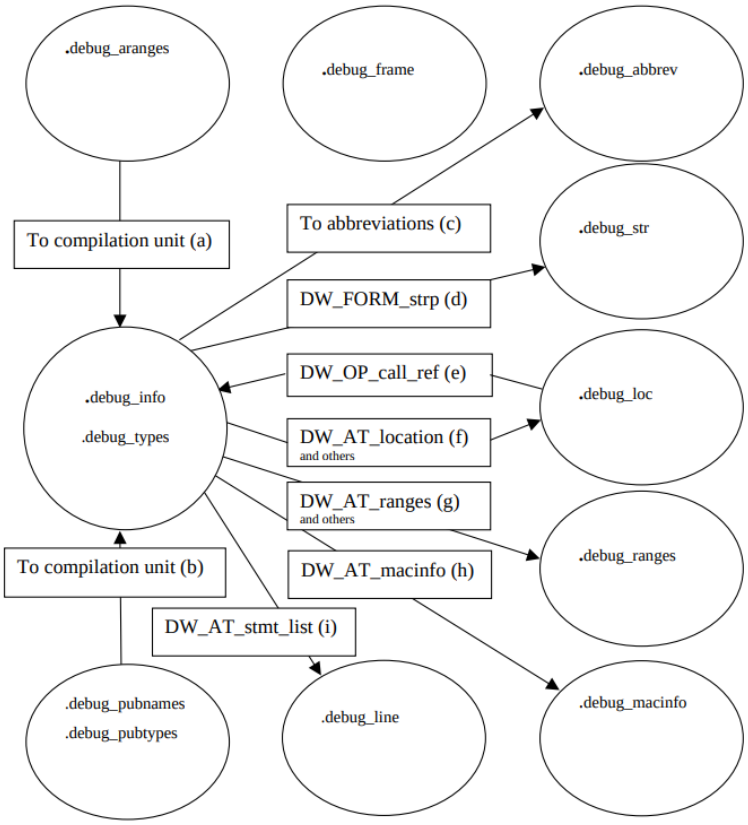

Figure 2: Diagram of all the different DWARF sections and there relations to each other.

about the die tag, attributes and if it has children. The library *gimli-rs* simplifies the process of using this table and thus removes the need to know the detail of how to read it, but checkout section 7.5.3 in [Com10] to know more.

The DWARF section *debug_aranges* is used to lookup which machine address corresponds to which compilations unit. This address information is stored in ranges where a compilation unit can have multiple ranges. These ranges consists of a start address followed by length. Thus to lookup the user only needs to check if the search 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].

In the DWARF section *.debug_frame* the information needed to virtually unwind the call stack is kept. Unwinding the call stack is complex and is hardware specific but the *gimli-rs* library simplifies the process a lot. This section is completely self-contained and is made up of two structures called Common Information Entry (CIE) and Frame Description Entry (FDE). To learn more of this section checkout section 6.4.1 in [Com10]

Information about the source code are store in Debugging Information Entrys (DIEs) which are low-level representation of the source code. These 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 Tree structure that represents a compilation unit or a partial one. These Tree structures are structure like the source program and makes it possible to relate the source code to the machine code. The section *.debug_info* contains a number of units that all have one of these DIE Trees and some more important debug information. Thus this is one of the most important sections in DWARF because it is used to understand the relation ship between the source code and the machine code.

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 director, 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.

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. The pointers are most commonly found in the attribute *DW_AT_location* which most dies representing variables have. The relation between these sections can be seen in the figure 2.

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 complex thus to learn more about it read section 6.3 in [Com10].

There are two sections for looking up compilation units by name. The first one is *.debug_pubnames* which is for finding functions and objects. And the other one is for finding types, this section is called *.debug_pubtypes*. Both of these work in the same way and are fairly simple to understand, checkout section 6.1.1 in [Com10] for more information.

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. Checkout section 2.17 in [Com10] to learn more about code addresses and ranges.

The 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*.

The last section is *.debug_type* it 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 instead of a representation of the source program.

4.2.2 Dwarf Compilation Unit

The compiler when compiling a source program will most often generate one compilation unit for each source file, there are some cases when partial compilation units will be made. These compilation units holds most of the debug information generated from the source file. This naturally creates that the debug information in most of the different DWARF sections are divided into these compilation units. It also makes it easy for the debugger to find information because the debug information is structured similar to the source code. The main use of these compilation units in a debugger is to first find which unit the program is currently in. Then it is easy to search through the DIE Tree in the *.debug_info* section to find the information needed. Finding the right compilation unit can take time but there is the DWARF sections *.debug_aranges*, *.debug_pubnames* and *.debug_pubtypes* that provides a fast way of finding it.

The fist DIE in the Tree from the section *.debug_info* 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 directory and path of the source file. The Tree is structured in a way that the children of a DIE representing a function will all be declared inside of the function in the source file. Thus it is easy for the debugger to know everything about the function by going through all of its children.

4.2.3 Dwarf Debugging Information Entry

One of the most important data structures in the DWARF format is the DIE which are found in the *.debug_info* and *.debug_type* sections. These DIEs are used to represent parts of the source code and hold information about were in the source code there are declared. They also contain information about which range of addresses they are applicable to in the machine code and more. There are two types of DIEs one of them are dies that only describe types in the source code and are located in a Tree in the section *.debug_types*. The other type of DIE are dies that describe the structure of the source code and describes where that information is in the machine code.

Let 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 tree this DIE is located, the next number is the offset into this section for that line. Lastly the information stored on that line is a 9 which represents the tag of the DIE which is in this case *DW_AT_variable*. This first line is also the first line of the die. For the rest of the line the first number is also the offset into the *.debug_info* section that the information is located and these lines are attributes to in the

DIE. The attribute *DW_AT_location* has the information of where the variable is stored on the Debugee. The name attribute *DW_AT_name* has an offset into the DWARF section *.debug_str* that the tool has evaluated to "str" and it is the name of the variable. The attributes *DW_AT_decl_file* and *DW_AT_decl_line* has offset into the section *.debug_line* and will reveal the source file and line that this DIE comes from. Lastly the attribute *DW_AT_type* has a offset into the section *.debug_types* that points to a type DIE that has the type information for this variable.

```
<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.

4.2.4 Dwarf Attribute

The information in the DIEs are stored in attributes these attributes consists of a attribute name and a value. The name of the attribute is used to know what the value the attribute hold 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 number that can be decoded using the *.debug_abbrev* section. A die can only have one of each attributes and it thus limited to the number of attributes it can have.

4.2.5 Evaluate Variable

The process of evaluating the value of a variable or argument is a bit complicated because there is a lot of parts to it that all depend on each other. Thus to simplify the explanation a simple example will be used to explain the main part of evaluating the value. 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 example in the figure is the output of the program *objdump* run on an DWARF file and it is shown because it is much easier to read then the raw DWARF file. The function has a argument called *val* which is the DIE with the tag *DW_TAG_formal_parameter* in the figure 4, it is a child of the function DIE which means that it is a argument to the function *my_function*. It is this argument *val* that will be used as an example for evaluating the value of a variable or argument.

Examining the DIE for the argument *val* there is a attribute there called *DW_AT_location* this attributes value is a number of operations that when ex-

ecuted will result in the location of the value. In this example the value of the *DW_AT_location* attribute is *DW_OP_fbreg* -2, this operation means that the value is stored in memory at the address of the *frame base* plus the offset -2(see [Com10] page 18). *Frame base* is a address in memory that is fixed to the first value in the stack for the functions stack frame(see [Com10] page 56). To get the value of the *frame base* it also has to be evaluated in the same way that the argument *val* will be. The operation for evaluating the *frame base* are located in the attribute *DW_AT_frame_base* in the parent function DIE to the *val* die, it is the one called *my_function*. Looking at figure 4 at the attribute *DW_AT_frame_base* of the function DIE the operations to evaluating the *frame base* consist only of the operation *DW_OP_reg7*. This operation means that the value for the *frame base* is stored in register 7(see [Com10] page 27) thus to continue the evaluation of the argument *val* register 7 has to be read. Continuing evaluation of the argument *val* it is now know were it is located because the operations describe it to be -2 from the *frame base* in memory. Thus the value of the argument can then be read from that memory location but to do that the size and type must first be known.

```
<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.

Now the fist problem with parsing the value of the argument into the correct type is knowing what type the argument has. Luckily DWARF has a attribute for that called *DW_AT_type* which value is a offset into the DWARF section called *.debug_types*. The *val* argument DIE has this attribute and as can be seen in figure 4 it has the value $0x6233$ which points to the type DIE seen in figure 5. Note that the type DIE in the figure has the offset 6233 which is the same offset that the *val* DIE has in its type attribute. The type DIE has the tag *DW_TAG_base_type* which means that it a type that is built into the languages and is not defined in terms of other data types(see [Com10] page 75). The DIE also have three attributes the first one is the name attributes which in this case says that the base types name is *i16*, see figure 5. Then there is the attribute *DW_AT_encoding* describes how the raw bytes should be interpreted and in this case it has the value 5 that stands for signed. Thus from the encoding attribute it is known that the type of *val* is a signed integer, but the size of the integer is still unknown. The last attribute in the DIE is *DW_AT_byte_size* and it has the value 2 which means that the size of the signed integer is 2 bytes. Now the location, size and type of *val* is known and thus the value can be evaluated.

```
<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.

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* that describe the pc range in which the function applies. This means that the value of the argument *val* will only be in memory when the pc is in the range describe by the two attributes. Thus the value of *val* can only be determined when the pc is in that range. There is also some other attributes in the example that are not mention here because they are not needed for determining the value of the attribute.

Another important thing to note is that the example explained is a very simple one meaning that there is a lot more different DWARF operation for finding the location of a variable and also a lot more different tags for the type DIEs. All the various tags for type dies requires a unique explanation to how it should be used thus the explanation of each one of them can be read about in the DWARF specification [Com10], the same goes for the different DWARF operations.

4.2.6 Virtually Unwind Call Stack

To virtually unwind the *call stack* entail the task of restoring the code location and the register values for each *subroutine activation*, it also entail the task of find the location of the corresponding *call frame* on the stack. Each *subroutine activation* has a code location within the subroutine that show where it stopped for any reason, the reason could be that a breakpoint was hit, it was interrupted by a event or it could be location were it made a call. As mention the *subroutine activation* also has some register values at the mention code location that may or may not need to restored depending on if the activation is the last one or not. Lastly a *subroutine activation* has a corresponding *call frame* on the stack that is identified by the Canonical Frame Address (CFA). The CFA is the value of the stack pointer in the previous stack frame when it is at the call location of the current *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).

Accessing one of the *subroutine activation* requires that the activation stack is virtually unwind to get to the desired activation, there is only one exception to this and that is if the top activation is desired. That is because all of *activation* information location is known thus it is easy to read that information from the registers and memory. It is called *virtually unwinding* because none of the value in the registers or memory are changes which means that the state of the target is not changed. In general the steps to virtually unwinding the *activation* is to begin with calculating the CFA of the previous *activation*, the code location of

the previous *activation* and to virtually restore any registers if needed. Then the same thing can be done to the previous *activation* and so it repeats until the desired *activation* is reached. This process is required to start with the top *activation* because it is the only one that is known and the process has to be stopped if the bottom *activation* is reached.

Any subroutine can have some prologue code that is in the beginning of the subroutine and epilogue code that is at the end. The prologue code is used to preserve the values of registers over the duration of the subroutine, this is done by allocating some extra space on the call stack for the *call frame* which is used to store the register values. Then the epilogue code is used to restore these values to the registers before returning to the previous frame. Using this fact the compiler generates debug information that enables a user to virtually restore these preserved registers and that is what is done when virtually unwinding the *activations*. 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.

The location information for finding CFA and the registers for a *activation* is stored in a table, that consist of virtual unwinding rules and addresses. There are multiple of these tables for each frame that are stored in a data structure called Frame Description Entry (FDE) that is meant to hold all the needed frame information for that frame. Then there is also the data structure Common Information Entry (CIE) that holds information that is shared between some Frame Description Entry's. All of this frame information is stored in a separate section called *.debug_frame* and each instance of this section is guaranteed to have at least one Common Information Entry.

Going back to the table with the virtual unwinding rules the first column of that table the code addresses are located, they are used to identify that all the virtual unwinding rules on that row applies for a particular code address. Every code line is meant to have these rules but because many of the rows are exactly the same many of those rows are removed from the table to save storage space. The second column is also special because it contains the virtual unwinding rules for CFA, these rules are either a DWARF expression that needs to be evaluated in the same way that the variables are in section 4.2.5 or a register values plus some singed offset. All the other columns contain virtual unwinding rules for all the register and they are in orders from 0 to n, see figure 6 for a visual example of the table.

There are a number of different virtual unwinding rules for the registers that are called register rules in the DWARF specification [Com10], these are listed on page 128. Some of them are very easy to use such as the register rule *undefined* that says that a register is not preserved by the callee and thus is impossible to know the value of. If the register value is unchanged then register rule *same value* is used to denote that. The must common rule is *offset(N)* where N is a signed offset, this rule means that the register value is stored at the address $CFA + N$. All of the remaining rules can be read about in the

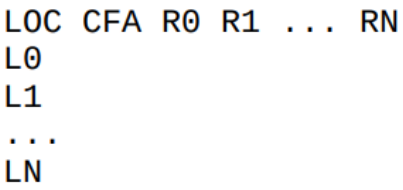

Figure 6: This is how the table for reconstructing the CFA and registers looks like. *LOC* means that it is the column containing the code locations for 0 to N. The column with CFA has the virtual unwinding rules for CFA. The rest of the column *R0* to *RN* holds all the virtual unwinding rules for the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

DWARF specification [Com10] on page 128.

Now understanding what information is stored about the call frames the call stack can be virtually unwind by first finding the relevant CIE and FDE. Finding the relevant CIE and FDE is done by doing a lookup in the DWARF section *.debug_frame* using the current machine code location that the program is stopped on. Then the virtual unwinding rules in the table of the FDE can be evaluated to get the value of the CFA and the preserved register. Lastly the code location of the previous frame can be calculated using the return register that is must often one of the preserved registers, this can not always be done because sometimes the information needed is not present. The most common case were the value of the return register is not preserved is when the bottom of the stack is reach, thus the virtual unwinding is complete. This way of virtually unwind a *activation* can then be repeat for all the actiavations in the stack starting from the top *activation* of the stack to the bottom one.

The virtual unwinding rules in the table actually needs to be evaluated from some special DWARF operations. There are a lot of these operation which many are similar to the operation used to evaluate the location of a value in section 4.2.5. Thus there is no real need of explain them here but to learn more about them read section 6.4.2 in the DWARF specification [Com10] on pages 131-136. The way to evaluate these operations is also describe in section 6.4.3 in the DWARF specification [Com10] on pages 136-137 in 4 steps.

5 Implementation

The implementations of the debugger is separade into three diffrent smaller projects, the fist being a debug library that simplifies the process of retriving infomation from the DWARF format. To learn more about how the debug library project is implemented checkout the subsection 5.1. The second project is to create a debugger using the debug library and the last project is to make a debug adapter extension for *VSCode*. The structure of the debugger can be seen in the figure 7, as can be seen in the figure the debugger uses the debug library *rust-debug* to read the ELF file and the probe-rs library for interfacing with the microcontroller being debugged. It can also be noted that the debugger is devided into two separate thread, one for handeling the users input and displaying information. An another one for interacting with the Debugee and the ELF file. These are not the only threads but they are the most important ones and to simplify this general overviw it can be seen as only two threads. The *Main Thread* is the thread that can eiter be started as a cli that handles the users commands from the command line or as a debug adapter server that handles DAP commands over a Transmission Control Protocol (TCP) connection. The main purpos of this thread is to handle input from the user and to display or forwad inormation from the *Debug Thread*. The purpose of the debug thread is to handle requests from the *Main Thread* and then send back a response, it also cheacks the state of the Debugee and sends events to the *Main Thread* if the state of the debugee changes. These two threads comunicates using a asyncronus channel that both sides polls to check for any new messages.

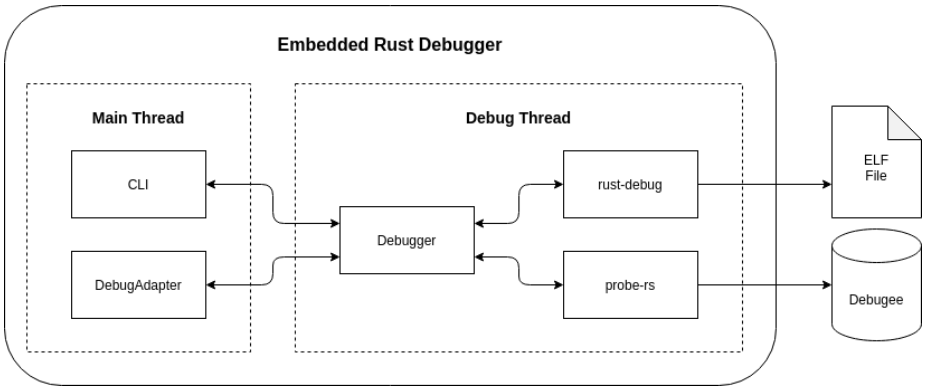

Figure 7: Diagram showing all the moduals of emmbedded rust debugger and there relations to each other.

5.1 Debug Library

Retrieving the 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

DWARF format is made to be space efficient so that the binary file doesn't get to large. This feature of the DWARF format has a side effect in that it makes it much more complicated for retrieving the information wanted. Luckily there exist a library called *gimli-rs* that simplifies reading the DWARF format. But the library still is very complicated to use because it required a lot of knowledge about the DWARF format. Thus the *rust-debug* library was made to simplify this problem even more so that almost no knowledge of the DWARF format is needed.

The library *rust-debug* is built apone the *gimli-rs* library and doesn't restrict the user from accessing the functionality of that library. It uses the *gimli-rs* library for paring the DWARF sections into more workable data structures, the figure 8 shows how they are connected. The goal of the design of the library is that a user should be able to just call a function for retrieving debug information such as a stack trace and at the same time be able to use the *gimli-rs* functionality to retrieve the same information if wanted. The library is has the features to restive the call stack, stack frames, variables, sourceinfomaiton and a evaluation function for evaluating the values of variables.

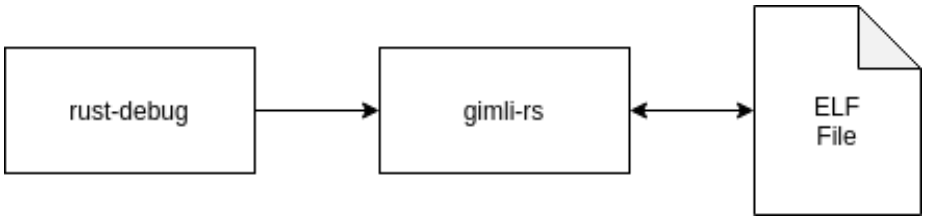

Figure 8: A diagram showing the relation between the *ELF* file and the two libraries *rust-debug* and *gimli-rs*.

Some of the dies have attributes that starts with *DW_AT_decl_*, these attributes describe which source file and where in it this dies was declared. The library has a function for retrieving the value of all these attributes and it returns as *SourceInformation* struct. This struct contain the directory, file name, line number and column number where the die was declared in the source files. The line and column number is very simple to retrieve because it is the same as the value of the attributes *DW_AT_decl_line* and *DW_AT_decl_column*. It is much more complicated to get the directory and file name because the value of the attributes *DW_AT_decl_file* is a file index into the current units line program. The line program indexes all the file entries and can thus be used to find the directory and file name from the file index. If any of these attributes are not present in the die then the result of this function will be that that entry has a *None* value.

As mention before one of the requirements for evaluating the value of a variable is access to the registers and memory of the debugee. This library does not have that functionality because the problem of accessing memory and register is out of scope for this library. The reason being that it would limit the

number of systems it could be used for too the systems it support and also the goal of this library is to make it simpler to get debug information from DWARF, and not to provide an interface to the systems memory and register. That lead to the motivation of creating a data structure that holds these needed values of the system. The implementation is just a struct with two hashmaps, one for the memory values and one for the register values. It also has some methods for retrieving the stored values. This struct is then used as an argument to the functions in the library and if a value that is not present in the data structure is needed the result of the function call will be a request to add the to the data structure and call the function again. This has a negative effect in that some of the calculations will be repeated multiple times but it is not very notable.

The library has a structure called *VariableCreator* which takes a reference to the DWARF unit and die of the variable. Then there is a evaluation method that takes the memory and registers data structure and preforms calculation to evalue and retrieve variable information. This method return an enum that either tell the user what memory or register values are needed, or that the it is done and the method *get_variable* can be used to get the variable. The variable structure contains the name, value, type, sourcelocaiton and the location of where the value was evaluated from.

Using the call stack result each call frame can be used to create a stack frame. These stack frames are like the call frames except that they have more information like the function name of the frame, where the function was declared and the value of all the variables. The way this library constructs a *StackFrame* struct works the same way as when creating a variable. Meaning that there is a helper struct for creating the stack frame which requires the memory and register struct as an argument. This helper struct when created will find the name of the function for this frame and where it was declared, then it will go through all dies that belong to that function to find all the variable dies and store it in a list. This list of variable dies is then used to evaluate all variables and each entry is removed when evaluated and the result stored in a variables attribute. Thus if one of the variables requires a value from memory that is not present in the memory and register struct then all the variables already evaluated doesn't need to be evaluated again. The evaluating of the variables is done in the same way as describe above the only difference is that the registers values is set to be the ones evaluated in the call frame. Thus adding the value of a register to the memory and register struct won't do anything when evaluating a stack frame. Then lastly the stack frame can be retrieve using the method *get_stack_frame* from the stack frame helper struct.

There is also a function for finding out what machine code address a certain line in a source file corresponds to. It works by first finding out which compilation unit that the source file corresponds to by looping trough all the file entries in all of the compilation units. Then it loops through all the rows in that file entry and adds all the rows that match the source line number. If the vector is empty after that then there is no machine code instruction that matches that source line otherwise it is not empty there is at least one instruction that corresponds to that line. When there are multiple match the function will take the

one that has the nearest column value to the source lines column value.

5.2 Debugger

The debug thread has two main state that it changes between, the first state is when it is not attached to any Debugee and is called DebugHandler. The second state is when it is attach to the Debugee 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 Debugee 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 9 show how all of these parts interact. The *rust-debug* library as mention above and seen in the figure 9 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 Debugees memory and/or registers are needed. Thus when the *rust-debug* library gives a response to a requires that it needs some specific value the *Debugger Rust* struct can use *probe-rs* to read those values. Then those values can be sent in as a argument or the *rust-debug* library. This repeats until the *rust-debug* library returns the requested value or an error.

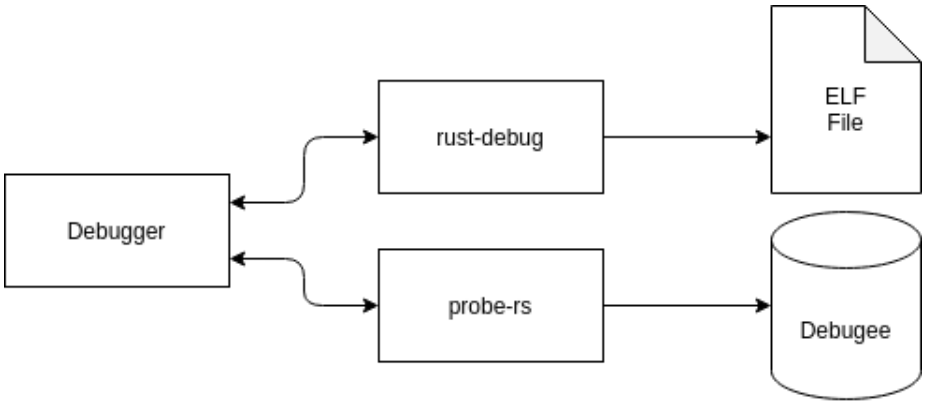

Figure 9: A diagram showing the relations between the debugger, the *ELF* file, the Debugee and the two libraries *rust-debug* and *probe-rs*.

To simultaneously handle incoming request from the user and events that happens in the Debugee, the debugger polls the channel for incoming request and the state of the debugger. To reduce the amount of polling of the state of the Debugee the debugger has a boolean that keeps track of the state of the Debugee. This boolean is to track if the Debugee is running or is stopped. And because events can only occur if the Debugee is running the polling only needs to be done when it thinks the Debugee is running. Thus this is how the debugger can handle request from the user and near simultaneously handle events that happen in the Debugee.

To improve of the performance of the debugger the debugger stores the value of the stack frames every time they are calculated. This allows for fast repetitive look up of stack frame information and variables. The stored stack frame values are only stored when the Debugee is stopped and are removed when the Debugee starts. Thus the wrong values will never be shown to the user. Also another feature of the debugger is that if the value of a variable is request the debugger will restive all the stack frames instead and then search for the requested variable. This simplifies the implementation a lot and will also make the debugger faster when repeated request are made.

5.3 Command Line Interface

The *CLI* is very simple and works by having a thread that constantly waits for an input from the command line. When an input is given to the thread it tries to parses the input into a request for the debugger. It parses the input by first comparing the fist word of the input to all the commands, if it matches a command then the rest of the input is parsed by using the specific parser for that command. If the input does not mach any of the commands then an error is printed to the user. When the input has been parsed into a request it is then sent through a channel to the main thread which forwards it to the debug thread. Then when the main thread gets a response back from the debug thread it prints the result to the user and sends a boolean back to the thread that reads the input. The boolean tells the thread if it should continue reading inputs or if it should stop. The main tread constantly awaits a request from the input thread or a response or event from the debug thread. It does this by constantly polling the two channels. If the main thread receives a event from the debug event is displays it to the user and continues as usual.

5.4 Debug Adapter

The debug adapter is implemented as a TCP server that listens for new connection on a specified port. Then when a new connection is made it communicates with the client using the *Microsoft Debug Adapter Protocol*. Looking at figure 11 show the flow of communication between the different processes from the user to the debugger. When a user interacts with the debugger extension in *VSCode*, it will intern send a DAP message to the debug adapter server over the TCP connection. The debug adapter will then process the message and translate it

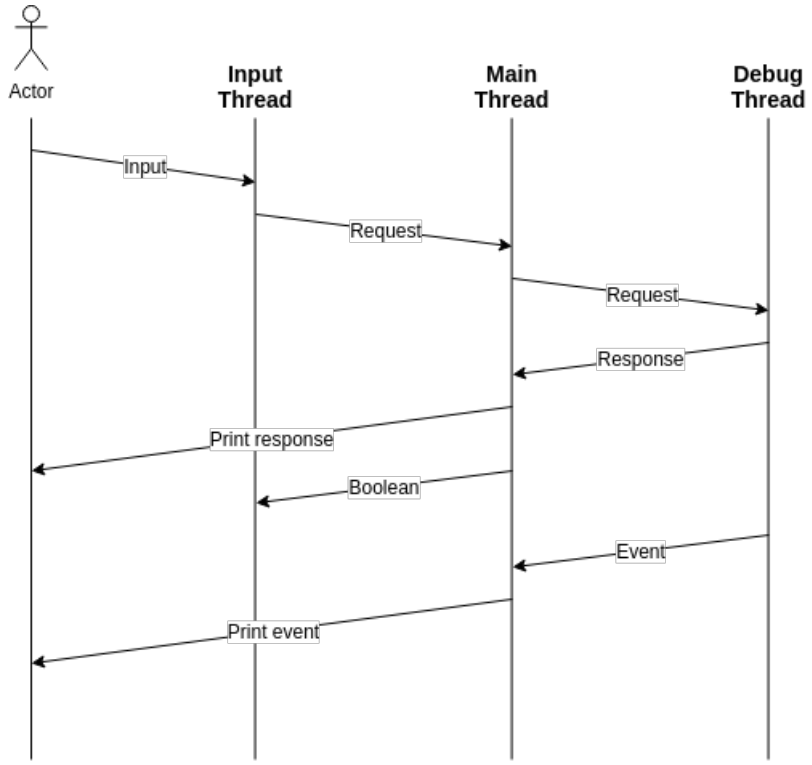

Figure 10: A diagram showing the comunication between the user/actor and the three different threads.

to commands that the debugger in the debug thread can understand and send them through a channel to the debug thread. The debug thread will then intern process the commands and send responses back to the debug adapter which in tern can translate the response and forward it back to the *VSCode* extension. That is the normal flow of communication but there is another case that can happen. Which is when a event happens on the debugee, this event could be that the debugee has stopped for some reason. In this case the flow of communication starts at the debug thread which can be seen in figure 11. The debug thread then sends the event to the debug adapter which in tern sends it to the *VSCode* extension, where it is then shown to the user by *VSCode*.

It is required the the first few DAP messages sent to the debug adapter is for configuring the debug adapter by communicating the supported capabilities of the debugger and *VSCode*. This debugger doesn't support any of the optional capabilities that are defined in the DAP protocol and will thus not show up in *VSCode*. After everything is configured then the debugger will get a requires to flash the connected debugee with the program that is going to be debugged. Now the debug adapter is started and will work as a middle man between the

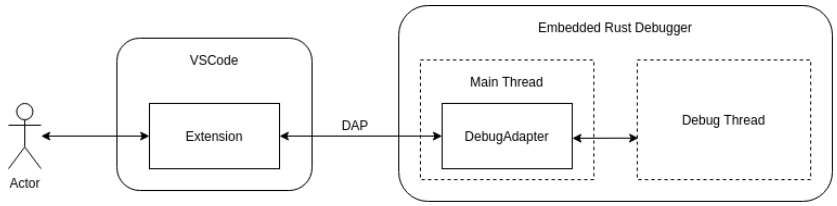

Figure 11: A diagram showing the comunication between the user/actor, *VSCode* and the debugger *Embedded Rust Debugger*.

debugger and the GUI.

The debug adapter and the debugger is running on separate threads and thus communicate asynchronously using channels. The channels uses a different set of commands then that is defined in DAP which means that the debug adapter translates these commands and forwards them. This means that a single DAP message can result in multiple commands being sent from the debug adapter to the debugger.

Because the debug adapter can get messages from both the GUI and the debugger at any time it uses continues polling on the TCP connection and the channel. This enables the debug adapter to forward messages sent by both *VSCode* and the debugger.

5.5 VSCode Extension

The *VSCode* extension for the debugger *Embedded Rust Debugger* is a very simple and bare bones implementation. *Microsoft* provides *API* which can be used to starting a debugging session and trackers for logging what is happening in the debugging session. The implementation of the extensions uses the *API* to create a tracker that logs all the sent and received messages, it also logs all the errors. It also creates a session that tries to connect to a DAP server on a configurable port or uses a already existing session. Then there is some extra arguments added to the configuration DAP message sent to the debug adapter.

6 Evaluation

To evaluate the soultion to the problem(see section 2.3) there is three important kriterias. 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 embedded *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 popular and have been used for a while thuse they should be able to retrive almost all of the debug information stored in DWARF. 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

Going all the possible options that can be sent into the *rust* compiler there are two that are key for getting the most amount of debug information. 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. 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 infomation got less and less when a higher optimization level was set on a simple blink code example. The optimiztaion level 3 resulted in amlost all variables being optimized out which made the debugging extreamly hard. But if the optimiztaion level was set to 2 there was in most cases enugh infomation to debug code. Optimization level of 1 hade the most debug information and was eisest to debug with.

6.2 Debugger Comparison

The example code used to test is not totaly a real program that would be worked on because it would take to much time making or finding such examples that covers all the different cases that a debugger encounters. But that said all debuggers have been tested on real code examples as well to see that they work.

Now comparing the debugger is done by manuly debugging example code [**debugexample**] and looking for diffrences in the result. The version of GDB used is GDB-11.0 because it is the latest version released on the latest *Ubuntu* version which is *Ubuntu 21.04*. For the same reason the version of LLDB used is version 10.1.

When running the example code on optimization 2 and with a software breakpoint set in the middle of a function to ensure that both debugger stop on the same machine code location. 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 12. 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 12: 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 13. 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 13: 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 14. 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 14: 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 14 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 13 shows the values they should have. Also looking at figure 12 shows the same result as in figure 13 thus both GDB and the debugger presented in this thesis is able to evaluate the correct value. Going back again to figure 14 which shows that the value of the attribute *flag* is equal to *false*, but looking at figure 12 and 14 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 15, 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 12).

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

Figure 15: 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 16.

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

Figure 16: 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 17). 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 17: 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

8 Conclusions and future work

TODO

References

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

[GNU] GNU. *GDB: The GNU Project Debugger*. URL: `https://www.gnu.org/software/gdb/`. (accessed: 26.01.2021).

[Tea] The LLDB Team. *The LLDB Debugger*. URL: `https://lldb.llvm.org/`. (accessed: 13.07.2021).

[Yat] Yatekii. *probe-rs Homepage*. URL: `https://probe.rs/`. (accessed: 26.01.2021).