# OpenBMC Cross-Debugging with GDB

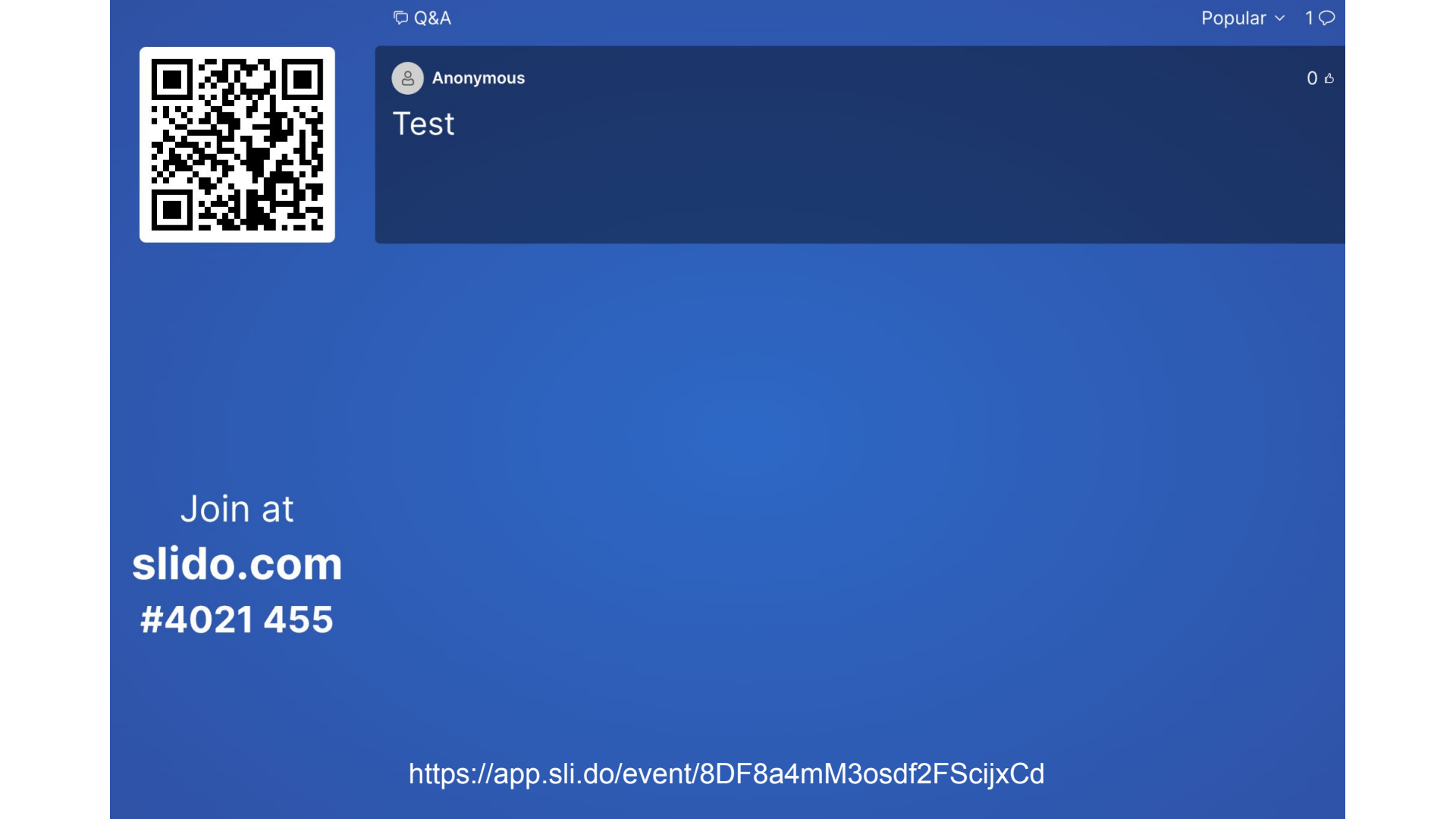
## Techniques for Memory-Constrained BMCs

Slido Q/A

Doc

**COSCUP 2025 · Taipei, Taiwan**

**Speaker:** Alan Tseng | **Date:** 10 Aug 2025 (Sun.)


Q&A
Popular
1
Anonymous
0
Test
Join at
slido.com
#4021 455
https://app.sli.do/event/8DF8a4mM3osdf2FScijxCd

# About me

- BMC Engineer @ Server ODM
- 1 year Kubernetes & OpenBMC / Buildroot
- Recent OSS:
  - nuta/operating-system-in-1000-lines (RISC-V)

- Links
  - hackmd.io/@alanhc
  - https://www.linkedin.com/in/alanhc316/

# Learning Objectives

After this talk you will be able to…

1. Configure GDB for cross-debugging on a 64 MB BMC.
2. Explain the core GDB architecture (remote stub, gdbserver, symbols).
3. Apply userspace debugging workflows (gdbserver, core-dump).

# Out-of-Scope

Not covered today – happy to chat afterwards

1. Yocto build internals
2. Kernel-space debugging
3. Perf / eBPF profiling
4. AI tooling (MCP)

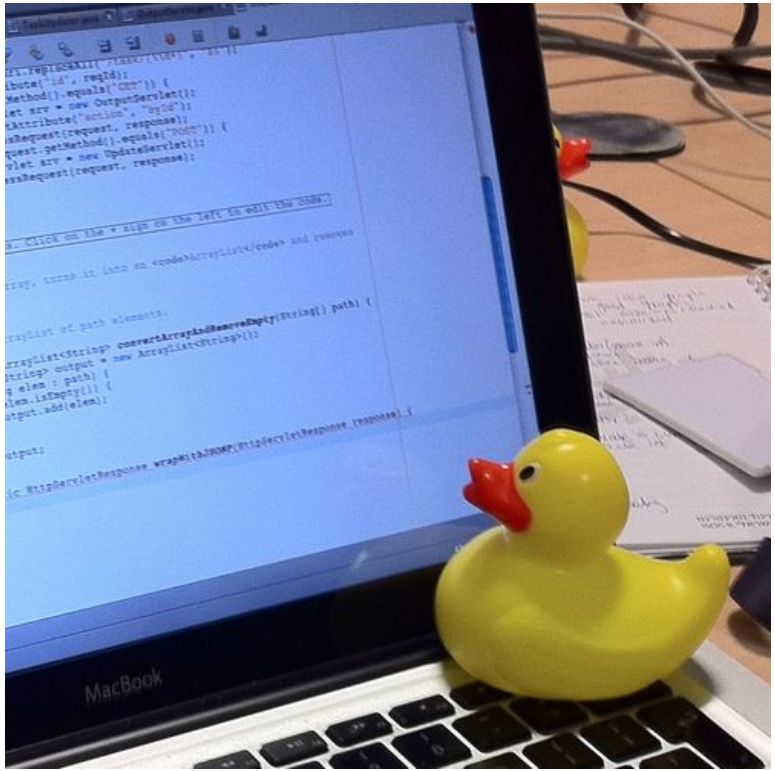
For Yocto basics, feel free to check out my notes. I've written a step-by-step guide on how to set up everything on a Raspberry Pi.

# Agenda

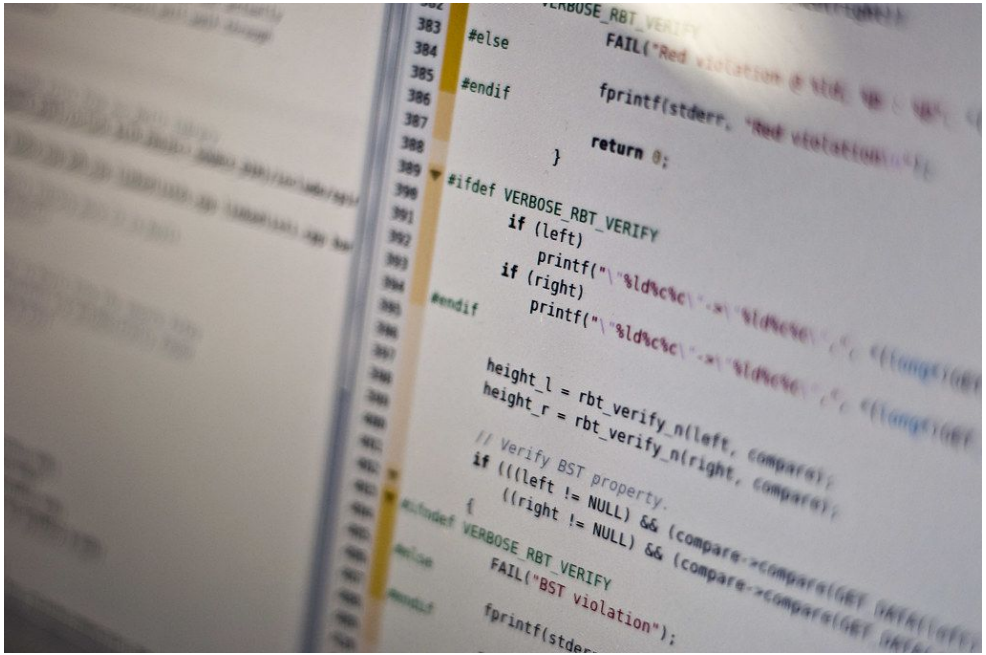
1. Motivation & real-world crash
2. Toolchain
3. Cross-debug setup
4. SIGSEGV walk-through
5. Deep dive: reading a core file
6. Q-A & Resources

If debugging is the process of removing software bugs, then programming must be the process of putting them in. - Edsger W. Dijkstra

# You might heard this before…




Rubber Duck Debugging

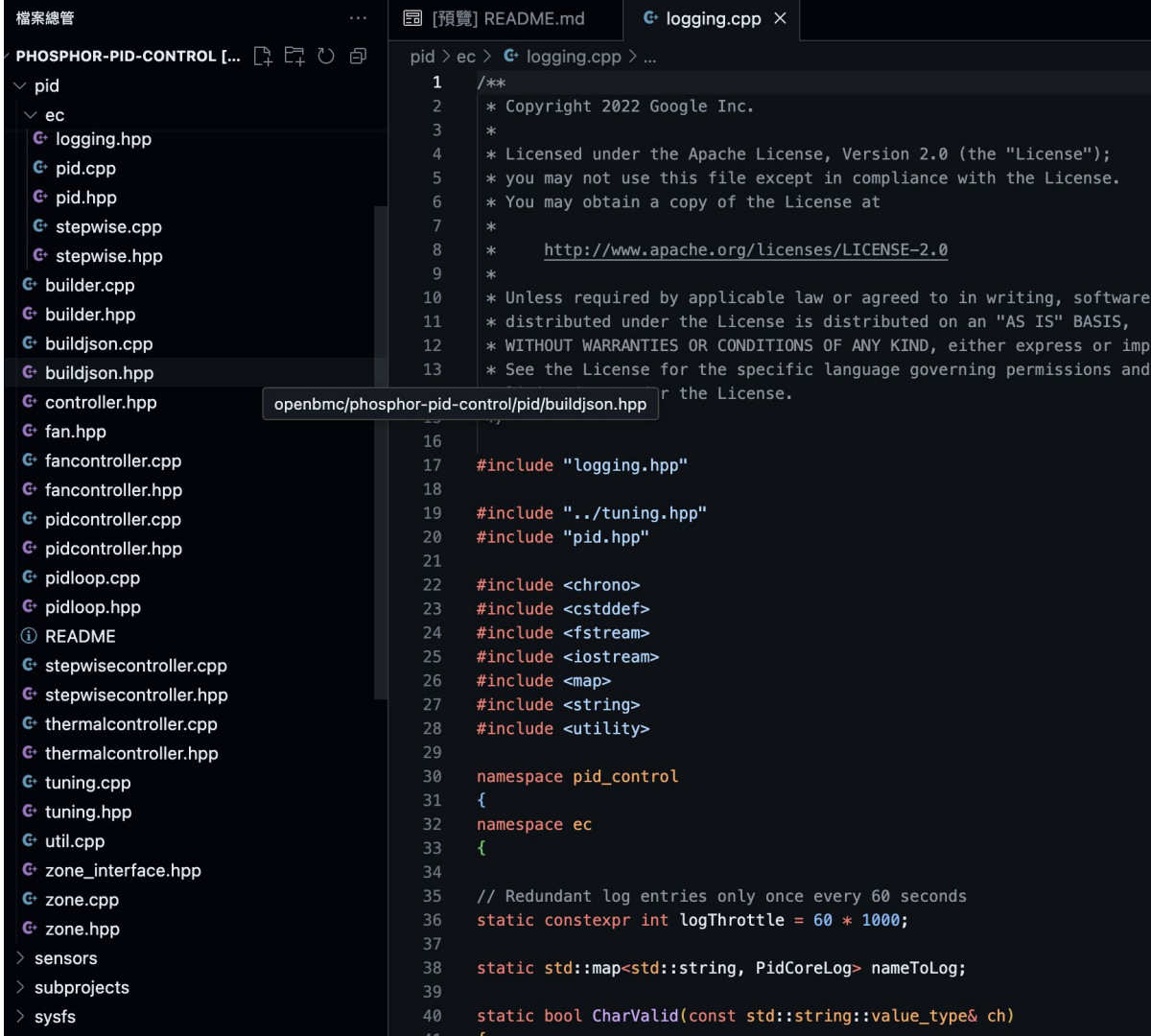



`printf` Debugging

# In real scenario

| | | printf |
|---|---|---|
| pros | ● Requires no tools | ● Quick way to observe values and control flow. |
| cons | ● Cannot directly identify bugs or runtime issues.<br>● Less effective for complex or low-level problems. | ● Requires recompilation and rerunning the program multiple times.<br>● Output can get messy and hard to interpret. |

# But in real world




https://github.com/openbmc/phosphor-pid-control

# Motivation

- Why bother with GDB on a tiny BMC?
  a. Logs show only Segmentation fault → need line-level context.
  b. Recompiling printf() is slow → need interactive breakpoints.

```
root@192.168.5.6's password:
sh-5.2# systemctl status hello.service
x hello.service - Hello OpenBMC Meson Demo
     Loaded: loaded (/usr/lib/systemd/system/hello.service; enabled; preset: enabled)
     Active: failed (Result: core-dump) since Mon 2025-08-04 22:49:43 UTC; 1 day 12h ago
   Duration: 514ms
 Invocation: abaff80d6408466a9b4ef13e3d890bc2
    Process: 388 ExecStart=/usr/bin/hello (code=dumped, signal=SEGV)
   Main PID: 388 (code=dumped, signal=SEGV)

Aug 04 22:49:41 raspberrypi3 systemd[1]: hello.service: Failed with result 'core-dump'.
Aug 04 22:49:43 raspberrypi3 systemd[1]: hello.service: Scheduled restart job, restart counter is at 2.
Aug 04 22:49:43 raspberrypi3 systemd[1]: hello.service: Start request repeated too quickly.
Aug 04 22:49:43 raspberrypi3 systemd[1]: hello.service: Failed with result 'core-dump'.
Aug 04 22:49:43 raspberrypi3 systemd[1]: Failed to start Hello OpenBMC Meson Demo.
```

# GDB

```
#0  0x004ba72e in main () at /usr/src/debug/hello/1.0/main.cpp:7
(gdb)
(gdb)
(gdb) list
2       #include <iostream>
3
4       int main() {
5               std::cout << "Hello, World!" << std::endl;
6               int* ptr = nullptr;
7               std::cout << *ptr << std::endl; // Dereferencing a null pointer causes a segmentation fault (core dump)
8               return 0;
9       }
```

pros

- Provides deep insight into program internals: memory, registers, stack, etc.
- Excellent for analyzing segmentation faults, crashes, and core dumps.

cons

- Steeper learning curve for beginners.
- lower than printf debugging for simple issues.
- Might be affected by compiler optimizations

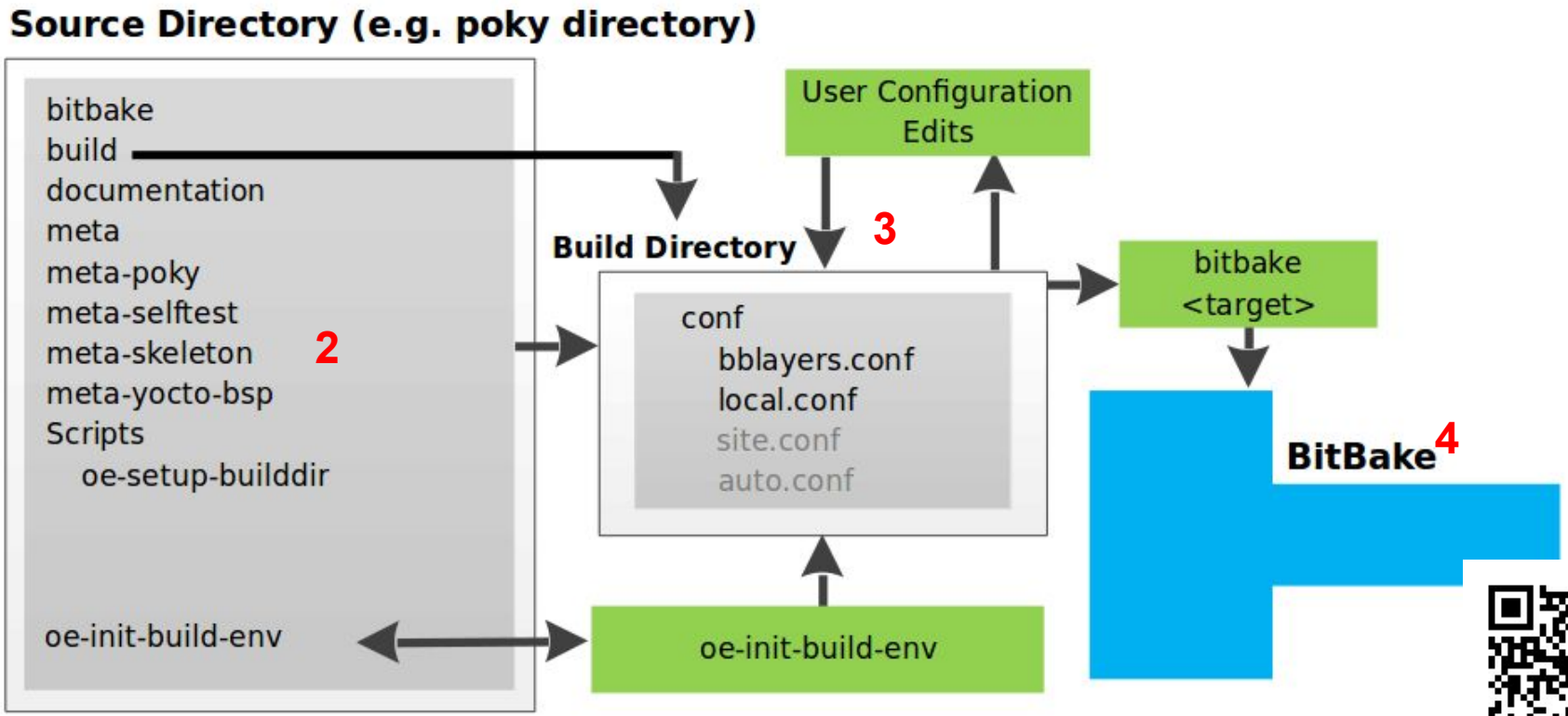
# Debugging Methods

| | printf() | GDB |
|---|---|---|
| Setup | none | one-time gdbserver |
| Visibility | values only | full stack / registers |
| Turnaround | recompile | live |
| Best for | simple logic | crashes, memory |

# Backgrounds - Yocto Project

- **meta-layer**: A collection of recipes, configuration files, and classes that define how software is built in Yocto.
- **local.conf**: The main user configuration file in Yocto.
- **layer.conf**: A configuration file inside each meta-layer.
- **BB file**: Defines how to fetch, configure, compile, and install a package.
- **meson.build**: The project definition file for the Meson build system.

# Backgrounds - Yocto Project




https://docs.yoctoproject.org/singleindex.html

# Backgrounds - Toolchain

- **GNU Compiler Collection (GCC)**: A widely used compiler suite that supports C, C++, and other languages.
- **GNU Binutils**: A set of binary tools (e.g., as, ld, objdump) used for assembling, linking, and inspecting binaries during the build process.
- **Meson / GNU Make**: Build systems that control how software is compiled.
- **GNU Debugger (GDB)**: A powerful debugger for tracking down bugs in compiled programs.
- **ABI (Application Binary Interface)** defines how binary code components (like object files or libraries) interact at the machine-code level.

# Yocto + Toolchain backgrounds

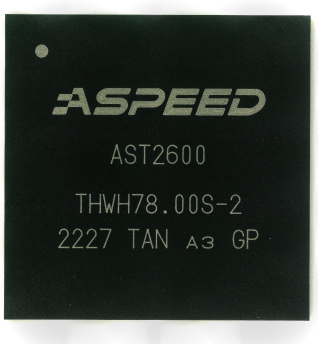
- Layer → recipes → .bb → bitbake
- SDK: oecore-x86_64 ⇒ cross compiler + sysroot

# Compiler Flags for Debugging and Optimization

- **Optimization Level**: Specifies how aggressively the compiler optimizes the code for performance or size
- **Debug Symbol**: Debug symbols map compiled machine code back to source code (function names, line numbers, variable names).
- **Frame Pointer**: A frame pointer (usually ebp/rbp/fp) is used to keep track of function call stacks.

```
CFLAGS:append   = " -O0 -g3 -fno-omit-frame-pointer"
CXXFLAGS:append = " -O0 -g3 -fno-omit-frame-pointer"
```

# Problem




BMC

ARM Cortex-A7


Host

x86_64

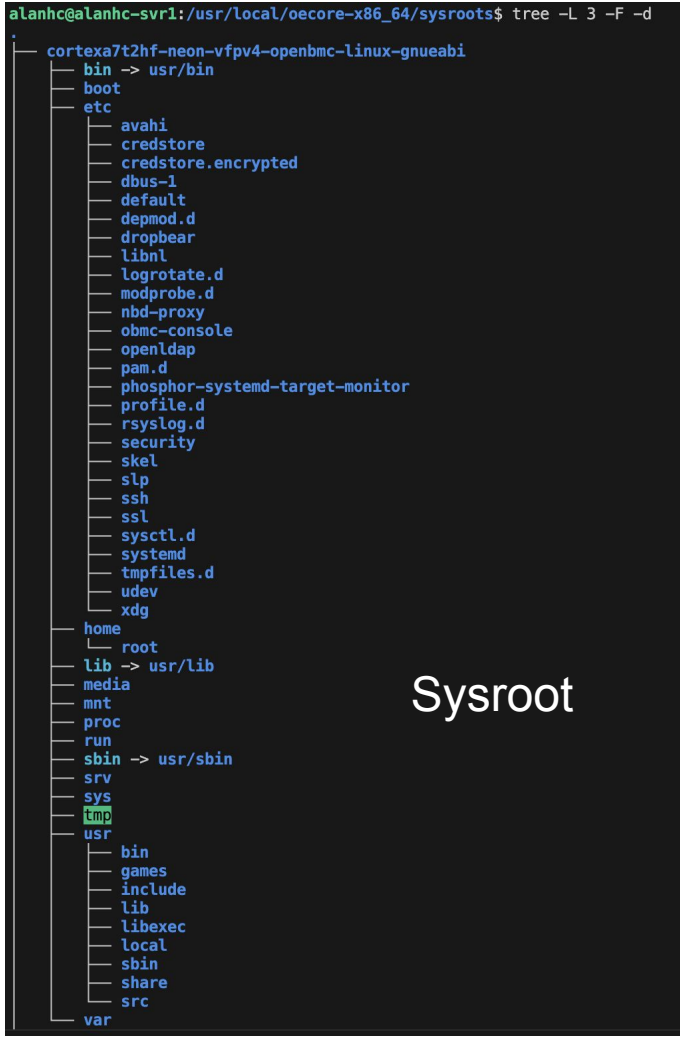
x86_64 host
└── cross-compiler ──▶ ARM Cortex-A7 BMC (OpenBMC)

# Cross Debug

- **Cross Compiler**: Generates binaries and debug symbols for the target architecture, not the host.
- **Sysroot**: Provides GDB access to the target system's shared libraries (e.g., libc.so, libm.so), enabling accurate symbol resolution and call stack tracing.
- **ABI(Application Binary Interface)**: Defines how registers, stack frames, and floating-point operations are handled on the target.
- **Debug symbol**: Maps memory addresses to source code, allowing GDB to show file names, line numbers, and variable names during debugging.
- **Substitute-path**: Maps build-time source paths (on the target) to local paths (on the host), so GDB can locate and display the correct source code.

# Setup Sysroot

| step | Command | Generate path |
|---|---|---|
| Build sdk | bitbake -c populate_sdk obmc-phosphor-image | tmp/deploy/sdk |
| install | ./tmp/deploy/sdk/oecore-obmc-phosphor-image-x86_64-cortexa7t2hf-neon-vfpv4-raspberrypi3-toolchain-nodistro.0.sh | /usr/local/oecore-x86_64 |



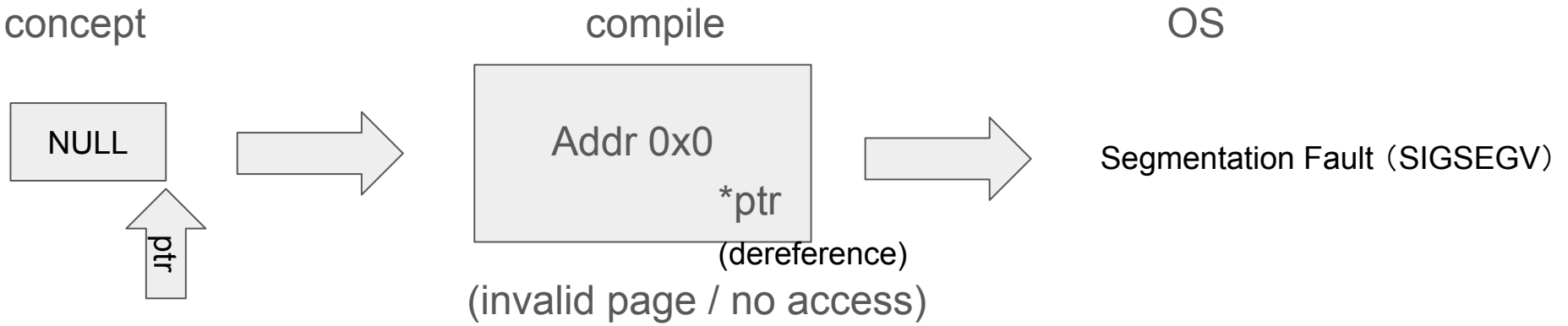
# Cross Environment commands

| Description | command |
| --- | --- |
| Source env | . /usr/local/oecore-x86_64/environment-setup-cortexa7t2hf-neon-vfpv4-openbmc-linux-gnueabi |
| cross compiler | echo $CC |
| target sysroot | echo $PKG_CONFIG_SYSROOT_DIR |
| toolchain | echo $PATH |
| Yocto SDK rootfs | echo $OECORE_TARGET_SYSROOT |

# SIGSEGV Example

```cpp
// main.cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    int* ptr = nullptr;
    std::cout << *ptr << std::endl; // Dereferencing a null pointer causes a segmentation fault (core dump)
    return 0;
}
```



# In Yocto Project doc…

In your `local.conf` file, set the following:

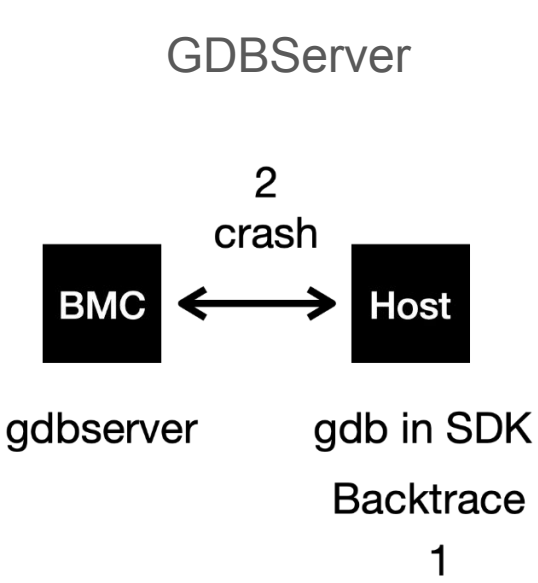
```
IMAGE_GEN_DEBUGFS = "1"
IMAGE_FSTYPES_DEBUGFS = "tar.bz2"
```

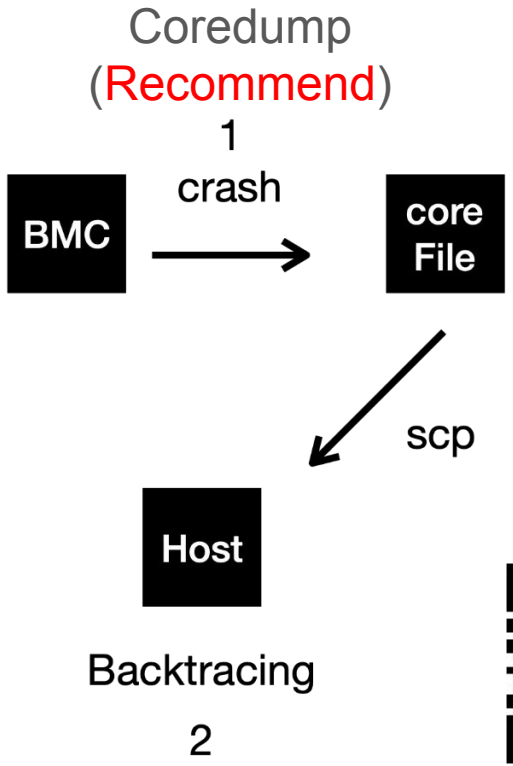
Make the following addition in your `local.conf` file:

```
EXTRA_IMAGE_FEATURES:append = " tools-debug"
```

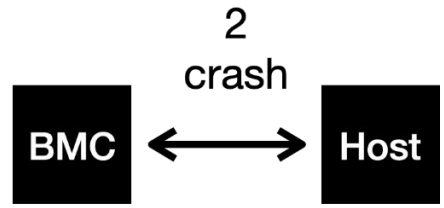
Your Image will become really large!

# Two less memory GDB Debugging methods



# GDBServer Remote debug

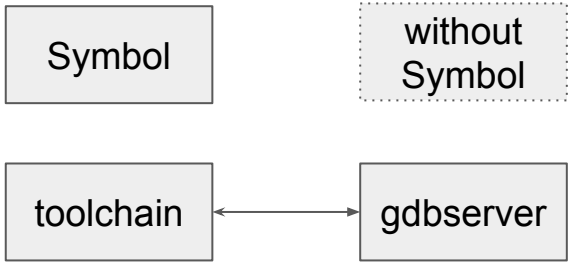



Host

BMC

Host Terminal:

(gdb) target remote $BMC_IP:1234




BMC Terminal:

gdbserver :1234 {program_binary}

2

1

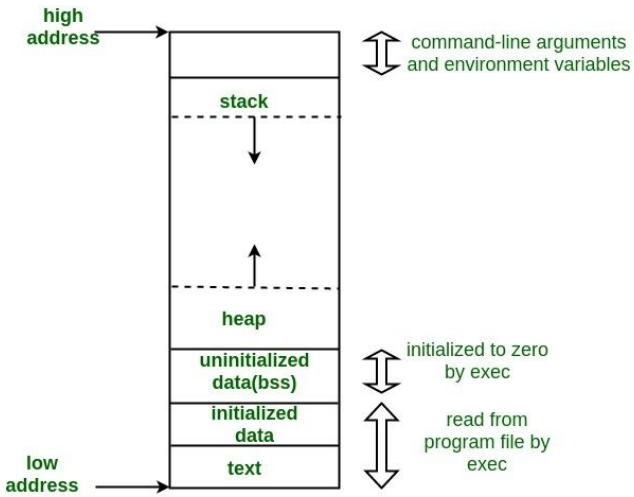
# Core Dump Analysis Backgrounds

When will core file be generate?

- When the process crashes, e.g., SIGSEGV
- System allows dumping (check ulimit -c)
- /proc/sys/kernel/core_pattern is valid and writable

# Core Dump Analysis Backgrounds

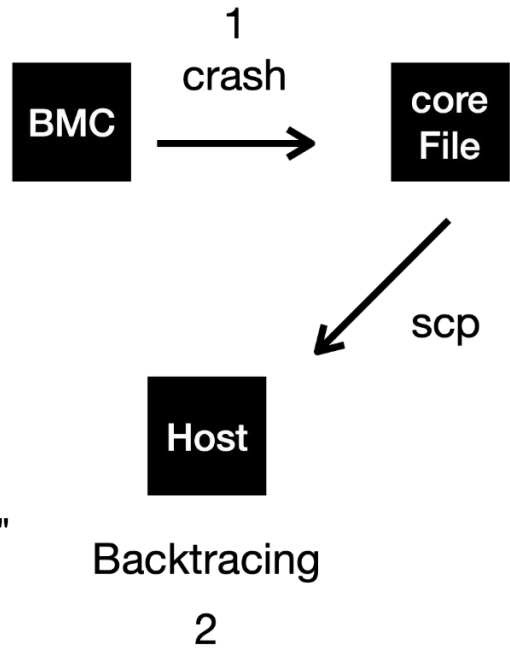
[Recall]




Memory Layout of C Programs

| Memory Section | Description |
|---|---|
| Stack | Function calls, local variables, backtrace info |
| Heap | Dynamically allocated memory (`malloc`, etc.) |
| .bss (Data) | Uninitialized global/static variables (known size) |
| .data | Initialized global/static variables |
| .text | Executable code (machine instructions) |
| Mapped Libraries (mmap) | Shared libraries (like libc, libm) |
| VDSO / Linker | Virtual dynamic shared object (e.g., for `gettimeofday`) |

# Core Dump Analysis Configuration




**local.conf**

```
IMAGE_INSTALL:append = " coreutils"
EXTRA_OECONF += "ULIMIT_CORE=unlimited"
EXTRA_OECONF +=
"KERNEL_PARAM_CORE_PATTERN=/var/lib/systemd/coredump/core.%e.%p.%t"
```

**.bb file**

```
CFLAGS:append = " -O0 -g3 -fno-omit-frame-pointer"
CXXFLAGS:append = " -O0 -g3 -fno-omit-frame-pointer"

INHIBIT_PACKAGE_STRIP = "1"
INHIBIT_PACKAGE_DEBUG_SPLIT = "0"
```

`-O0`: Disable all optimizations to preserve original code structure.

`-g3`: Include full debug information, including macro expansions.

`-fno-omit-frame-pointer`: Keep the frame pointer for stack tracing and better debugging (especially with GDB or perf).

**Don't strip binaries** (keep debug symbols in the binary)
Enable debug split

# .gdbinit

```
16  # === 1. 載入執行檔（含 .text 區段） ===
15  file /usr/local/oecore-x86_64/sysroots/cortexa7t2hf-neon-vfpv4-openbmc-linux-gnueabi/usr/bin/hello
14
13  # === 2. 載入完整除錯符號 ===
12  symbol-file /usr/local/oecore-x86_64/sysroots/cortexa7t2hf-neon-vfpv4-openbmc-linux-gnueabi/usr/bin/.debug/hello
11
10  # === 3. 載入 core dump ===
9   core-file hello.core
8
7   # === 4. 指定 sysroot 與共享函式庫路徑 ===
6   set sysroot /usr/local/oecore-x86_64/sysroots/cortexa7t2hf-neon-vfpv4-openbmc-linux-gnueabi
5   set solib-search-path $sysroot/usr/lib
4   set debug-file-directory $sysroot/lib/.debug
3
2   # === 5. 原始碼路徑映射 ===
1   set substitute-path /usr/src/debug/hello/1.0 /home/alanhc/workspace/coscup/openbmc/meta-mytest/recipes-example/hello/fil
17
1   # === 6. 顯示簡要狀態 ===
2   info sources
3   info sharedlibrary
4   bt
```

```
NORMAL  master  openbmc  hello.gdb   13  80%  17:1
```

Usage:  arm-openbmc-linux-gnueabi-gdb -x hello.gdbinit

# Congratulation, You find Bug!

```
(gdb) backtrace
#0  0x004ba72e in main () at /usr/src/debug/hello/1.0/main.cpp:7
```

# [Deepdive] How to calculate by hand?

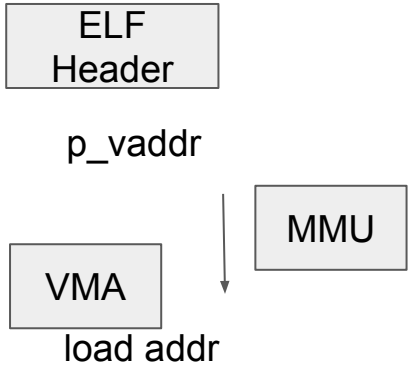
```
#0  0x004ba72e
```

(gdb) info files
0x004ba000 - 0x004bb000 is load1

Load Base

PC = 0x004ba72e
Offset = 0x004ba72e - 0x004ba000 = 0x72e

ELF offset




```
arm-openbmc-linux-gnueabi-addr2line:  a.out : No such file
alanhc@alanhc-svr1:~/workspace/coscup/openbmc/build/tmp/deploy/sdk$ arm-openbmc-linux-gnueabi-addr2line -e /usr/local/oecore-x86_64/sysroots/cortexa7t2hf-neon-vfpv4-openbmc-linux-gnueabi/usr/bin/.debug/hello 0x72e -f -C
main
/usr/src/debug/hello/1.0/main.cpp:7
```

# [Deepdive] How to calculate by hand?

```
/usr/src/debug/hello/1.0/main.cpp:7
alanhc@alanhc-svr1:~/workspace/coscup/openbmc/build/tmp/deploy/sdk$ arm-openbmc-linux-gnueabi-objdump -d hello.d

hello.debug:     file format elf32-littlearm


Disassembly of section .init:
```

```
00000604 <.text>:
 604:   0b00f04f        bleq    3c748 <__cxa_finalize@plt+0x3c150>
 608:   0e00f04f        cdpeq   0, 0, cr15, cr0, cr15, {2}
 60c:   466abc02        strbtmi fp, [sl], -r2, lsl #24
 610:   b401b404        strlt   fp, [r1], #-1028        @ 0xfffffbfc
```

# [Deepdive] How to calculate section address

## (gdb) info files

```
Local core dump file:
        `/home/alanhc/workspace/coscup/openbmc/build/tmp/deploy/sdk/h
        0x004ba000 - 0x004bb000 is load1
        0x004bb000 - 0x004bc000 is load2
        0x004bc000 - 0x004bd000 is load3
        0x00fbc000 - 0x00fdd000 is load4
        0x76cb6000 - 0x76cb8000 is load5
        0x76cb8000 - 0x76cb9000 is load6a
        0x76cb9000 - 0x76cfd000 is load6b
        0x76cfd000 - 0x76cfe000 is load7
        0x76cfe000 - 0x76cff000 is load8
        0x76cff000 - 0x76d00000 is load9a
        0x76d00000 - 0x76e01000 is load9b
        0x76e01000 - 0x76e03000 is load10
        0x76e03000 - 0x76e04000 is load11
        0x76e04000 - 0x76e0e000 is load12
        0x76e0e000 - 0x76e0f000 is load13a
        0x76e0f000 - 0x76e26000 is load13b
        0x76e26000 - 0x76e27000 is load14
        0x76e27000 - 0x76e28000 is load15
        0x76e28000 - 0x76e29000 is load16a
        0x76e29000 - 0x76fb5000 is load16b
        0x76fb5000 - 0x76fbc000 is load17
        0x76fbc000 - 0x76fbd000 is load18
        0x76fbd000 - 0x76fbf000 is load19
        0x76fc1000 - 0x76fc3000 is load20
        0x76fc3000 - 0x76fc4000 is load21a
        0x76fc4000 - 0x76fdf000 is load21b
        0x76fdf000 - 0x76fe1000 is load22
        0x76fe1000 - 0x76fe2000 is load23
        0x7ee8b000 - 0x7eeac000 is load24
        0x7ef79000 - 0x7ef7a000 is load25
        0x7ef7a000 - 0x7ef7b000 is load26
        0x7ef7b000 - 0x7ef7c000 is load27
        0xffff0000 - 0xffff1000 is load28
```

```
Local exec file:
        `/usr/local/oecore-x86_64/sysroots/cortexa7t2hf-neon-vfpv4-openbmc-linux-gnueabi/usr/bin/hello', file type elf32-littlearm.
        Entry point: 0x604    When program runs, cpu start from this
        0x00000174 - 0x00000198 is .note.gnu.build-id
        0x00000198 - 0x000001b5 is .interp
        0x000001b8 - 0x000001d0 is .gnu.hash
        0x000001d0 - 0x000002d0 is .dynsym
        0x000002d0 - 0x00000468 is .dynstr
        0x00000468 - 0x00000488 is .gnu.version
        0x00000488 - 0x00000508 is .gnu.version_r
        0x00000508 - 0x00000558 is .rel.dyn
        0x00000558 - 0x00000590 is .rel.plt
        0x00000590 - 0x0000059c is .init
        0x0000059c - 0x00000604 is .plt
        0x00000604 - 0x00000764 is .text
        0x00000764 - 0x0000076c is .fini
        0x0000076c - 0x000008a4 is .rodata
        0x000008a4 - 0x000008b0 is .ARM.extab
        0x000008b0 - 0x000008c8 is .ARM.exidx
        0x000008c8 - 0x000008cc is .eh_frame
        0x000008cc - 0x000008ec is .note.ABI-tag
        0x00001eac - 0x00001eb0 is .init_array
        0x00001eb0 - 0x00001eb4 is .fini_array
        0x00001eb4 - 0x00001fbc is .dynamic
        0x00001fbc - 0x00002000 is .got
        0x00002000 - 0x00002008 is .data
        0x00002008 - 0x0000200c is .bss
        0x7ef7b0d4 - 0x7ef7b100 is .hash in system-supplied DSO at 0x7ef7b000
        0x7ef7b100 - 0x7ef7b160 is .dynsym in system-supplied DSO at 0x7ef7b000
        0x7ef7b160 - 0x7ef7b1cf is .dynstr in system-supplied DSO at 0x7ef7b000
        0x7ef7b1d0 - 0x7ef7b1dc is .gnu.version in system-supplied DSO at 0x7ef7b000
        0x7ef7b1dc - 0x7ef7b214 is .gnu.version_d in system-supplied DSO at 0x7ef7b000
        0x7ef7b214 - 0x7ef7b268 is .note in system-supplied DSO at 0x7ef7b000
        0x7ef7b268 - 0x7ef7b2f0 is .dynamic in system-supplied DSO at 0x7ef7b000
        0x7ef7b2f0 - 0x7ef7b330 is .ARM.exidx in system-supplied DSO at 0x7ef7b000
```

file VMA

# [Deepdive] How to calculate section address

arm-openbmc-linux-gnueabi-addr2line -e hello 0x628

(range in memory) 0x004ba604 - 0x004ba764

This range in
(core file) 0x004ba000 - 0x004bb000 is load1

Entry point: 0x604 CPU start

The `.text` section starts at offset `0x604` in the ELF file.

0x004ba000 + 0x604 = 0x004ba604

Page base offset

0x00000764 - 0x00000604 = 0x160 bytes

.text end .text start start

ELF .text offset 0x604

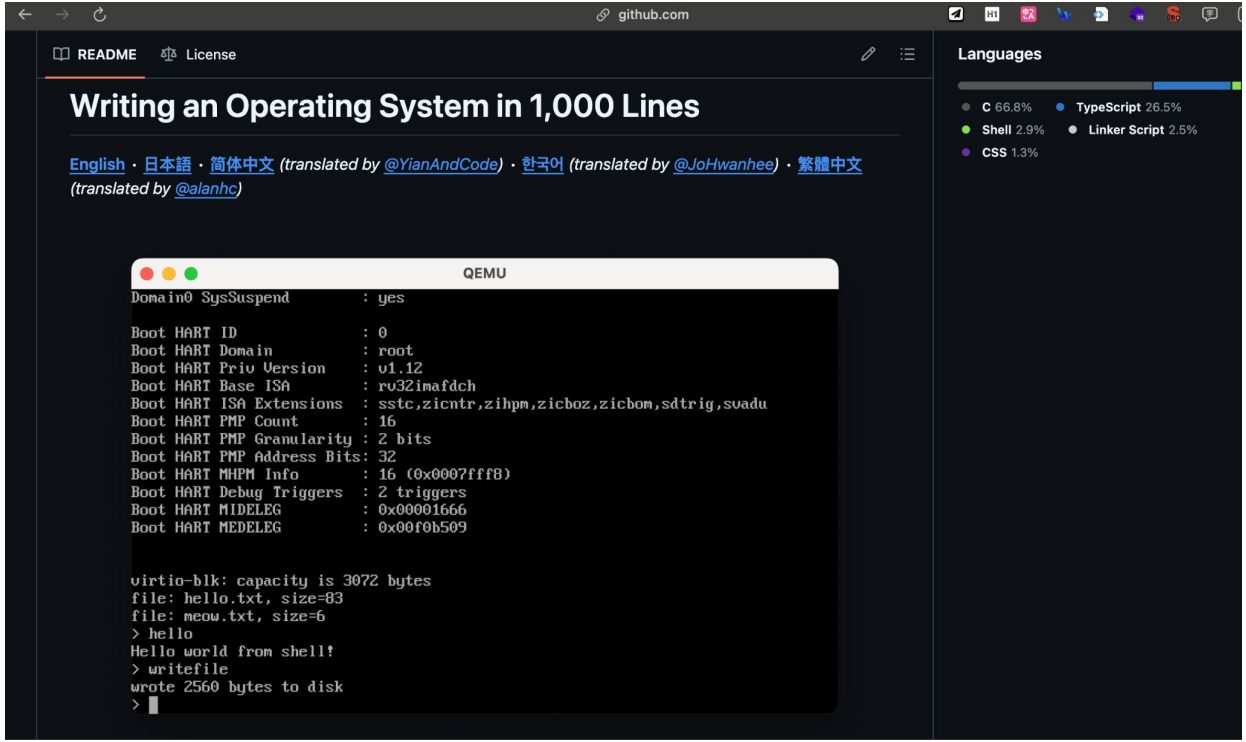
|

+ load base 0x004ba000

▼

Run-time PC 0x004ba604

# Deep dive to OS!




https://github.com/nuta/operating-system-in-1000-lines

# Acknowledgments

Jim Huang (jserv)


You can focus on
Week 1-6

Checkout this best Embedded related course!

# Useful Resources


https://youtu.be/JGhAgd2a_Ck

# Thank You 🙌

*Ask me anything!*

*If you have any question or opportunities*

*Please send to*

alan.tseng.cs@gmail.com or connect

https://www.linkedin.com/in/alanhc316/