**1/15/2025:** Day 1. The challenge was released today. I am trying to get a rust toolchain so that we can begin developing an HAL for our microprocessor. I have been able to relatively easily generate a .bin file, but have been struggling to create a .img file to flash to the device with the competition tools. I finally figured out how the image is generated. It looks like the compiler generates a .elf file. This is then used to generate a raw binary. Finally, the utils python script written by the competion generates the .img file to flash.

A screenshot of a computer program

Description automatically generated

The insecure bootloader appears to start at 0x00002124 before jumping into the bootloader at 0x100043b6

A screenshot of a computer program

Description automatically generated

**1/16/2025:** I’m trying to figure out why the processor hard faults. So I am stepping through using a debugger and trying to match it to the image file that I provide. The first step was to disassemble the ‘insecure.bin’ bootloader

|  |
| --- |
| arm-none-eabi-objdump -D -b binary -m arm -Mforce-thumb --adjust-vma=  0x10000000 ./rust\_example/insecure.bin > disassembled\_insecure.txt |

This gave me the first matching entry in the below table.

|  |  |  |
| --- | --- | --- |
| PC Address | True Memory Value (mww) | Disassembled |
| 0x00002124 | 0xd000 f8d0 |  |
| 0x00002128 | 0xe004f 8d0 |  |
| 0x0000212C | 0xbf00 4770 |  |
| 0x100043b4 | 0x4685 4847 | 100043b4: 4847 ldr r0, [pc, #284] @ (0x100044d4)  100043b6: 4685 mov sp, r0  This loaded 0x2002000 into R0 (the value that was at 0x100044d4)  The stack pointer (reg sp) is then set to R0=0x2002000  Lr is 0x100043b5 |
| 0x100043b6 | 0x4847 4685 | 100043b8: 4847 ldr r0, [pc, #284] @ (0x100044d8)  100043ba: 4780 blx r0  The value at 0x100044d8 is 0x10004565  LR is 0x100043b5 |
| 0x100043b8 | 0x47804847 | 100043b8: 4847 ldr r0, [pc, #284] @ (0x100044d8) |
| 0x100043ba | 0xb9804780 | 100043ba: 4780 blx r0 |
| 0x10004564 | 0x47702000 | 10004564: 2000 movs r0, #0  loads zero into r0  Updates status flag (sets zero flag, clears negative flag)?? |
| 0x10004566 |  | 10004566: 4770 bx lr  Transfers program control to the lr |
| 0x100043bc |  | 100043bc: b980 cbnz r0, 0x100043e0  This checks if R0 is equal to zero. If r0 is not zero, this jumps to 0x100043e0 |
| 0x100043be |  | 100043be: 4947 ldr r1, [pc, #284] @ (0x100044dc) |
|  |  | 100043c0: 4a47 ldr r2, [pc, #284] @ (0x100044e0)  Loads 0x20000000 into r2  100043c2: 4b48 ldr r3, [pc, #288] @ (0x100044e4)  Loads 0x20000514 into r3  100043c4: 1a9b subs r3, r3, r2  R3 = R3-R2  100043c6: dd03 ble.n 0x100043d0  Branch if the subtraction is less than zero (to 0x100043d0) (skip loop if starting condition invalid) |
|  |  | THIS IS A LOOP THAT CAN BE SKIPPED BY THE BRANCH AT 100043c6  100043c8: 3b04 subs r3, #4  Subtract 4 from R3  100043ca: 58c8 ldr r0, [r1, r3]  Load [r1 + R3] into R0  100043cc: 50d0 str r0, [r2, r3]  Store R0 into [R2 + R3]  100043ce: dcfb bgt.n 0x100043c8  Branches to 0x100043c8 if the condition is greater than zero  Debugging aid: chatGPT  this is used to copy the memory from one range into another range and operates like a loop which makes sense |
|  |  | Before execution,  R0=0x40042000  R1=0x10006008  R2=0x20000000  R3=0x00000000 |
| 0x100043d0  0x100043d2  0x100043d4 |  | 100043d0: 4945 ldr r1, [pc, #276] @ (0x100044e8)  100043d2: 4a46 ldr r2, [pc, #280] @ (0x100044ec)  100043d4: 2000 movs r0, #0  Load registers 1 and 2 with the values at 0x100044e8 and 0x100044ec  Place 0 into r0 |
| 0x100043d6  0x100043d8  0x100043da  0x100043de | Loop | 100043d6: 4291 cmp r1, r2  100043d8: bfbc itt lt  100043da: f841 0b04 strlt.w r0, [r1], #4  100043de: e7fa blt.n 0x100043d6  Increase R1 by 4 until it reaches r2. Each iteration stores r0 into r1 before incrementing (r1?) |
| 0x100043e0 |  |  |
| 0x100043e2 |  |  |
| 0x10004568  0x1000456a  0x1000456c  0x1000456e  0x10004570  0x10004572 |  |  |
| 0x10004d1c |  |  |
| 0x10004d24  0x10004d28  0x10004d2a  0x10004d2e  0x10004d32  0x10004d36  0x10004d3a  0x10004d3c  0x10004d3e |  |  |
| 0x10004d42  0x10004d46  0x10004d4a  0x10004d4e  0x10004d50  0x10004d52 |  |  |
| 0x10004576  0x1000457a  0x1000457e  0x10004582  0x10004586  0x1000458a |  |  |
| 0x100044fc |  |  |
| 0x10004500  0x10004502  0x1000450a  0x1000450c |  |  |
| 0x10004532 |  |  |
| 0x1000454a  0x1000454c |  |  |
| 0x10004520  0x10004524  0x10004526  0x10004528  0x1000452c  0x1000452e  0x10004530 |  |  |
| 0x1000458e |  |  |
|  |  | A bunch more stuff is done |
| 0x100001f8 |  | 100001f8: f00e b802 b.w 0x1000e200 |
| 0x1000e200 |  |  |
| 0x1000daf2 |  | Current mode: thread |
| 0x100043f2 |  | Current mode: Handler HARD FAULT |
| 0x100043f2 |  | Current mode: Handler HARD FAULT |

To set a breakpoint, continue, remove breakpoint (in telnet):

|  |
| --- |
| bp 0x100043ca 2 hw  Resume  Rbp 0x100043ca |

Aha! I figured out where the hard fault occurs. To observe it, set a breakpoint at 0x100001f8. This is the final instruction of the bootloader

|  |
| --- |
| bp 0x100001f8 2 hw  reset  step  step  step |

|  |  |
| --- | --- |
| A screenshot of a computer screen  Description automatically generated |  |

My current plan is to shift the entry point assumed by the rust compiler so that it aligns with the bootloader at 0x1000e200. This involves adding a dummy section in the memory.x configuration and truncating the start of the FLASH region.

Memory.x

|  |
| --- |
| MEMORY {  ROM (rx) : ORIGIN = 0x00000000, LENGTH = 0x00010000 /\* 64kB ROM \*/  BOOTLOADER (rx) : ORIGIN = 0x10000000, LENGTH = 0x0000E000 /\* Bootloader flash \*/  filler\_FLASH (rx) : ORIGIN = 0x1000E000, LENGTH = 0x00000200 /\* filler flash to work with bootloader and rust toolchain \*/  FLASH (rx) : ORIGIN = 0x1000E200, LENGTH = 0x00037E00 /\* Location of team firmware, skipping 200 bytes to make it work for this toolchain \*/  RESERVED (rw) : ORIGIN = 0x10046000, LENGTH = 0x00038000 /\* Reserved \*/  ROM\_BL\_PAGE (rw) : ORIGIN = 0x1007E000, LENGTH = 0x00002000 /\* Reserved \*/  RAM (rwx): ORIGIN = 0x20000000, LENGTH = 0x00020000 /\* 128kB SRAM \*/  }  SECTIONS {  .filler\_section ORIGIN(filler\_FLASH) :  {  \_\_filler\_section = .;    . += 0x200;  FILL(0xFFFFFFFF); /\* Fill the section with all 1s \*/  KEEP(\*(.filler\_section)); /\* Keep the section in the ELF file \*/  } > filler\_FLASH  } |

After some more work, my first run without an eventual hard-fault (stays in thread mode) used the following memory.x file. Basically, I had to add a short segment at 0x1000e200 that caused a jump to the reset handler.

|  |
| --- |
| /\* With some help from ChatGPT, but its not very good at linker files so Aidan Jacobsen reworked most of this from firmware.ld\*/  MEMORY {  ROM (rx) : ORIGIN = 0x00000000, LENGTH = 0x00010000 /\* 64kB ROM \*/  BOOTLOADER (rx) : ORIGIN = 0x10000000, LENGTH = 0x0000E000 /\* Bootloader flash \*/  START\_FLASH (rx) : ORIGIN = 0x1000E000, LENGTH = 0x0000020C /\* start flash to work with bootloader and rust toolchain \*/  FLASH (rx) : ORIGIN = 0x1000E20C, LENGTH = 0x00037E04 /\* Location of team firmware, skipping 200 bytes to make it work for this toolchain \*/  RESERVED (rw) : ORIGIN = 0x10046000, LENGTH = 0x00038000 /\* Reserved \*/  ROM\_BL\_PAGE (rw) : ORIGIN = 0x1007E000, LENGTH = 0x00002000 /\* Reserved \*/  RAM (rwx): ORIGIN = 0x20000000, LENGTH = 0x00020000 /\* 128kB SRAM \*/  }  SECTIONS {  /\* Combined Section: start + Firmware Startup \*/  .combined\_section ORIGIN(START\_FLASH) :  {  /\* Start the start Section \*/  \_\_start\_section = .;  . += 0x200; /\* Space for the start data \*/  FILL(0xFFFFFFFF); /\* Fill with all 1s \*/  /\* Start the Firmware Startup Section \*/  firmware\_startup = .; /\* Label for Disassembly \*/  . = ALIGN(4); /\* Align to a 2-byte boundary \*/  SHORT(0x4800) /\* LDR R0, [PC, #0] \*/  SHORT(0x4780) /\* BLX R0 \*/    /\* Insert the reset handler address \*/  LONG(Reset) /\* Address of the reset vector \*/  . = ALIGN(4); /\* Align to a 4-byte boundary \*/    KEEP(\*(.start\_section)); /\* Keep the section in the ELF file \*/  KEEP(\*(.firmware\_startup)); /\* Keep the startup section in the ELF file \*/  } > START\_FLASH  } |

I then added back in the GPIO functionality and the code ran! The gpio write turns on the red LED.

A close up of a circuit board

Description automatically generated

Now, it looks like I just need to clean up the repository and push it to GitHub so that others can use it.