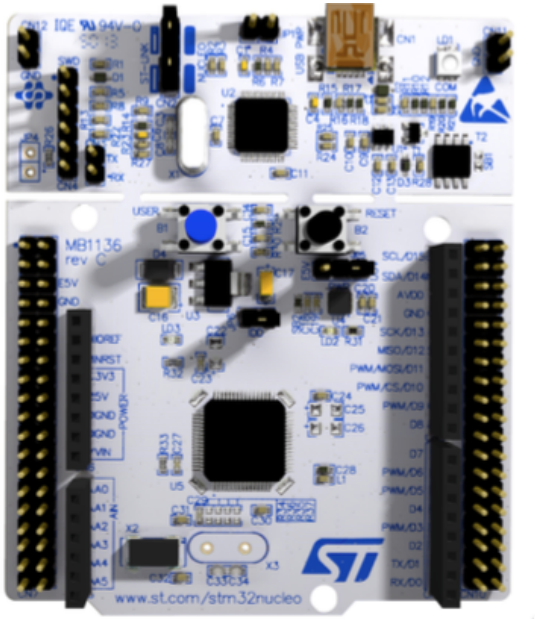
HOW DO WE PROGRAM OUR TARGETS?

- Whatever the chosen development model, executable code is written to some storage to be executed at the proper time:
  - whenever needed or on user's demand by the OS
  - at startup in the case of OS and bare metal code
- In both cases, some code must be executed at startup as even an OS must be started in some way when the target is switched on
- Targets are provided with some startup mechanism that perform a basic initialization of the machine and executes code installed in some permanent storage
- This process is called **bootstrapping** or simply **booting**

Raspberry Pi 3 B+

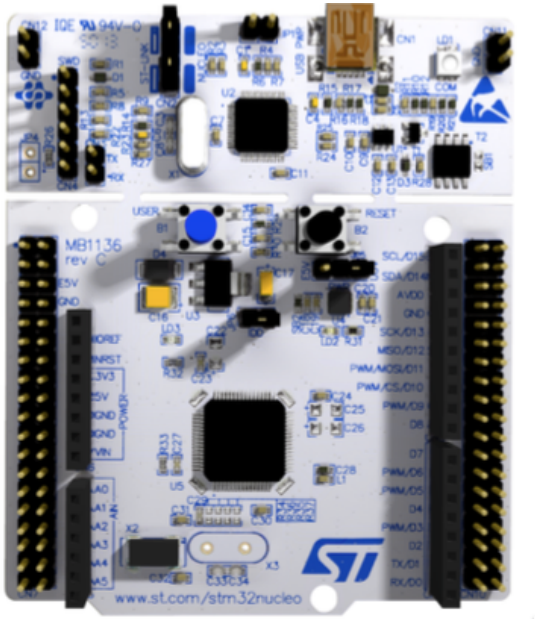


Nucleo-64 STM32F446

BOOTSTRAPPING

- At startup CPUs simply execute code at some fixed start address in the program memory
- Some permanent program storage, usually some Flash RAM, may be included in the system architecture and mapped at the CPU start address (e.g. UEFI, STM32F446)
  - a hardware mechanism is used to copy executable code to the storage from outside (**programming** or *flashing*)

Raspberry Pi 3 B+

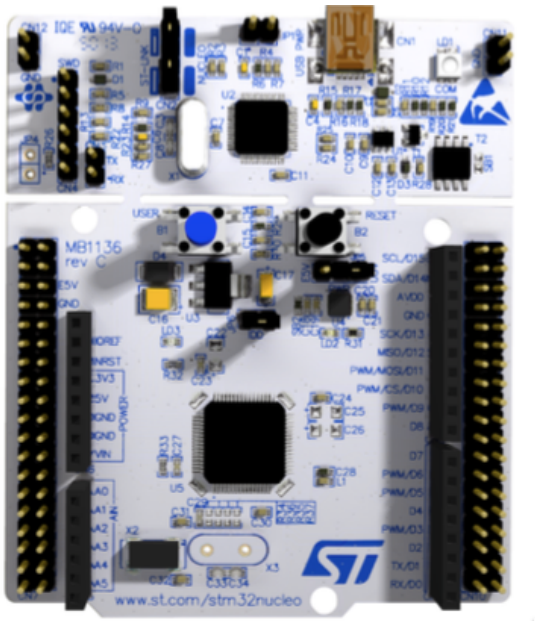


Nucleo-64 STM32F446

BOOTSTRAPPING

- If program memory is not permanent (e.g. Raspberry Pi)
  - some mechanism must load the executable code from a permanent storage to program memory at the start address
    - the permanent storage can be either fixed or removable
      - The latter (e.g. a microSD) can be simply extracted from the target, reprogrammed on the development machine and reinserted into the target.
      - If the storage is fixed, hardware programming is again needed

Raspberry Pi 3 B+



Nucleo-64 STM32F446

BOOTSTRAP AND BOOTLOADERS

- Besides internal storage, it may be more convenient to load startup code from external storage or even other machines, for instance those used in developing software for the target, through some network connection
- Bootstrapping is most of the time performed by some **bootloader** code installed in the target storage
- There exist many types of bootloaders
- Often multi-stage bootloaders are used

Raspberry Pi 3 B+

Nucleo-64 STM32F446

BOOTSTRAPPING AND BOOTLOADERS

- Special storage can be reserved to bootloader code by the manufacturer
- SoCs are often provided with a pre-installed bootloader that eases programming the target through USB, network, Over-the-Air (OTA) upgrading
  - Recovery modes (e.g. Device Firmware Update or DFU) are based on a boot loader able to program the target so that it can be restored to the factory state in case the user software (or even the OS) gets corrupted
  - Other debugging and diagnostic modes can be included in the bootloader

Raspberry Pi 3 B+

Nucleo-64 STM32F446

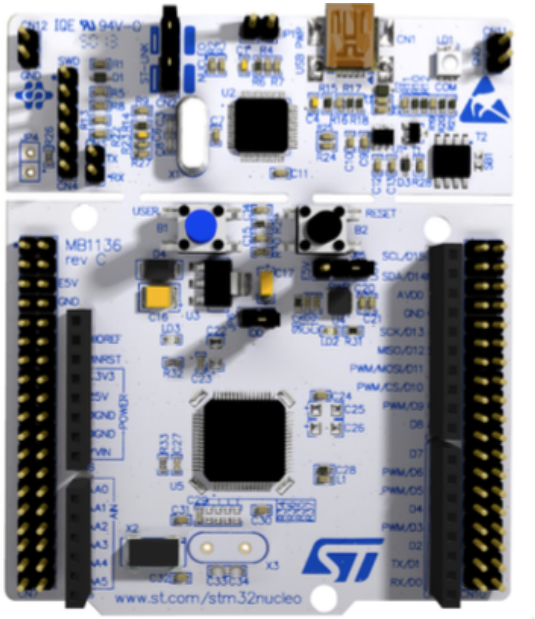
BOOTSTRAPPING AND BOOTLOADERS

- In most cases, as any other on-board stored code, even bootloaders can be upgraded by code running on the target
- With a separate bootloader storage if an issue happens during user code upgrade (e.g. power outage) the procedure can simply be repeated
- But what if an issue happens during a bootloader upgrade?
  - The target becomes a useless weight (a brick, so it is said to be *bricked*)

PROGRAMMING, BOOTSTRAPPING AND DEBUGGING

- Some hardware interfaces support programming (including bootloaders) and bootstrapping:
  - JTAG (Joint **Test** Action Group)
  - SWD (Serial Wire **Debug**)
- and also verifying code at runtime (**debugging**)
- and recovering most bricked devices (if one can find these, sometimes concealed, interfaces)

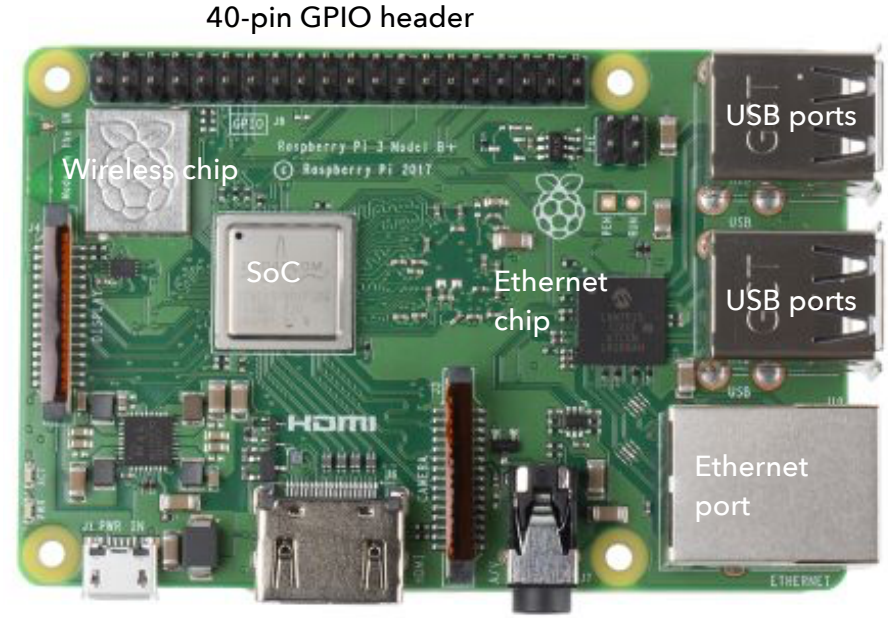
Raspberry Pi 3 B+

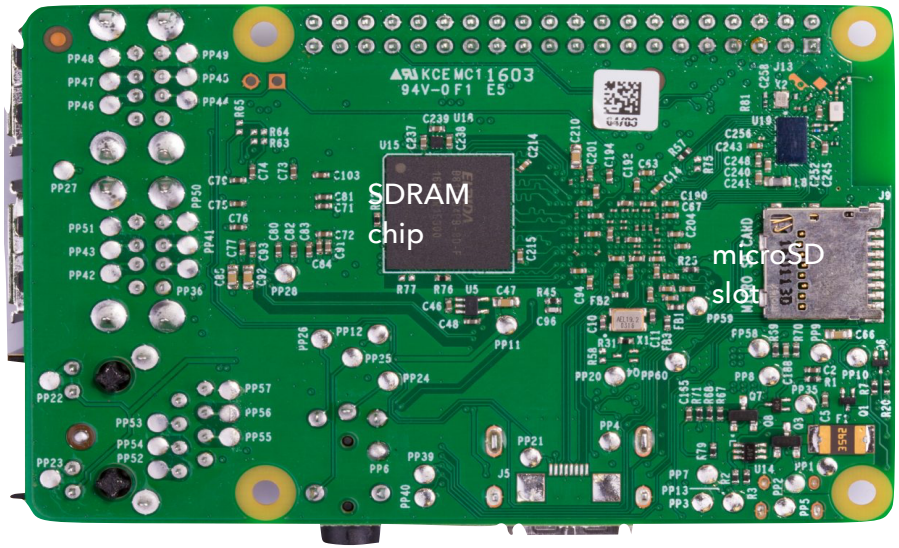


Nucleo-64 STM32F446

HARDWARE SPECS:

- SoC: Broadcom BCM2837B0 quad-core A54 (ARMv8) 64-bit @ 1.4GHz
  - GPU: Broadcom VideoCore IV
- On-board Components
  - 1GB LPDDR2 SDRAM
  - Gigabit Ethernet (via USB channel)
  - Wireless networking: 2.4GHz and 5GHz 802.11b/g/n/ac wireless LAN, Bluetooth 4.2, Bluetooth Low Energy (BLE)
  - Storage: microSD
  - One CPU controllable green LED
  - Ports
    - HDMI, 3.5 mm analog audio-video jack
    - 40-pin GPIO header
    - 4x USB 2.0
    - Camera Serial Interface (CSI) and Display Serial Interface (DSI)

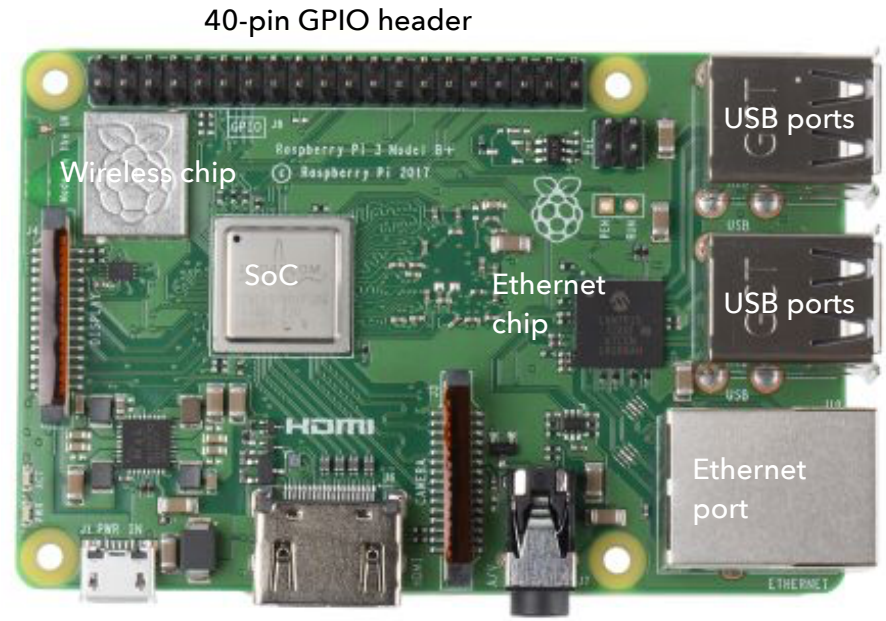


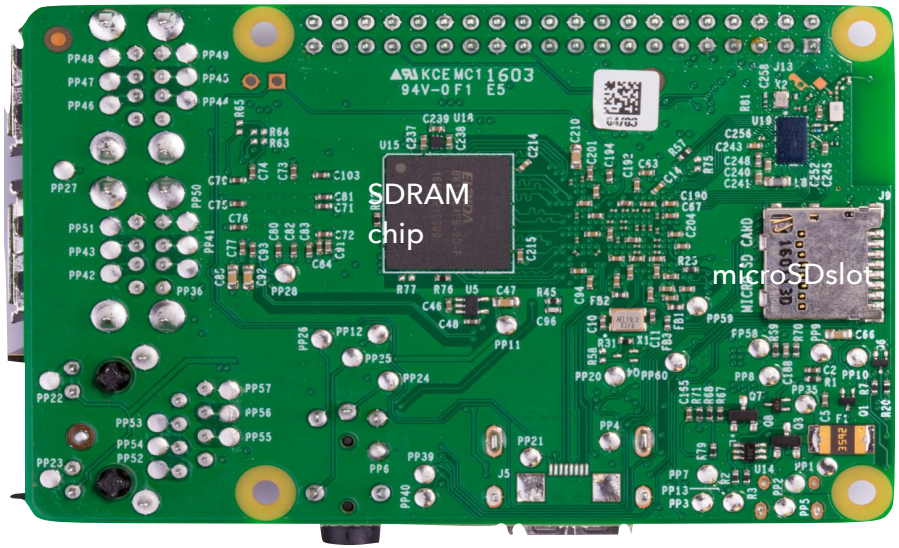


Raspberry Pi 3 B+

HARDWARE SPECS:

- The quad-core 64-bit CPU in the Raspberry PI 3 B+ (and in the Pi 2 too) supports all the ARM ISA variants:
  - aarch64
  - aarch32
  - Thumb

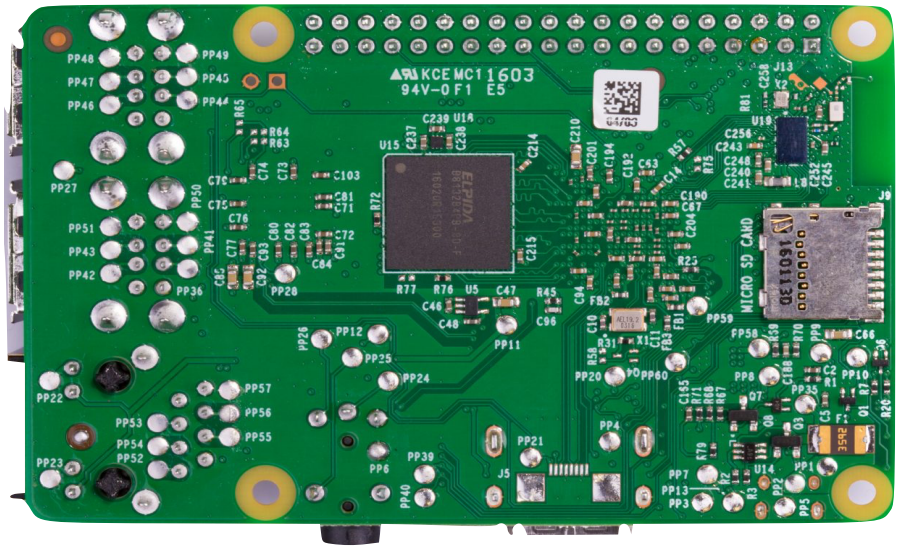




Raspberry Pi 3 B+

NATIVE COMPILATION

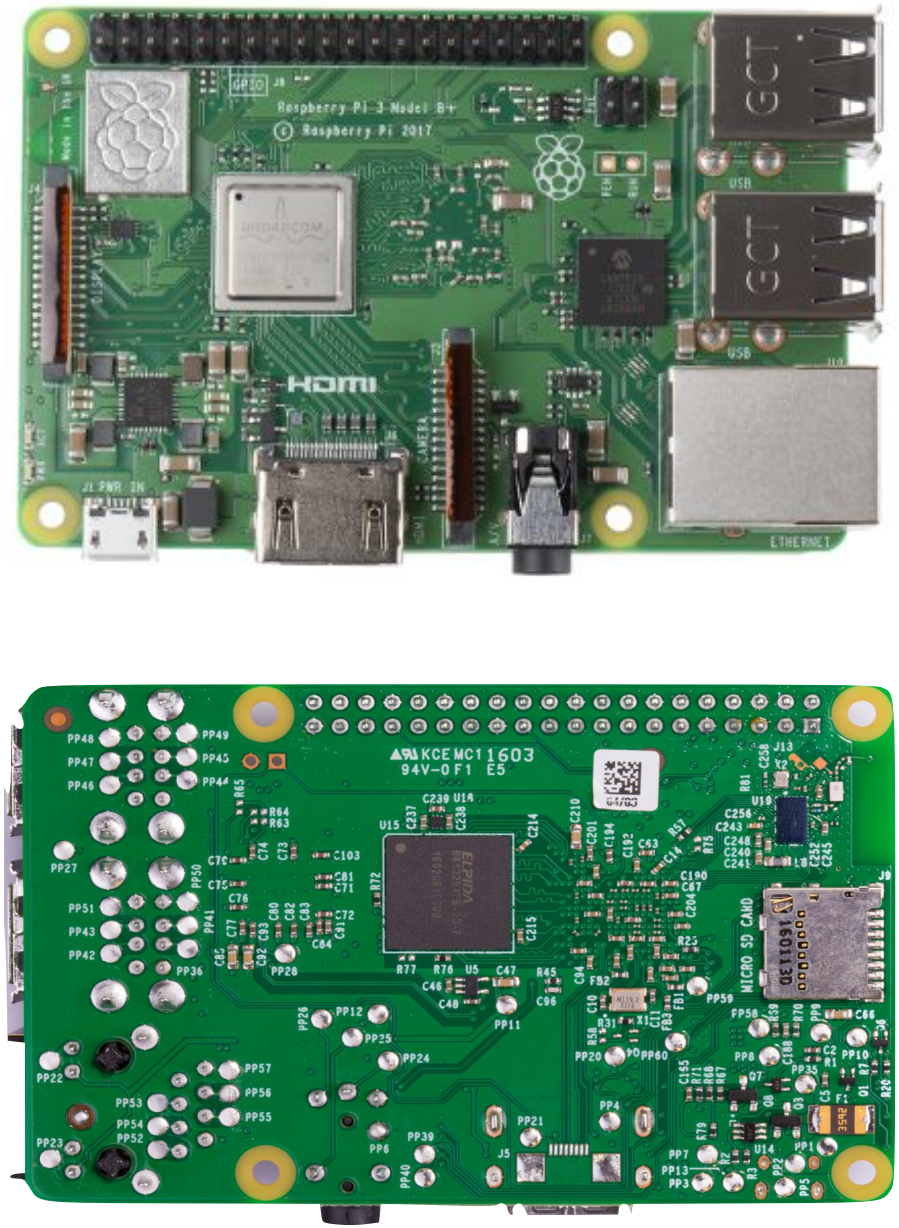
- This system is powerful enough to run full OSs, such as:
  - Linux distributions (e.g. Raspbian)
  - RISCOS
  - Windows
  - ...
- along with complete development environments (e.g. the GNU toolchain in Raspbian)



Raspberry Pi 3 B+

CROSS COMPILATION

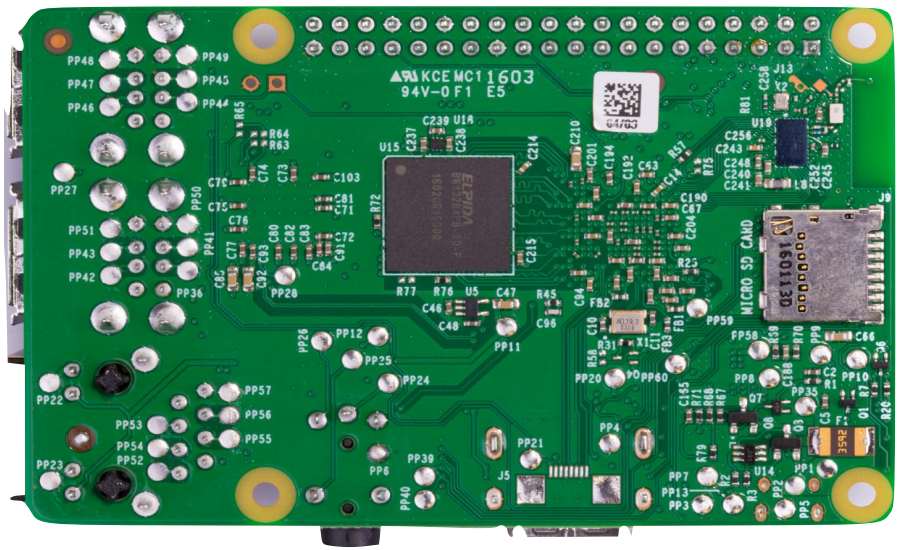
- The GNU ARM Embedded Toolchain (gcc-arm-none-eabi-*) is used on a development machine (even another Raspberry Pi)
  - https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads



Raspberry Pi 3 B+

BOOTSTRAPPING

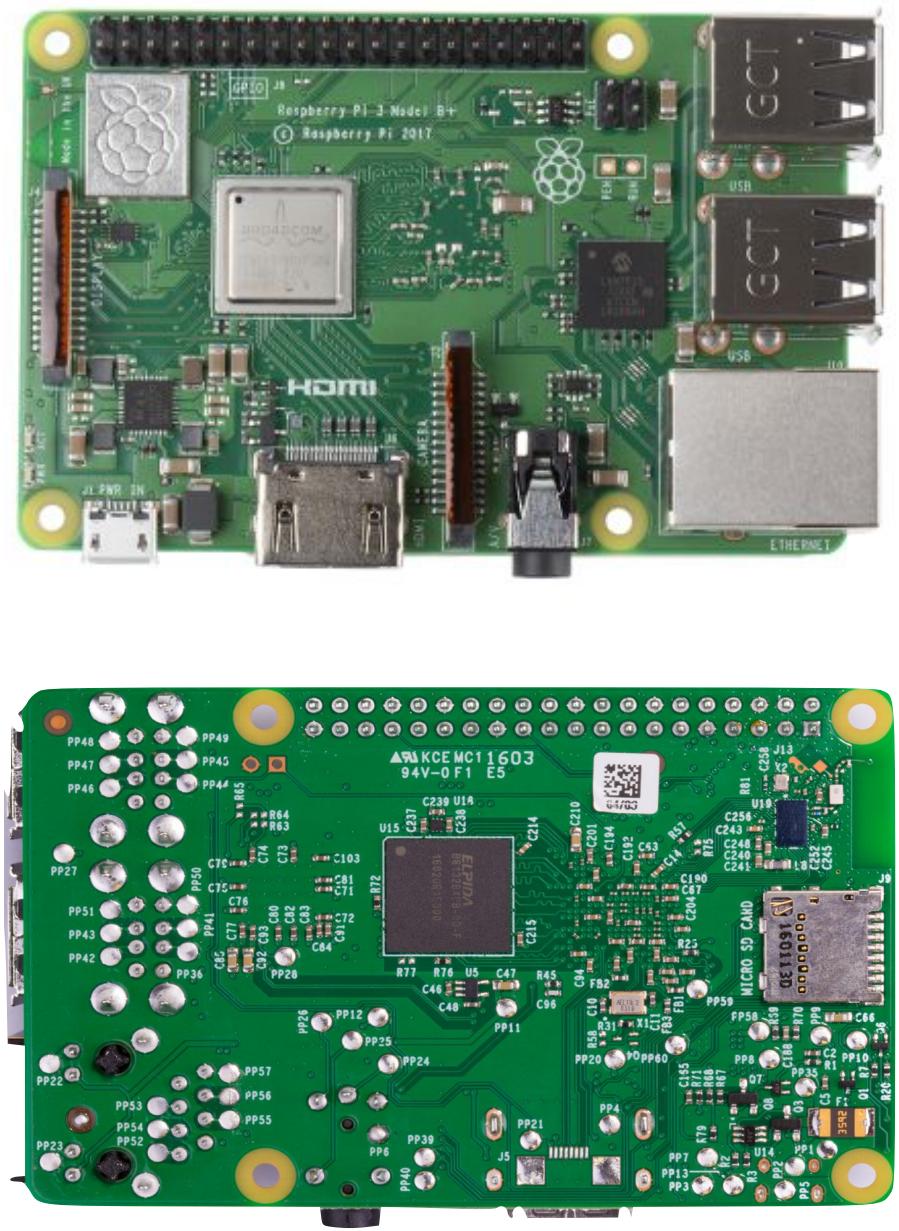
- Bootstrapping is handled by the VC (Broadcom VideoCore IV), a proprietary part of the system that also includes a GPU and video signal generation circuitry
- The VC is able to look for files in the first partition (boot partition) of the microSD if formatted as VFAT (FAT32)



Raspberry Pi 3 B+

BOOTSTRAPPING

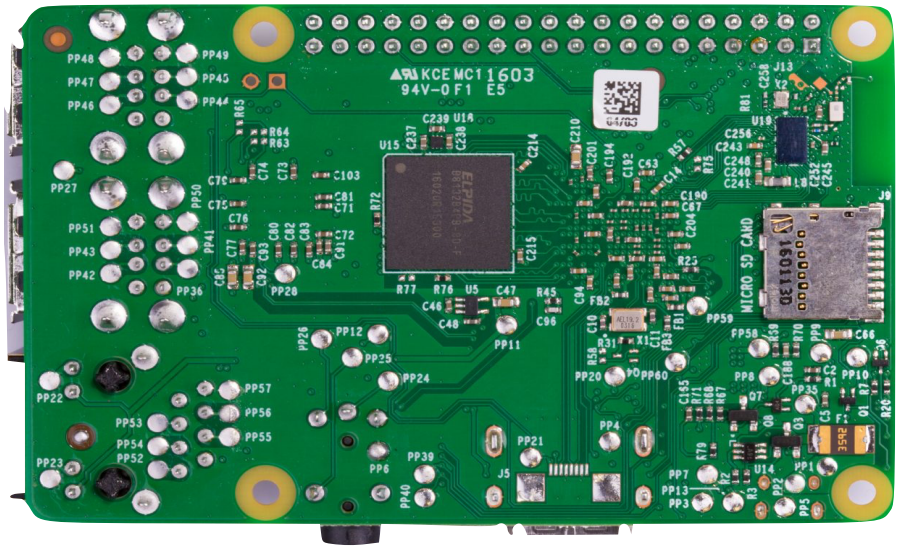
- At startup, before the ARM CPU starts running, the VC bootstraps the machine:
  - the VC loads its bootloader code from file `bootcode.bin` in the boot partition and executes it
  - the bootloader looks for the optional `config.txt` file in the boot partition. If the file exists, the bootloader examines the file reading the configuration parameters it contains; otherwise, default values for all the parameters are used
  - the bootloader then loads the second-stage bootloader from another file (default: `start.elf`) and, optionally, a matching linker file (default: `fixup.dat`). The file contains other VC code and initialization data



Raspberry Pi 3 B+

BOOTSTRAPPING

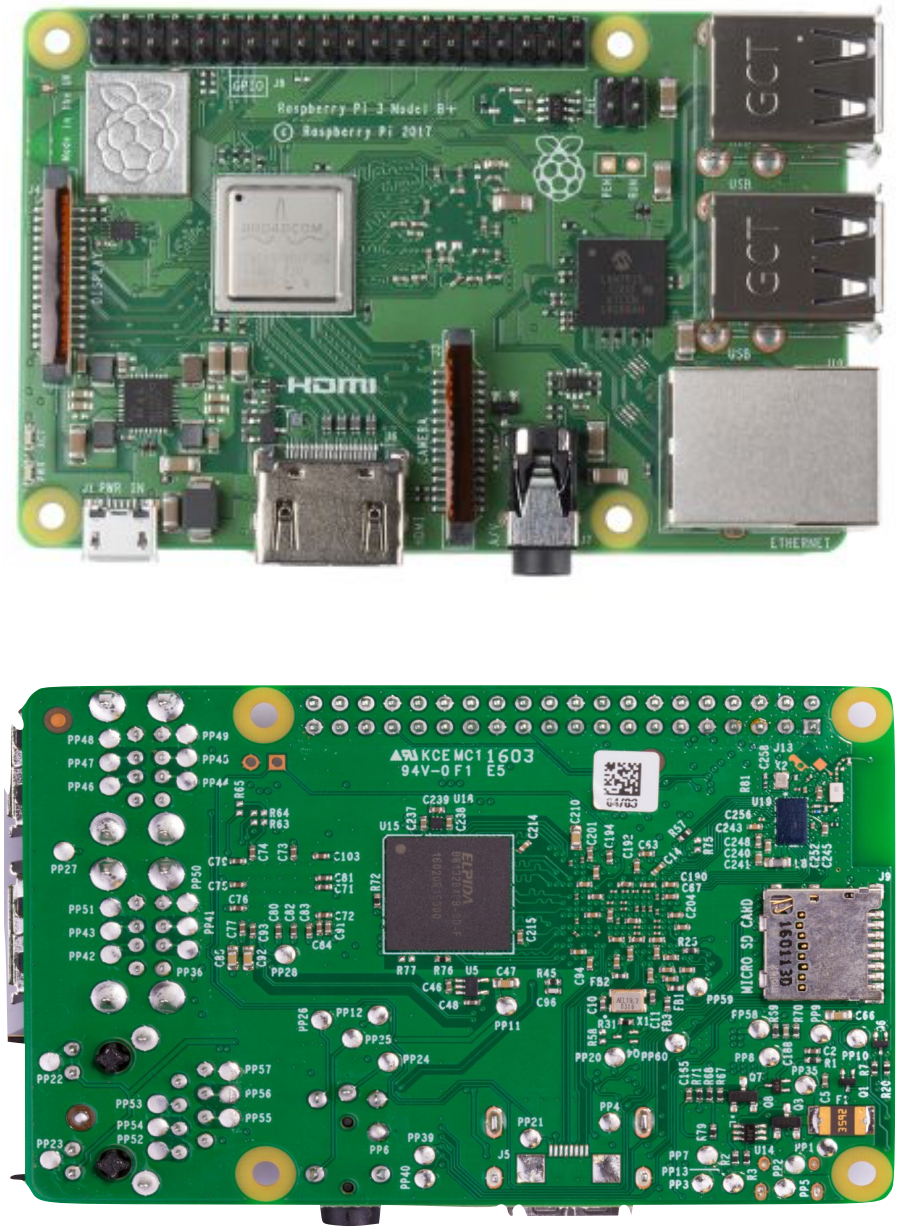
- The code in second-stage bootloader is then executed by the VC, which then looks for a binary image file (.img) containing executable code for the ARM CPU in the boot partition
  - files are searched in the following order:
    - `kernel8.img`
    - `kernel8-32.img`
    - `kernel7.img`
    - `kernel.img`
  - `kernel8.img` must contain 64-bit code, the other files must contain 32-bit code



Raspberry Pi 3 B+

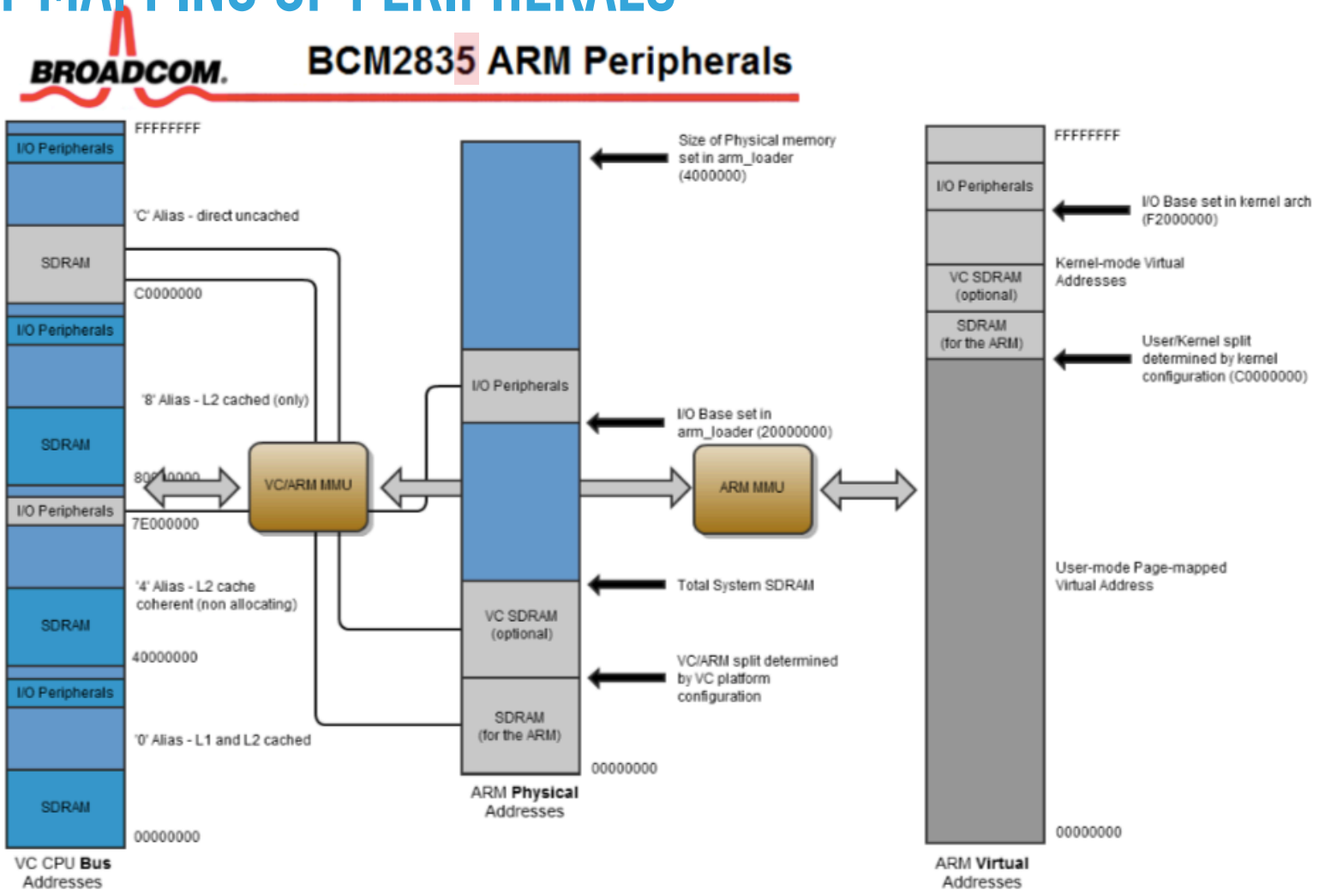
BOOTSTRAPPING

- The first found file in the search list is loaded to memory
- the load address can be specified through the `kernel_address` parameter in `config.txt`
- the load address defaults to 0x8000 (32-bit code) or 0x80000 (64-bit code)
- after loading the image, the VC initializes the ARM CPU (first core) and makes it jump to the load address; other cores are made running **w**ait **f**or **e**vent (**wfe**) idle loops
- if `kernel8.img` is found the ARM CPU is switched to 64-bit mode before executing the code



Raspberry Pi 3 B+

MEMORY MAPPING OF PERIPHERALS



06 February 2012 Broadcom Europe Ltd. 406 Science Park Milton Road Cambridge CB4 0WW Page 5

BCM2837-ARM-Peripherals.-.Revised.-.V2-1 pp. 4-7 - (BCM2837-ARM-Peripherals.-.Revised.-.V2-1.pdf)

BARE METAL PROGRAMMING

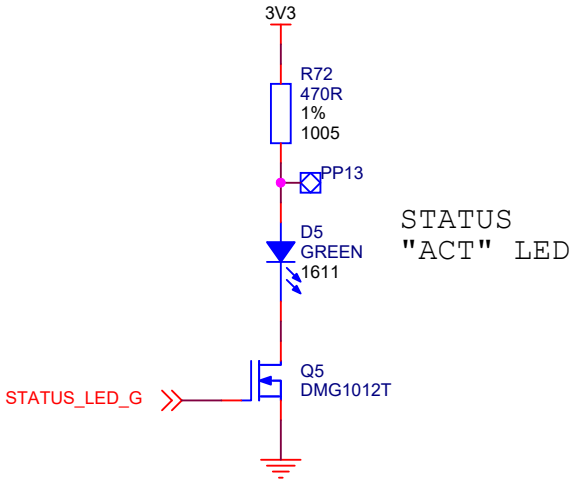
- Following the default conventions (no `config.txt`), only the following files are needed in the microSD to execute properly crafted bare metal code
- Our target code
  - `kernel7.img`
- Binary blobs from the Raspberry Pi repository
  - `bootcode.bin`
  - `start.elf`
  - `fixup.dat`

BARE METAL PROGRAMMING – ARM ASSEMBLY

- Our target code is obtained through the following steps:
  - Code is written into a source (`.s`) file
    - `vi test.s`
  - The assembly file is assembled into an Executable and Linkable Format (ELF) object file (`.o`)
    - `arm-none-eabi-as test.s -o test.o`
  - The object file is processed by the linker to resolve addresses for a fixed memory model producing a finalized ELF file
    - `arm-none-eabi-ld -T kernel7.ld test.o -o test.elf`
  - The objectdump tool can be used to obtain the finalized ML and assembly code in a listing file (`.list`)
    - `arm-none-eabi-objdump -D test.o > test.list`
  - The objectcopy tool is used to extract only the ML code from the finalized ELF file in binary format
    - `arm-none-eabi-objcopy test.elf -O binary test.bin`
  - The binary file is renamed to follow the bootstrap conventions
    - `mv test.bin kernel7.img`
- The image file can then be simply copied onto the microSD to be executed at startup

BARE METAL PROGRAMMING - ARM ASSEMBLY - EXAMPLE 01

- Let's light up the green LED




```
/* Example bm-01: LED on
 *
 * Target: Raspberry Pi 2/3
 *
 */

/* Pi 2/3
 * GPIO register addresses
 */
GPFSEL2=0x3F200008
GPSET0 =0x3F20001C

.global _start

_start:
    ldr r0,=GPFSEL2
    ldr r0,[r0]
    bic r1, r0, #0x38000000
    orr r1, r1, #0x08000000
    ldr r0,=GPFSEL2
    str r1, [r0]

    ldr r0,=GPSET0
    mov r1, #0x20000000  /* GPIO pin 29 on (LED ON) */
    str r1, [r0]

1:
    b 1b
```

ARM-Assembly/4-Bare metal/RPi3Bp/bm-01-led_on-0/led_on.s

BARE METAL PROGRAMMING – ARM ASSEMBLY – EXAMPLE 01

- The disassembly obtained with
  - `arm-none-eabi-objdump -d led_on.elf > led_on.elf.list`
- shows that the information about the fixed loading address (`0x00008000`) is contained in the ELF file and that the two register addresses (`0x3f200008`, `0x3f20001c`) were transformed by the assembler into two data words (`.word` directives) at the end of code to be read by PC-relative load instructions. Even if the information about the loading address is not copied to the binary image file (`arm-none-eabi-objcopy led_on.elf -O binary led_on.bin`), this dump is useful to see how the code will look like when loaded into memory

```
led_on.elf:     file format elf32-littlearm


Disassembly of section .text:

00008000 <_start>:
    8000:       e59f0020        ldr     r0, [pc, #32]   ; 8028 <_start+0x28>
    8004:       e5900000        ldr     r0, [r0]
    8008:       e3c0130e        bic     r1, r0, #939524096      ; 0x38000000
    800c:       e3811302        orr     r1, r1, #134217728      ; 0x8000000
    8010:       e59f0010        ldr     r0, [pc, #16]   ; 8028 <_start+0x28>
    8014:       e5801000        str     r1, [r0]
    8018:       e59f000c        ldr     r0, [pc, #12]   ; 802c <_start+0x2c>
    801c:       e3a01202        mov     r1, #536870912  ; 0x20000000
    8020:       e5801000        str     r1, [r0]
    8024:       eafffffe        b       8024 <_start+0x24>
    8028:       3f200008        .word   0x3f200008
    802c:       3f20001c        .word   0x3f20001c
```

In the ARM ISA the offset of a PC-relative address is given with respect to the second instruction following the instruction using it, e.g:

- the first constant (`0x3f200008`) is stored in the word (32-bits) at address `0x00008028`
- the first `ldr` loads its value from the address `pc+32 = pc+0x20 = 0x00008008 + 0x20 = 0x00008028`
  - `0x00008008` is the address of the second instruction after the `ldr` (`bic`)
- the second `ldr` accesses the same value using the PC-relative offset 16=`0x10` with the address `0x00008018`

ARM-Assembly/4-Bare metal/RPi3Bp/bm-01-led_on-0/led_on.elf.list