Master Thesis

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

August 25, 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 4.3.

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 4.3 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.

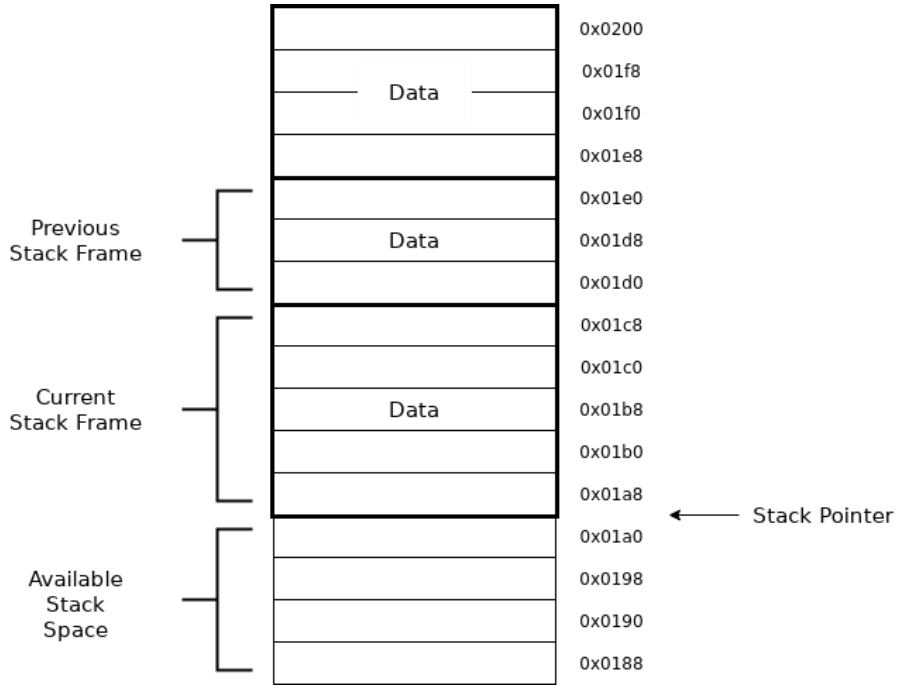

Figure 1: An visual example of how a stack and stack frames can look.

4.1.3 Heap

TODO

4.2 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.

4.2.1 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.

4.3 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.

4.3.1 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.

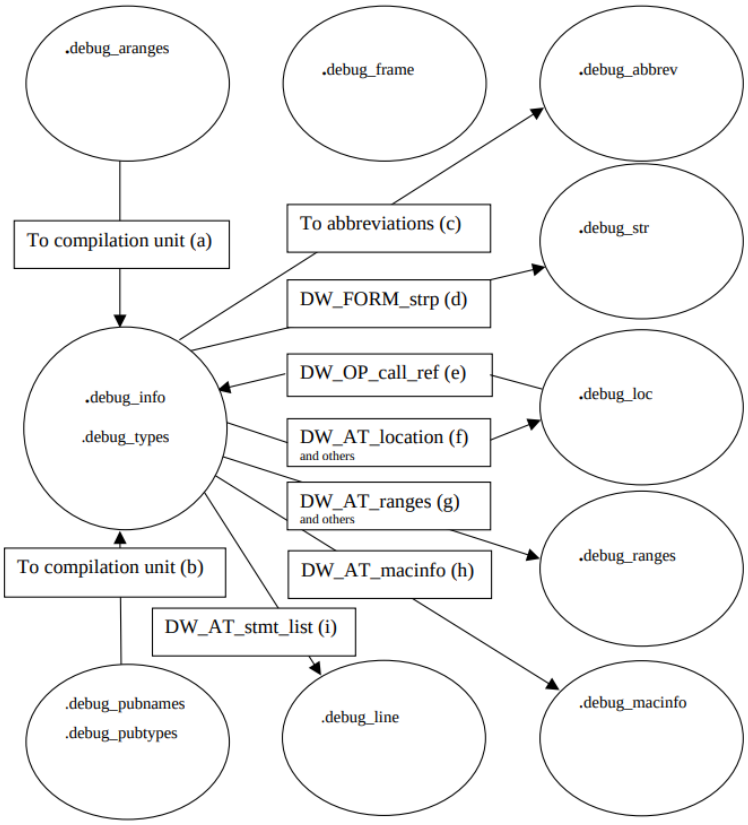

Figure 2: Diagram of all the different DWARF sections and there relations to each other. This diagram is taken directly from the DWARF specification [Com10].

4.3.1.1 *.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.

4.3.1.2 *.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].

4.3.1.3 *.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 4.3.5 and section 6.4.1 in [Com10]

4.3.1.4 *.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.

4.3.1.5 *.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.

4.3.1.6 *.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.

4.3.1.7 *.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].

4.3.1.8 *.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.

4.3.1.9 *.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.

4.3.1.10 *.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*.

4.3.1.11 *.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.

4.3.2 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 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.

4.3.3 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.

4.3.3.1 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.

4.3.3.2 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.

4.3.4 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.

4.3.4.1 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 4.3.4.2 for how that is done.

4.3.4.2 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).

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

```
<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.

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.

4.3.5 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 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 at first 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 contains the code addresses, they are used to identify that all the virtual unwinding rules on that row applies for a particular code address. Every line of code 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 4.3.4 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.

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

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

There are a number of different virtual unwinding rules 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 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 4.3.4. 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 debug target and the ELF file. These are not the only threads but they are the most important ones and to simplify this general overveiw it can be seen as only two threads. The *Main Thread* is the thread that can eiter be started as a Command Line Interface (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 debug target and sends events to the *Main Thread* if the state of the debug target changes. These two threads comunicates using a asyncronus channel that both sides polls to check for any new messages.

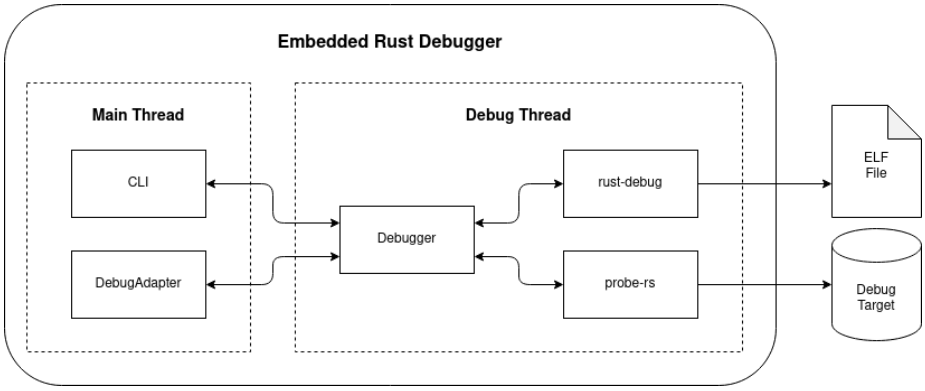

Figure 7: Diagram showing all the moduals of emmbedded rust debugger and there relations to each other.

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*[gim] 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 function for evaluating the values of variables, the code for this library can be found in the git repository [Lunc].

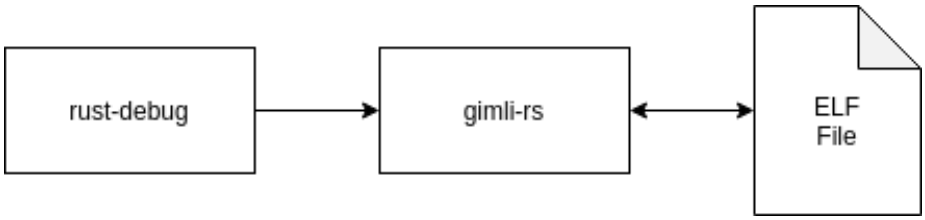

Figure 8: A diagram showing the relation between the *ELF* file and the two libraries *rust-debug* and *gimli-rs*.

5.1.1 Finding Source Location

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 that needs to be looked up in 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.

5.1.2 Accessing Memory And Registers

As mention before one of the requirements for evaluating the value of a variable is access to the registers and memory of the debug target. 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.

5.1.3 Evaluating Variables

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.

5.1.4 Evaluating Call Stack

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.

5.1.5 Finding Breakpoint Location

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 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 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 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.

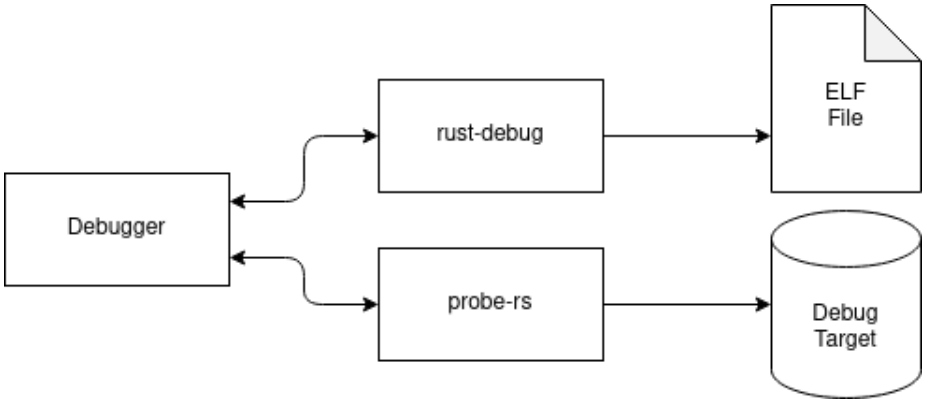

Figure 9: A diagram showing the relations between the debugger, the *ELF* file, the debug target and the two libraries *rust-debug* and *probe-rs*.

5.2.2 Simultaneous Handling Of Request And Events

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

5.2.3 Optimization Of Repeated Variable Evaluation

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 debug target is stopped and are removed when the debug target 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.

5.2.4 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.

5.2.5 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 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 debug target, this event could be that the debug target 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

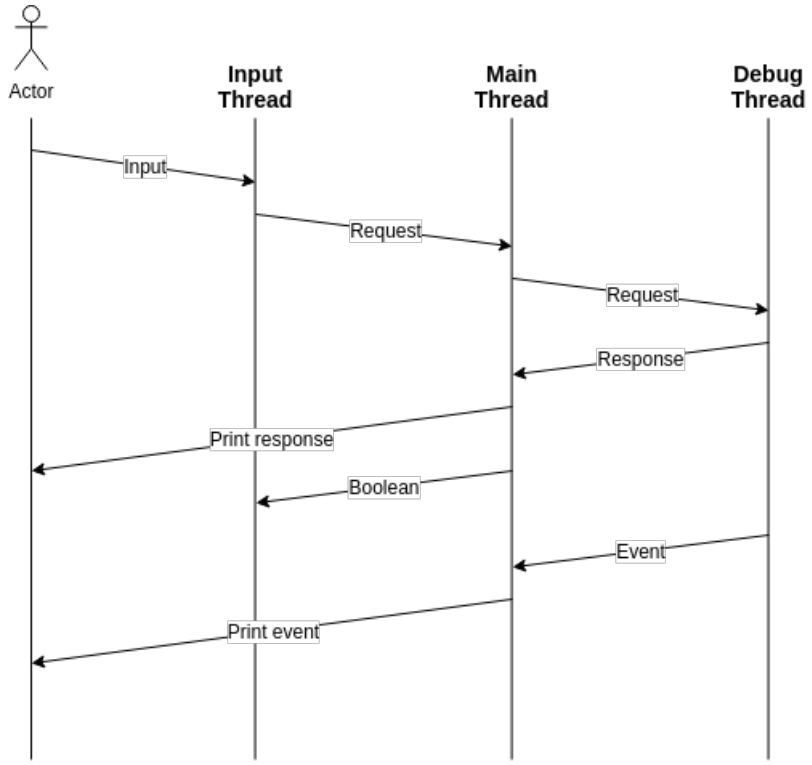

Figure 10: A diagram showing the comunication between the user/actor and the three different threads.

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*.

5.2.5.1 Initial DAP Messages

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 debug target with the program that is going to be debugged. Now the debug adapter is started and will work as a middle man between the debugger and the GUI.

5.2.5.2 Translating DAP Messages To Internal Messages

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

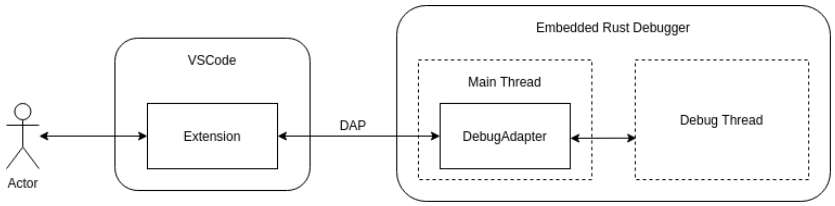

Figure 11: A diagram showing the comunication between the user/actor, *VSCode* and the debugger *Embedded Rust Debugger*.

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.

5.2.5.3 Simultaneous Handling Of DAP Messages And Events

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.

5.3 VSCode Extension

The *VSCode* extension for the debugger *Embedded Rust Debugger* is a very simple and bare bones implementation, the code for the extension is in the git repository [Lunb]. *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 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 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

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

Acronyms

CFA Canonical Frame Address. 19–21

CIE Common Information Entry. 14, 20, 21

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, 12–26, 30–32, 35, 36

ELF Executable and Linkable Format. 16, 22, 23, 26

FDE Frame Description Entry. 14, 20, 21

GUI Graphical User Interface. 28, 29

MCU Microcontroller Unit. 6, 8

PC Program Counter. 8, 10, 17

TCP Transmission Control Protocol. 22, 27, 29

References

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

[gim] gimli-rs. *gimli github page*. URL: `https://github.com/gimli-rs/gimli`. (accessed: 17.08.2021).

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

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

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

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

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

[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).