ELEC3850 - Embedded Systems 1

STM32 I/O
Interrupts

Brenton Schulz

University of Newcastle

September 15, 2020

Summary

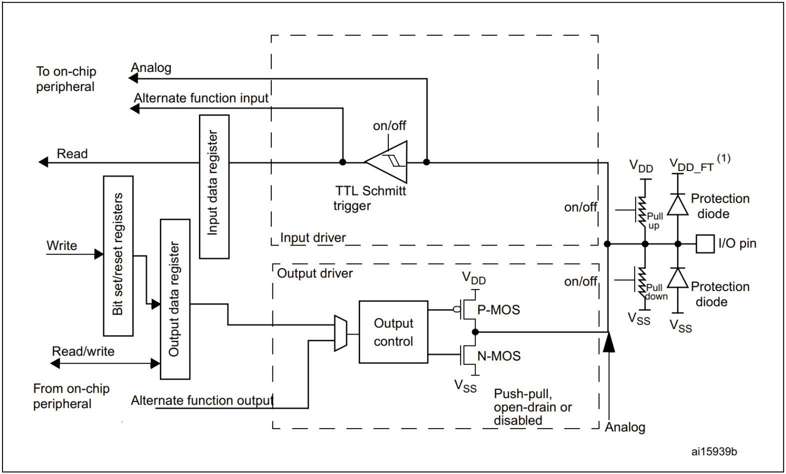
- Software driven GPIO
- Peripherals
    - SPI
    - I2C
    - UART
- Interrupts

Levels of Understanding

- Fundamental electronics
  - Transistors drive pins
  - What is push-pull Vs. open drain?
  - What are "pull-ups"?
- Datasheet / reference manual
  - Low-level configuration registers drive the GPIO circuit
- CubeMX
  - How does the datasheet translate to CubeMX?
  - In general: CubeMX assumes you have read the reference manual
- HAL
  - Using the HAL with confidence requires an understanding over everything above

GPIO Hardware

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



1. V_{DD_FT} is a potential specific to five-volt tolerant I/Os and different from V_{DD}.

GPIO Hardware

- STM32 pins can be configured as:
  - Digital outputs
    - Push-pull
    - Open drain
  - Digital inputs
    - With or without pull-up
    - With or without pull-down
  - Alternate Function I/Os
    - Outputs can also push-pull or open drain
    - AF inputs are analog when they drive internal ADCs
- STM32 outputs also have output bandwidth control
- NB: Outputs can have pull up/down enabled
  - This wastes a small amount of power
  - Useful if a pin swaps between input and output

GPIO Control Bits

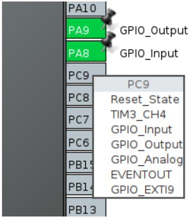
Table 35. Port bit configuration table(1)

MODER(i) [1:0]	OTYPER(i)	OSPEEDR(i) [B:A]	PUPDR(i) [1:0]		I/O configuration	
	0		0	0	GP output	PP

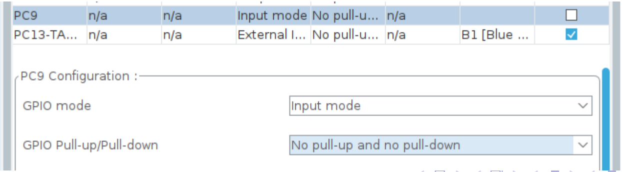
- Control bits:
  - MODER - Input/output control
  - OTYPER - Push-pull or open drain
  - OSPEEDR - Output bandwidth limiting
  - PUPDR - Pull up/down enable bits
- These control bits are packed into 32-bit GPIO registers:
  - GPIOx_MODER
  - GPIOx_OTYPER
  - GPIOx_OSPEEDR
  - GPIOx_PUPDR

CubeMX

- To configure GPIOs in CubeMX click pins and select a mode:



- Pull mode is then found in the left panel:



HAL Translation

- Open STM32CubeIDE, perform example configuration
- Observe:
  - `GPIO_TypeDef` - compare to GPIO registers in reference manual
  - `HAL_GPIO_*` functions in `stm32f4xx_hal_gpio.h`
  - HAL `GPIO_PinState` datatype

Interrupts Review

- An *interrupt* is an event which causes the CPU to:
  - Stop what it was doing
  - Execute an *interrupt service routine* (ISR)
  - Resume what it was doing
- An ISR is a C function with a specific name
  - These names are listed in the HAL in `startup_xxx.s`
  - The memory locations of the ISRs get packed into the interrupt vector table - at the start of program memory on an ARM CPU
  - By default ISRs are declared `weak` so you can define your own versions without removing the "default" versions

Interrupts Review

- ARM CPUs use a nested vector interrupt controller (NVIC) to control the interrupt process
- The *nested* behaviour is the ability of interrupts to trigger (and their ISRs execute) while another ISR is already executing
  - Interrupts have *priorities* which control when they can preempt each other
- The "vector" term relates to ISRs having unique program memory addresses which are jumped to when an interrupt triggers

GPIO Interrupts

- STM32s can have interrupts triggered by GPIO state changes:
  - Rising
  - Falling
  - Both
- There are 16 GPIO external interrupt (`EXTI`) sources on most STM32s
  - Each can be triggered off 1 GPIO pin as-per Figure 42, p382 of RM0090
  - tl;dr: `PORTx`N triggers `EXTI`N
    - eg: PORTB2 can trigger EXTI2

GPIO Interrupts

- The possible interrupt vectors are:
  - EXTI0
  - EXTI1
  - EXTI2
  - EXTI3
  - EXTI4
  - EXTI9_5
  - EXTI10_15
- ie: GPIO pins 0-4 have unique interrupts, the others trigger a “grouped” interrupt
  - The ISR needs to determine which pin triggered the interrupt

GPIO Interrupts

- To use an external GPIO interrupt:
  1. Configure the pin as EXTI in CubeMX
  2. Write the ISR function
  3. (Optional) Set the interrupt priority
     - This is a big topic - not covered in this lecture
  4. Enable the interrupt

GPIO Interrupts - Demonstration

- Observe Nucleo-F103RB project
- Note it includes `EXTI15_10_IRQHandler()`
  - It did once then disappeared - CubeMX is weird
- This function, in turn, calls `HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13)`
  - The HAL includes other interrupt functions with various names - these are **NOT** interrupt service routines called by the NVIC
- If the `EXTI*_IRQHandler()` function does not exist you need to write it
  - It is the function *actually called* by the NVIC when the interrupt triggers

GPIO Interrupts

- Very few interrupts are enabled by default!
- Enable interrupts with `HAL_NVIC_EnableIRQ()`
  - eg: `HAL_NVIC_EnableIRQ(EXTI15_10_IRQn)`

GPIO Interrupts - Demonstration

- Crucial note 2: EXTIs are not cleared by hardware!
  - Software must clear the appropriate interrupt flag by writing a 1 to the correct bit in `EXTI_PI`
  - Recommended to use `__HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN);`
- Crucial note 3: When using shared EXTI interrupts use `__HAL_GPIO_EXTI_GET_IT(GPIO_PIN)` to test which EXTI triggered the ISR
- `stm32f1xx_hal_gpio.h` must be included for both the macro and `GPIO_PIN` definitions

UART

- Review
  - The UART is a serial communications device
  - It is *asynchronous* - only a data signal is sent
    - The clock rate is configured at sender and receiver
    - The start of a transmission is synchronised with a “start bit”
  - UARTs are *byte oriented* - they send or receive 8-bits
- Demonstration: Configuration of a UART and code required to use `printf()` on a NUCLEO-F103RB
- NB: STM32s contain UARTs and USARTs - the synchronous USART hardware can be configured with a clock pin but that will not be demonstrated here

UART - CubeMX Configuration

- Creating a project with the NUCLEO-F103RB automatically configures USART2 for pins PA2 (Tx) and PA3 (Rx)
  - These pins are connected to a virtual COM port on the ST-Link debugger - other boards may require a USB to UART adapter

UART - libc

- Low level *system calls* needs to be written to handle data reads and writes
- The `newlib` embedded version of `glibc` contains `printf()`
- `printf()` eventually calls `__io_putchar()` to write characters
- `__io_putchar()` is declared `weak` in `syscalls.c` - we need to write our own
  - You can also write a `__io_putchar()` version which prints to other hardware, such as an LCD screen, i2c peripheral, etc

UART - libc

- We will write `__io_putchar()` to write characters to the USART2
- This will use `HAL_UART_Transmit()`
  - `HAL_UART_Transmit()` is *blocking* - it won't return until data has been sent
  - Non-blocking and interrupt-driven methods use far less CPU time