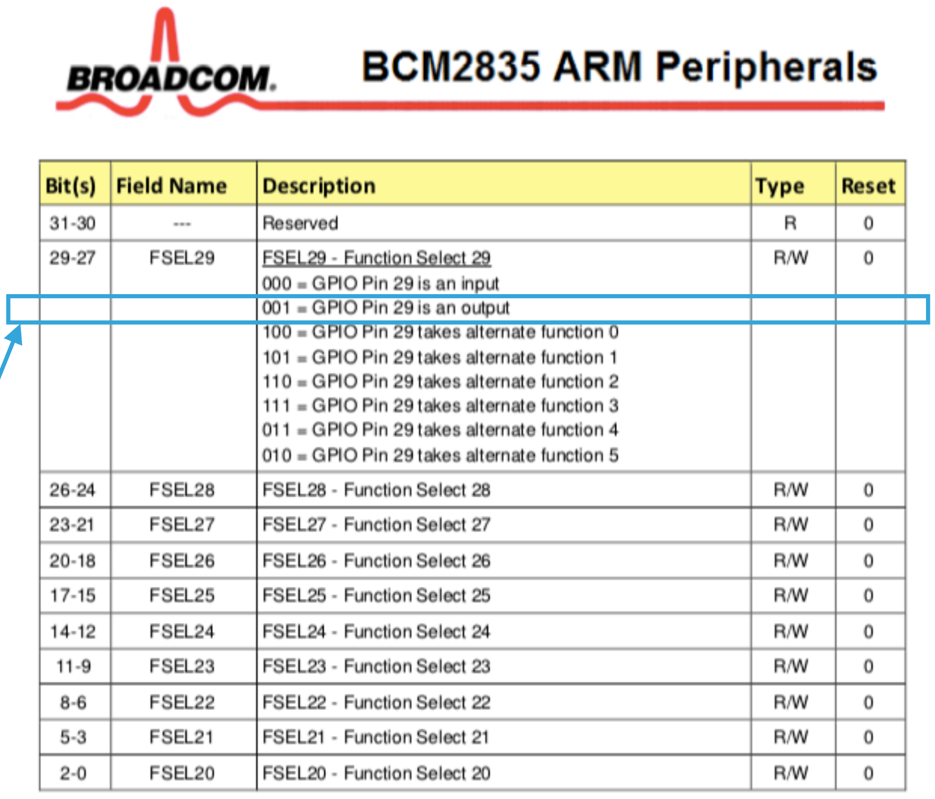
BARE METAL PROGRAMMING – ARM ASSEMBLY – EXAMPLE 01

- Two General Purpose I/O (GPIO) registers are used to light up the green LED on the board: <u>GPFSEL2</u> and GPSET0
  - GPFSEL2 at address `0x3F200008` contains three bits (29-27) controlling the function of the GPIO pin 29, which is physically connected to the LED
  - In this case the function "output" must be selected by setting bits 29-27 to `0b001`



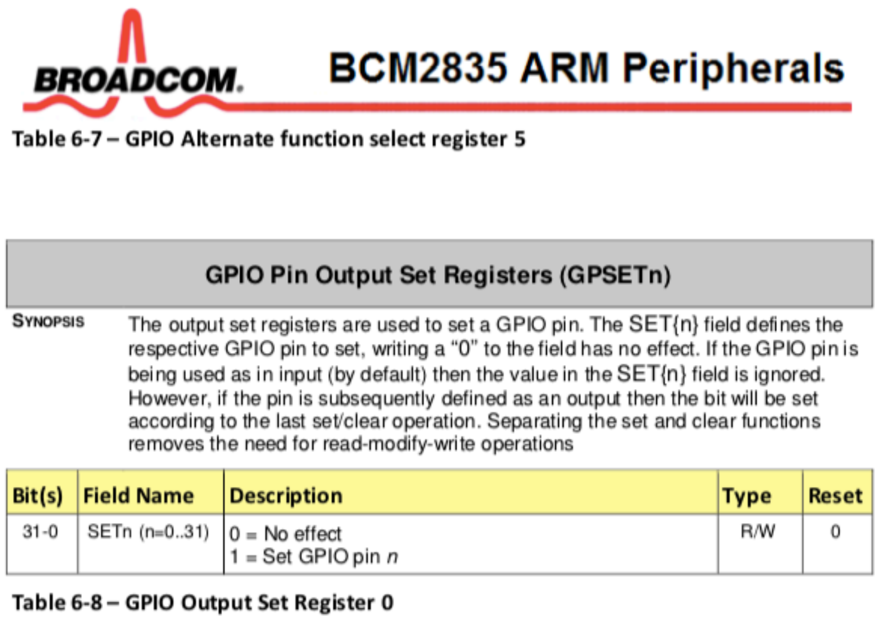
BROADCOM. BCM2835 ARM Peripherals

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL29	<u>FSEL29 - Function Select 29</u> 000 = GPIO Pin 29 is an input 001 = GPIO Pin 29 is an output 100 = GPIO Pin 29 takes alternate function 0 101 = GPIO Pin 29 takes alternate function 1 110 = GPIO Pin 29 takes alternate function 2 111 = GPIO Pin 29 takes alternate function 3 011 = GPIO Pin 29 takes alternate function 4 010 = GPIO Pin 29 takes alternate function 5	R/W	0
26-24	FSEL28	FSEL28 - Function Select 28	R/W	0
23-21	FSEL27	FSEL27 - Function Select 27	R/W	0
20-18	FSEL26	FSEL26 - Function Select 26	R/W	0
17-15	FSEL25	FSEL25 - Function Select 25	R/W	0
14-12	FSEL24	FSEL24 - Function Select 24	R/W	0
11-9	FSEL23	FSEL23 - Function Select 23	R/W	0
8-6	FSEL22	FSEL22 - Function Select 22	R/W	0
5-3	FSEL21	FSEL21 - Function Select 21	R/W	0
2-0	FSEL20	FSEL20 - Function Select 20	R/W	0

Table 6-4 – GPIO Alternate function select register 2

BARE METAL PROGRAMMING – ARM ASSEMBLY – EXAMPLE 01

- Two General Purpose I/O (GPIO) registers are used to light up the green LED on the board: GPFSEL2 and GPSET0
    - GPSET0 at address `0x3F20001C` contains the bit (#29) that is used to set the pin high so to turn the LED on



BROADCOM. BCM2835 ARM Peripherals

Table 6-7 – GPIO Alternate function select register 5

GPIO Pin Output Set Registers (GPSETn)

SYNOPSIS The output set registers are used to set a GPIO pin. The SET{n} field defines the respective GPIO pin to set, writing a "0" to the field has no effect. If the GPIO pin is being used as in input (by default) then the value in the SET{n} field is ignored. However, if the pin is subsequently defined as an output then the bit will be set according to the last set/clear operation. Separating the set and clear functions removes the need for read-modify-write operations

Bit(s)	Field Name	Description	Type	Reset
31-0	SETn (n=0..31)	0 = No effect 1 = Set GPIO pin *n*	R/W	0

Table 6-8 – GPIO Output Set Register 0

BARE METAL PROGRAMMING - ARM ASSEMBLY - EXAMPLE 01

- The `bic` and `orr` instructions are used to set the three-bit value FSEL29 to “output”
- Writing `0x20000000` (1<<29) to GPSET0 sets the output level of pin 29 to HIGH leaving all the others pin levels unchanged

```
/* 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
/* Position              7    6    5    4    3    2    1    0
   0x38000000 is: 0011 1000 0000 0000 0000 0000 0000 0000
   (hex:            0x3  0x8  0x0  0x0  0x0  0x0  0x0  0x0)

   bic (BIt Clear) performs a bitwise AND of the bits in
   r0 and the complements of the corresponding bits in
   the third operand (#0x38000000):
   r1= r0 & ~(0x38000000);
   Indicating the generic bit value with X:
   r0: XXXX XXXX XXXX XXXX XXXX XXXX XXXX XXXX
   after bic execution r1 contains:
   r1: XX00 0XXX XXXX XXXX XXXX XXXX XXXX XXXX
   after orr execution r1 contains:
   r1: XX00 1XXX XXXX XXXX XXXX XXXX XXXX XXXX
 */
    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 01-1

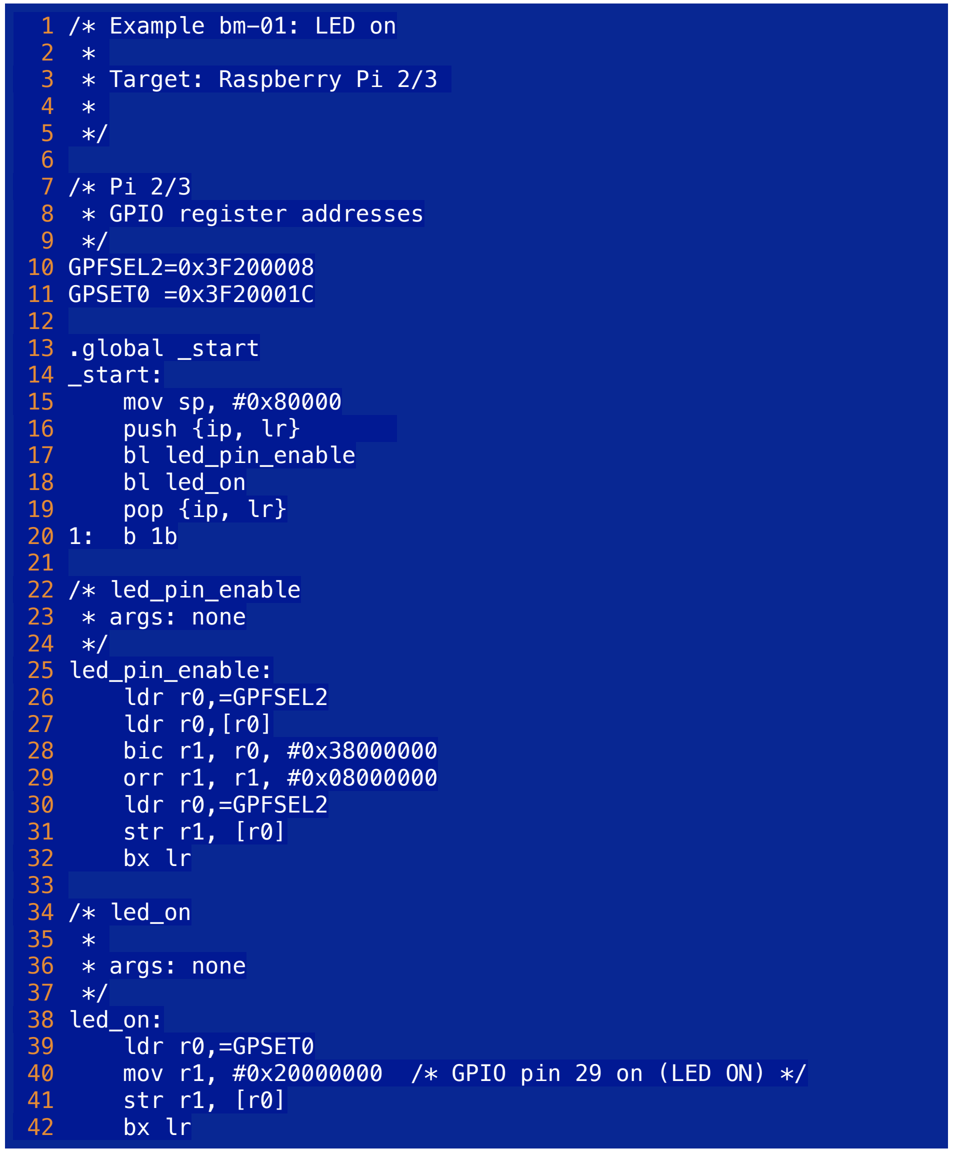
- Code can be structured using subroutines
  - the stack pointer must be initialized before being used as there is no OS doing it
    - in this case the stack base address is set to `0x80000`, far enough from the loaded code
  - preserving the link register (`push`, `pop`) is not needed in this example as the main subroutine will never return
  - output enabling code is moved to subroutine `led_pin_enable`
    - the instruction branch and link (`bl led_pin_enable`) saves the return address into `lr` and moves the address of the subroutine into `pc`
    - the instruction branch and exchange (`bx lr`) copies the value of `lr` into `pc` ending the subroutine

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

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

.global _start

_start:
    mov sp, #0x80000

    push {ip, lr}
    bl led_pin_enable

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

    pop {ip, lr}
1:  b 1b

led_pin_enable:
    ldr r0,=GPFSEL2
    ldr r0,[r0]
    bic r1, r0, #0x38000000
    orr r1, r1, #0x08000000
    ldr r0,=GPFSEL2
    str r1, [r0]
    bx lr
```

ARM-Assembly/4-Bare metal/RPi3Bp/bm-01-led_on-1/led_on.s

BARE METAL PROGRAMMING - ARM ASSEMBLY - EXAMPLE 01-2

- Code can be structured using subroutines
    - also the code that turns the LED on is moved to a subroutine (`led_on`)



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

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

.global _start
_start:
    mov sp, #0x80000
    push {ip, lr}
    bl led_pin_enable
    bl led_on
    pop {ip, lr}
1:  b 1b

/* led_pin_enable
 * args: none
 */
led_pin_enable:
    ldr r0,=GPFSEL2
    ldr r0,[r0]
    bic r1, r0, #0x38000000
    orr r1, r1, #0x08000000
    ldr r0,=GPFSEL2
    str r1, [r0]
    bx lr

/* led_on
 *
 * args: none
 */
led_on:
    ldr r0,=GPSET0
    mov r1, #0x20000000  /* GPIO pin 29 on (LED ON) */
    str r1, [r0]
    bx lr
```

ARM-Assembly/4-Bare metal/RPi3Bp/bm-01-led_on-2/led_on.s

BARE METAL PROGRAMMING - ARM ASSEMBLY - EXAMPLE 02

- Let’s make the LED blink!

BARE METAL PROGRAMMING – ARM ASSEMBLY – EXAMPLE 02

- Blinking LED
  - The pin which the LED is connected to is enabled as output by the `led_pin_enable` subroutine
  - Then, in a never ending loop,
    - the LED is switched on by calling the `led_on` subroutine
    - the delay subroutine makes the CPU wait for a time proportional to the value passed in r0
    - the LED is switched off by calling the `led_off` subroutine
    - the delay subroutine is called again with a smaller time value in `r0`

```
/* Example bm02: Blinking LED
 *
 * Target: Raspberry Pi 2/3
 *
 */

/* Pi 2/3
 * GPIO register addresses
 */
GPFSEL2=0x3F200008
GPSET0 =0x3F20001C
GPCLR0 =0x3F200028

.global _start

_start:
    mov sp, #0x80000
    push {ip, lr}

    bl led_pin_enable

1:  bl led_on
    mov r0, #0x00400000 /* duration=64*65536 - LED stays longer (8x) on than off */
    bl delay
    bl led_off
    mov r0, #0x00080000 /* duration=8*65536 */
    bl delay
    b 1b

/* led_pin_enable
 * args: none
 */
led_pin_enable:
    ldr r0,=GPFSEL2
    ldr r0,[r0]
    bic r1, r0, #0x38000000
    orr r1, r1, #0x08000000
    ldr r0,=GPFSEL2
    str r1, [r0]
    bx lr
```

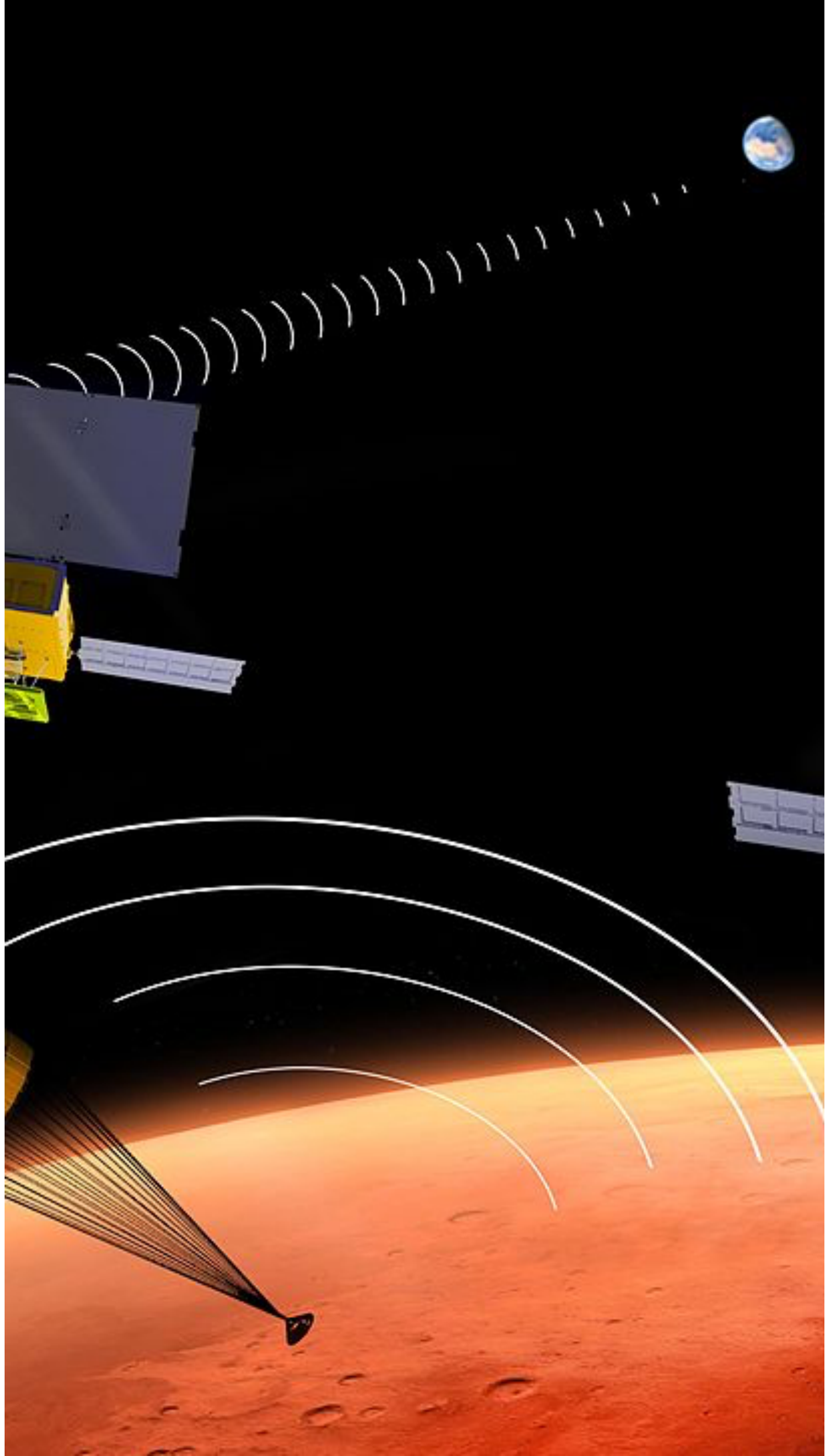
ARM-Assembly/4-Bare metal/RPi3Bp/bm-02-blinking_led-0/blinking_led.s

BARE METAL PROGRAMMING – ARM ASSEMBLY – EXAMPLE 02

- The `led_on` and `led_off` subroutines are very similar
- The same value, `0x20000000`, written to `GPSET0` turns the LED on, written to `GPCLR0` turns it off
    - This is because the bits that are set (in this case the 29th) in the immediate operand only carry the information on the position of the bits to be set or clear
    - The real bit write in the corresponding pin-level register bits is performed by the peripheral circuitry
        - A write to `GPSET0` sets the bits at the indicated positions
        - A write to `GPCLR0` clears the bits at the indicated positions
        - All the other pin-level bits are untouched
- To make the CPU wait for a given time, the delay subroutine simply executes a counting loop until the number of iterations reaches the value in `r0`

```
/* led_on
 *
 * args: none
 */
led_on:
    ldr r0,=GPSET0
    mov r1, #0x20000000  /* GPIO pin 29 on (LED ON) */
    str r1, [r0]

    bx lr

/* led_off
 *
 * args: none
 */
led_off:
    ldr r0,=GPCLR0
    mov r1, #0x20000000  /* GPIO pin 29 off (LED OFF) */
    str r1, [r0]
    bx lr

/*
 * delay
 *
 * args (r0) delay (iterations)
 */
delay:
    mov r1, #0

1:  add r1, r1, #1
    nop
    cmp r1, a1
    bne 1b

    bx lr
```

ARM-Assembly/4-Bare metal/RPi3Bp/bm-02-blinking_led-0/blinking_led.s

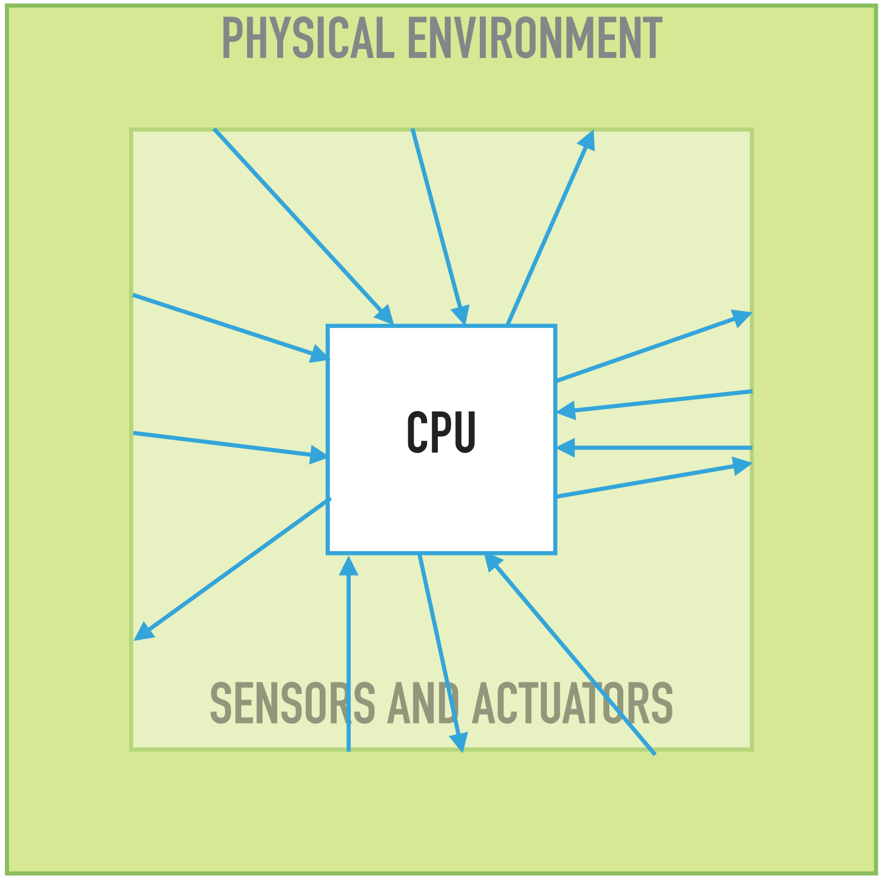


TARGETS

I/O & PERIPHERALS

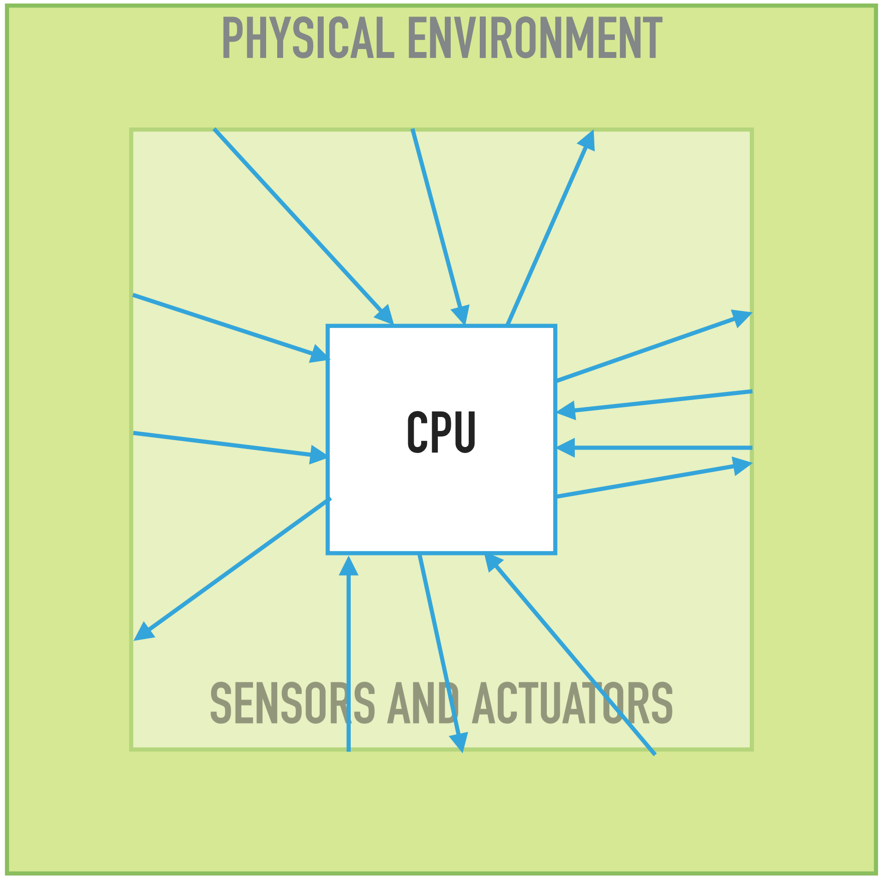
I/O SIGNALS

- The CPU in an Embedded System exchanges information with the physical environment through Input/Output (I/O) signals
- I/O signals can be either **digital** or **analog**
    - A **digital signal** carries a single logic value (0-1, SET-CLEAR, FALSE-TRUE, …) as one of two possible voltage levels called LOW and HIGH (e.g. 0V for LOW and 5V for HIGH or 0V for LOW and 3.3V for HIGH)
        - Sometimes a signal is said to be **asserted low**, that is, its logical value TRUE is associated to the LOW level and the value FALSE to the HIGH level
        - Digital signals can be grouped to form a **bus**
    - An **analog input** signal is handled by specialized circuitry that transforms it into some digital value. For example, an Analog-to-Digital Converter (ADC) is available to sample analog signal values
    - Analog output signal can be produced by Digital-to-Analog Converters (DACs) or by modulating digital outputs (e.g. sigma-delta, Pulse Width Modulation)



I/O SIGNALS

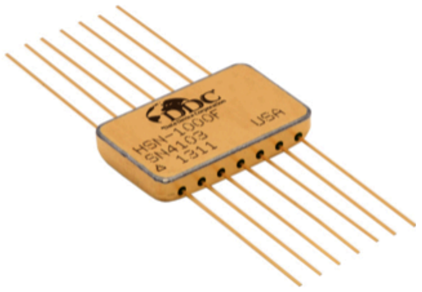
- Physical quantities (temperature, luminosity, acceleration of a body, radiation levels, rotation of an axle) in the physical environment can be measured and transformed into input signals to the CPU by **sensors**
- Some sensors can detect a change in a physical quantity and generate an input signal for this **event** to the CPU
- In most cases, event signals are **asynchronous**, that is, they can reach the CPU at any time
- I/O circuits must be able to deal with **asynchronous** signals by holding information about the event and even triggering a timely response by the CPU (e.g. by interrupts) or other circuitry (e.g. counters)
- The CPU can also act on the environment by sending signals to **actuators,** which transform signals into changes in physical quantities

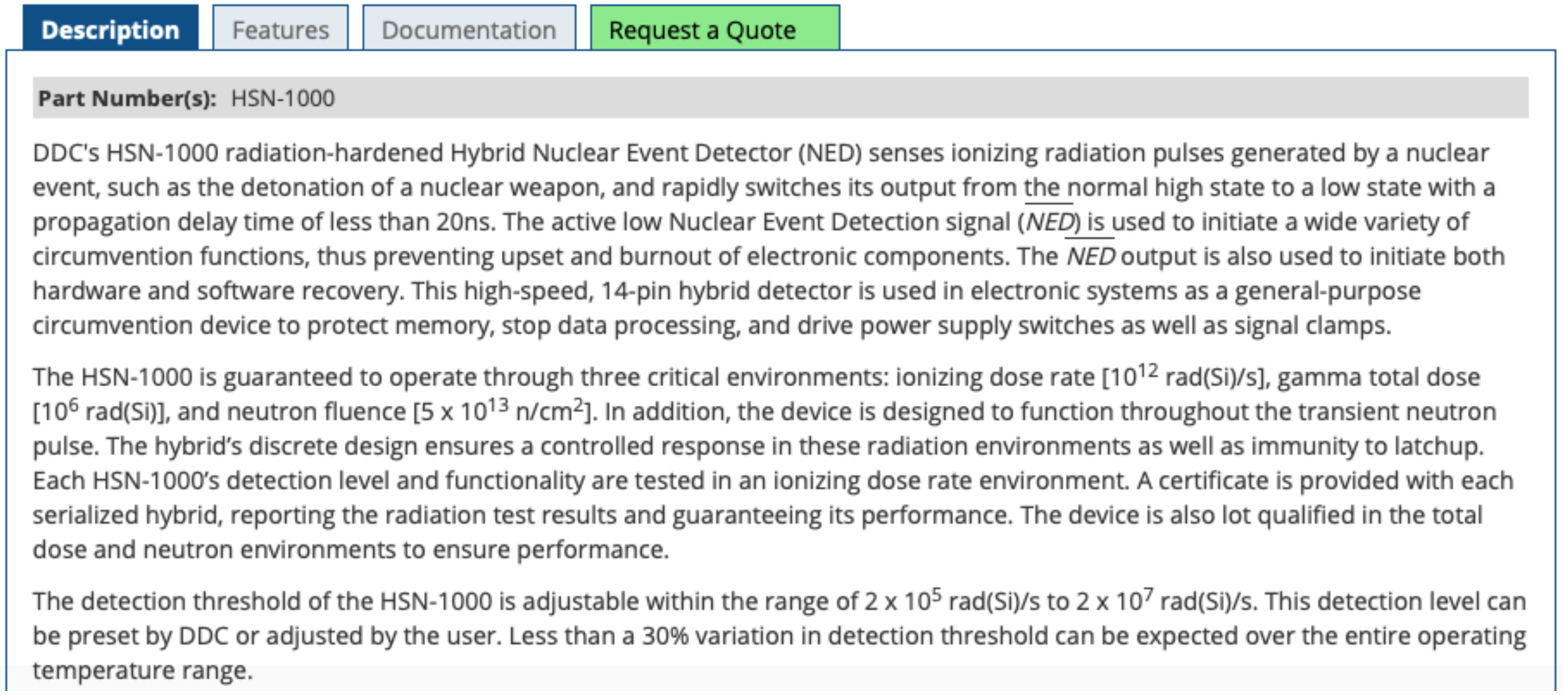


I/O SIGNALS

- There are many types of sensors to measure a wide range of **physical quantities** and **events** and send corresponding input signals to CPUs

HSN-1000 Nuclear Event Detector





Part Number(s): HSN-1000

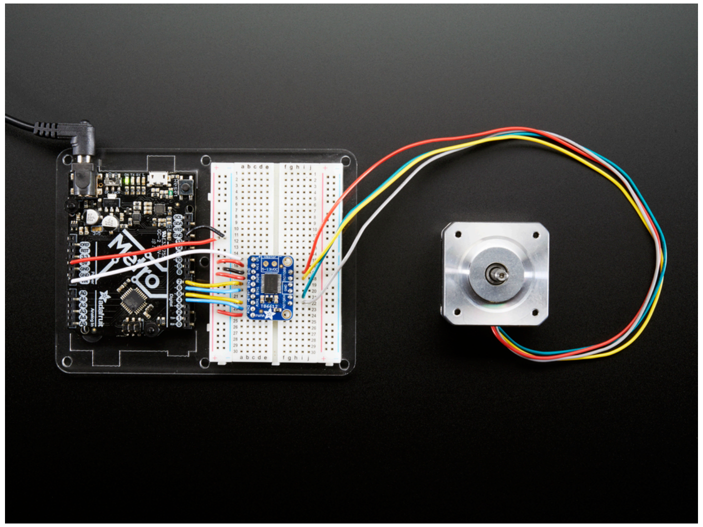
DDC's HSN-1000 radiation-hardened Hybrid Nuclear Event Detector (NED) senses ionizing radiation pulses generated by a nuclear event, such as the detonation of a nuclear weapon, and rapidly switches its output from the normal high state to a low state with a propagation delay time of less than 20ns. The active low Nuclear Event Detection signal (*NED*) is used to initiate a wide variety of circumvention functions, thus preventing upset and burnout of electronic components. The *NED* output is also used to initiate both hardware and software recovery. This high-speed, 14-pin hybrid detector is used in electronic systems as a general-purpose circumvention device to protect memory, stop data processing, and drive power supply switches as well as signal clamps.

The HSN-1000 is guaranteed to operate through three critical environments: ionizing dose rate [10^{12} rad(Si)/s], gamma total dose [10^6 rad(Si)], and neutron fluence [5×10^{13} n/cm^2]. In addition, the device is designed to function throughout the transient neutron pulse. The hybrid's discrete design ensures a controlled response in these radiation environments as well as immunity to latchup. Each HSN-1000's detection level and functionality are tested in an ionizing dose rate environment. A certificate is provided with each serialized hybrid, reporting the radiation test results and guaranteeing its performance. The device is also lot qualified in the total dose and neutron environments to ensure performance.

The detection threshold of the HSN-1000 is adjustable within the range of 2×10^5 rad(Si)/s to 2×10^7 rad(Si)/s. This detection level can be preset by DDC or adjusted by the user. Less than a 30% variation in detection threshold can be expected over the entire operating temperature range.

I/O SIGNALS

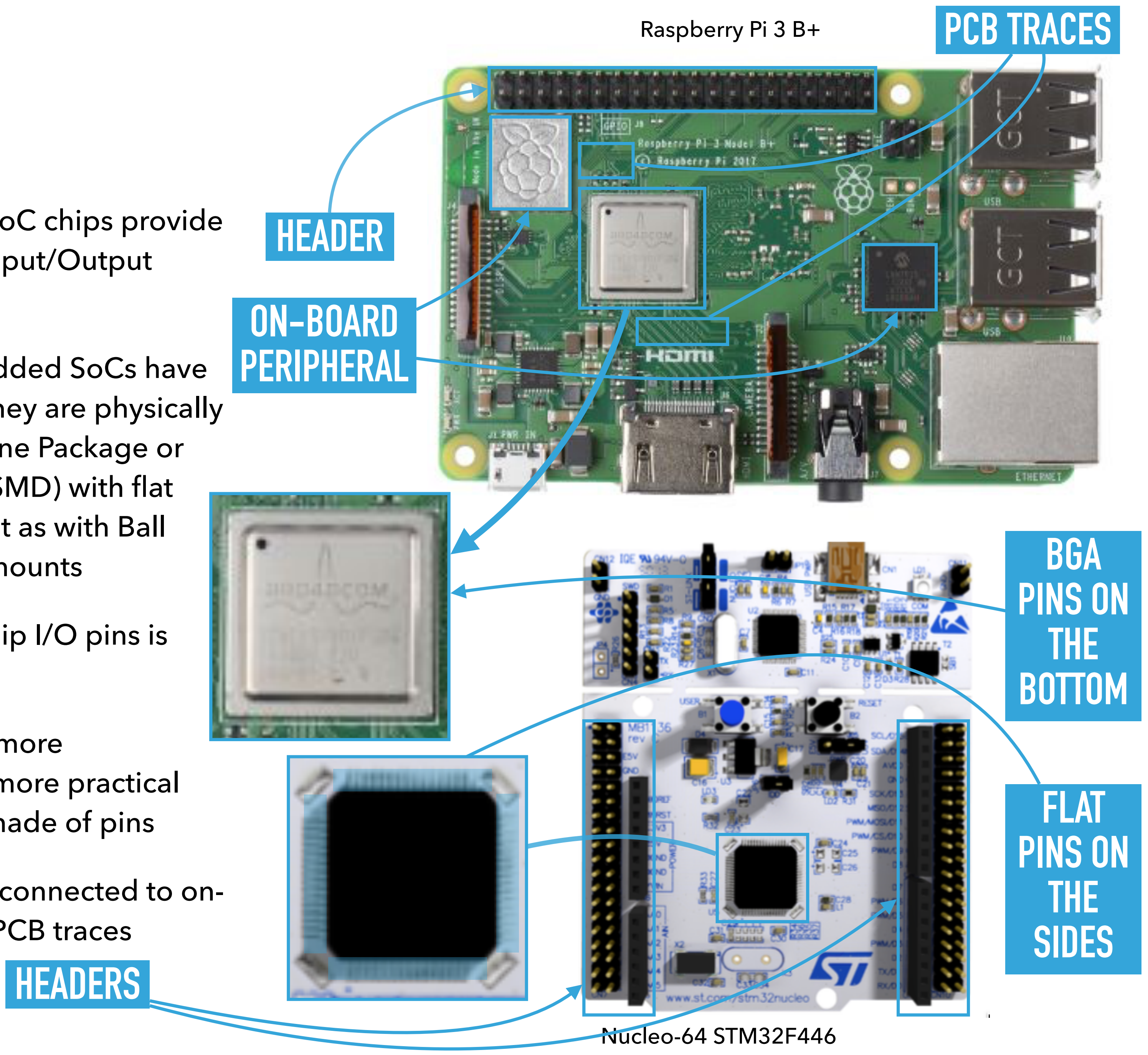
- Output signal are used to control devices such as displays, speakers, motors, servos and other actuators
- Often electronic devices called **drivers** are used to transform the **low-power** output signals from the CPU into signals matching the electric characteristics (voltage, current) required by actuators
- Other devices, called **controllers**, relieve the CPU of the burden to generate complex sequences of signal patterns to drive output devices (e.g. display controllers) and actuators (e.g. stepper motor controller)
- More in general, circuits connected to a processor to handle input signals and produce output signals are called **I/O peripherals**


A stepper motor controller between an MCU board (left) and a stepper motor (right)

I/O PINS

- Physical pins on MCU and SoC chips provide the electrical interface for Input/Output signals
- Microcontrollers and Embedded SoCs have plenty of pins; sometimes they are physically inspectable, as for Dual in-line Package or Surface Mounted Devices (SMD) with flat pins, sometimes they are not as with Ball Grid Array (BGA) package mounts
- Often just a subset of the chip I/O pins is used
- The used on-chip pins may more conveniently connected to more practical headers. Also headers are made of pins
- Other chip pins are directly connected to on-board peripherals through PCB traces

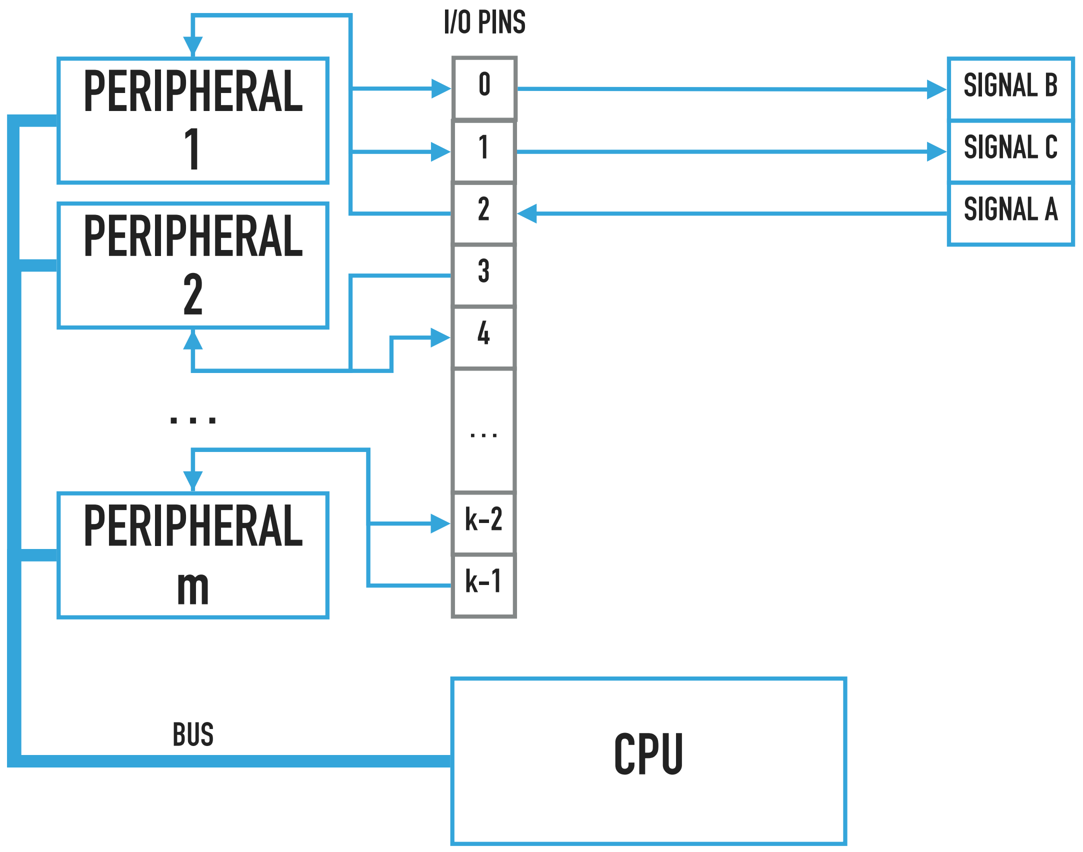
Raspberry Pi 3 B+



Nucleo-64 STM32F446

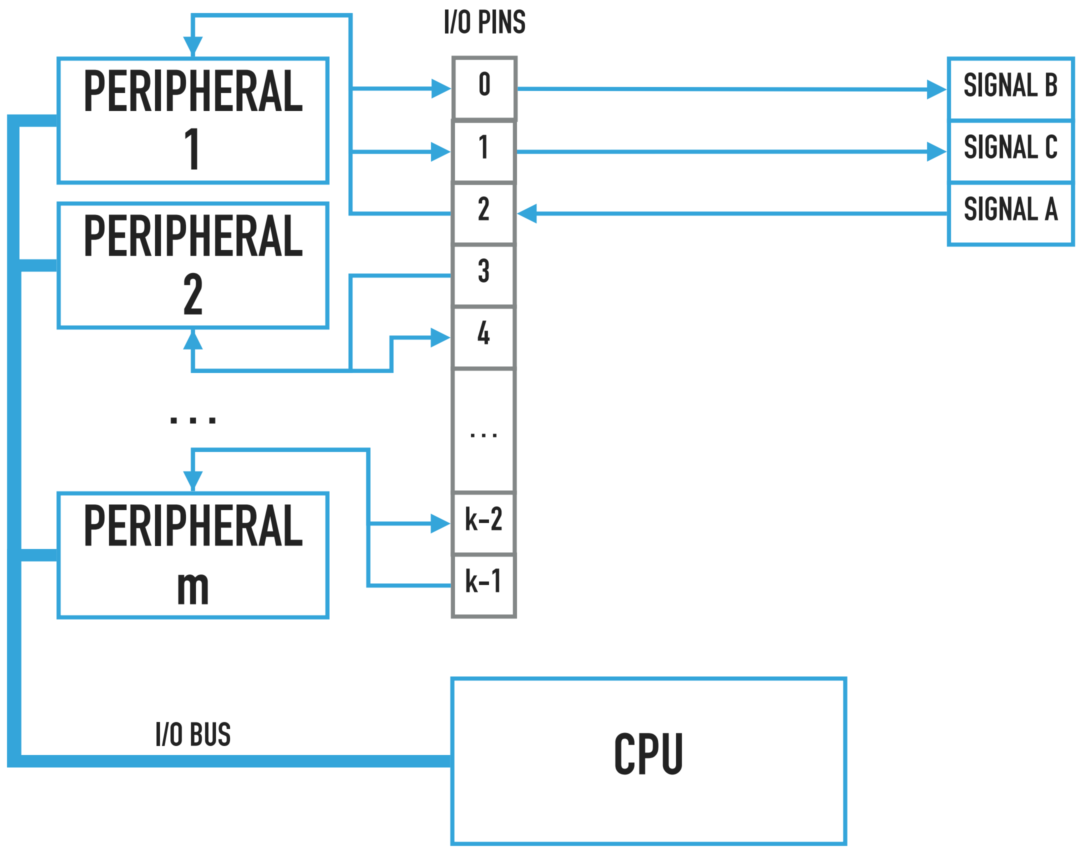
I/O PERIPHERALS

- I/O peripherals range from very simple to extremely complicated
- I/O peripherals may work concurrently with the CPU
- One or more **bus** connect I/O peripherals to the CPU
  - Some buses support Interrupts, Direct Memory Transfer (DMA), or both, reducing the burden of I/O on the CPU
- Each peripheral contains **control registers**
- The CPU reads and writes control registers to configure and operate the peripheral
- Microcontrollers and Embedded SoCs have plenty of peripherals



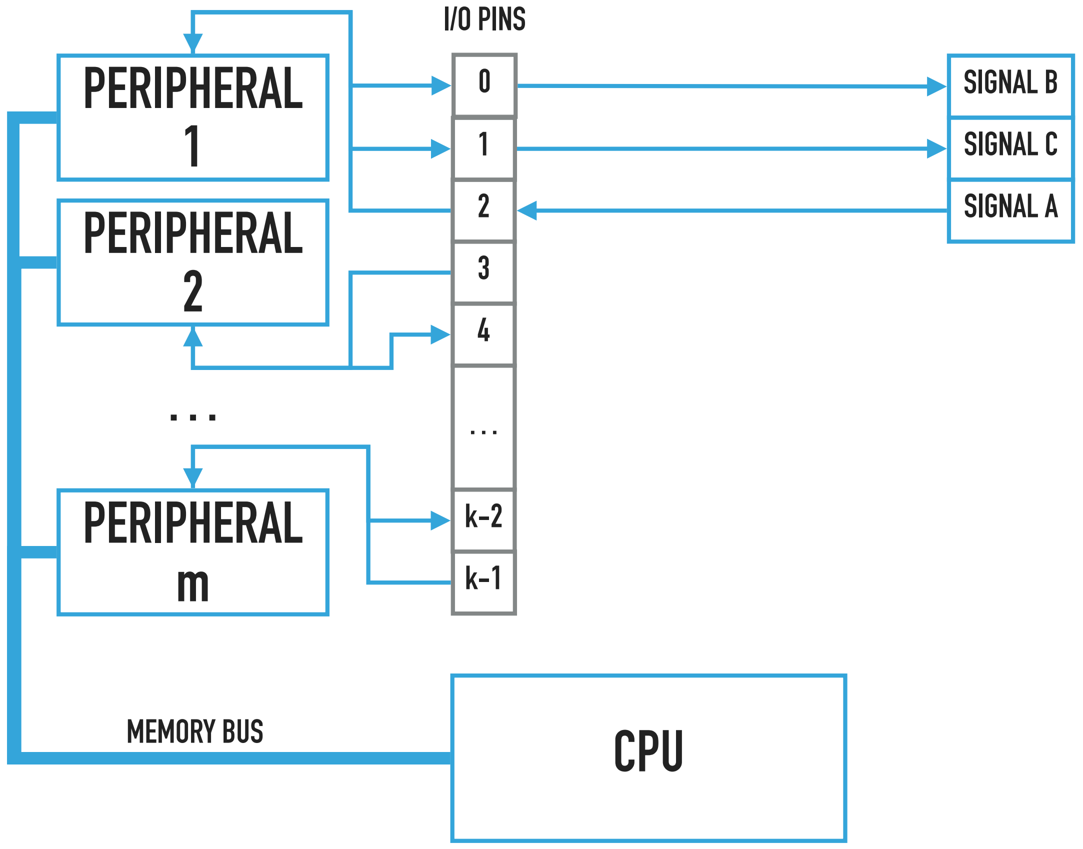
I/O PERIPHERALS

- The CPU can read and write registers using
  - An I/O bus (port-mapped I/O), but
    - Most CPUs do not have an I/O bus
    - Special ML instructions (e.g. `in`, `out`) are needed



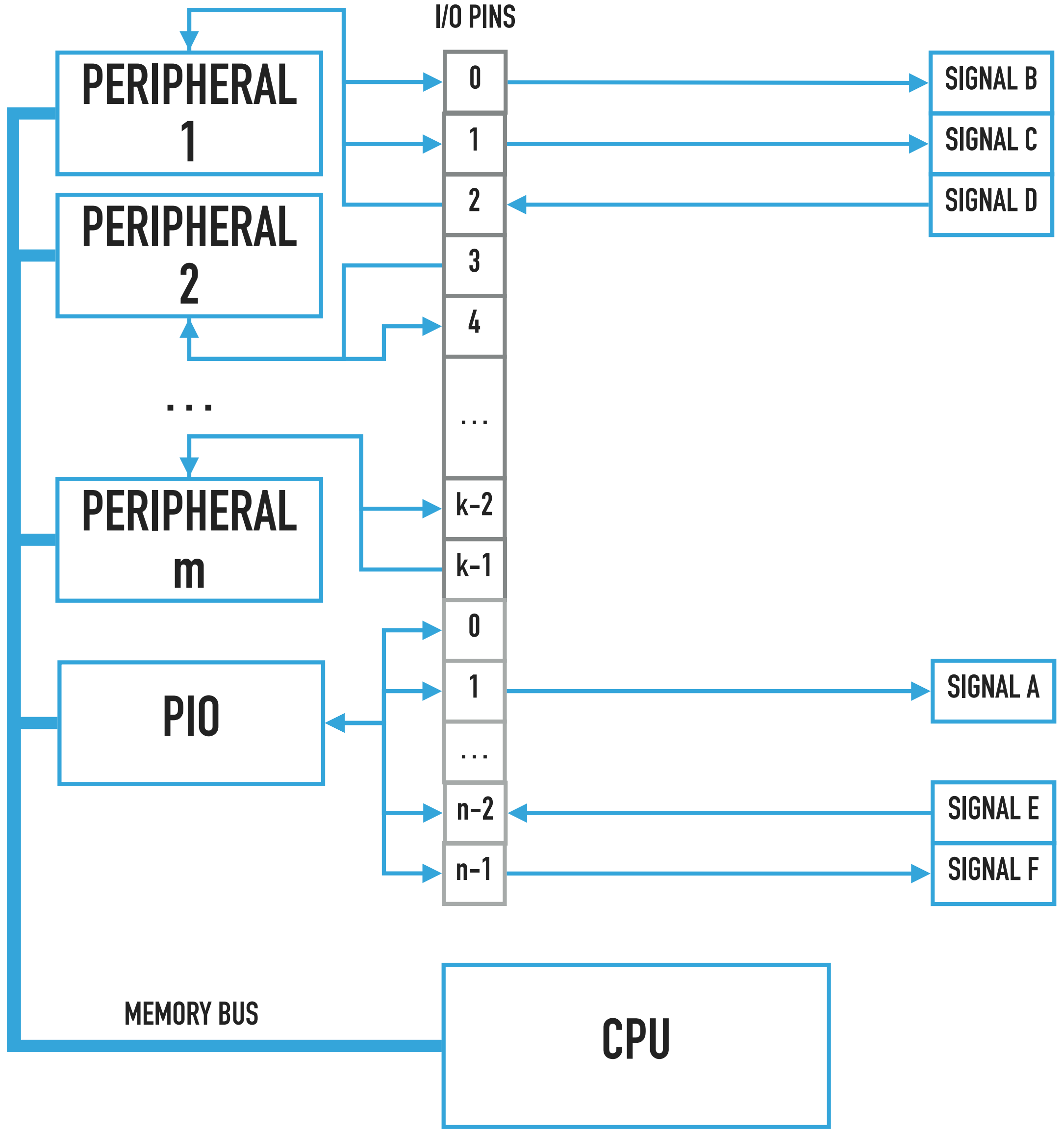
I/O PERIPHERALS

- More simply, the CPU can read and write registers using the memory bus (memory-mapped I/O)
    - All the CPUs have a memory bus
    - Normal load and store instructions can be used to drive peripherals as their registers appear to the CPU as memory words
- Most embedded SoCs are based on memory-mapped I/O



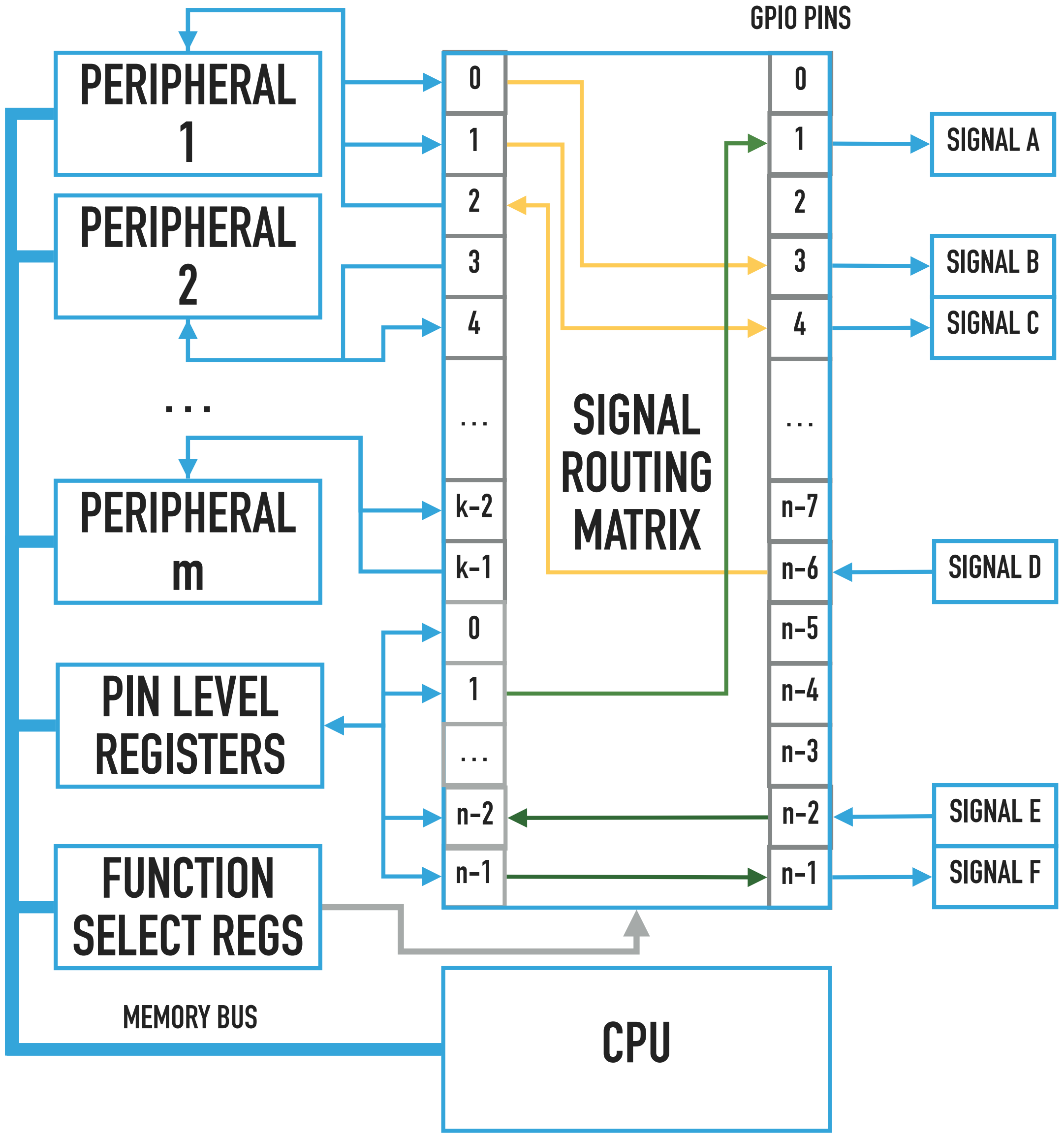
I/O PERIPHERALS

- Behavior of peripherals can be fixed, as for a serial interface or a disk controller, or programmed
- Parallel I/O (PIO) peripherals provide a set of **ports** each containing a number of simple digital logic I/O pins
    - A configuration register for each port permits to configure at any time each of the pins of the port as either input or output
    - Other registers for each port permit to set, reset or read the status of each pin



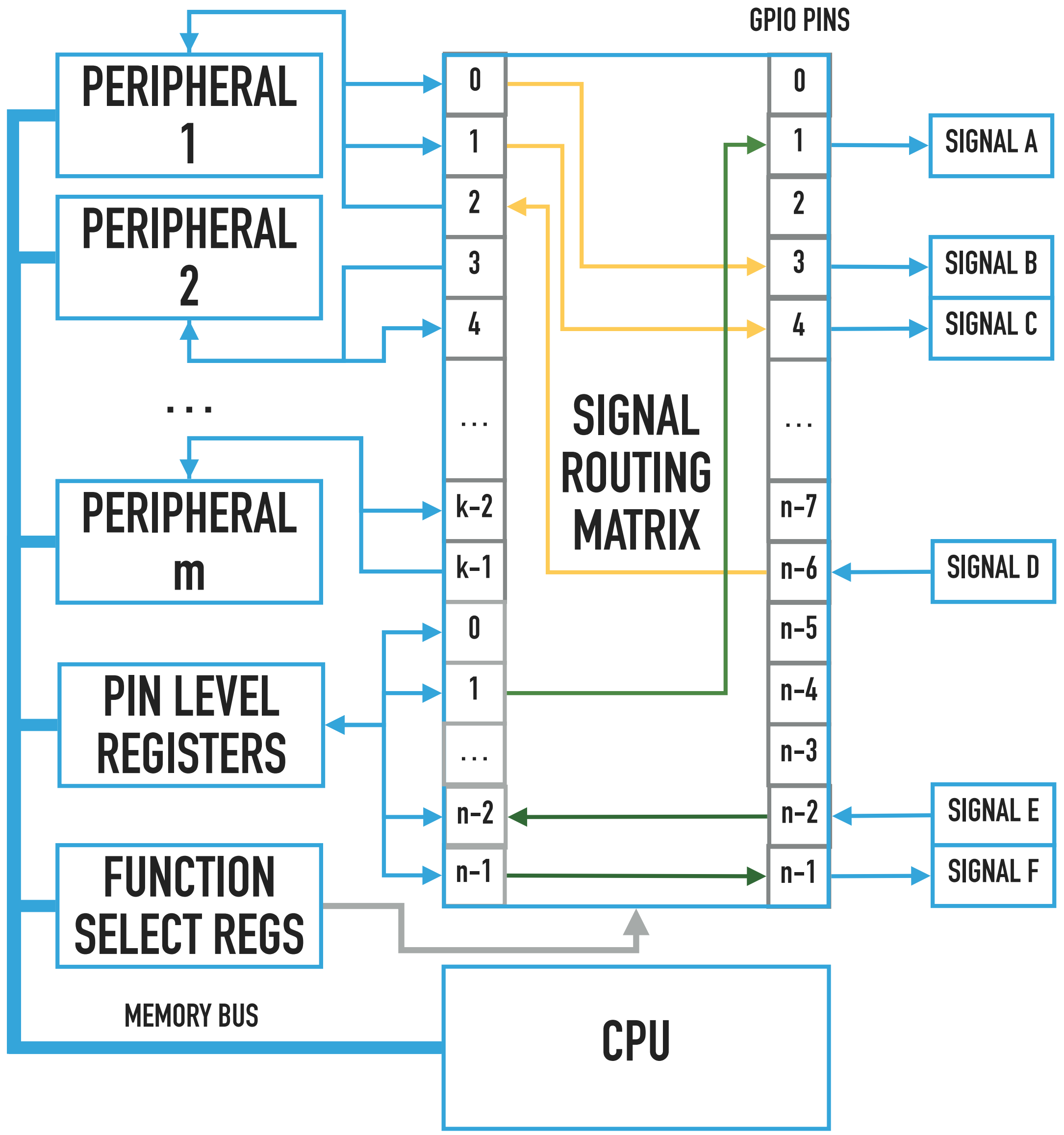
GENERAL-PURPOSE I/O (GPIO)

- Extending the concept of Parallel I/O, a General-Purpose I/O (GPIO) is a pin that, besides digital input and output, can be switched also to **alternate (or alternative) functions (AFs)** provided by internal peripherals
- Digital input and output bits are managed through Pin Level Registers that are similar to PIO registers



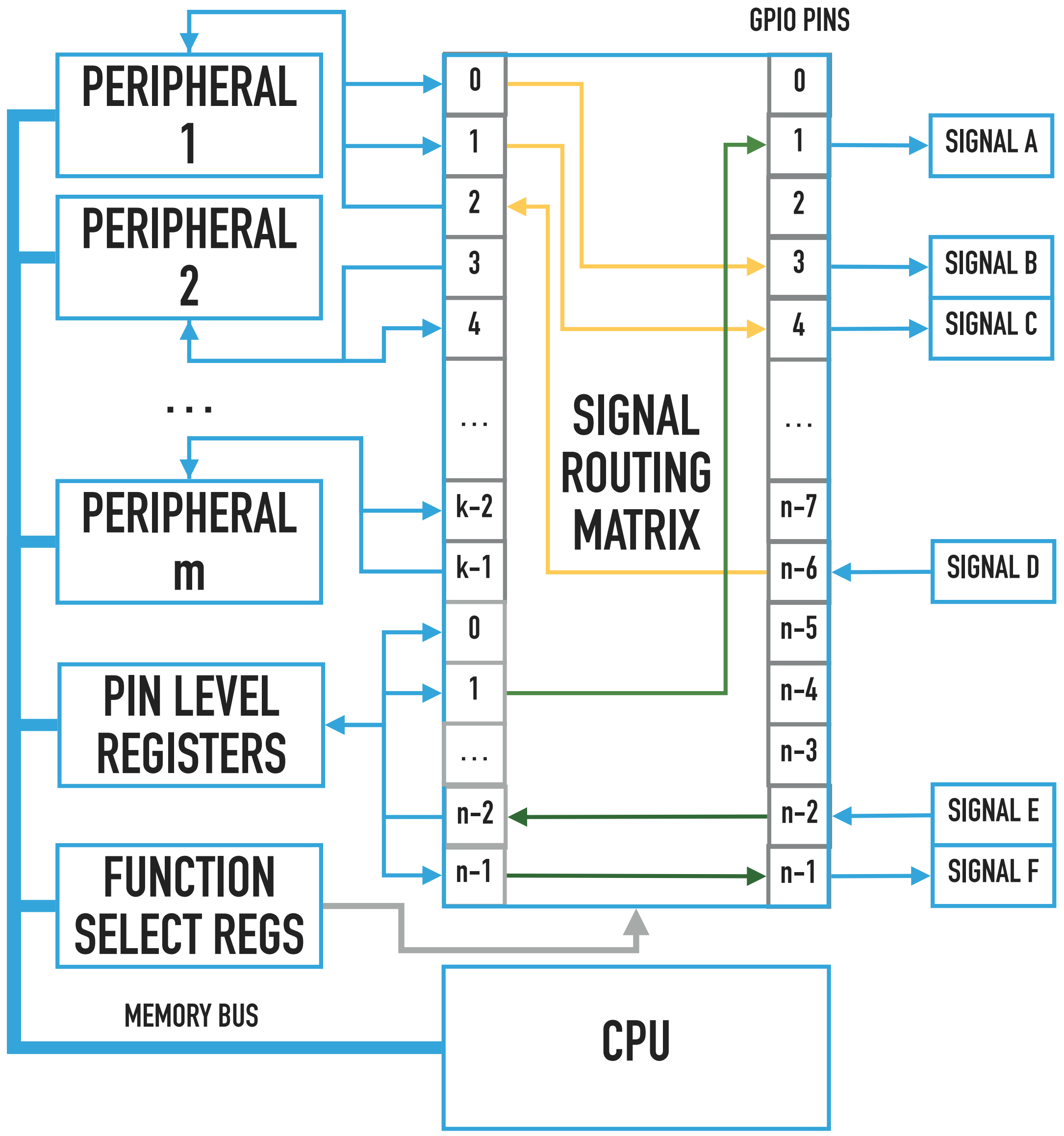
GENERAL-PURPOSE I/O (GPIO)

- Function selection for the GPIO pins is managed through a routing matrix that is configurable through registers
- A Function Select Register of p bits controls the function of each pin
  - For each pin is it possible to choose only at most a subset of 2^p of the available functions
- Given a register size rs, several pin-level ($\geq n/rs$) and AF-selection registers ($\geq (n * p)/rs$) may be required for a set of n GPIO pins
- Microcontrollers and Embedded SoCs have plenty of GPIOs



GENERAL-PURPOSE I/O (GPIO)

- In the simplified scheme in the figure:
    - Green arrows indicate digital input or output signals managed through pin level registers
    - Yellow arrows indicate signals associated to peripheral functions
- More complex interconnection through several bridges are often encountered (e.g. STM32F446)


GPIO PINS
PERIPHERAL 1
PERIPHERAL 2
...
PERIPHERAL m
PIN LEVEL REGISTERS
FUNCTION SELECT REGS
MEMORY BUS
CPU
SIGNAL ROUTING MATRIX
0
1
2
3
4
...
k-2
k-1
0
1
...
n-2
n-1
0
1
2
3
4
...
n-7
n-6
n-5
n-4
n-3
n-2
n-1
SIGNAL A
SIGNAL B
SIGNAL C
SIGNAL D
SIGNAL E
SIGNAL F