64 bit Bare Metal Programming on RPI-3

Tristan Gingold
gingold@ada.core.com
What is Bare Metal?

- No box
What is Bare Metal?

Your application is the OS

No Operating System
Why Bare Board?

Not enough ressources for an OS
Why Bare Board?

It’s fun
(YMMV)
Why Bare Board?

To learn low-level stuff
Why Raspberry PI-3?

It’s popular:
- Many tutorials (like [github.com/dwelch67/raspberrypi.git](https://github.com/dwelch67/raspberrypi.git))
- It’s safe (you cannot brick it)
Why Raspberry Pi-3? But...

It’s poorly documented:
• It’s a Broadcom SOC
• Data sheet of BCM2835 is available
  • But it’s Raspberry Pi 1
  • It’s incomplete (watchdog ?)
• Differences between Pi 1 and Pi 2 are (partially) documented
• What about BCM2837? Wifi? Bluetooth?
• Only 1 page schematic of Pi 3 (IO)
• GPU is partially documented
Why Raspberry Pi-3 ? But…

This is the Broadcom chip used in the Raspberry Pi 3.

It has an ARMv8 CPU.

Also see the Raspberry Pi 2's chip BCM2836 and the Raspberry Pi 1's chip BCM2835.
Raspberry PI-3 Platform

PI-1: ARM1176JZF

PI-2: 4 * Cortex A7

PI-3: 4 * Cortex A53 (Aarch-64)
Raspberry Pi Boot (1/2)

1. VideoCore GPU boots, Cortex cores are off
2. GPU initialise HW, load config and ELF file
Raspberry PI Boot (2/2)

3. GPU starts the cores (*)
Note: Boot process is very safe - you cannot brick the board
Files on the SD Card (FAT32)

- `bootcode.bin`
  First file read by the ROM. Enable SDRAM, and load…
  Boot loader: load `start.elf`
- `start.elf`
  GPU firmware, load the other files and start the CPUs
- `config.txt`
  configuration
- `fixup.dat`
  Needed to use 1GB of memory
- `kernel7.img`
  Your bare metal application (or the Linux kernel)
  https://github.com/raspberrypi/firmware/tree/master/boot
config.txt

Start in 64 bit mode!

Load at address 0x0

Don’t write ATAGS at 0x100

arm_control=0x200
kernel_old=1
disable_commandline_tags=1

https://github.com/raspberrypi/documentation/blob/master/configuration/config-txt.md
Your First Bare Metal Program

“Hello World” on the console

You need:

• A 3.3v to serial USB converter

• A terminal emulator

• https://github.com/gingold-adacore/rpi3-fosdem17.git
Console (Mini-UART)

Serial-to-USB

GND (0V)
RPI Tx
RPI Rx
Makefile

CROSS=aarch64-elf-
CC=$(CROSS)gcc
CFLAGS=-Wall -0 -ffreestanding

HELLO_OBJCScrt0.o hello.o

all: hello.bin

hello.bin: hello.elf
  $(CROSS)objcopy -O binary $< @$

hello.elf: $(HELLO_OBJCScram.ld
  $(CROSS)ld -o @$ $(HELLO_OBJCScram.ld -Map hello.map


clean:
  rm -f $(HELLO_OBJCScrt0.o hello.bin hello.elf hello.map
Crt0

- C Run Time 0
  - Traditional name for the entry point file (before main)
  - Generally written in assembly
  - Has to initialise the board
  - Simpler on RPI as the GPU does initialisation
  - Still have to create a C friendly environment
Crt0: Setup (before calling main)

```assembly
.section .traps,"ax"
__start:
    b __start_ram1
.text
.type __start_ram1, %function
__start_ram1:
    # Read processor number, move slave processors to an infinite loop
    mrs x7, mpidr_el1
    and x7, x7, #3
    cbz x7, __start_master
0:
    wfe // Busy loop
    b 0b

__start_master:
    # Load stack pointer (on 32bit)
    adrp x2,__cpu0_stack_end
    add x2, x2, #:lo12:__cpu0_stack_end
    mov sp,x2

    # Clear BSS
    ldr w0,bss_segment + 0
    ldr w1,bss_segment + 4
0:
    cbz x1,1f
    str xzr,[x0],#8
    sub x1,x1,#1
    cbnz x1,0b
1:
    bl main /* Call the main routine */
0:
    b 0b /* Wait forever in case of exit. */
.size __start_ram1, . - __start_ram1
bss_segment:
    .word __bss_start
    .word __bss_dwords
```

- Start point (at 0x00)
- Keep only cpu #0
- Set stack pointer
- Clear bss
- Call C main

No need for more assembly
C code

- Crt0 calls main()
- You can execute C code
- But no syscalls, you have to write your own IO code
- There might be no C library (you write all the code)
- Write your own drivers
  - Essentially writing and reading words at special addresses, with side effects
- First driver on RPI3: Serial port
Main()

```c
void raw_putchar (char c)
{
    while (!(MU_LSR & 0x20))
        MU_IO = c;
}

void putchar (char c)
{
    if (c == '\n')
        raw_putchar ('\r');
    raw_putchar (c);
}

void puts (const char *s)
{
    while (*s)
        putchar (*s++);
}

int main (void)
{
    init_uart ();
    puts ("Hello world!\n");
    return 0;
}
```

- Wait until ready
- Send one byte to the UART
- Write to the TX shift register
- Handle \n
Send a string

Next slide
UART init

```c
#define IO_BASE 0x3f000000
#define GP_BASE (IO_BASE + 0x200000)
#define MU_BASE (IO_BASE + 0x215000)

#define AUX_ENB (*(volatile unsigned *)(MU_BASE + 0x04))
#define MU_IO (*(volatile unsigned *)(MU_BASE + 0x04))
#define MU_LCR (*(volatile unsigned *)(MU_BASE + 0x4c))
#define MU_LSR (*(volatile unsigned *)(MU_BASE + 0x54))
#define MU_CNTL (*(volatile unsigned *)(MU_BASE + 0x60))
#define MU_BAUD (*(volatile unsigned *)(MU_BASE + 0x68))

#define GPFSEL1 (*(volatile unsigned *)(GP_BASE + 0x04))
#define GPPUD (*(volatile unsigned *)(GP_BASE + 0x94))
#define GPPUDCLK0 (*(volatile unsigned *)(GP_BASE + 0x98))

static void
init_uart(void)
{
    int i;

    AUX_ENB |= 1;    /* Enable mini-uart */
    MU_LCR = 3;     /* 8 bit. */
    MU_BAUD = 270;  /* 115200 baud. */
    GPFSEL1 &= ~(7 << 12) | (7 << 15); /* GPIO14 & 15: alt5 */
    GPFSEL1 |= (2 << 12) | (2 << 15);

    /* Disable pull-up/down. */
    GPPUD = 0;
    for (i = 0; i < 150; i++)
        asm volatile("nop");
    GPPUDCLK0 = (2 << 14) | (2 << 15);
    for (i = 0; i < 150; i++)
        asm volatile("nop");
    GPPUDCLK0 = 0;

    MU_CNTL = 3;    /* Enable Tx and Rx. */
}```
Linker script

MEMORY
{
  SRAM (rwx) : ORIGIN = 0, LENGTH = 32M
}

SECTIONS
{
  .text :
  {
    KEEP (*(.traps))
    . = 0x1000; /* Space for command line. */
    *(.text .text.* .gnu.linkonce.t*)
  }
  .rodata : { *(.rodata .rodata.* .gnu.linkonce.r*) }
  .ARM.extab : { *(.ARM.extab* .gnu.linkonce.armextab.*) }
  PROVIDE_HIDDEN (__exidx_start = .);
  .ARM.exidx : { *(.ARM.exidx* .gnu.linkonce.armexidx.*) }
  PROVIDE_HIDDEN (__exidx_end = .);
  .data : { *(.data .data.* .gnu.linkonce.d*) }
  .bss (NOLOAD): {
    __bss_start = ALIGN(0x10);
    *(.bss .bss.*)
    *(COMMON)
    __bss_end = ALIGN(0x10);
    . = ALIGN(0x10);
    . += 0x1000;
    __cpu0_stack_end = .;
    __end = .;
  }
  __bss_dwords = (__bss_end - __bss_start) >> 3;
What next?

• Make your own program

• Write drivers
  • GPIO are very easy
  • I2C, SPI, MMC aren’t difficult
  • Video is easy too (mainly handled by the Firmware)
  • USB, Bluetooth, Wifi, Ethernet need doc

• At this point it’s like an Arduino…
Performance

- You must enable cache
  - Performances are abysmal without cache
- But IO regions must not be cacheable
  - As IO regions have side effects
- So you need to setup MMU
  - To mark IO regions as uncachable
- Static 1-1 tables are enough (and easy to generate)
SMP

• RPI-3 has 4 cortex-A53 cores

• Use multi-processors
  • All processors start
  • Use mpidr to get core number
  • Assign different stack to each processor
  • Initialise hardware only once!
Processor mode

- Cores start at EL3 (Exception Level) Secure Monitor
- Usually boot is handled by some firmware
- Need to switch to lower EL: EL1 is OS, EL2 is hypervisor
  - EL0 is not recommended (user applications)
- Per EL exceptions handlers
  - Could be used for debug (dump registers in case of crash)
- See smp/ directory in the github repo for the code
Demo: ray casting

- Written in Ada 2012
  - (Could have been guessed from the company name)
- Realtime kernel (Ada ravenscar tasking profile)
- Use 4 cores
- DMA-2D, Vsync interrupt
- No GPU uses
- ~60 fps
Demo (photo of the display)
Demo: ray casting