Improving Debugging For Optimized Rust Code

Master Thesis

Niklas Lundberg

Department of Computer Science, Electrical and Space Engineering
Luleå University of Technology

September 15, 2021

What is debugging?

- Debugging is the process of finding an resolving errors, flaws, or faults.
- There are many different debugging techniques
- A debugger is a tool for debugging.
  - Used for controlling debugged target.
    - Start
    - Stop
    - Reset
    - Step
    - Breakpoints
  - Used to retrieve and visualize debug information.
    - Evaluate values of variables
    - Stack trace
    - And more

Unoptimized Vs Optimized Rust Code

- Rust is a programming language.
- Unoptimized Rust Code
  - Slow to run compared to other similar languages and optimized Rust code.
  - The machine code is very similar to the source code.
  - Values of variables are stored in memory on the stack.
  - Easy to debug.
- Optimized Rust Code
  - Fast compared to unoptimized code.
  - The machine code is very different form source code.
  - Values of variables have short life spans because they are stored in registers.
  - Hard to debug.

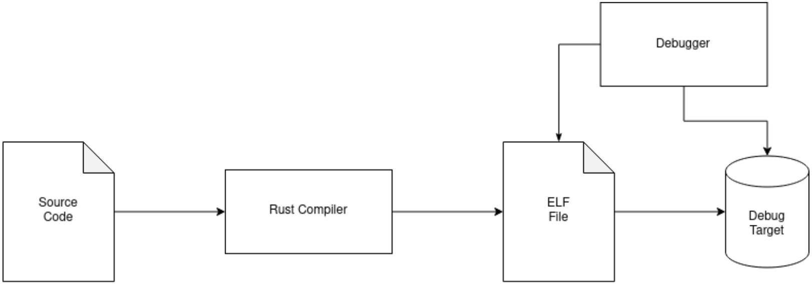
Problems

- Hard to debug because of short life spans of values.
- Existing debuggers like GDB often say that variables are optimized out.
- Existing debuggers also gives wrong result, this is especially true for LLDB.
- Both LLDB and GDB are hard to use for beginners.

Solution and motivation

- Research if better and more debug information can be generated.
  - Why? Because debuggers are restricted by the amount of debug information generated.
- Create a debugger for embedded systems that solve the mentioned problems.
  - Why? Because debuggers are very useful for embedded systems.
    - They often have high requirements to meet.
    - Important that the optimized code work correctly.
    - Small enough that the system can fully be understood.

How dose debuggers work?



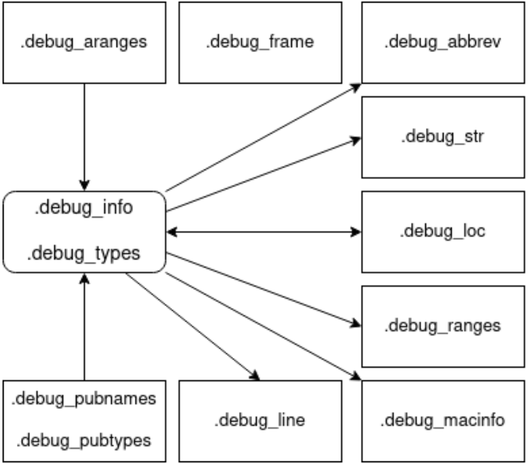
DWARF

DWARF

How to get the debug information?

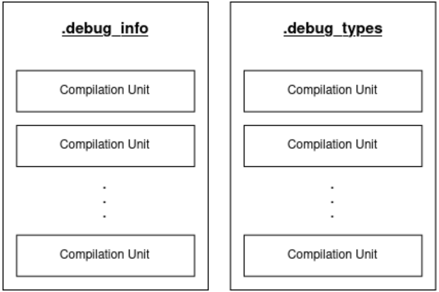
- DWARF is used to understand how the source and machine code relates to each other.
- ELF contains the Debugging with Attributed Record Formats(DWARF).
- Rust uses DWARF version 4.
- Dwarf is divided into 12 sections.

DWARF Sections



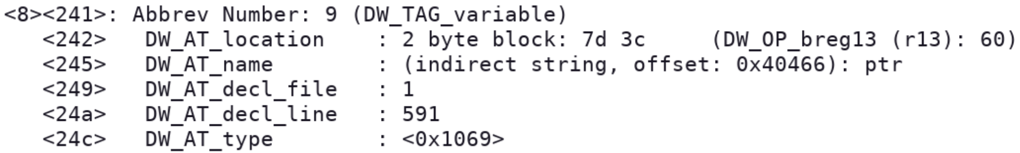
Compilation unit

- Sections .debug_info and .debug_types contain a number of compilation units.
- A compilation unit can be a program or a library.
- Each compilation unit contains all the related debug information for that unit.
- Each compilation unit contains a tree of DIEs.



Debug Information Entry(DIE)

- Debug Information Entry(DIE).
    - All DIEs have a DWARF TAG.
    - DIEs also contain a number of DWARF attributes.
- DWARF DIE example from the .debug_info section.



```
<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>
```

An example on how to use DWARF

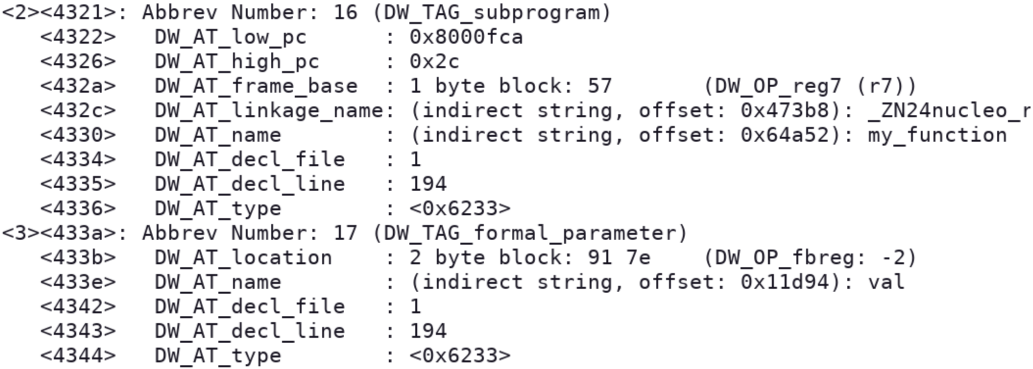
- How can DWARF be used to evaluate the value of a variable?
  - Two things are needed from DWARF.
    - Location of the value in the debug target.
    - The type of the variable.

Evaluating a variable

Finding the relevant debug information

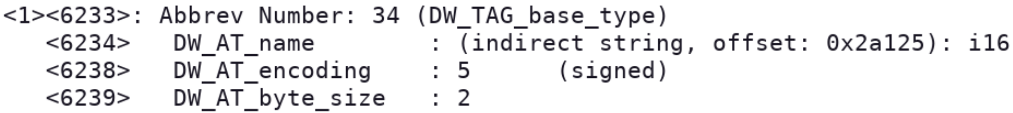
1. Read the current code location from the debug target.
2. Find the current compilation unit using the current code location.
3. Find the current subprogram DIE using the current code location.
4. Find the searched variable DIE in the sub tree of the subprogram DIE.

Evaluating the location of a variable



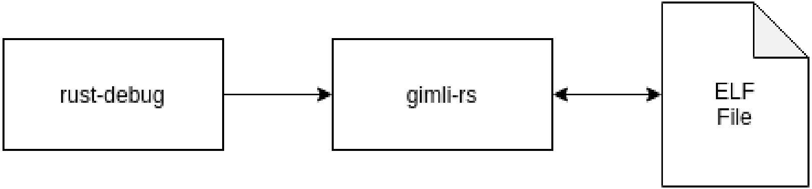
```
<2><4321>: Abbrev Number: 16 (DW_TAG_subprogram)
   <4322>   DW_AT_low_pc      : 0x8000fca
   <4326>   DW_AT_high_pc     : 0x2c
   <432a>   DW_AT_frame_base  : 1 byte block: 57       (DW_OP_reg7 (r7))
   <432c>   DW_AT_linkage_name: (indirect string, offset: 0x473b8): _ZN24nucleo_r
   <4330>   DW_AT_name        : (indirect string, offset: 0x64a52): my_function
   <4334>   DW_AT_decl_file   : 1
   <4335>   DW_AT_decl_line   : 194
   <4336>   DW_AT_type        : <0x6233>
<3><433a>: Abbrev Number: 17 (DW_TAG_formal_parameter)
   <433b>   DW_AT_location    : 2 byte block: 91 7e    (DW_OP_fbreg: -2)
   <433e>   DW_AT_name        : (indirect string, offset: 0x11d94): val
   <4342>   DW_AT_decl_file   : 1
   <4343>   DW_AT_decl_line   : 194
   <4344>   DW_AT_type        : <0x6233>
```

Parsing the type of a variable



```
<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
```

Debugging library rust-debug

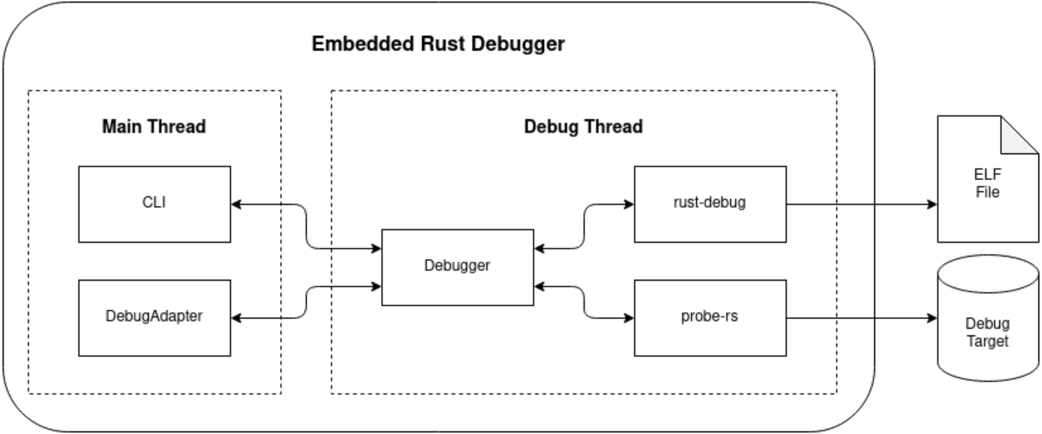


Debugging library rust-debug

Features

- Stack trace
- Evaluating Variables
- Finding breakpoint location
- Retrieving source location information from a DIE
- And more

Embedded Rust Debugger(ERD)



Embedded Rust Debugger(ERD)

Features

- The features that rust-debug has.
- Start, stop, step and reset execution.
- Flash target.
- Set and remove hardware breakpoints.
- And more.

Result

- Evaluated the result be comparing ERD to GDB and LLDB.
- Compiled an example blink program with different optimization.
- The blink program has a inline assembly breakpoint set.
- To ensure that the program stops a the same location each time.
- Added some different types of variables to test the debugger.

Comparing Evaluation Of Rust Enums

Rust Source Code

```
let mut test_enum3 = TestEnum::Struct(TestStruct { flag: true, num: 123 });
```

ERD

```
test_enum3 = TestEnum { < OptimizedOut >}
```

GDB Version 11.0.90

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

Comparing Evaluation Of Rust Enums

Rust Source Code

```
let mut test_enum3 = TestEnum::Struct(TestStruct { flag: true, num: 123 });
```

LLDB Version 13.0.0

```
(nucleo_rtic_blinking_led::TestEnum) test_enum3 = {
  ITest = (0 = 0)
  UTest = (0 = 0)
  Struct = {
    0 = (flag = false, num = 0)
  }

  Non = {}
}
```

Comparing Evaluation Of Rust Enums

Rust Source Code

```
let mut test_struct = TestStruct { flag: true, num: 123 };
```

ERD

```
test_struct = TestStruct { num::123, flag::< OptimizedOut >}
```

GDB Version 11.0.90

```
(gdb) p test_struct
$ 1 = nucleo_rtic_blinking_led::TestStruct {flag: <sybthetic pointer>, num: 123}
```

LLDB Version 13.0.0

```
(nucleo_rtic_blinking_led::TestEnum) test_struct = (flag = false, num = 123)
```

Comparing Evaluation Of Rust Enums

Rust Source Code

```
let mut test_u16: u16 = 500;
```

ERD

```
test_u16 = <OutOfRange>
```

GDB Version 11.0.90

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

LLDB Version 13.0.0

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

Conclusion

- Was not able to generate better or more debug information.
- Able to do some small improvements.
- ERD lacks some of the features that LLDB and GDB has.
- Contributed with a Debugging library for Rust.
- Improved somewhat on the user experience.
- Still a lot that needs to be done.

Questions?