



POLITECNICO DI MILANO



Embedded from Scratch: Peripherals and interrupts

Advanced Operating Systems (2017/2018)

Federico Terraneo
federico.terraneo@polimi.it

A blinking led example

- The RCC and GPIO peripherals in the stm32 microcontroller
- Writing the main
- Compiling
- Analyzing the produced code
- Programming
- In-circuit debugging

A serial port example

- The serial port protocol
- The need for hardware peripherals
- The stm32 USART peripheral
- GPIOs and alternate functions
- Serial port transmission example
- Interrupts in the Cortex-M4 and stm32
- Serial port reception using interrupts

These slides explain general concepts present in a conceptually similar way in most microcontrollers, using a “by example” approach.

All the code (C/C++ sources, linker scripts, makefiles, ...) is available on the course website (`embedded_from_scratch.tar.gz`) while a guide on installing the compiler and required tools can be found at <https://miosix.org>

To understand how to program a microcontroller at the peripheral register level you always need the reference manual with the documentation of the peripheral registers.

You can find the one for the STM32F407VG microcontroller by searching online for “STM32F407VG reference manual” and downloading the RM0090 document from ST’s website. Also download the “STM32F407VG datasheet” and “UM1472 stm32f4discovery user manual”

A microcontroller has a certain number of electrical pins

- Some of them are dedicated to power, ground, clock, etc.
- Most of them are GPIOs
- Some of the GPIO can also be configured as alternate function (more on that later)



GPIO (General Purpose Input Output) are software-controllable pins

- They can be configured as inputs

Software can read the logic level (0 or 1) to sense conditions coming from the rest of the circuit

- They can be configured as outputs

Software can apply a logic level to control the rest of the circuit

GPIO pins are grouped in ports to

- Allow software to read and write to multiple GPIOs at the same time
- Make more efficient use of the memory layout of peripheral register

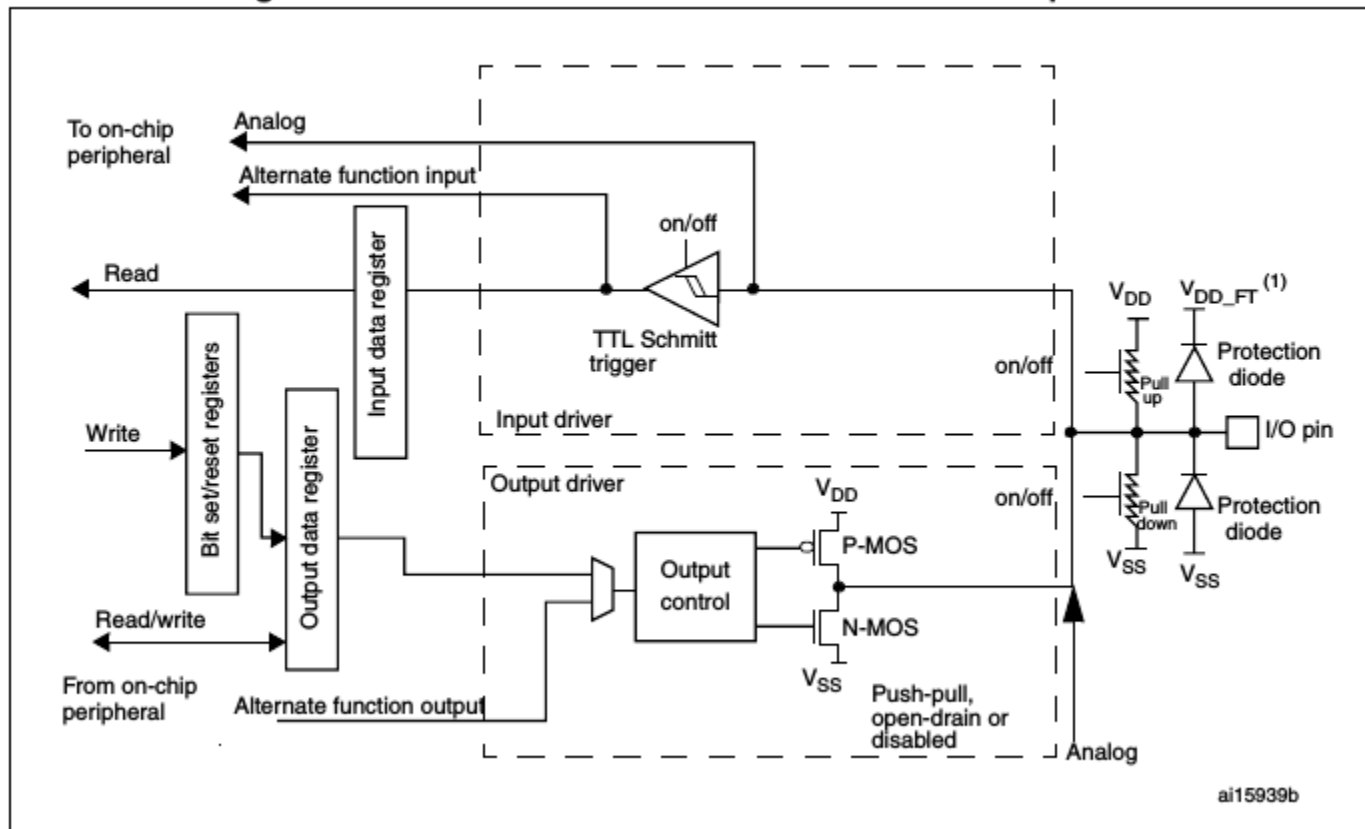
GPIO ports

- are identified by the manufacturer by either a letter or a number. stm32 microcontrollers use letters.
- The number of pins that compose a port depends on the microcontroller. stm32 microcontrollers have 16 pins per port

For example, PB2 identifies pin 2 of port B.

Each GPIO in the stm32 has the following structure

Figure 25. Basic structure of a five-volt tolerant I/O port bit



To configure pins in a port as input or output, use the MODER register

- Each port has its own register, so there is GPIOA->MODER, GPIOB->MODER, ...

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

00: Input (reset state)

01: General purpose output mode

10: Alternate function mode

11: Analog mode

To read the state of pins configured as input, use the IDR register

- Each port has its own register, so there is GPIOA->IDR, GPIOB->IDR, ...

8.4.5 GPIO port input data register (GPIOx_IDR) (x = A..I/J/K)

Address offset: 0x10

Reset value: 0x0000 XXXX (where X means undefined)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **IDRy**: Port input data (y = 0..15)

These bits are read-only and can be accessed in word mode only. They contain the input value of the corresponding I/O port.

Note how

- The IDR_x bits are read-only
- The upper 16 bits of the 32bit register are not present (reserved)

To change the state of pins configured as output, use the BSRR register

- Each port has its own register, so there is GPIOA->BSRR, GPIOB->BSRR, ...

8.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A..I/J/K)

Address offset: 0x18

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BRy**: Port x reset bit y (y = 0..15)

These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Resets the corresponding ODRx bit

Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BSy**: Port x set bit y (y= 0..15)

These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Sets the corresponding ODRx bit

Accessing the BSRR register

- All bits are write only, and cannot be read back
- The register is divided in two parts
 - ♦ Writing 1 to one of the first 16 bits causes the corresponding GPIO to be set high
 - ♦ Writing 1 to one of the second 16 bits causes the same GPIO to be cleared low

This is done to allow different parts of the program to concurrently access GPIOs of the same port without requiring locks and without causing race condition

- It allows to set a bit without a read-modify-write operation
 - ♦ To clear PB2, you can do `GPIOB->BSRR=1<<(2+16);`
 - ♦ To set PB5, you can do `GPIOB->BSRR=1<<5;`
- Notice the `=` and not `|=` operator, which is meaningless for registers that can't be read.

Hardware designers can implement special logic to modify bit in registers to achieve specific goals

To make these registers accessible from C/C++ we need to declare a struct

This is the complete list of the register of the GPIO peripheral

The same struct is mapped multiple times in the address space, once per port, since all ports have separate registers but share the same layout

```
typedef struct
{
    volatile uint32_t MODER;
    volatile uint32_t OTYPER;
    volatile uint32_t OSPEEDR;
    volatile uint32_t PUPDR;
    volatile uint32_t IDR;
    volatile uint32_t ODR;
    volatile uint32_t BSRR;
    volatile uint32_t LCKR;
    volatile uint32_t AFR[2];
} GPIO_TypeDef;

#define GPIOA ((GPIO_TypeDef *)0x40020000)
#define GPIOD ((GPIO_TypeDef *)0x40020C00)
```

Clock gating is a common solution to reduce the power consumption of digital logic

- Consists in leaving the unused parts of a hardware design unclocked, thus saving power
- Is not transparent to the application developer, before using a peripheral, it must be turned on explicitly

The RCC, “Reset and Clock Control” is the peripheral in the stm32 which controls clock gating

- Is the only peripheral that is always turned on
- Contains bits to enable the clock to all other peripherals
- So to use a GPIO port we first have to enable the GPIO peripheral

Register AHB1ENR in the RCC peripheral controls the clock to the GPIO peripherals

- One bit for each port: GPIOAEN, GPIOBEN, ...

6.3.10 RCC AHB1 peripheral clock register (RCC_AHB1ENR)

Address offset: 0x30

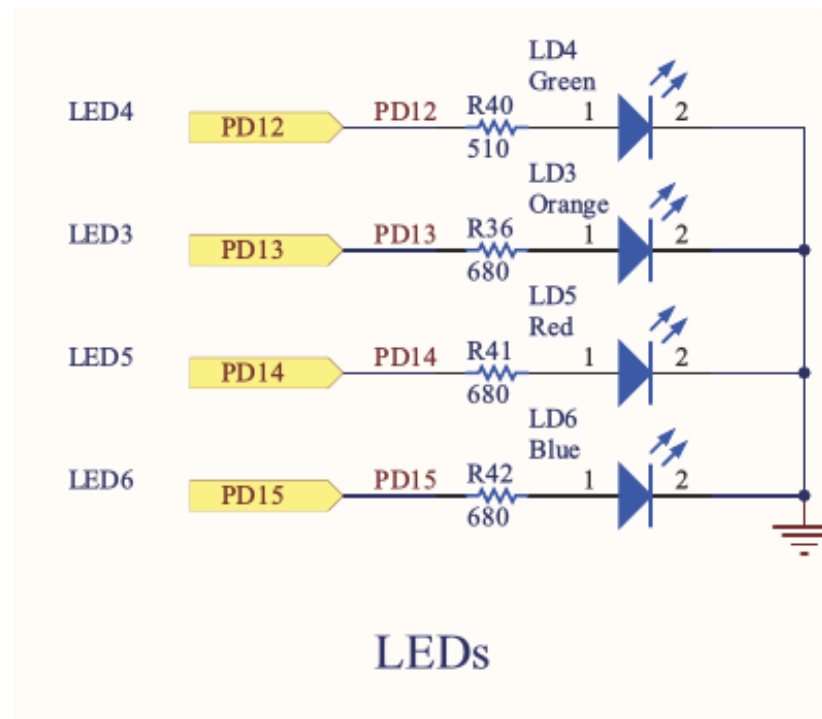
Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved	OTGHS ULPIEN	OTGHS SEN	ETHMACPT EN	ETHMACRX EN	ETHMACTX EN	ETHMACEN	Res.	DMA2D EN	DMA2E N	DMA1E N	CCMDAT ARAMEN	Res.	BKPSR AMEN	Reserved	
	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCE N	Res.	GPIOK EN	GPIOJ EN	GPIOIE N	GPIOH EN	GPIOG EN	GPIOFE N	GPIOEEN	GPIOD EN	GPIOC EN	GPIO BEN	GPIO AEN
			rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Now that we know how to access GPIOs, we need to know to which pin the LED is connected

- Does not depend on the microcontroller, but on the circuit it is soldered in
- In the stm32vldiscovery board, the red LED is connected to PD14
 - ♦ The schematics of the board are in the document UM1472 stm32f4discovery user manual



Now that the necessary registers have been introduced, we can write a main to blink the LED

In the absence of an OS, we can implement a delay with a loop that keeps the CPU busy for some time. Note the use of volatile to prevent the compiler from optimizing away the loop

First, we enable the clock gating to PORTD

Then, we configure PD14 as output

In the loop we set and reset pin 14 through the BSRR register

```
#include "registers.h"

void delay()
{
    volatile int i;
    for(i=0;i<1000000;i++)
    {
    }
}

int main()
{
    RCC->AHB1ENR |= 1<<3;
    GPIOD->MODER &= ~(3<<28);
    GPIOD->MODER |= 1<<28;
    for(;;)
    {
        GPIOD->BSRR=1<<14;
        delay();
        GPIOD->BSRR=1<<(14+16);
        delay();
    }
}
```

To compile our project we need to write a Makefile

We define variables for


- The C compiler
- The C++ compiler
- The assembler
- A tool to convert binary files

Note that we have to use tools designed for the architecture of the microcontroller (ARM)


Since the ARM architecture has multiple instruction sets, we must tell the tools for which instruction set we are compiling

Here we specify the flags for each tool we use


Here we list source files and produce the list of object files




```
CC := arm-miosix-eabi-gcc
CXX := arm-miosix-eabi-g++
AS := arm-miosix-eabi-as
CP := arm-miosix-eabi-objcopy
```



```
CPU := -mcpu=cortex-m4 -mthumb -mfloat-abi=hard \
      -mfpu=fpv4-sp-d16
```



```
AFLAGS := $(CPU)
CFLAGS := $(CPU) -O0 -g -c
CXXFLAGS := $(CPU) -O0 -fno-exceptions -fno-rtti \
              -g -c
LFLAGS := $(CPU) -Wl,-T./linker.ld,-Map,main.map \
              -nostdlib
```




```
SRC := startup.s main.c
OBJ := $(addsuffix .o, $(basename $(SRC)))
...
```


We use the C++ compiler to invoke the linker for us. This is required if at least one source file is written in C++

Objcopy strips metadata from the elf file to produce the bin file, which contains byte-per-byte what will be written in the microcontroller's flash

Rules for compiling assembly, C and C++ files



```
...
all: $(OBJ)
    $(CXX) $(LFLAGS) -o main.elf $(OBJ)
    $(CP) -O binary main.elf main.bin

clean:
    -rm $(OBJ) main.elf main.bin main.map

%.o : %.s
    $(AS) $(AFLAGS) $< -o $@

%.o : %.c
    $(CC) $(CFLAGS) $< -o $@

%.o : %.cpp
    $(CXX) $(CXXFLAGS) $< -o $@
```

The output of the compilation is

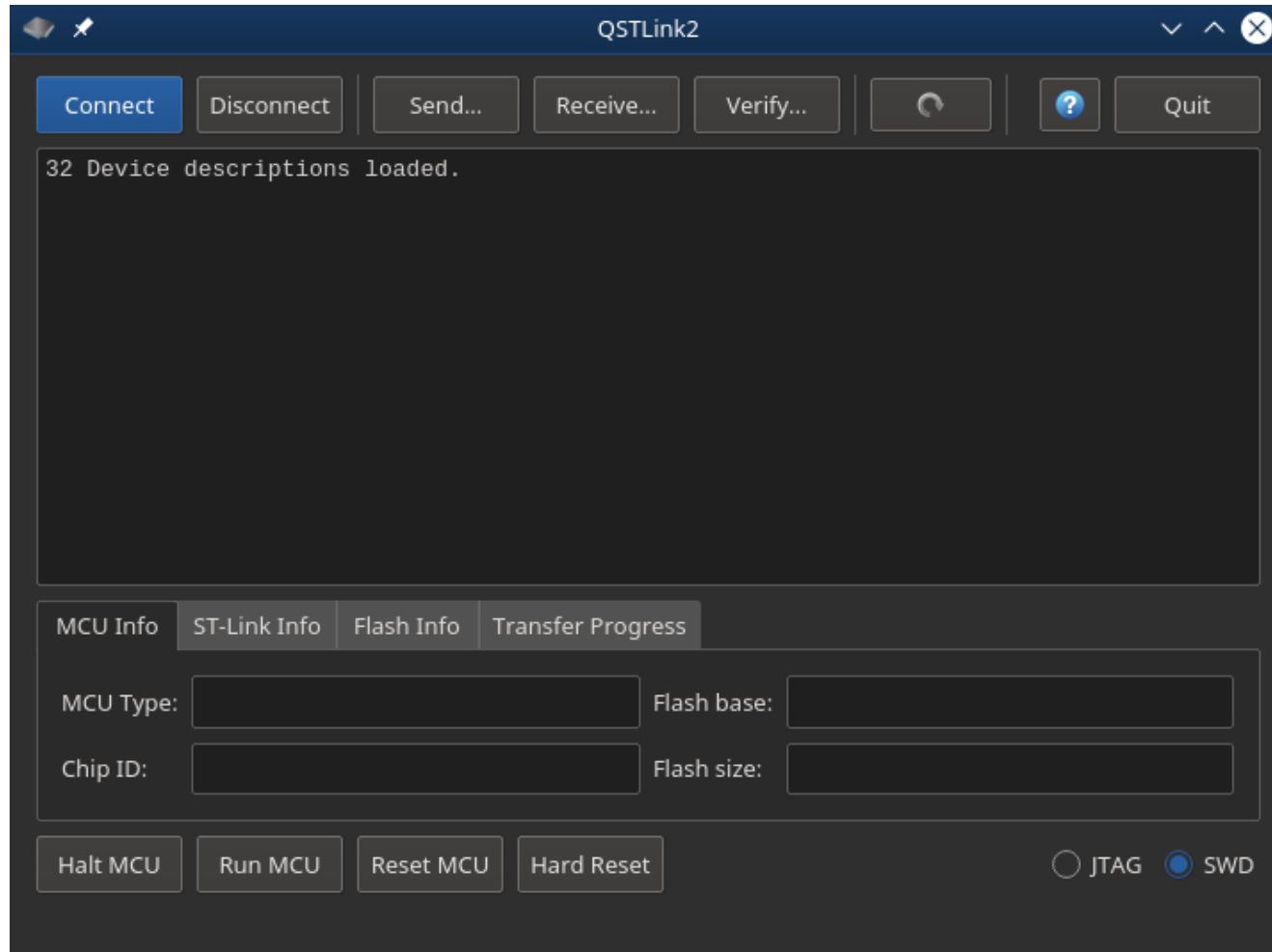
- The main.elf file
 - ♦ Contains both the output sections as specified by the linker script and metadata used by debuggers
 - ♦ Cannot be loaded directly in the FLASH memory of the microcontroller, the CPU cannot understand the metadata
 - ♦ The file for the blinking LED example is around 35KB
- The main.bin file
 - ♦ Contains only the output sections with no metadata
 - ♦ Is a firmware image that can be written to the microcontroller FLASH memory
 - ♦ The file for the blinking LED example is around 200 bytes

```
00000000 00 00 02 20 09 00 00 00 0E 48 0F 49 0F 4A 88 42 00 F0 08 80 ... ..H.I.J.B....
00000014 04 3A 52 F8 04 3F 40 F8 04 3B 81 42 7F F4 F9 AF 0A 48 0B 49 ..R..?@...;B.....H.I
00000028 88 42 00 F0 06 80 00 23 40 F8 04 3B 81 42 7F F4 FB AF 00 F0 .B.....#@...;B.....
0000003C 2B F8 FF F7 FE BF 00 00 00 00 00 20 00 00 00 20 F0 00 00 00 +.....
00000050 00 00 00 20 00 00 00 20 80 B4 83 B0 00 AF 4F F0 00 03 7B 60 ... ..O...{'`
00000064 4F F0 00 03 3B 60 07 E0 7B 68 03 F1 01 03 7B 60 3B 68 03 F1 0...;`...{h....{`;h..
00000078 01 03 3B 60 3A 68 44 F2 3F 23 C0 F2 0F 03 9A 42 F0 DD 07 F1 ..;`:hD.?#.....B....
0000008C 0C 07 BD 46 80 BC 70 47 80 B5 00 AF 4F F4 60 53 C4 F2 02 03 ...F..pG....O.`S....
000000A0 4F F4 60 52 C4 F2 02 02 12 6B 42 F0 08 02 1A 63 4F F4 40 63 0.`R.....kB....cO.@c
000000B4 C4 F2 02 03 4F F4 40 62 C4 F2 02 02 12 68 42 F0 80 52 1A 60 ....O.@b.....hB..R.`
000000C8 4F F4 40 63 C4 F2 02 03 4F F4 80 42 9A 61 FF F7 BF FF 4F F4 0.@c....O..B.a....O.
000000DC 40 63 C4 F2 02 03 4F F0 80 42 9A 61 FF F7 B6 FF EC E7 00 BF @c....O..B.a.....
```

The main.bin file contains

- At offset 0, the word 0x20020000 (in little-endian format). This is stored by the CPU in the stack pointer register during boot, and is the address at the top of the RAM memory
- At offset 4, the word 0x00000009 (in little-endian format). This is a pointer to the first instruction of the Reset_Handler function, where the boot begins
- The rest of the bytes are the opcodes of the functions Reset_handler, delay and main
 - The blinking LED example has no data section, otherwise the read-only copy would be there too
 - Since we did not use external libraries, every single byte of the firmware is related to code that we wrote

Loading the bin file to the microcontroller is done using the QSTlink tool



Modern microcontroller support a feature called in-circuit debugging through a special debugging port. When in debug mode

- The clock to the CPU can be stopped
 - ♦ This allows halting the code execution
- While the CPU is halted, it is possible to examine and modify the memory
 - ♦ This allows to inspect and modify the program state at a certain point during the program execution
- One or more hardware comparators exist on the CPU address bus, triggering a CPU halt
 - ♦ This allows putting breakpoints that stop code execution once a certain line of code is reached

Note: when debugging the code needs to be compiled with optimizations disabled. Optimizations blur the mapping between source code and assembly code, confusing debuggers

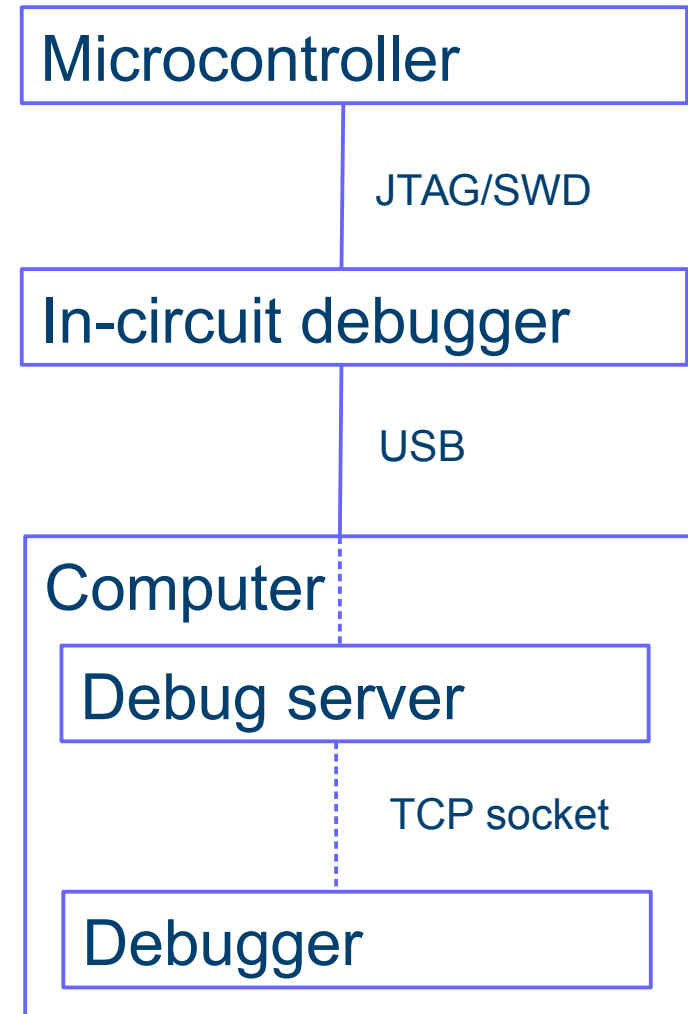
Debugging a microcontroller requires the following

An in-circuit debugger is connected to the microcontroller through a special debug port using protocols such as JTAG or SWD. The stm32f4discovery has the in-circuit debugger built-in.

The in-circuit debugger is connected to a computer, usually through USB.

A debug server such as openocd listens to a TCP socket, and sends commands to the in-circuit debugger.

A source-level debugger such as GDB reads the source code metadata from the elf file and connects to the debug server.



To debug

- Connect the stm32f4discovery board to the computer
- Download the openocd configuration file for the board from
 - ♦ https://github.com/fedetft/miosix-kernel/blob/master/miosix/arch/cortexM4_stm32f4/stm32f407vg_stm32f4discovery/stm32f4discovery.cfg
- Open a shell and launch openocd

```
[fed@asus miosix-kernel]$ openocd -f miosix/arch/cortexM4_stm32f4/stm32f407vg_stm32f4discovery/stm32f4discovery.cfg
Open On-Chip Debugger 0.9.0 (2015-09-02-10:42)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v14 API v2 SWIM v0 VID 0x0483 PID 0x3748
Info : using stlink api v2
Info : Target voltage: 2.887346
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

The last line indicates that the microcontroller was found

Open a second shell and type the following commands

The first line starts the debugger. You must pass to it the elf file and the corresponding bin file must be already flashed to the board.

The second line tells gdb to connect to openocd through a TCP socket on port 3333

The third line resets the board and leaves the CPU halted

The fourth line puts a breakpoint at the beginning of main

The last line releases the CPU, which will perform the boot and stop at the main

```
$ arm-miosix-eabi-gdb main.elf
(gdb) target remote :3333
(gdb) monitor reset halt
(gdb) break main
(gdb) continue
```


Useful gdb command

- `break <function name>`
 - ♦ Puts a breakpoint to the beginning of the given function
 - ♦ Example: `break main`
- `break <source file name>:<line number>`
 - ♦ Puts a breakpoint on the specified source code line
 - ♦ Example `break main.c:8`
- `continue`
 - ♦ Restarts execution, stops at the next breakpoint
- `step`
 - ♦ Single steps program execution
 - ♦ If the line to be executed is a function, stop at the first line inside the function
- `next`
 - ♦ Single steps program execution
 - ♦ If the line to be executed is a function, execute it till the end

Useful gdb command

- `print <variable>`
 - ♦ Prints the content of the specified variable
 - ♦ The variable must be in scope
 - ♦ Example: `print i`
- `Set var <variable name>=<value>`
 - ♦ Modifies the variable
 - ♦ The variable must be in scope
 - ♦ Example: `set var i=10`

Microcontrollers have hardware support for many communication interfaces

- Some of them are designed to connect the microcontroller to a PC
 - ♦ USB
 - ♦ Ethernet
 - ♦ ...
- Other are specific to interfacing to other integrated circuits on the same board
 - ♦ SPI
 - ♦ I2C
 - ♦ ...

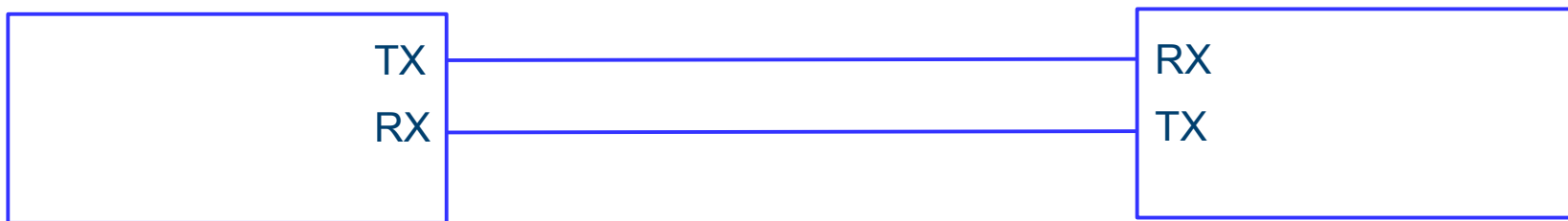
Of the many available interfaces, we will explore the serial port, a versatile interface that

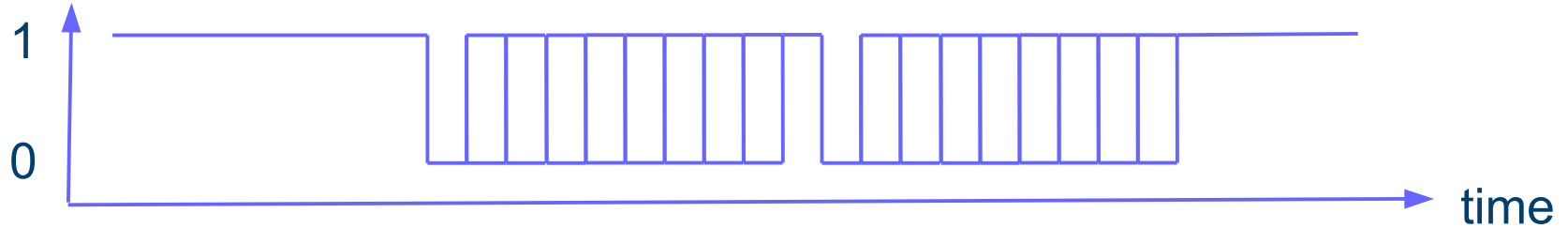
- Can be used for interfacing to other integrated circuits as well as to a PC
- Does not require a complex software driver to operate

- A serial port is a character-oriented interface that allows to transfer characters between two devices.
- Unlike other interfaces such as USB or Ethernet there is no standard stack of protocols layered on top of each other that must be implemented for proper operation
 - ♦ Although higher layer protocols exist for certain classes of devices, for example the NMEA protocol to interface to GPS receivers, for simple PC interfacing no higher level protocol is needed

The minimal serial port interface uses two wires (+ground) to connect two devices

- Both devices have a transmit end (TX) that is connected to the receive end (RX) of the other one





In a 3.3V or 5V serial port

- The line is idle high
 - As long as the transmitter has nothing to send, the line is driven high
- All bits have the same length
 - The bit length or its inverse, the “baud rate” is a configuration parameter. Both the transmitter and receiver must agree on this parameter for communication
- When transmitting a character
 - The transmitter sends a start bit which is always a 0
 - Then the bits of the character are sent. The number of bits per character is another configuration parameter, but is usually 8
 - Finally, a stop bit is sent, always a 1
- A new character can then be transmitted immediately

It is possible to implement the serial port protocol in software

- GPIOs can be used for input (RX) and output (TX)
- Appropriate delay functions can be used to wait one bit time

However, there are many problems

- The CPU is kept busy for the entire time of a transmission, and even worse, for the entire time when data is expected to be received
- The protocol is timing-critical, errors in time measurements cause wrong characters to be sent or wrong decoding of incoming characters
 - Interrupts may add short delays (a few microseconds), which may create issues at high baud rates
 - If an OS is used, context switches may introduce long delays (milliseconds), which will create issues even at the slowest baud rates

Most modern microcontroller have one (or more) hardware peripherals dedicated to implementing the serial port protocol.

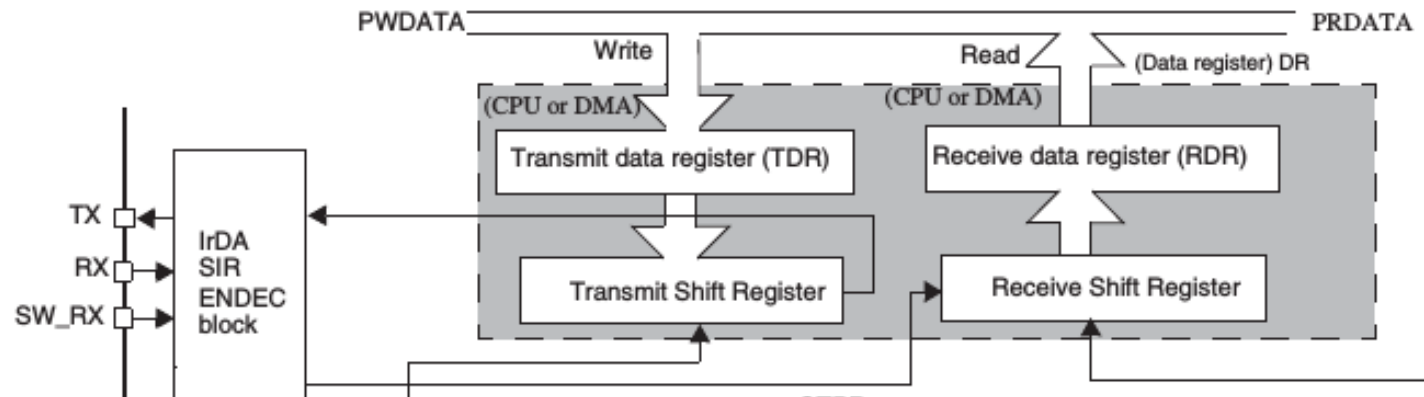
Software sees a higher level interface

- For the transmit side
 - ♦ A character is written to a peripheral register
 - ♦ The hardware takes care of sending the start bit, then one by one the bits of the character and finally the stop bit on a given GPIO
- For the receive side
 - ♦ The hardware monitors a given GPIO for the start bit
 - ♦ The character is reconstructed from the bits on the RX line
 - ♦ Only when the full character has been received, software is notified and can read the character from a peripheral register

The hardware peripheral offloads the CPU from the low-level part of the protocol

The stm32 hardware serial port is called USART

- Just like GPIO ports, there are more than one USART peripherals
 - ♦ They are called USART1, USART2, ...
 - ♦ Allow to perform multiple serial communications to different devices at the same time



At the heart of the peripheral there are two shift registers

- They construct/deconstruct characters one bit at a time

The USART registers are divided in

- Control registers
 - Allow to enable transmission and reception, and set the baud rate
- Data registers
 - Allow software to send characters to be transmitted, and get received characters
- Status registers
 - Allow to know when the peripheral has finished transmitting/receiving characters

The first control register is the USART CR1 register

- The UE bit enables the USART. If it is zero, the peripheral is off
- The TE bit enables the transmit side of the USART
- The RE bit enables the receive side
- The RXNEIE bit enables an interrupt when a character is received



We will see interrupts later

30.6.4 Control register 1 (USART_CR1)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw	Res.	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The second control register is the USART BRR register

- It is used to set the baud rate, that is the transmission speed
- It control a hardware prescaler, which divides the system clock to obtain the clock to drive the shift registers that transmit the bits. You have to program the division factor
- The stm32 when used in the embedded from scratch example runs at 8MHz (even though it can run up to 168MHz). We want the baud rate to be 19200bit/sec, but the prescaled clock has to be 8 times faster than the baud rate.
 - Thus the division factor is $8\text{MHz} / (19200 \times 8) = 52.0833333$
- We can only write fixed point numbers with 1/16 fractional part increments
 - The closest value is 52.0625
 - So our serial port actually runs at 19207.68bit/sec

30.6.3 Baud rate register (USART_BRR)

Note: The baud counters stop counting if the TE or RE bits are disabled respectively.

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV_Mantissa[11:0]												DIV_Fraction[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

There is a single data register, the USART DR register

- Writing a character to this register causes the character to be transmitted
 - ♦ Before writing, software should check that the transmit shift register is not busy transmitting the previous character, otherwise the written character is lost
- Reading this register returns the last received character
 - ♦ Software should read this register as soon as a character is received. If multiple characters arrive, the previous characters are lost
- Notice that despite the register is both readable and writable, reading and writing have two separate semantics (one accessing the transmit, the other the receive side)

Status bits are in the USART SR register

- The TXE bit is 1 when the transmit shift register is empty, and thus ready to accept a new character
- The RXNE bit is 1 when the receive shift register is not empty, and thus a full character has been received and can be read by software

30.6.1 Status register (USART_SR)

Address offset: 0x00

Reset value: 0x00C0 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

How does the USART peripheral interfaces with the outside world?

- It reuses the same pins on the microcontroller that are used as GPIO
- Each I/O pin can be either used as
 - ♦ GPIO, where it is software controlled
 - ♦ Alternate function, where it is driven by a hardware peripheral

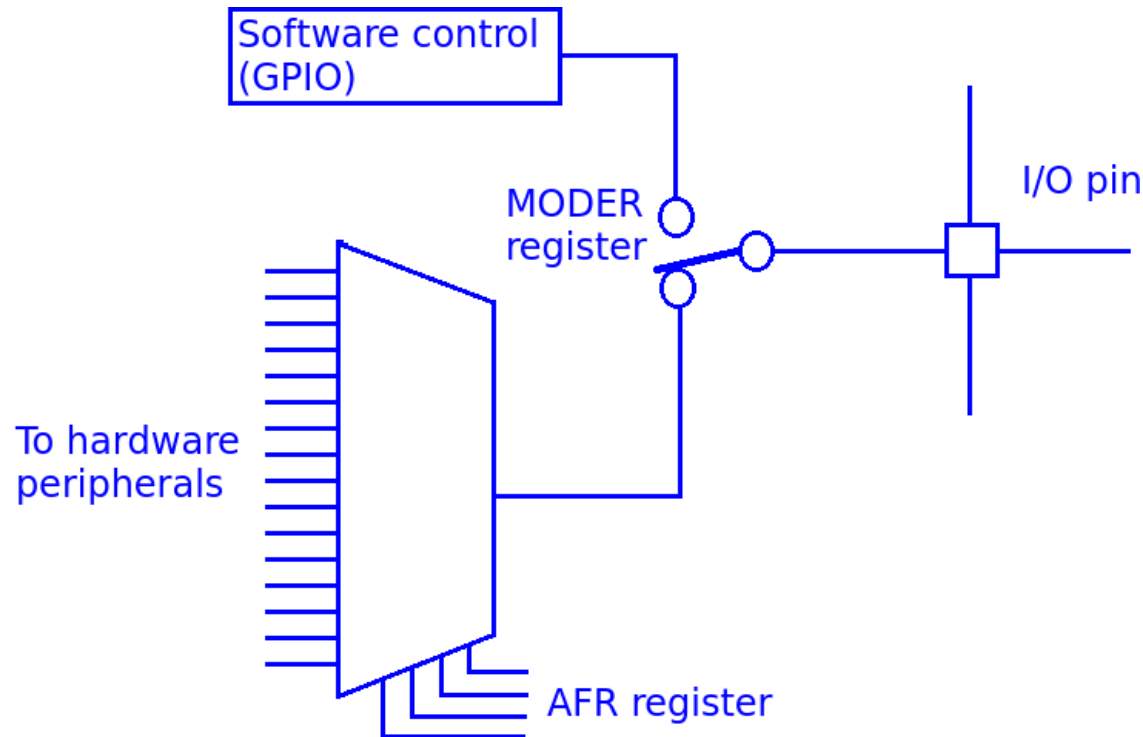
Already seen in the
blinking led example



Since the USART peripherals are connected only to specific I/O pins

- We must understand to which pins the USART is connected
 - ♦ We cant choose any I/O of any port and use them as an USART
- We must configure those pins as alternate function
- Moreover, in the STM32 multiple peripherals compete for the same I/O pins
 - ♦ We must select which peripheral (in this case, the USART) to connect to the pins

The management of I/O pins in the STM32 is conceptually as follows



To connect a pin to a hardware peripheral

- Set the bits of the MODER register to disconnect the pin from software control
- Write the bits in the AFR register to connect the pin to the wanted peripheral

The MODER register is the same we have used to configure GPIOs as input or output

- However, when setting the bits to “10” the I/O pin is configured as alternate function

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]**: Port x configuration bits (y = 0..15)

These bits are written by software to configure the I/O direction mode.

00: Input (reset state)

01: General purpose output mode

10: Alternate function mode

11: Analog mode

The AFR register is used to select which peripheral to connect

- There are 4 bits per I/O, up to 16 peripherals can compete for the same pin
- There are two registers per port, AFRL or AFR[0] for the pins 0 to 7, and AFRH or AFR[1] for the pins from 8 to 15.

8.4.9 GPIO alternate function low register (GPIOx_AFRL) (x = A..I/J/K)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

8.4.10 GPIO alternate function high register (GPIOx_AFRH) (x = A..I/J/K)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRH15[3:0]				AFRH14[3:0]				AFRH13[3:0]				AFRH12[3:0]			
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRH11[3:0]				AFRH10[3:0]				AFRH9[3:0]				AFRH8[3:0]			
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

A table in the datasheet shows which peripherals are connected to which pin

Table 12. STM32F427xx and STM32F429xx alternate function mapping

Port		AF0	AF1	AF2	AF3	AF4	AF5	AF6	AF7	AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SYS	TIM1/2	TIM3/4/5	TIM8/9/ 10/11	I2C1/ 2/3	SPI1/2/ 3/4/5/6	SPI2/3/S AI1	SPI3/US ART1/2/3	USART6/U ART4/5/7/8	CAN1/2/TIM 12/13/14/ LCD	OTG2_HS /OTG1_ FS	ETH	FMC/SDIO /OTG2_FS	DCMI	LCD	SYS
Port A	PA0	-	TIM2 CH1/TIM2 ETR	TIM5_ CH1	TIM8_ ETR	-	-	-	USART2_ CTS	UART4_TX	-	-	ETH_MII_ CRS	-	-	-	EVEN TOUT
	PA1	-	TIM2_ CH2	TIM5_ CH2	-	-	-	-	USART2_ RTS	UART4_RX	-	-	ETH_MII_ RX_CLK/E TH_RMII_ REF_CLK	-	-	-	EVEN TOUT
	PA2	-	TIM2_ CH3	TIM5_ CH3	TIM9_ CH1	-	-	-	USART2_ TX	-	-	-	ETH_ MDIO	-	-	-	EVEN TOUT
	PA3	-	TIM2_ CH4	TIM5_ CH4	TIM9_ CH2	-	-	-	USART2_ RX	-	-	OTG_HS_ ULPI_D0	ETH_MII_ COL	-	-	LCD_B5	EVEN TOUT
	PA4	-	-	-	-	-	SPI1_ NSS	SPI3_ NSS/ I2S3_WS	USART2_ CK	-	-	-	-	OTG_HS_ SOF	DCMI_ HSYNC	LCD_ VSYNC	EVEN TOUT
	PA5	-	TIM2 CH1/TIM2 ETR	-	TIM8_ CH1N	-	SPI1_ SCK	-	-	-	-	OTG_HS_ ULPI_CK	-	-	-	-	EVEN TOUT
	PA6	-	TIM1_ BKIN	TIM3_ CH1	TIM8_ BKIN	-	SPI1_ MISO	-	-	-	TIM13_CH1	-	-	-	DCMI_ PIXCLK	LCD_G2	EVEN TOUT
	PA7	-	TIM1_ CH1N	TIM3_ CH2	TIM8_ CH1N	-	SPI1_ MOSI	-	-	-	TIM14_CH1	-	ETH_MII_ RX_DV/ ETH_RMII_ CRS_DV	-	-	-	EVEN TOUT
	PA8	MCO1	TIM1_ CH1	-	-	I2C3_ SCL	-	-	USART1_ CK	-	-	OTG_FS_ SOF	-	-	-	LCD_R6	EVEN TOUT
	PA9	-	TIM1_ CH2	-	-	I2C3_ SMBA	-	-	USART1_ TX	-	-	-	-	-	DCMI_ D0	-	EVEN TOUT
	PA10	-	TIM1_ CH3	-	-	-	-	-	USART1_ RX	-	-	OTG_FS_ ID	-	-	DCMI_ D1	-	EVEN TOUT
	PA11	-	TIM1_ CH4	-	-	-	-	-	USART1_ CTS	-	CAN1_RX	OTG_FS_ DM	-	-	-	LCD_R4	EVEN TOUT
	PA12	-	TIM1_ ETR	-	-	-	-	-	USART1_ RTS	-	CAN1_TX	OTG_FS_ DP	-	-	-	LCD_R5	EVEN TOUT

We can now start writing a library for the USART peripheral

Since we can use C++,
our driver will be a class

The constructor will initialize the USART and configure the GPIOs as alternate function, while the write member function can be called to transmit data

```
//File: serial.h

#ifndef SERIAL_H
#define SERIAL_H

class SerialPort
{
public:
    SerialPort();

    void write(const char *s);
};

#endif //SERIAL_H
```

The C++ source file contains the driver implementation

First, we make sure the clock gating for both the GPIOA and USART2 are enabled

Then, we configure PA2 and PA3 as alternate function, and select alternate function 7, to connect them to USART2

Finally, we enable the USART, set the baud rate and enable the transmit side

```
//File: serial.cpp
#include "serial.h"
#include "registers.h"

SerialPort::SerialPort()
{
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
    RCC->APB1ENR |= RCC_APB1ENR_USART2EN;

    GPIOA->AFR[0] &= ~( (15<<(2*4)) | (15<<(3*4)) );
    GPIOA->AFR[0] |= (7<<(2*4)) | (7<<(3*4));
    GPIOA->MODER |= (2<<(2*2)) | (2<<(3*2));

    USART2->CR1 = USART_CR1_UE;
    USART2->BRR = (52<<4) | (1<<0);
    USART2->CR1 |= USART_CR1_TE;
}
```

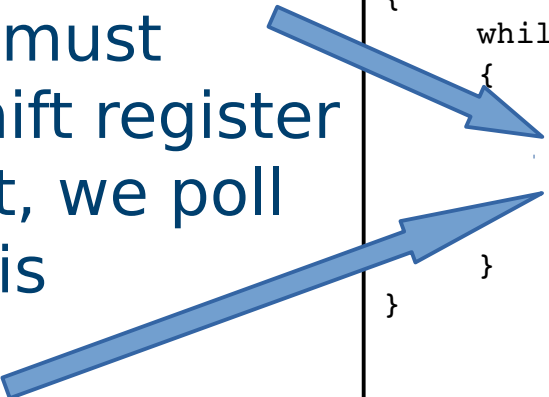
...

The driver write function sends characters one at a time

Before sending each character, we must check if the shift register is empty. If not, we poll the bit until it is

Then we send the current character and increment the pointer

```
//File: serial.cpp
...
void SerialPort::write(const char *str)
{
    while((*str)!='\0')
    {
        //Wait until the hardware fifo is ready
        while((USART2->SR & USART_SR_TXE)==0) ;
        USART2->DR=*str;
        str++;
    }
}
```



The main just repeatedly blinks an LED and sends a string

To see the output,
connect the serial cable to
PA2 (TX from the stm32,
RX of the serial cable)
PA3 (RX to the stm32, TX
of the serial cable)

Use

screen /dev/ttyUSB0 19200
on the PC side to read the
data from the serial port

```
//File: main.cpp
...
#include "registers.h"
#include "serial.h"

void delay()
{
    volatile int i;
    for(i=0;i<1000000;i++) ;
}

SerialPort serial;

int main()
{
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;
    GPIOD->MODER |= 1<<28;
    bool led=false;
    for(;;)
    {
        serial.write("Hello world\r\n");
        delay();
        if(led) GPIOD->BSRR=1<<14;
        else GPIOD->BSRR=1<<(14+16);
        led=!led;
    }
}
```

We have seen that writing to peripheral registers is the way for software to communicate with hardware

On the other hand, for hardware to communicate with software, we can

- Have software check status bits
- Use interrupts

Why we need interrupts?

- Repeatedly checking status bits is called polling
- There is an inherent tradeoff between responsiveness to events, and polling period
- Some events may be sporadic
 - ♦ No character arrives to a serial port for a long time
 - ♦ Then, two characters arrive in a row
 - ♦ If the polling is not fast enough, and software does not read the first character before the second arrives, the first character may be lost

Interrupts can be thought as letting hardware call a software function when an event occurs

When the interrupt occurs, the CPU is executing other code, that we call the “main code”

Interrupts can pause the execution of the main code in between any two assembly instructions, and cause it to jump to a function, the “interrupt service routine (ISR)”

- Hardware/compiler/OS take care of saving the CPU registers so as not to interfere with program execution
- Accessing variables from both main code and interrupts causes concurrency issues, we will see how to handle them

When the interrupt service routine has completed, the processor reverts back to executing the main code

- Interrupts are always run to completion, an interrupt must NEVER block, or it will block the main code as well
- Interrupts should be written to be as fast as possible, in order to minimize the time interference with the main code (interrupts can insert pauses anywhere in the main code, reducing its time determinism)

Interrupts require cooperation between three distinct hardware components

- The peripheral
 - Generates events
 - Has local status and enable bits for the events
 - Usually groups all events in a single output signal to the interrupt controller
- The interrupt controller
 - Receives the interrupt signals from all the peripherals
 - Has enable, pending and active bits for all peripherals
 - Handles interrupt prioritization, and generates the output signal to the CPU
- The CPU
 - Has the global interrupt enable bit
 - Has logic to stop main code execution and jump to the correct ISR

In a microcontroller

- All three components are inside the microcontroller, all signals between them are internal
- However, they are not transparent to software
 - Device drivers need to interact with all three for interrupts to work

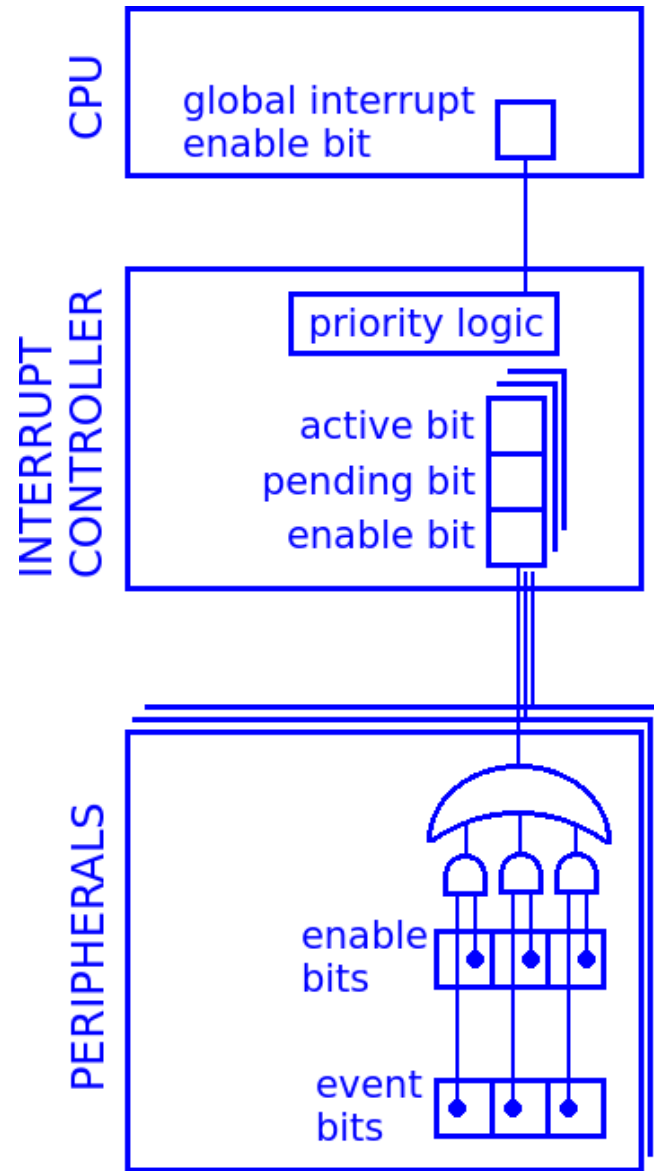
The CPU has a bit to enable/disable interrupts

In the ARM Cortex-M4 CPU, there are two special assembly instructions to modify this bit

“cpsid i” disables interrupts

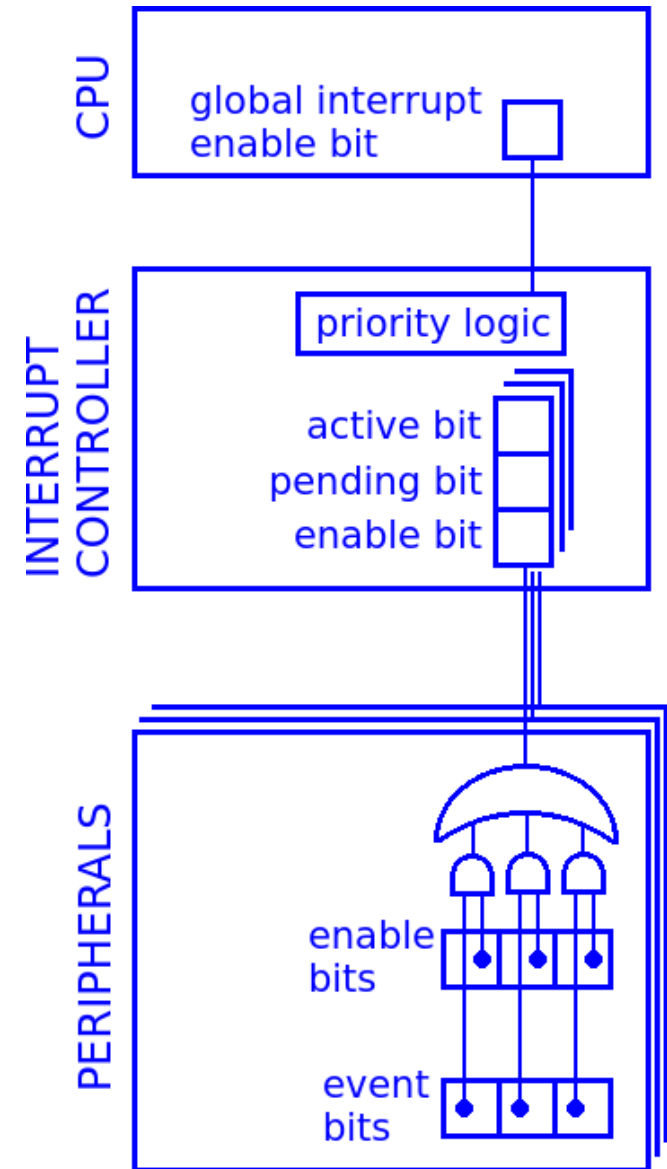
“cpsie i” enables interrupts

When the CPU is first powered up, interrupts are already enabled



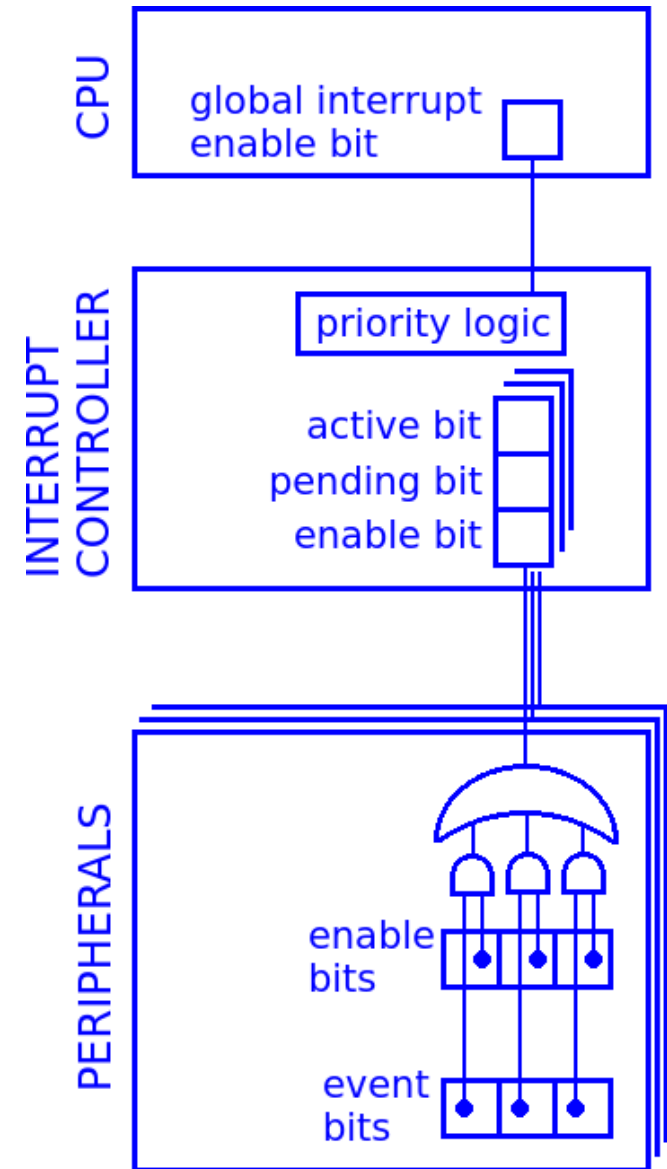
The interrupt controller has three bits for each peripheral

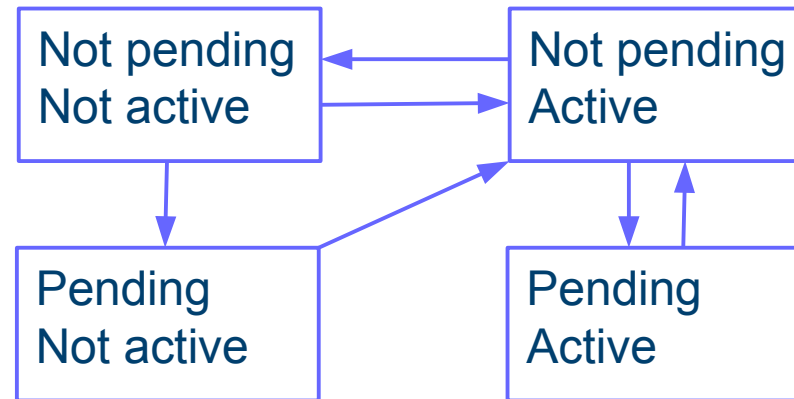
- If the enable bit is 0, the peripheral cannot generate interrupts
- The pending and active bit will be explained in detail later
 - The active interrupt is the one whose interrupt service routine is executing on the CPU
 - Pending interrupts are interrupts scheduled for execution
- If multiple interrupts are raised by different peripherals at the same time, a priority logic decides which interrupt is executed first
 - The other become pending



Peripherals often have multiple internal events that can generate an interrupt

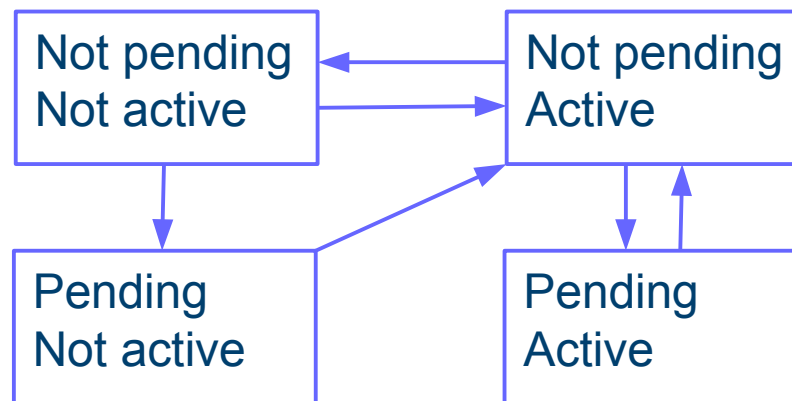
- Each event has an enable bit to let software decide which events should generate interrupts
- Each event has a status bit, signaling that the event occurred
 - If an interrupt is not enabled for that event, software can still do polling on the event bit
- A single line goes from the peripheral to the interrupt controller



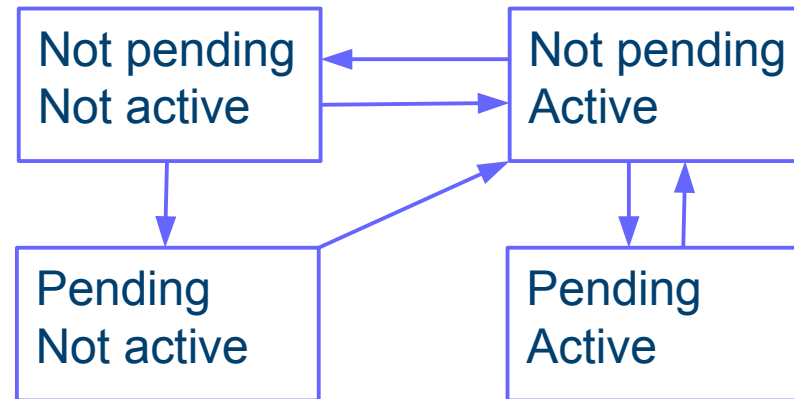


For each interrupt source, the interrupt controller implements a state machine with the pending and active bits

- Assume a peripheral signals an interrupt while the global enable bit is set, and the CPU is executing the main code
- The interrupt state goes from not pending, not active to not pending, active
 - It remains in this state as long as the CPU is executing the ISR
- After that, the interrupt state reverts back to not pending, not active



- Assume a peripheral signals an interrupt while the global enable bit is not set, or the CPU is executing another interrupt
- The interrupt state goes from not pending, not active to pending, not active
 - ♦ It remains in this state as long as the conditions that prevent the ISR execution persist
- After that, the interrupt state changes to active, not pending and the ISR is executed
- When the ISR completes, the state reverts back to not pending, not active



- If while the CPU is executing an ISR (thus, the state is active, not pending) the same peripheral signals again an interrupt
- The state goes to pending, active
- When the ISR completes its execution, the same ISR is called again, and the state changes to active, not pending

The last detail that we need to address is how the CPU locates the ISR to execute

- Remember the table that is placed at address 0 with the stack pointer and pointer to the function that is executed to begin the boot?
 - It is not actually made of just two entries
- After the first two entries, there are many more function pointers
- The first ones are for “system” interrupts, which are condition that occur inside the CPU
 - accessing an unmapped part of the address space, an invalid opcode being fetched, etc.
- Then, there are more function pointers, one for each peripheral interrupt
- The index into this table that corresponds to a specific peripheral, such as the USART depends on how the hardware designers who made the microcontroller connected the peripherals to the interrupt controller
 - Is documented in the manuals for the microcontroller

C++ name mangling, or how to call a C++ function from assembler code

- C++ supports function overloading, that is, letting programmers declare multiple functions with the same name, but different parameters
- C++ supports namespaces, which again allows to declare functions with the same name in multiple namespaces
- How can the linker resolve calls to functions with the same name?

The solution is name mangling

- The C++ compiler transforms the name of each function, “decorating” it with parameter information, and thus making an unique name
- This process is transparent to developers, unless they want to reference a C++ function from assembly code
 - Here we want to put a pointer to an interrupt service routine written in C++ in a table of function pointers written in assembler

C++ name mangling, or how to call a C++ function from assembler code

Let's consider this function:

```
void USART2_IRQHandler()
```

- Every C++ mangled name begins with “_Z”
- Then comes the number of characters of the function name
- Then comes the function name
- Finally, comes the list of parameters
 - All ISR take no parameters (remember, they're called by the hardware!)
 - The parameter list is thus “v” which stands for void
 - That's all we need to know to mangle ISR names
- The return value is not encoded (overloading the return value is not possible)
- The resulting mangled name is thus

```
_Z17USART2_IRQHandlerv
```

To write an interrupt-based driver for the serial port receive side

- We extend the assembler startup script to call our ISR
- We add a producer-consumer queue where the interrupt can push the received characters, and the main code can read them
- We extend the driver class to add accessor member functions to the queue
- In the class constructor, we add the necessary code to enable the interrupt

Extending the assembler startup script

The Reset_Handler function is identical to the previous ones

We add a function that is called when an entry in the interrupt table other than the ones we want is executed. We just have it loop forever

Then, we extend the __Vectors table with pointers to the unimplemented IRQ until we reach the index into the table for the USART2. There, we place the pointer to the C++ function

```
/* Reset_Handler function same as the previous
startup.s */

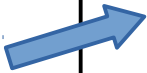
...

/* Unimplemented interrupt function. */
.global UnimplementedIrq
.type UnimplementedIrq, %function
UnimplementedIrq:
    b    UnimplementedIrq
.size   UnimplementedIrq, .-UnimplementedIrq

.section .isr_vector
.global __Vectors
__Vectors:
    .word _stack_top
    .word Reset_Handler
    .word UnimplementedIrq      /*NMI*/
    ...
    .word UnimplementedIrq      /*USART1*/
    .word _Z17USART2_IRQHandler /*USART2*/
```

Extending the driver interface

The serial port driver class is extended with an available member function that returns if the queue has at least one character, and a read member function that returns the first character in the queue, or blocks until a character arrives if the queue is empty



```
#ifndef SERIAL_H
#define SERIAL_H

class SerialPort
{
public:
    SerialPort();

    void write(const char *s);

    bool available() const;

    char read();
};

#endif //SERIAL_H
```

Adding the queue and writing the ISR

The queue is a hardcoded fixed length fifo, holding up to 16 characters

The ISR function checks if we have actually received a valid character (the interrupt is also called when errors occurred while receiving).

The character is enqueued if there is space

```
...
static const int bufsize=16;
static char rxbuffer[bufsize];
static int putpos=0;
static int getpos=0;
static volatile int numchar=0;

void USART2_IRQHandler()
{
    //Read status of usart peripheral
    unsigned int status=USART2->SR;

    //Read possibly received char
    char c=USART2->DR;

    //Did we receive a char?
    if(status & USART_SR_RXNE)
    {
        if(numchar==bufsize) return; //Buffer full
        rxbuffer[putpos]=c;
        if(++putpos >= bufsize) putpos=0;
        numchar++;
    }
}

...
```

Implementing the driver interface

The available member function just return if there are characters

The read member function polls the numchar variable, blocking if the queue is empty (client code can avoid this by checking available before calling read).

Then, it reads a character from the queue, and returns it

```
...
bool SerialPort::available() const
{
    return numchar>0;
}

char SerialPort::read()
{
    //Wait until the interrupt puts one char in
    //the buffer
    while(numchar==0) ;

    asm volatile("cpsid i"); //Disable interrupts
    char result=rxbuffer[getpos];
    if(++getpos >= bufsize) getpos=0;
    numchar--;
    asm volatile("cpsie i"); //Enable interrupts
    return result;
}
...
```

Synchronization between threads and interrupts

Notice the two “asm volatile” surrounding the code that read from the queue. The first disables the global interrupt enable bit in the CPU, the second one enables back interrupts.

Why is this necessary?

```
char SerialPort::read()
{
    while(numchar==0) ;
    asm volatile("cpsid i"); //Disable interrupts
    char result=rxbuffer[getpos];
    if(++getpos >= bufsize) getpos=0;
    numchar--;
    asm volatile("cpsie i"); //Enable interrupts
    return result;
}
```

- The queue is a data structure shared between the interrupts and the main code.
 - ♦ Just like with threads, race conditions can occur
- However, we can't use mutexes in this case
 - ♦ With a mutex the first thread entering the critical section “wins”, the other blocks
 - ♦ Threads are all created equal, it's ok if either of the two blocks
- When synchronizing an interrupt with the main code
 - ♦ An interrupt can interrupt the main code, but the reverse is not possible
 - ♦ Interrupts are always run to completion and must never block

Synchronization between threads and interrupts

To solve this issue, we disable interrupts during the critical section

- An interrupt either happens before interrupts are disabled
 - ♦ First the interrupt enter the critical section then the main code, no race condition
- Or if it occurs while interrupts are disabled, it becomes pending, and gets executed when interrupts are enabled again
 - ♦ First the main code enter the critical section, then the interrupt, no race condition
- The interrupt service routines and critical sections have to be kept as short as possible

Enabling interrupts in the constructor

GPIO alternate function configuration, same as before

In USART->CR1 we also set the RXNEIE bit, which tells the peripheral to signal an interrupt to the interrupt controller when a character is received.

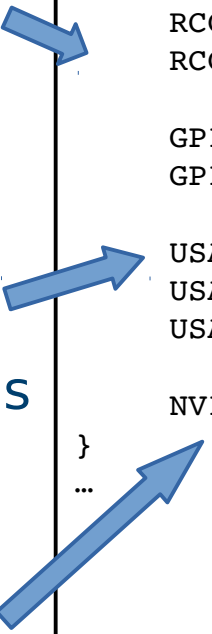
Then, we set the enable bit for the USART2 peripheral in the interrupt controller

```
...
SerialPort::SerialPort()
{
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIOAEN;
    RCC->APB1ENR |= RCC_APB1ENR_USART2EN;

    GPIOA->AFR[0] |= (7<<(2*4)) | (7<<(3*4));
    GPIOA->MODER |= (2<<(2*2)) | (2<<(3*2));

    USART2->CR1 = USART_CR1_UE | USART_CR1_RXNEIE;
    USART2->BRR = (52<<4) | (1<<0);
    USART2->CR1 |= USART_CR1_TE | USART_CR1_RE;

    NVIC->ISER[1]=1<<6;
}
...
```



Main code

The main code echoes back the received character, or if it is a +, it blinks the led 10 times.

Experiment: after writing a +, type other characters while the LED is blinking, what happens?

While the led is blinkine the main code is not consuming characters from the queue in the serial driver, so up to 16 characters can be placed in the queue. The remaining will be lost. When the led stops blinking, the characters in the queue are echoed back.

```
#include "registers.h"
#include "serial.h"
...
void blink() {
    for(int i=0;i<10;i++) {
        GPIOD->BSRR=1<<14;
        delay();
        GPIOD->BSRR=1<<(14+16);
        delay();
    }
}

SerialPort serial;

int main() {
    RCC->AHB1ENR |= RCC_AHB1ENR_GPIODEN;
    GPIOD->MODER |= 1<<28;
    bool led=false;
    For(;;) {
        char c=serial.read();
        if(c=='+') blink();
        else {
            char str[24]="Received chatacter:  \r\n";
            str[20]=c;
            serial.write(str);
        }
    }
}
```

In these slides we have seen, using a “by example” approach, how software can interact with hardware in the absence of an operating system.

Experiments performed in class using the stm32 have shown the presented code examples in operation.

The techniques presented here (accessing peripheral registers, polling and interrupts) can be applied broadly for other peripherals and other microcontrollers.

The techniques presented here are the basis of device driver development in an OS environment, where the interaction with context switches and the scheduler have to be considered.