and the Last Nasal Demon Riders

Ahmad Fatoum – a.fatoum@pengutronix.de

About Me

Ahmad Fatoum

Pengutronix

a3f

a.fatoum@pengutronix.de

@a3f@fosstodon.org

- Kernel and Bootloader Porting
- Driver and Graphics Development
- System Integration
- Embedded Linux Consulting

Overview

- Nasal Demons
- What and why of barebox multi-image
- Funky Stuff
  - Stack Setup
  - Relocations
  - Dereferencing Pointers
- Responsible Fun
- Future Outlook

Keep safe!

Content Warning

ARM (Dis-)Assembly
Creative Uses of C
Disturbance of the Dead
Imitable behavior

Nasal Demons

- Who has seen any recently?

Nasal Demons

- Who has seen any recently?

> *In the C programming community, undefined behavior may be humorously referred to as "nasal demons", after a comp.std.c post that explained undefined behavior as allowing the compiler to do anything it chooses, even "to make demons fly out of your nose".[2]*
>
> — https://en.wikipedia.org/wiki/Undefined_behavior

Nasal Demons

- One possible penalty for violating the standard
- May also spawn when you go beyond what the implementation (compiler, architecture, …) documents as supported
  - This is what we'll talk about today!

BAREBOX in (some) tidbits

- Bootloader, mostly for embedded systems (ARM/x86/RISC-V/MIPS/kvx/or1k/PPC)
- Emphasis on adopting ideas and sharing code (**driver model and frameworks**) with the **Linux kernel**
- **VFS** with common / with ramfs, devfs and mounted file systems
- File descriptor API and **familiar libc** for implementing commands
- Opinionated **frameworks**, less ad-hoc: Complex redundant/fallback boot scenarios possible with **no scripting required**
- **Multiplatform**: `multi_v7/v8_defconfig` builds dozens of boards at once

Building barebox

```
$ make ARCH=arm64 multi_v8_defconfig && make ARCH=arm64 -j$(nproc)
  ...
  LD      images/start_dt_2nd.pbl          # one PBL link per entry point (x52)
  OBJCOPYB images/start_dt_2nd.pblb
  ...
images built:
barebox-raspberry-pi.img
barebox-exynos.img
barebox-qemu-virt.img
barebox-zynqmp-zcu102.img
barebox-beagleplay.img
barebox-dt-2nd.img
** barebox-nxp-imx8mm-evk.img skipped due to missing firmware **
** barebox-rock5b.img skipped due to missing firmware **
   (+40 more: bring your own DDR training blobs / TF-A)
```

Multiple Images: barebox proper is built once, compressed and then linked against the individual *prebootloaders* (PBLs)

Building barebox

```
$ make ARCH=arm64 multi_v8_defconfig && make ARCH=arm64 -j$(nproc)
  ...
  LD      images/start_dt_2nd.pbl          # one PBL link per entry point (x52)
  OBJCOPYB images/start_dt_2nd.pblb
  ...
images built:
barebox-raspberry-pi.img
[…]
```

barebox proper is built once, compressed and then linked against the individual *prebootloaders* (PBLs)

+ Nice for developers: Less variants to build for coverage
+ Nice for users: forces board code to be non-intrusive,
  e.g. one barebox-raspberry-pi.img for all of Raspberry Pi 1, 2, 3, 4, ...

- Same build: How to ensure no unused code/firmware ends up in PBLs?
  e.g. Rockchip barebox image has no need for i.MX DDR firmware

multi-image: The beginning

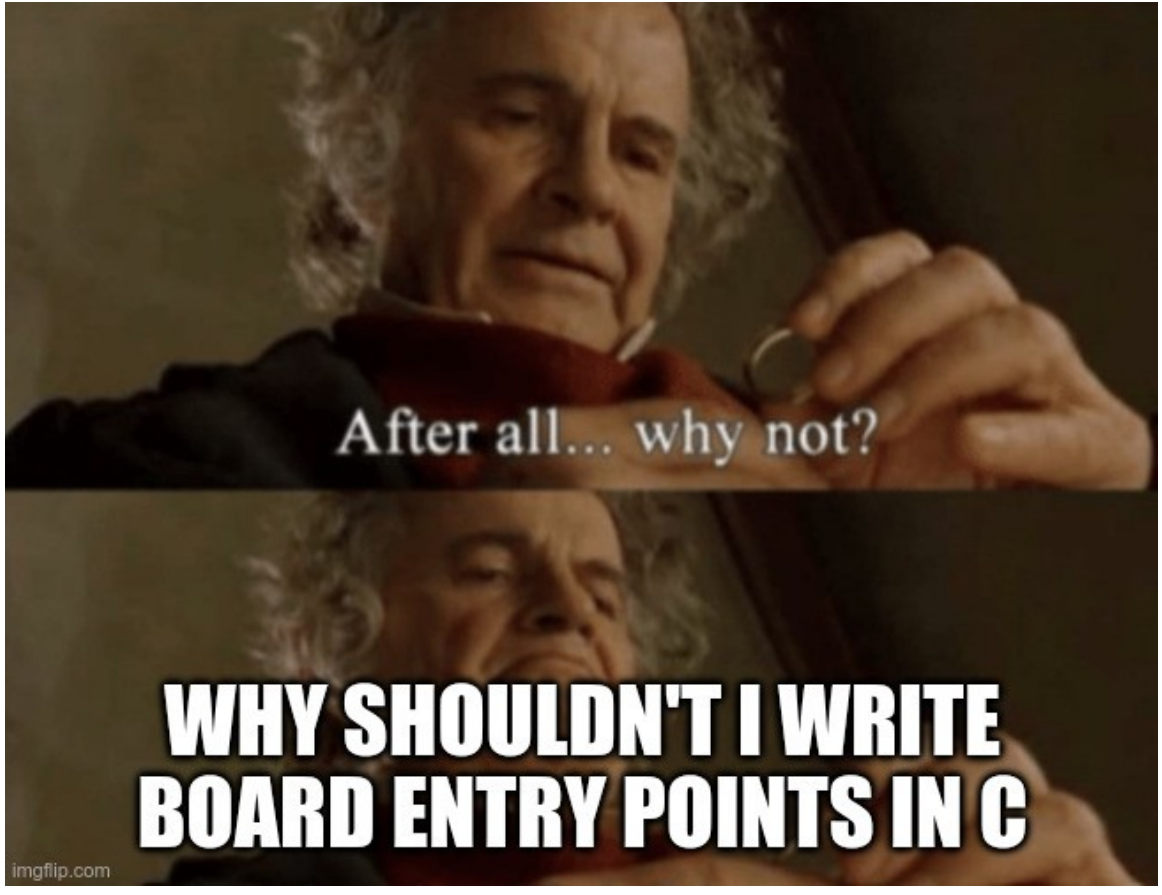
- Each board defines its own entry point(s)
- Linker Garbage Collection cuts off anything not reachable from the entry point
- e.g. `barebox_arm_entry()` called →barebox proper is linked in

multi-image: The beginning

- Each board defines its own entry point(s)
- Linker Garbage Collection cuts off anything not reachable from the entry point
- e.g. `barebox_arm_entry()` called →barebox proper is linked in

"I like writing my board entry point in assembly"
– said *nobody* porting a board *ever*

multi-image: The beginning



(SPOILER: You'll see why soon enough)

A naive C entry function

```
void start_imx6s_riotboard(ulong r0, ulong r1, ulong r2)
{
        imx6_cpu_lowlevel_init();

        relocate_to_current_adr();
        setup_c();

        continue_imx6s_riotboard();
}
```

Local Variables

(and how to shoot yourself into the foot with them)

Setting up a stack pointer in C

- Fun fact: The C standard speaks of no stack!
  - Yet you'll see a stack push prologue in nearly every function.

```
$ arm-linux-gnueabihf-objdump -d arch/arm/boards/embest-riotboard/lowlevel.pbl.o

00000000 <__start_imx6s_riotboard.isra.0>:
     0: push    {r3, lr}
     2: bl      imx6_cpu_lowlevel_init
     6: bl      relocate_to_current_adr
     a: bl      setup_c
     e: bl      continue_imx6s_riotboard
```

⚡ BootROM may or may not have left a usable `sp` when it jumped to us.

__naked temptation

```
static inline void arm_setup_stack(unsigned long top)
{
        __asm__ __volatile__("mov sp, %0" : : "r"(top));
}
```

```
#define __naked __attribute__((naked)) notrace
void __naked start_imx6ul_ccimx6ulsbcpro(ulong r0, ulong r1, ulong r2)
{
        void *fdt;

        imx6ul_cpu_lowlevel_init();

        arm_setup_stack(0x00910000);

        relocate_to_current_adr();
        setup_c();
        /* ... */
        imx6ul_barebox_entry(fdt);
}
```

No prologue and no epilogue, but:

Nothing stops GCC from spilling `fdt` to the stack before `arm_setup_stack()` has run...

What GCC says about naked

> *Only basic asm statements can safely be included in naked functions. While using extended asm or a mixture of basic asm and C code may appear to work, they cannot be depended upon to work reliably and are not supported.*
>
> — GCC manual, *Common Function Attributes*

- `"mov sp, %0" : : "r"(top)` is extended asm.
- clang: `error: non-ASM statement in naked function is not supported`
- AArch64: no `naked` support at all.
  - Supported on RISC-V though (including 64-bit)

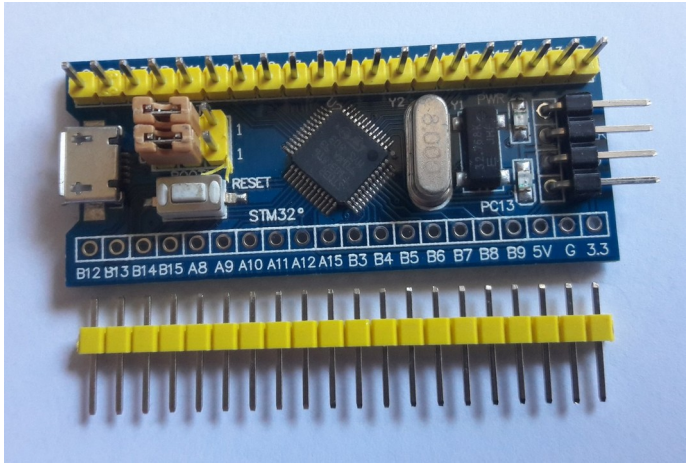
GCC logo, Free Software Foundation, CC BY-SA 3.0 · commons.wikimedia.org/wiki/File:GNU_Compiler_Collection_logo.svg

Some Cortex-M inspiration

Cortex-M vector table: offset `0x0` holds the initial SP, offset `0x4` the reset vector.

> *On reset, the processor loads the MSP with the value from address 0x00000000.*
>
> — Arm DUI 0552A

- The hardware sets up the stack before the first instruction runs.
- So let's get the same behavior on a Cortex-A..



STM32F103 "Blue Pill", Popolon, CC BY-SA 4.0 · commons.wikimedia.org/wiki/File:Core_Learning_Board_module_Arduino_STM32_F103_C8T6.jpg

ENTRY_FUNCTION_WITHSTACK

```
ENTRY_FUNCTION_WITHSTACK(start_beagleplay, 0x80800000, r0, r1, r2)
{

    arm_cpu_lowlevel_init();

    relocate_to_current_adr();
    setup_c();

    beagleplay_continue();
}
```

The linker script ties head, entry and stack top together

```
        .text : {
                *(.text_head_prologue*)      <-- head_64.S: first bytes of the image
                *(.text_head_entry*)         <-- the one board entry that survived GC
                ...
        __pbl_board_stack_top = .;
        .rodata.pbl_board_stack_top : {
                *(.pbl_board_stack_top_*)
                ASM_LD_PTR(0x00000000)       /* Dummy for when BootROM sets up usable stack */
        }
        ASSERT(. - __pbl_board_stack_top <= 2 * ASM_SZPTR, "Only One PBL per Image allowed")
```

```
ENTRY(__barebox_arm64_head)             /* arch/arm/cpu/head_64.S */
        nop
        adr x9, __pbl_board_stack_top
        ldr x9, [x9]
        cbz x9, 1f
        mov sp, x9
1:      nop
        nop
        b __pbl_board_entry                 /* --defsym on the ld command line */
        .org 0x20
        .asciz "barebox"
```

`-e start_board` roots the entry, `--gc-sections` discards every other board's, and `adr` finds the one that is referenced.

ENTRY_FUNCTION_WITHSTACK

```
#define ENTRY_FUNCTION_WITHSTACK_HEAD(name, stack_top, head, arg0, arg1, arg2) \
        void name(ulong r0, ulong r1, ulong r2);                                 \
        static void __##name(ulong, ulong, ulong);                              \
        void __section(.text_head_entry_##name) name                            \
                                (ulong r0, ulong r1, ulong r2)                  \
                {                                                               \
                        static __section(.pbl_board_stack_top_##name)           \
                                const ulong __stack_top = (stack_top);          \
                        __keep_symbolref(head);                                 \
                        __keep_symbolref(__stack_top);                          \
                        __##name(r0, r1, r2);                                   \
                }                                                               \
                static void noinline __##name                                   \
                        (ulong arg0, ulong arg1, ulong arg2)
```

```
/* This generates a useless load from the specified symbol
 * to ensure linker garbage collection doesn't delete it */
#define __keep_symbolref(sym)   __asm__ __volatile__("": :"r"(&sym) :)
```

No __naked. The stack top is a data word in its own section.

Disabling the MMU

barebox ARMv8 code from 2016 to 2025:

```
void mmu_disable(void)
{
        unsigned int cr;

        cr = get_cr();
        cr &= ~(CR_M | CR_C);

        set_cr(cr);
        /* Caches disabled → accesses bypass the still dirty cache */

        v8_flush_dcache_all();
        /* Data-Cache is now clean */
        tlb_invalidate();

        dsb();
        isb();
}
```

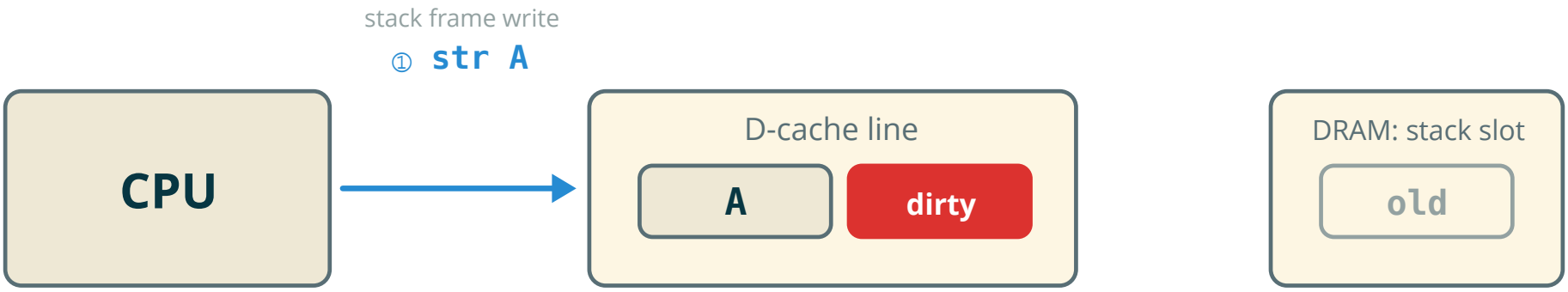
Disabling the MMU vs Stack Frame

barebox ARMv8 code from 2016 to 2025:

```
void mmu_disable(void)
{
        unsigned int cr;

        cr = get_cr();
        cr &= ~(CR_M | CR_C);

        set_cr(cr);
        /* Caches disabled → accesses bypass the still dirty cache */
        /* Insert a stack spill here */
        v8_flush_dcache_all();
        /* Data-Cache is now clean */
        tlb_invalidate();

        dsb();
        isb();
}
```

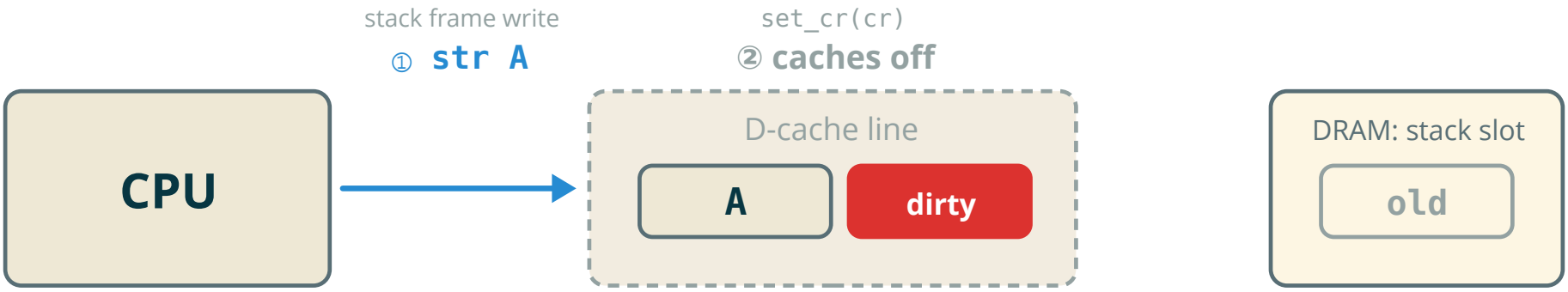
Disabling the MMU: inverted writes

① The stack frame write lands in the D-cache



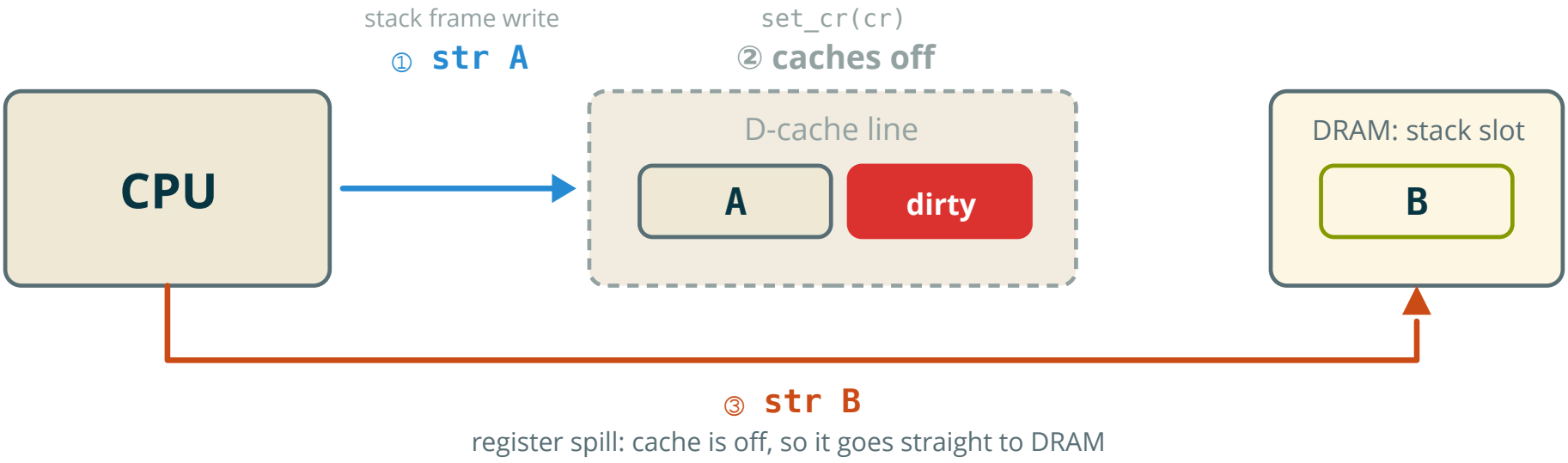
Disabling the MMU: inverted writes

② Caches go off: the dirty line stays behind



Disabling the MMU: inverted writes

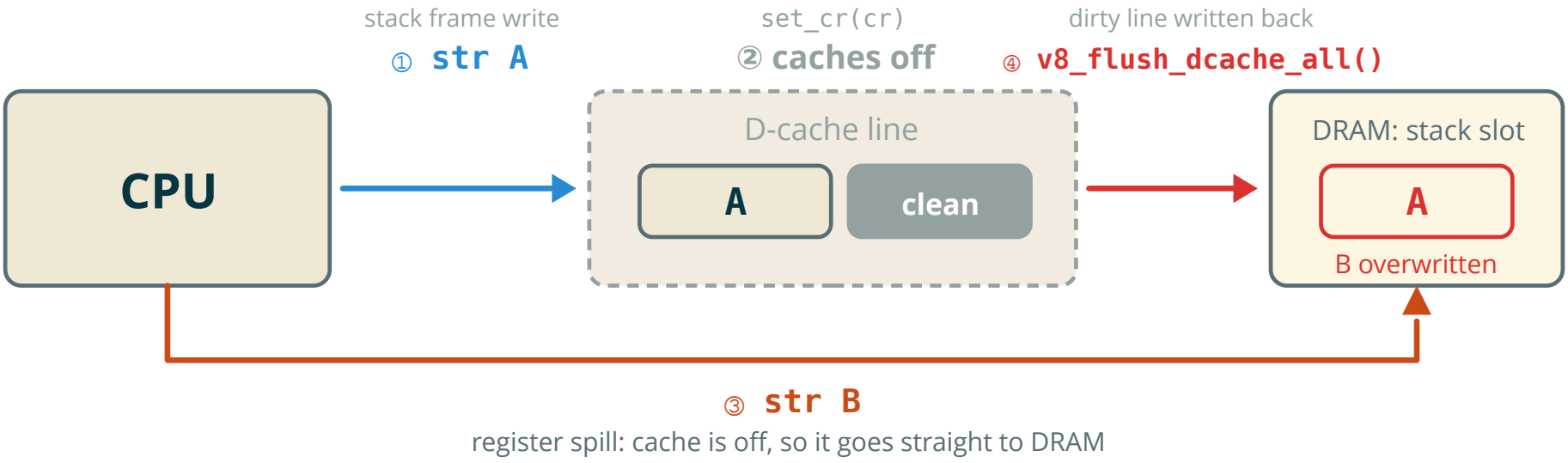
③ The spill bypasses the cache, straight to DRAM



register spill: cache is off, so it goes straight to DRAM

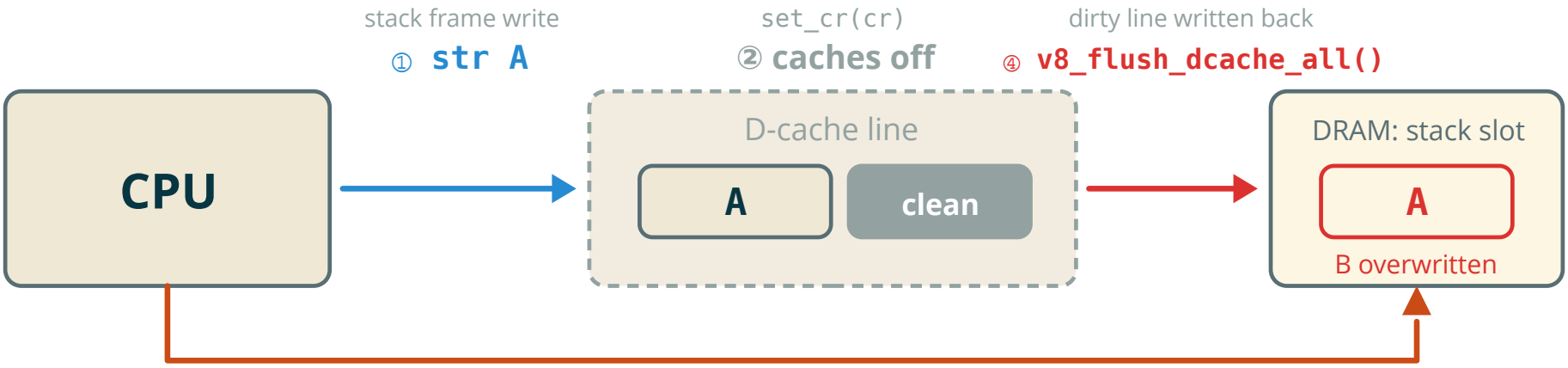
Disabling the MMU: inverted writes

④ Cleaning writes the stale line back over it



Disabling the MMU: inverted writes

⑤ Both writes landed: in the wrong order



program order	str A	→	str B	**inverted!**
what DRAM sees	str B	→	str A	

Disabling the MMU sans Stack Frame

barebox ARMv8 code from 2016 to 2025:

```
void mmu_disable(void)
{
        unsigned int cr;

        cr = get_cr();
        cr &= ~(CR_M | CR_C);

        set_cr(cr);
        /* Caches disabled → accesses bypass the still dirty cache */
        /* Insert a stack spill here */
        v8_flush_dcache_all();
        /* Data-Cache is now clean */
        tlb_invalidate();

        dsb();
        isb();
}
```

🔧 Routine must be implemented in assembly only using registers

Global Variables

(and how to shoot yourself in the foot with them)

Relocations

- PBL and barebox proper are position-independent and linked at `0x0`
  They run wherever the BootROM, the previous stage or
  `bootm /mnt/tftp/barebox-raspberry-pi.img` put them.
- Every absolute address in the image needs fixing up.
  The code that does it is written in C and runs *before* it is relocated itself.

```
        relocate_image(offset,
                       runtime_address(__rel_dyn_start),   /* a global */
                       runtime_address(__rel_dyn_end),     /* a global */
                       runtime_address(__dynsym_start),    /* a global */
                       runtime_address(__dynsym_end));     /* a global */
```

How do we access a global variable before global variables are set up?

Prerelocation &global_variable

- To get a global variable's absolute address, we need:
  - its unrelocated address
  - The relocation offset that fixing up relocations would add

Prerelocation &global_variable

- To get a global variable's absolute address, we need:
  - **its unrelocated address**
  - The relocation offset that fixing up relocations would add

```
.rel_dyn_start : { *(.__rel_dyn_start) }
.rela.dyn : { *(.rela*) }
.rel_dyn_end : { *(.__rel_dyn_end) }
```

```
char __rel_dyn_start[0] __attribute__((section(".__rel_dyn_start")));
char __rel_dyn_end[0]   __attribute__((section(".__rel_dyn_end")));
```

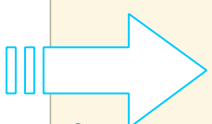
Prerelocation &global_variable

- To get a global variable's absolute address, we need:
  - its unrelocated address
  - **The relocation offset that fixing up relocations would add**

```
static inline __prereloc unsigned long global_variable_offset(void)
{
        unsigned long text;

        /* Open-code position-independent access in Assembly */
        __asm__ __volatile__("adrp   %0, _text\n"
                             "add    %0, %0, :lo12:_text\n" : "=r" (text)
                             : : "memory");
        /* Compute actual address minus what the compiler emitted */
        return text - (unsigned long)_text;
}
```

Putting it all together

```
void relocate_to_current_adr(void)
{
        /* […] */

        /* Get offset between linked address and runtime address */
        offset_var = global_variable_offset();

        dstart = (void *)__rel_dyn_start + offset_var;
        dend = (void *)__rel_dyn_end + offset_var;

        dynsym = (void *)__dynsym_start + offset_var;
        dynend = (void *)__dynsym_end + offset_var;

        while (dstart < dend) {
                /* […] */
        }


        __memset(dynsym, 0, (unsigned long)dynend - (unsigned long)dynsym);
        sync_caches_for_execution();
}
```



Putting it all together

```
void relocate_to_current_adr(void)
{
        /* […] */

        /* Get offset between linked address and runtime address */
        offset_var = global_variable_offset();

        dstart = (void *)__rel_dyn_start + offset_var;
        dend = (void *)__rel_dyn_end + offset_var;

        dynsym = (void *)__dynsym_start + offset_var;
        dynend = (void *)__dynsym_end + offset_var;

        while (dstart < dend) {
                /* […] */
        }

        dynsym = (void *)__dynsym_start + offset_var;
        dynend = (void *)__dynsym_end + offset_var;

        __memset(dynsym, 0, (unsigned long)dynend - (unsigned long)dynsym);
        sync_caches_for_execution();
}
```

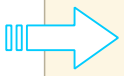
Putting it all together

```
void relocate_to_current_adr(void)
{
        /* […] */

        /* Get offset between linked address and runtime address */
        offset_var = global_variable_offset();

        dstart = (void *)__rel_dyn_start + offset_var;
        dend = (void *)__rel_dyn_end + offset_var;

        dynsym = (void *)__dynsym_start + offset_var;
        dynend = (void *)__dynsym_end + offset_var;

        while (dstart < dend) {
                /* […] */
        }

        __memset(dynsym, 0, (unsigned long)dynend - (unsigned long)dynsym);
        sync_caches_for_execution();
}
```

Compiler is within rights though to assume variables to be already relocated though and thus proves that &__dynsym_end may not change in the loop

Materialization of variable references

GCC 11.1.1, THUMB2, `relocate_to_current_adr()`:



```
    while (dstart < dend) {
179c:       f1a3 0608       sub.w   r6, r3, #8
17a0:       42b7            cmp     r7, r6
17a2:       d80a            bhi.n   17ba <relocate_to_current_adr+0x46>
    dynend = (void *)__dynsym_end + offset_var;
17a4:       4b43            ldr     r3, [pc, #268]   ; <-- after the loop that rewrote this word
    __memset(dynsym, 0, (unsigned long)dynend - (unsigned long)dynsym);
17a8:       58eb            ldr     r3, [r5, r3]
17aa:       441a            add     r2, r3
17ac:       1a12            subs    r2, r2, r0
17ae:       f000 fda5       bl      22fc <__memset>   ; <-- size off by `offset`
```

⚡ Offset ended up being added twice..

`runtime_address()`: three layers of paranoia

```
#define runtime_address(sym) ({                                         \
        void *volatile __addrof_sym = (sym);                            \  /* materialize now */
        if (!__is_incomplete_byte_array(sym))                           \  /* only extern char sym[] */
                __unsafe_runtime_address();                             \
        RELOC_HIDE(__addrof_sym, global_variable_offset());             \  /* hide from the optimizer */
})

#define RELOC_HIDE(ptr, off)                                                \
({                                                                          \
        unsigned long __ptr;                                                \
        __asm__ ("" : "=r"(__ptr) : "0"(ptr));                              \
        (typeof(ptr)) (__ptr + (off));                                      \
})
```

ℹ️ *a simple compiler barrier() with "memory" clobber is not immune to compiler proving that &sym fits in a register and as such is unaffected by the memory clobber!*

* -> 

(and how to shoot yourself in the foot dereferencing pointers)

Accessing the zero page

- In C, dereferencing a null pointer is undefined behavior.
- All barebox-supported platforms have `(uintptr_t)NULL == 0`
  - GCC can conclude that code following a dereference of address 0 is unreachable.
- But address 0 frequently has valid memory at it, e.g.
  - MaskROM
  - Start of DRAM

Hiding pointers from the optimizer

```
/* Make the optimizer believe the variable can be manipulated arbitrarily. */
#define OPTIMIZER_HIDE_VAR(var) \
        __asm__ ("" : "=r" (var) : "0" (var))
```

```
int imx8mp_romapi_load_image(void *bl33)
{
        struct rom_api *rom_api = (void *)0x980;

        OPTIMIZER_HIDE_VAR(rom_api);

        /* calls function pointers in page 0 */
        return imx_romapi_load_image(rom_api, bl33);
}
```

Linux has a better name for it: `absolute_pointer()`.

`readl()` / `writel()` on 64-bit ARM

```
static __always_inline void ___raw_writel(u32 val, volatile void __iomem *addr)
{
        volatile u32 __iomem *ptr = addr;
        asm volatile("str %w0, %1" : : "rZ" (val), "Qo" (*ptr));
}

static __always_inline u32 ___raw_readl(const volatile void __iomem *addr)
{
        u32 val;
        asm volatile("ldr %w0, [%1]" : "=r" (val) : "r" (addr));
        return val;
}
```

Device memory already guarantees access size and order,
so why not `*(volatile u32 *)addr`?

Because a hypervisor emulating the access needs to know *which* instruction caused it.
Linux KVM **doesn't** decode all the different instructions a compiler could emit for AArch64 memory accesses

Responsible Fun

(without commandeering nasal demons)

KASAN for DMA buffer ownership

`CONFIG_DMA_API_DEBUG` checked function calls, but not memory accesses.
(K)ASAN's job *is* checking memory accesses...

```
#define KASAN_DMA_DEV_MAPPED    0xF8  /* DMA-mapped buffer currently owned by device */
```

```
        /* dma_map / dma_sync_for_device: */
        kasan_poison_shadow(addr, DMA_KASAN_ALIGN(size), KASAN_DMA_DEV_MAPPED);
        /* dma_sync_for_cpu / dma_unmap: */
        kasan_unpoison_shadow(cpu_addr, kasan_size);
```

Poison memory while the device owns it

- Unsafe CPU access to buffer reports `BUG: KASAN:` with a backtrace.

Green Threads

All barebox architectures implement standard C's

```
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

... and also extend them with `initjmp`:

```
int initjmp(jmp_buf jmp, void __noreturn (*func)(void), void *stack_top);
```

```
ENTRY(initjmp)                          /* arch/arm/lib64/setjmp.S, complete */
        str  x2, [x0, #96] /* stack pointer */
        str  x1, [x0, #88] /* return address */
        mov  x0, #0
        ret
ENDPROC(initjmp)
```

- Used to mimic Linux kernel kthreads; simplifies driver porting
- Co-operative multi-tasking: scheduled in delay loops
  - 🔗 bareDOOM renders DOOM while waiting

barebox proper as proper ELF

- PBL is nowadays an ELF loader
- barebox segment laid out for direct execution, then compressed
- Flexibility greatly simplifies the memory management code
  - Multiple text, data, rodata segments without detriment to W^X

```
PHDRS
{
        text PT_LOAD FLAGS(5);          /* PF_R | PF_X */
        rodata PT_LOAD FLAGS(4);        /* PF_R */
        dynamic PT_DYNAMIC FLAGS(4);    /* PF_R */
#ifdef CONFIG_EFI_RUNTIME
        efirt_text PT_LOAD FLAGS(5);    /* PF_R | PF_X */
        efirt_data PT_LOAD FLAGS(6);    /* PF_R | PF_W */
#endif
        data PT_LOAD FLAGS(6);          /* PF_R | PF_W */
}
```

From arch/arm/lib64/barebox.lds.S

W^X from ELF segment flags

```
static unsigned int elf_flags_to_mmu_flags(u32 p_flags)      /* pbl/mmu.c */
{
        if (readable && writable)
                return MAP_CACHED;          /* data, bss */
        else if (readable && executable)
                return MAP_CODE;            /* text */
        else if (readable)
                return MAP_CACHED_RO;       /* rodata */
```

The PBL decompresses, relocates in place, programs the MMU from `p_flags` and jumps to `e_entry`.

No stack setup or relocation code *at all* in barebox proper!

Future Outlook

[WIP] Declarative entry points

- Relocations and stack setup can't be omitted from the prebootloader,
  but can we make it statically configurable from C?
  - `struct pbl_board_info` instead of only a stack pointer (`ENTRY_FUNCTION_WITHSTACK`)
  - Per-architecture init code written once in position-independent Assembly
  - Sets up stack, relocate and only then call board's C entry point

```
struct pbl_stage {
        unsigned long stacktop;         /* 0: don't touch SP */
        unsigned long relocaddr;        /* 0: don't relocate to specific addr */

        unsigned long mem_start. mem_minsz, mem_maxsz;

        void (*soc_init)(void);
        void (*entry)(unsigned long entry_data);
        void (*soc_chainload)(void *boarddata);
/* […] */ };

struct pbl_board_info {
        unsigned int soc_type;
        struct pbl_console console;
        struct pbl_stage rom_stage;          /* pre-sram */
        struct pbl_stage first_stage;        /* pre-ram  */
        struct pbl_stage second_stage;       /* some ram */
};
```

[WIP] Declarative entry points

```
#define PBL_SOC_IMX8MM_DEFAULTS                                                 \
        .console.setup = imx8mm_console_setup,                                  \
        .console.driver = &imx_uart_console,                                    \
        .first_stage = {                                                        \
                .soc_init = imx8mm_cpu_lowlevel_init,                           \
                .soc_chainload = imx8mm_load_and_start_image_via_tfa_pe,        \
        },                                                                      \
        .second_stage = {                                                       \
                .mem_start = MX8M_DDR_CSD1_BASE_ADDR,                           \
                .mem_minsz = SZ_2G + SZ_1G,                                     \
                .soc_init = imx8mm_cpu_lowlevel_init,                           \
                .soc_chainload = imx8mm_barebox_entry,                          \
        }
```

```
ENTRY_POINT(IMX8MM, imx8mm_evk_bdinfo)
        .console.arg = IOMEM(MX8M_UART2_BASE_ADDR),
        .first_stage.entry = first_stage_entry,
        .second_stage.entry = second_stage_entry,
ENTRY_POINT_END
```

- The board overrides SoC defaults with a second designated initializer.
- Remove Boilerplate without sacrificing flexibility. *C remains in the driver seat.*
- RFC under way, hopefully this month on 🔗 https://lore.kernel.org/barebox

Related commits to what we talked about

```
a81ec0225f5a ("ARM: Add relocatable binary support")                                  2012
74d44e7b2a74 ("ARM: provide accessor functions for linker variables")                 2012
cc27564e4c05 ("ARM: Add assembler function to get runtime offset")                    2012
d0c482359f89 ("Build with -fno-delete-null-pointer-checks")                           2015
b42a3e2f68f9 ("ARM: Allow to mask data aborts")                                       2015
47ea1f6b6df9 ("ARM: move away from ld_var")                                           2018
f9fc8254b23c ("ARM: Mark SP as being clobbered in arm_setup_stack()")                 2018
5f04e5e03e94 ("ARM: aarch64: Fix get_runtime_offset after relocation")                2019
1cd4bfe5f6a1 ("ARM: Fix global_variable_offset() for aarch64")                        2019
1d9cc19aa742 ("ARM: aarch64: Re-implement most of barebox_arm_entry() in assembly") 2019
b0348d677bb4 ("ARM: Compile with -fPIE")                                              2019
92e123a3d66b ("ARM: mmu64: allow to disable null pointer trap on zero page")         2020
64242776de86 ("common: introduce bthreads, co-operative barebox threads")            2021
fbcb07e192b5 ("usb: dwc3: Use _io functions on dma coherent memory")                  2021
880c9803b95a ("ARM: implement ENTRY_FUNCTION_WITHSTACK")                              2022
f36c980fd6e5 ("ARM: cpu: add compiler barrier around unrelocated access")             2022
e9c90d6afb53 ("include: asm-generic: reloc: implement runtime_address()")             2022
2d68037eb90a ("ARM64: asm: rewrite ENTRY_FUNCTION(_WITHSTACK) fully in assembly")     2022
6ccdd298ab69 ("ARM: i.MX: romapi: Hide using NULL pointers from gcc")                 2022
77b1f08efbb0 ("dma: debug: poison DMA buffers with KASAN while owned by device")      2024
ad6062f654a4 ("ARM64: io: implement I/O accessors in assembly")                       2024
c1395964b07f ("amba: support masking data abort during identification")               2025
09ff2fb801ae ("ARM64: mmu: implement mmu_disable completely in assembly")             2025
dbbca16895f5 ("ARM: link ELF image into PBL")                                         2026
43805e60fe38 ("ARM: add RELR relocation packing support")                             2026
```

No time for these, but interesting nonetheless

```
78104ae181f7 ("arm: reimplement startup code in C")                                  2010
b936cd4d6877 ("ARM: mark 'lr' as clobbered by inline assembler")                     2012
8b99fe895614 ("ARM omap3: reimplement setup_auxcr in pure asm")                      2012
b08e08506b97 ("ARN: fixup vector addresses for relocatable binaries")                2012
3fe6f23b76a4 ("arm: fix memset-related crashes caused by recent GCC (4.7.2) optimizations") 2013
b8b0a7f19956 ("ARM: start: Fix code reordering problem")                             2015
2b53005fe1ae ("ARM: do not ignore data aborts initially")                            2015
80c55cb48375 ("pbl: console: Let console pointer survive BSS clearing")              2016
e6b0633b51f9 ("ARM: i.MX27: Fix NAND boot with newer gcc")                           2018
0e284efa85a8 ("ARM: access __boot_cpu_mode with a function")                        2019
29974c5147a2 ("sandbox: compile with symbol -fvisibility=hidden")                   2019
1e4a948673d7 ("command: Use array of pointers to commands")                          2019
dd96bc73d798 ("ARM: qemu-virt64: convert to assembly entry")                         2020
b51b15ba1738 ("RISC-V: board-dt-2nd: move low level init into nonnaked function")   2021
ce7692081fb5 ("ARM: asm: fix miscompilation of 32-bit ENTRY_FUNCTION_WITHSTACK")     2022
631529d766a1 ("ARM: cpu: don't clobber sp when booted in HYP mode")                  2022
9f3477b0e9b3 ("Kbuild: disable ARM64 pointer authentication")                        2022
2401db9c04bb ("lib: add stackprotector support")                                     2023
34b36519d753 ("console: pbl: correctly handle relocate_to_adr after pbl_set_putc")   2023
c35c352679b7 ("compiler: define __ll_elem for linker list elements")                 2024
cf7081c0771f ("ARM64: board-dt-2nd: grow stack down from start of binary")           2024
133903c41840 ("mtd: cfi-flash: use I/O accessors for reads/writes of MMIO regions")  2024
b8454cae3b1e ("ARM64: mmu: flush cacheable regions prior to remapping")              2024
3ebd05809a49 ("virtio: don't use DMA API unless required")                           2024
2aba0017c95a ("sandbox: switch to using PBL")                                        2025
aeffa0386702 ("ARM: drop arm_fixup_vectors()")                                       2026
```

Thanks

Questions?

Web demo

On the web: barebox.org/demo
ML: barebox@lists.infradead.org
Archive: lore.kernel.org/barebox
Github: github.com/barebox
Mastodon: @barebox@fosstodon.org
Matrix: #barebox:matrix.org