General Purpose I/O

Copyright © ARM Ltd 2013

Overview

- **How do we make a program light up LEDs in response to a switch?**

- **GPIO**
  - Basic Concepts
  - Port Circuitry
  - Control Registers
  - Accessing Hardware Registers in C
  - Clocking and Muxing

- **Circuit Interfacing**
  - Inputs
  - Outputs

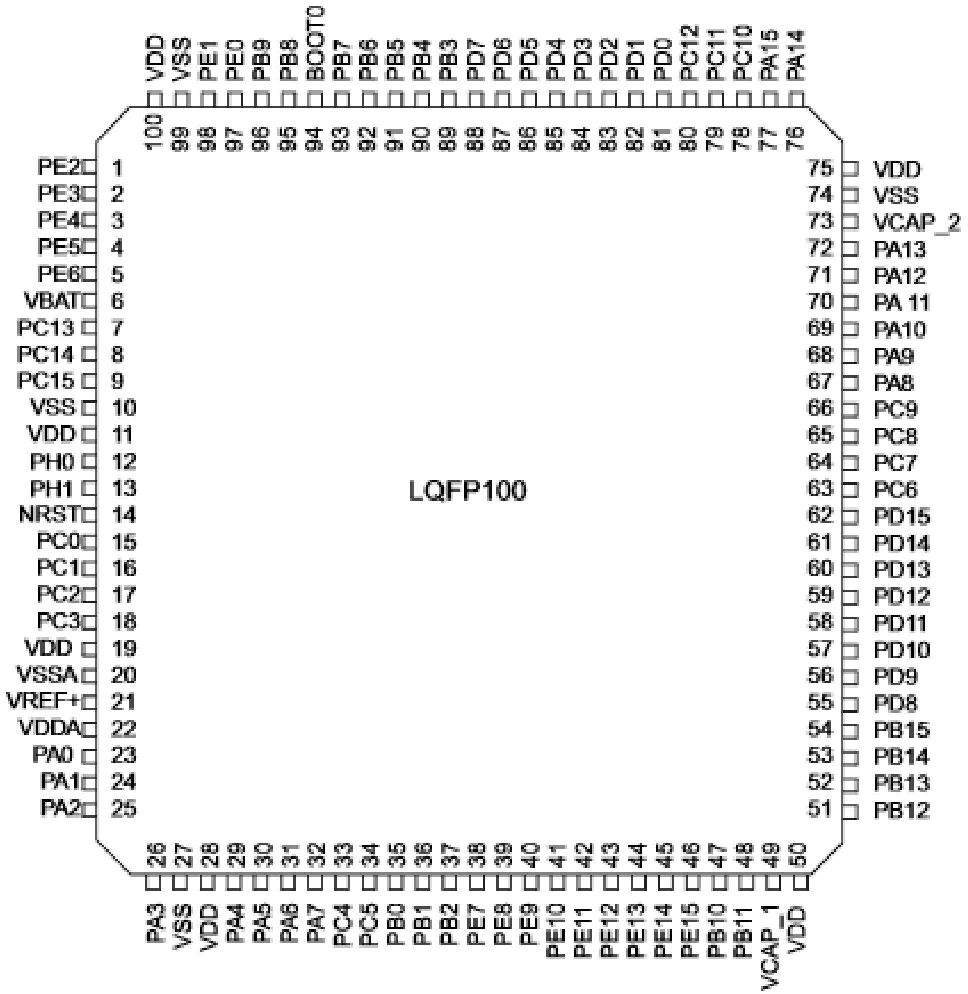
- **Additional Configuration**

Basic Concepts

- **GPIO = General-purpose input and output (digital)**
  - Input: program can determine if input signal is a 1 or a 0
  - Output: program can set output to 1 or 0
- **Can use this to interface with external devices or on board peripherals**
  - Input: switch, button……
  - Output: LEDs, speaker……

STM32F40x LQFP100 pinout

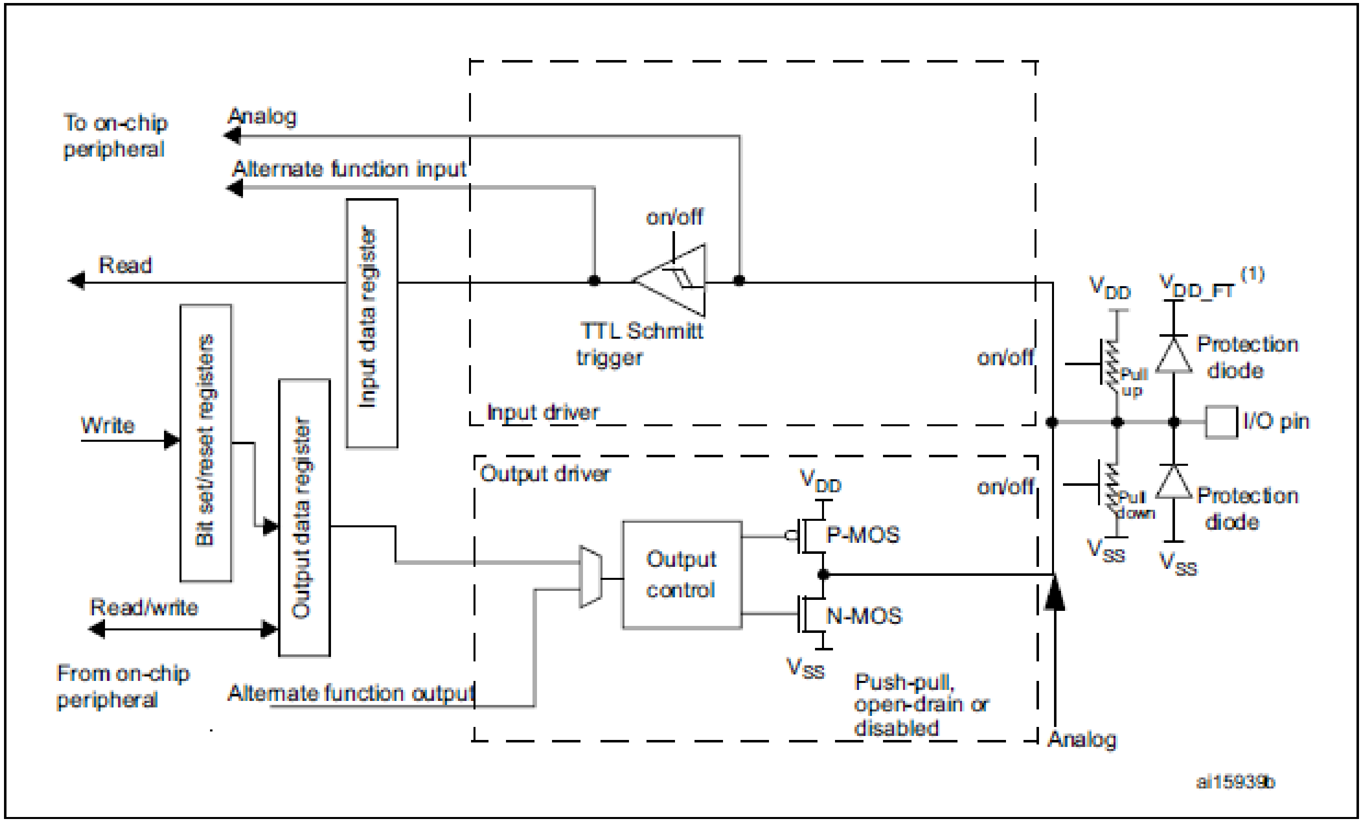
- Port A (PA) through Port E (PE)
- Not all port bits are available
- Quantity depends on package pin count



Copyright © ARM Ltd 2013

GPIO Port Bit Circuitry in MCU

- **Configuration**
  - Direction
  - MUX
  - Modes
  - Speed
- **Data**
  - Output (different ways to access it)
  - Input
  - Analogue
- **Locking**



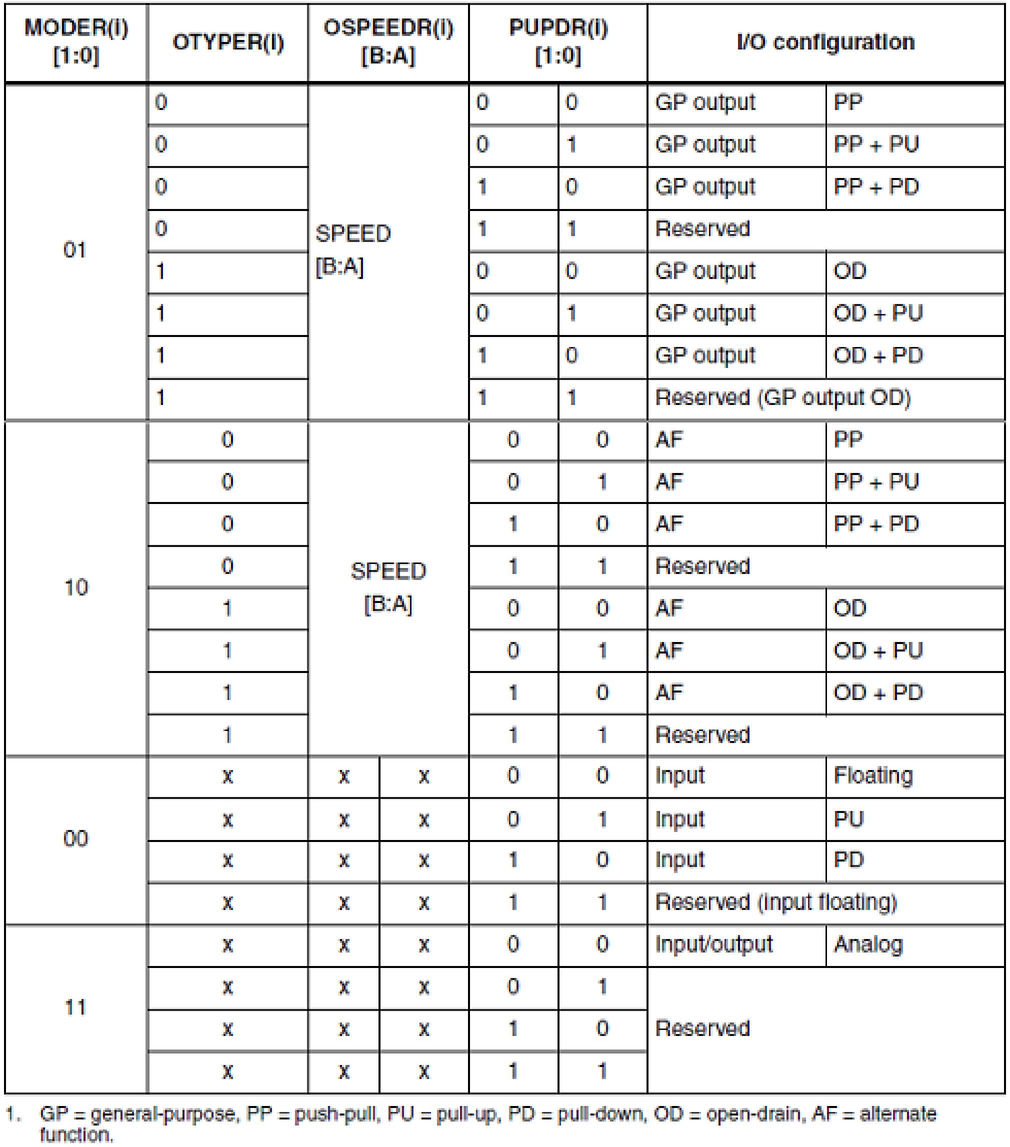
Control Registers

- Each general-purpose I/O port has
  - four 32-bit configuration registers (
    - GPIOx_MODER (input, output, AF, analog)
    - GPIOx_OTYPER (output type: push-pull or open drain)
    - GPIOx_OSPEEDR(speed)
    - GPIOx_PUPDR(pull-up/pull-down)
  - two 32-bit data registers(GPIOx_IDR and GPIOx_ODR)
  - a 32-bit set/reset register (GPIOx_BSRR)
  - a 32-bit locking register (GPIOx_LCKR)
  - two 32-bit alternate function selection register (GPIOx_AFRH and GPIOx_AFRL)
- **One set of control registers (10 in total) per port**
- **Each bit in a control register corresponds to a port bit**
- **All registers have to be accessed as 32-bit word**

Copyright © ARM Ltd 2013

GPIO Configuration registers

- **Each bit can be configured differently**
- **Reset clears port bit direction to 0**
- Output modes: push-pull or open drain + pull-up/down
- Output data from output data register (GPIOx_ODR) or peripheral (alternate function output)
- Input states: floating, pull-up/down, analog
- Input data to input data register (GPIOx_IDR) or peripheral (alternate function input)



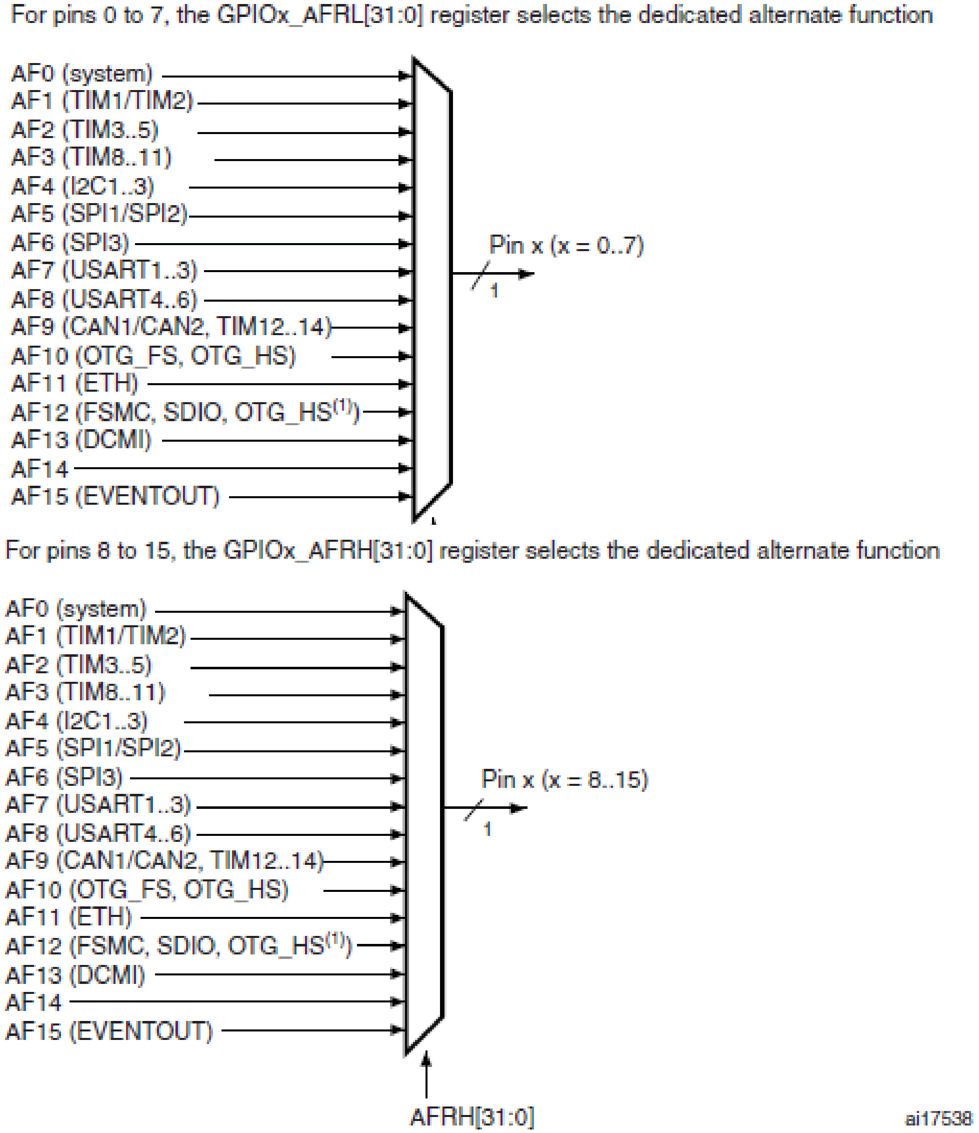
MODER(i) [1:0]	OTYPER(i)	OSPEEDR(i) [B:A]		PUPDR(i) [1:0]		I/O configuration	
01	0	SPEED [B:A]		0	0	GP output	PP
	0			0	1	GP output	PP + PU
	0			1	0	GP output	PP + PD
	0			1	1	Reserved	
	1			0	0	GP output	OD
	1			0	1	GP output	OD + PU
	1			1	0	GP output	OD + PD
	1			1	1	Reserved (GP output OD)	
10	0	SPEED [B:A]		0	0	AF	PP
	0			0	1	AF	PP + PU
	0			1	0	AF	PP + PD
	0			1	1	Reserved	
	1			0	0	AF	OD
	1			0	1	AF	OD + PU
	1			1	0	AF	OD + PD
	1			1	1	Reserved	
00	x	x	x	0	0	Input	Floating
	x	x	x	0	1	Input	PU
	x	x	x	1	0	Input	PD
	x	x	x	1	1	Reserved (input floating)	
11	x	x	x	0	0	Input/output	Analog
	x	x	x	0	1	Reserved	
	x	x	x	1	0		
	x	x	x	1	1		

1. GP = general-purpose, PP = push-pull, PU = pull-up, PD = pull-down, OD = open-drain, AF = alternate function.

Alternate function selection register

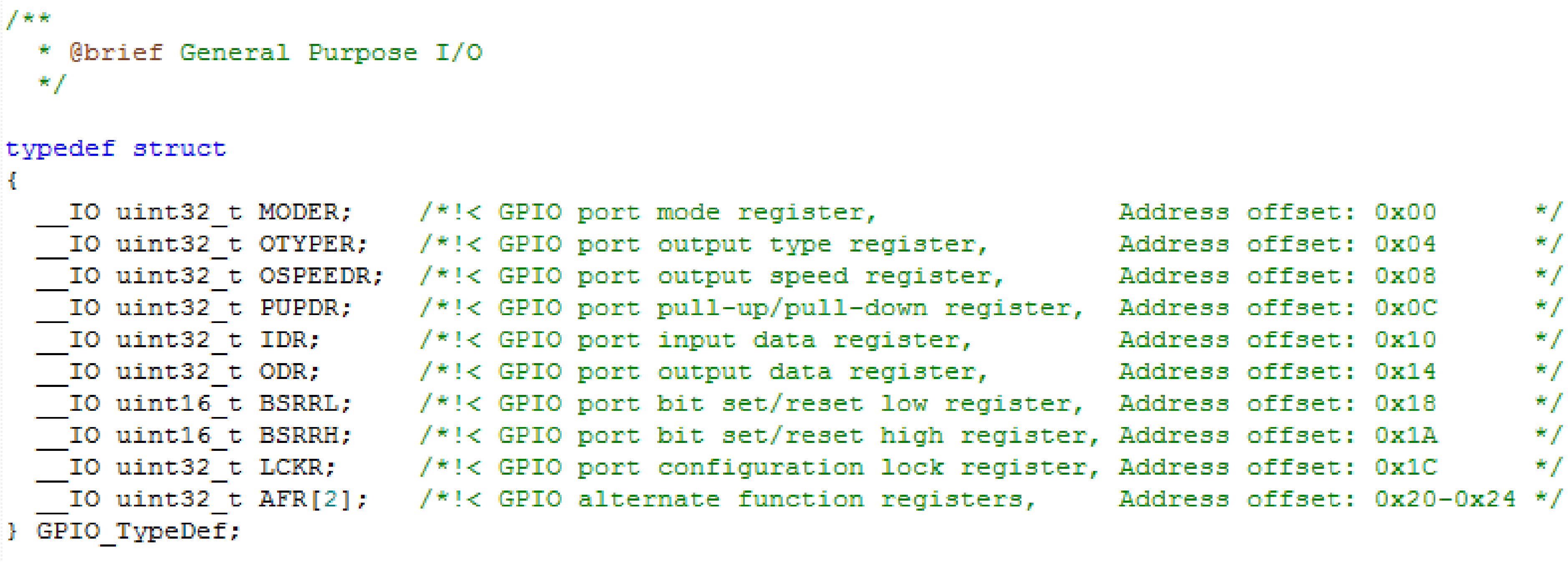
- **In AF mode, AFRL or AFRH needs to be configured to be driven by specific peripheral**
- **Can be seen as a select signal to the Mux**

- EVENTOUT is not mapped onto the following I/O pins: PC13, PC14, PC15, PH0, PH1 and PI8.



CMSIS - Accessing Hardware Registers in C

- **Header file stm32f4xx.h defines C data structure types to represent hardware registers in MCU with CMSIS-Core hardware abstraction layer**



```
/**
  * @brief General Purpose I/O
  */

typedef struct
{
  __IO uint32_t MODER;    /*!< GPIO port mode register,               Address offset: 0x00      */
  __IO uint32_t OTYPER;   /*!< GPIO port output type register,        Address offset: 0x04      */
  __IO uint32_t OSPEEDR;  /*!< GPIO port output speed register,       Address offset: 0x08      */
  __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register,  Address offset: 0x0C      */
  __IO uint32_t IDR;      /*!< GPIO port input data register,         Address offset: 0x10      */
  __IO uint32_t ODR;      /*!< GPIO port output data register,        Address offset: 0x14      */
  __IO uint16_t BSRRL;    /*!< GPIO port bit set/reset low register,  Address offset: 0x18      */
  __IO uint16_t BSRRH;    /*!< GPIO port bit set/reset high register, Address offset: 0x1A      */
  __IO uint32_t LCKR;     /*!< GPIO port configuration lock register, Address offset: 0x1C      */
  __IO uint32_t AFR[2];   /*!< GPIO alternate function registers,     Address offset: 0x20-0x24 */
} GPIO_TypeDef;
```

CMSIS C Support

- Header file stm32f4xx.h defines pointers to GPIO_Type registers

```
#define GPIOA_BASE                (AHB1PERIPH_BASE + 0x0000)
#define GPIOB_BASE                (AHB1PERIPH_BASE + 0x0400)
#define GPIOC_BASE                (AHB1PERIPH_BASE + 0x0800)
#define GPIOD_BASE                (AHB1PERIPH_BASE + 0x0C00)
#define GPIOE_BASE                (AHB1PERIPH_BASE + 0x1000)
#define GPIOF_BASE                (AHB1PERIPH_BASE + 0x1400)
#define GPIOG_BASE                (AHB1PERIPH_BASE + 0x1800)
#define GPIOH_BASE                (AHB1PERIPH_BASE + 0x1C00)
#define GPIOI_BASE                (AHB1PERIPH_BASE + 0x2000)
......
#define AHB1PERIPH_BASE           (PERIPH_BASE + 0x00020000)
......
#define PERIPH_BASE               ((uint32_t)0x40000000)
```

Clocking Logic

- Need to enable clock to GPIO module
- By default, GPIO modules are disabled to save power
- Writing to an unclocked module triggers a hardware fault!
- Control register RCC_AHB1ENR gates clocks to GPIO ports
- Enable clock to Port D

```
RCC->AHB1ENR|= (1UL <<  3);
```

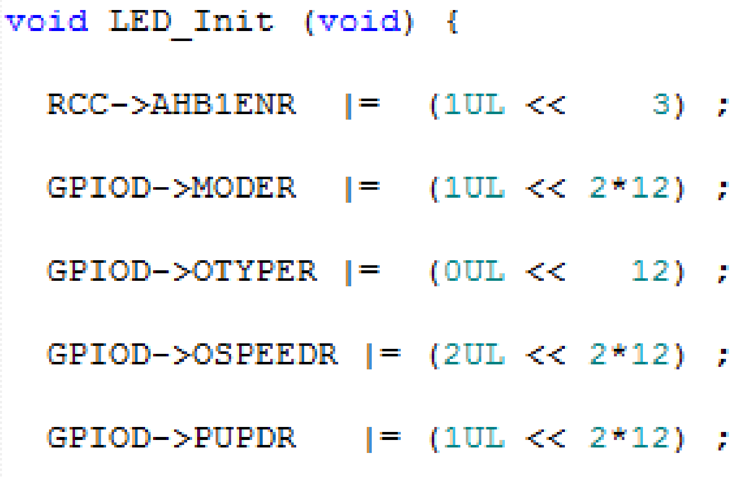
- Header file stm32f4xx.h has definitions

```
RCC->AHB1ENR|=RCC_AHB1ENR_GPIODEN;
```

Copyright © ARM Ltd 2013

Initializing GPIO

- **Enable clock for Port**
- **Set the mode**
- **Set the Output type**
- **Set the speed**
- **Set the pull-up or pull down**
- **Set the AF**
- **Not all of these are necessary, default setting is ok (usually all bits cleared after reset)**
- **Need to access the entire 32 registers**
- **Simple example for initializing the orange led on the board**
  - Port D pin 12



```
void LED_Init (void) {

  RCC->AHB1ENR   |=  (1UL <<    3) ;

  GPIOD->MODER   |=  (1UL << 2*12) ;

  GPIOD->OTYPER  |=  (0UL <<   12) ;

  GPIOD->OSPEEDR  |= (2UL << 2*12) ;

  GPIOD->PUPDR    |= (1UL << 2*12) ;
```

CMSIS C Support

- **Header file stm32f4xx.h also has bits definition for GPIO register**

```
#define GPIO_MODER_MODER0                    ((uint32_t)0x00000003)
#define GPIO_MODER_MODER0_0                  ((uint32_t)0x00000001)
#define GPIO_MODER_MODER0_1                  ((uint32_t)0x00000002)

   #define GPIO_OTYPER_OT_0                     ((uint32_t)0x00000001)

   #define GPIO_OSPEEDER_OSPEEDR0              ((uint32_t)0x00000003)
   #define GPIO_OSPEEDER_OSPEEDR0_0            ((uint32_t)0x00000001)
   #define GPIO_OSPEEDER_OSPEEDR0_1            ((uint32_t)0x00000002)

   #define GPIO_PUPDR_PUPDR0                   ((uint32_t)0x00000003)
   #define GPIO_PUPDR_PUPDR0_0                 ((uint32_t)0x00000001)
   #define GPIO_PUPDR_PUPDR0_1                 ((uint32_t)0x00000002)
```

Writing/Reading Output/Input Port Data

- **Direct: write value GPIOx_ODR**
- **Clear (to 0): Write 1 to BSRRL**
- **Set (to 1): write 1 to BSRRH**
  - GPIOD->ODR|=(1<<12);
  - Equivalent to: GPIOD->BSRRL=(1<<12);
  - Or with CMSIS: GPIOD-ODR|= GPIO_ODR_ODR_12
  - GPIOD->ODR&=~(<<12);
  - Equivalent to: GPIOD->BSRRH=(1<<12);
  - Or with CMSIS: GPIOD-ODR&=~GPIO_ODR_ODR_12

- **Read from IDR**
  - data=GPIOD->IDR&(1<<12)
  - Or with CMSIS: data=GPIOD->IDR&GPIO_IDR_IDR_12

Copyright © ARM Ltd 2013

Coding Style and Bit Access

- **Easy to make mistakes dealing with literal binary and hexadecimal values**
  - "To set bits 13 and 19, use 0000 0000 0000 1000 0010 0000 0000 0000 or 0x00082000"

- **Make the literal value from shifted bit positions**

```
n = (1UL << 19) | (1UL << 13);
```

- **Define names for bit positions**

```
#define POS_0 (13)
#define POS_1 (19)
n = (1UL << POS_0) | (1UL << POS_1);
```

- **Create macro to do shifting to create mask**

```
#define MASK(x) (1UL << (x))
n = MASK(POS_0) | MASK(POS_1);
```

Using Masks

- **Overwrite existing value in n with mask**

```
n = MASK(foo);
```

- **Set in n all the bits which are one in mask, leaving others unchanged**

```
n |= MASK(foo);
```

- **Complement the bit value of the mask**

```
~MASK(foo);
```

- **Clear in n all the bits which are zero in mask, leaving others unchanged**

```
n &= MASK(foo);
```

Using Masks with CMSIS

- #define SET_BIT(REG, BIT)   ((REG) |= (BIT))
- #define CLEAR_BIT(REG, BIT)   ((REG) &= ~(BIT))
- #define READ_BIT(REG, BIT)   ((REG) & (BIT))
- #define CLEAR_REG(REG)   ((REG) = (0x0))
- #define WRITE_REG(REG, VAL)   ((REG) = (VAL))
- #define READ_REG(REG)   ((REG))
- #define MODIFY_REG(REG, CLEARMASK, SETMASK) WRITE_REG((REG), (((READ_REG(REG)) & (~(CLEARMASK))) | (SETMASK)))
- `BIT = MASK(foo);`

C Code

```
#define LED1_POS (13)
#define LED2_POS (14)
#define SW1_POS (0)
#define MASK(x) (1UL << (x))
RCC->AHB1ENR|=RCC_AHB1ENR_GPIODEN;

/* Initialization of GPIO */

GPIOD->ODR = MASK(LED1_POS);  // turn on LED1, turn off LED2

while (1) {
  if (GPIOD->IDR & MASK(SW1_POS)) {
    // switch is pressed, then light LED 2
    GPIOD->BSRRL = MASK(LED2_POS);
    GPIOD->BSRRH = MASK(LED1_POS);
  } else {
    // switch is pressed, so light LED 1
    GPIOD->BSRRL = MASK(LED1_POS);
    GPIOD->BSRRH = MASK(LED2_POS);
  }
}
```

Atomic Access

- Unlike some of other MCU, the AHB1 on STM32F4Discovery provides atomic access to one or more bits.
- Which means do not have to disable the interrupt when programming the GPIOx_ODR at bit level.

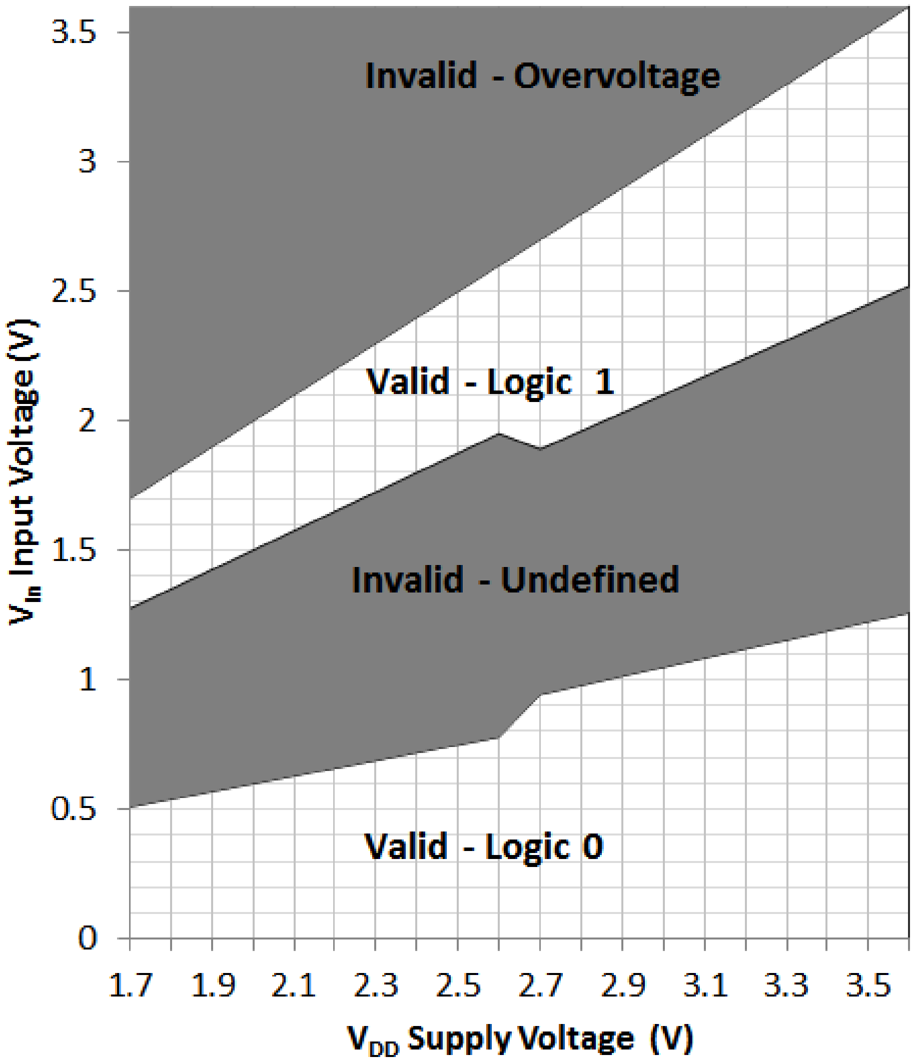
Inputs and Outputs, Ones and Zeros, Voltages and Currents

INTERFACING

Copyright © ARM Ltd 2013

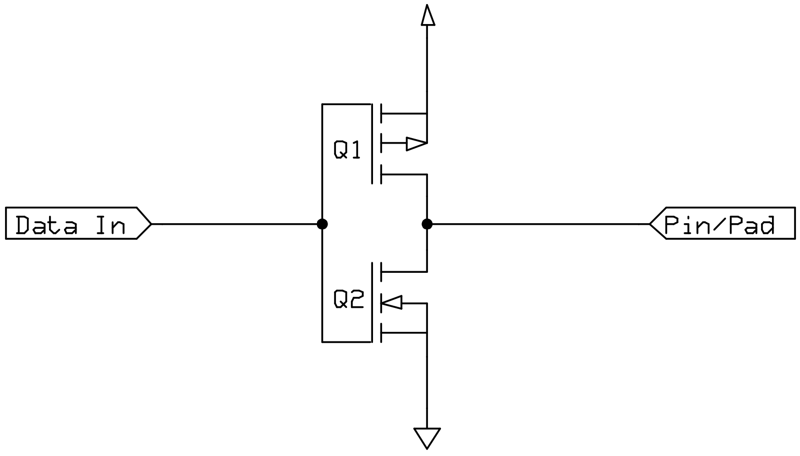
Inputs: What’s a One? A Zero?

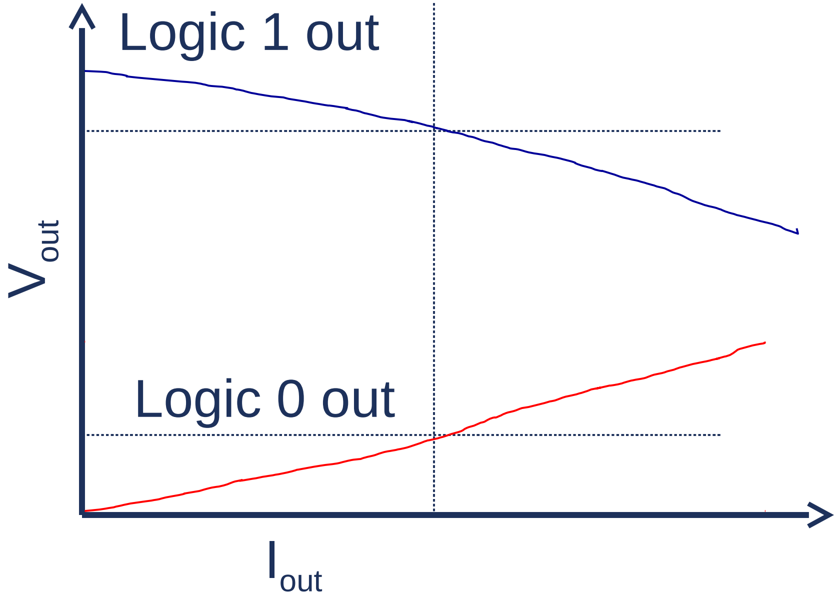
- Input signal’s value is determined by voltage
- Input threshold voltages depend on supply voltage V_{DD}
- Exceeding V_{DD} or GND may damage chip



Outputs: What's a One? A Zero?

- **Nominal output voltages**
  - 1: V_{DD}-0.5 V to V_{DD}
  - 0: 0 to 0.5 V
- **Note: Output voltage depends on current drawn by load on pin**
  - Need to consider source-to-drain resistance in the transistor
  - Above values only specified when current < 5 mA (18 mA for high-drive pads) and V_{DD} > 2.7 V

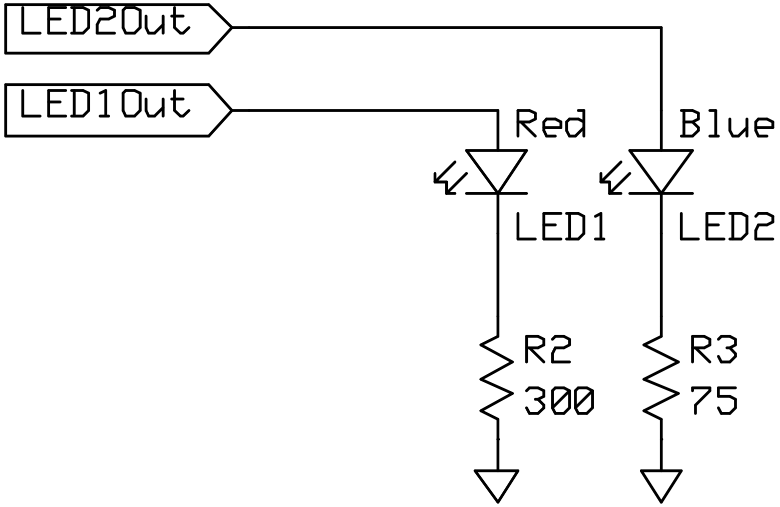




Copyright © ARM Ltd 2013

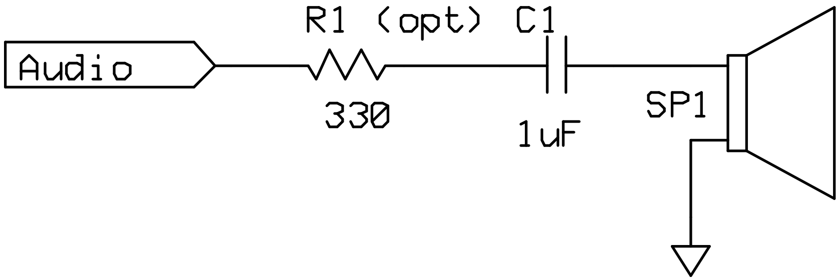
Driving External LEDs

- **Need to limit current to a value which is safe for both LED and MCU port driver**
- **Use current-limiting resistor**
  - $R = (V_{DD} – V_{LED})/I_{LED}$
- **Set I_{LED} = 4 mA**
- **V_{LED} depends on type of LED (mainly color)**
  - Red: ~1.8V
  - Blue: ~2.7 V
- **Solve for R given VDD = ~3.0 V**
  - Red: 300 Ω
  - Blue: 75 Ω

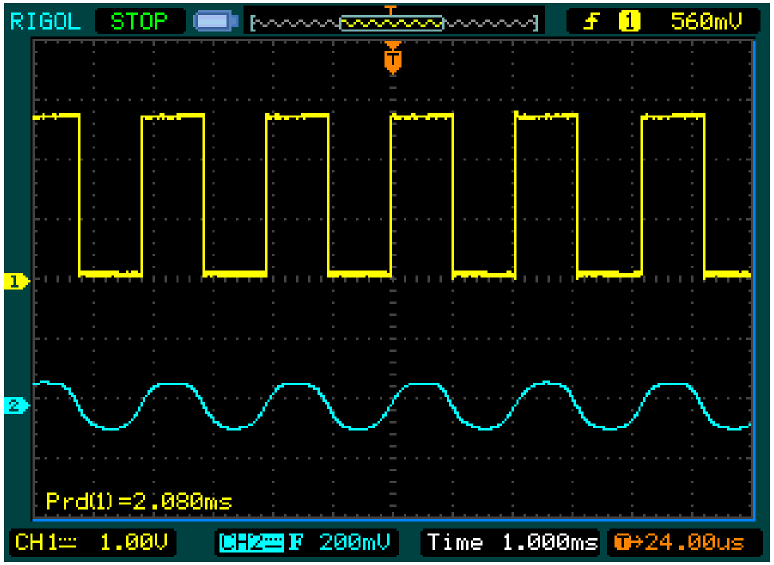


Output Example: Driving a Speaker

- Create a square wave with a GPIO output
- Use capacitor to block DC value
- Use resistor to reduce volume if needed



```
void Speaker_Beep(uint32_t frequency){
    Init_Speaker();
    while(1){
            GPIOD->BSRRL=(MASK(2));
            Delay(frequency);
            GPIOD->BSRRH=(MASK(2));
            Delay(frequency);
    }
}
```


RIGOL
STOP
560mV
Prd(1)=2.080ms
CH1 1.00V
CH2 200mV
Time 1.000ms
T→24.00us