Understanding STM32 ARM Microcontroller GPIOs

VISHNU MOHANAN / 19 OCTOBER 2018 / ELECTRONICS, TUTORIALS

When using any STM32 microcontrollers, GPIOs need to be initialized before you can use them in the application program. The GPIO peripheral is configured and controlled using a set of registers. Multiple registers are assigned to each port available in an STM32 MCU. These registers can be directly accessed and programmed but it can become a tedious task. To make it easier, we can use the **HAL (Hardware Abstraction Layer)** driver provided by ST or the **CMSIS (Cortex Microcontroller Software Interface Standard)** driver provided by ARM. The initialization code can be generated by **STM32CubeMX** application as per user configuration. In this tutorial we will see how to initialize GPIO directly using registers and do some port manipulations with it.

I must assume that you know about things like how data is represented in memory, and how individual in bits are manipulated with logical operators. I should also warn that this is not a comprehensive article on bit manipulation, but more of a starting guide if you are new to HAL or CMSIS.

Following are the registers associated with each GPIO port. Each port on STM32 can have up to 16 GPIO pins. A particular MCU variant can have multiple such ports. I hope you have some understanding of what each port is intended for. If not, try reading the part of the datasheet where the registers are explained.

Register	Description	Size in bits	Configuration Length in bits
GPIOx_MODER	GPIO port mode register	32	2
GPIOx_OTYPER	GPIO port output type register	16	1
GPIOx_OSPEEDR	GPIO port output speed register	32	2
GPIOx_PUPDR	GPIO port pull-up/pull-down register	32	2
GPIOx_IDR	GPIO port input data register	16	1
GPIOx_ODR	GPIO port output data register	16	1
GPIOx_BSRR	GPIO port bit set/reset register	32	1
GPIOx_BRR	GPIO port bit reset register	16	1
GPIOx_LCKR	GPIO port configuration lock register	16	1
GPIOx_AFRL	GPIO alternate function low register	32	4
GPIOx_AFRH	GPIO alternate function high register	32	4

The configuration length is simply the number of bits required to configure the state of a single pin. For 16-bit registers, it is usually a single bit and two bits for 32-bit registers. In software, you can access these registers as,

```
GPIOx->MODER
GPIOx->OTYPER
GPIOx->OSPEEDR
GPIOx->PUPDR
GPIOx->IDR
GPIOx->ODR
GPIOx->BSRR
GPIOx->BRR
GPIOx->LCKR
GPIOx->AFRL
GPIOx->AFRH
```

Where “**x**” can be the port identifier starting from letter A. For example if you want to set pin 5 of port A, ie PA5, as output, you can do so by,

```
GPIOA->MODER |= (1U << 5);
```

`(1U << 5)` simply means “shift unsigned integer 1 to 5 positions left”. This has the effect of writing `0b01` (`0b` is the prefix used to indicate a binary literal similar to `0x` for hex literal) to configuration bit pair of 5th pin of port A. `0b01` means the pin will be set as an output. The `|=` means the compiler is going to generate a “**Read Modify Write**” (RMW) sequence of instructions to,

1. First read the register
2. Use the source value to compute a new value.
3. And write the computed value back to the same register.

There are three steps involved. This “read-modify-write+ sequence can be bad at sometimes, and to know why, have a look at this page –

http://techref.massmind.org/techref/readmodwrite.htm. There is a better way to avoid RMW errors, which will be discussed further below.

As a programmer, you will most likely add multiple definitions to a header file to make all register operations easier, which is something already done by STM developers. For example `GPIOx->MODER` can accept the following values,

```
GPIO_MODER_MODER5_Pos     -  the relative position of the reg
GPIO_MODER_MODER5_Msk     -  a 0b11 binary mask pair
GPIO_MODER_MODER5         -  a 0b11 binary mask pair
GPIO_MODER_MODER5_0       -  a 0b01 binary set mask
GPIO_MODER_MODER5_1       -  a 0b10 binary set mask
```

The `5` is obviously to indicate the pin and therefore can have values from 0 to 15 for each pin of a port. The definitions of those values are,

```
#define GPIO_MODER_MODER5_Pos    (10U)
#define GPIO_MODER_MODER5_Msk    (0x3U << GPIO_MODER_MODER5_Po
#define GPIO_MODER_MODER5        GPIO_MODER_MODER5_Msk
#define GPIO_MODER_MODER5_0      (0x1U << GPIO_MODER_MODER5_Po
#define GPIO_MODER_MODER5_1      (0x2U << GPIO_MODER_MODER5_Po
```

These values are defined in the part specific header file in the `CMSIS\Device\ST\` folder for each MCU. The above snippets are copied from `\CMSIS\Device\ST\STM32F4xx\Include\stm32f446xx.h`.

If you have ever used bit manipulation in your programs, things would be clear by now on using those definitions accordingly. `GPIO_MODER_MODER5_Pos` is the relative position of a **bit pair** assigned to 5th pin of port A. For the first pair, the relative position would be zero. So we're cleverly using the shift operator to set the binary values in a register for each pin. Notice the relative position is defined as an `unsigned int 10U` ("u" suffix is to indicate a unsigned int literal) while the rest of the numbers as hex values. This is just arbitrary.

Now let’s see how we can configure the register using these definitions.

To set a bit pair to `0b11` simultaneously,

```
GPIOA->MODER |= GPIO_MODER_MODER5;
```

To clear a bit pair to `0b00` simultaneously,

```
GPIOA->MODER &= ~(GPIO_MODER_MODER5);
```

To set only the first bit to `0b1` while the second bit is unchanged,

```
GPIOA->MODER |= GPIO_MODER_MODER5_0;
```

To clear the first bit to `0b0` while the second bit is unchanged,

```
GPIOA->MODER &= ~(GPIO_MODER_MODER5_0);
```

To set the second bit to `0b1` while the first bit is unchanged,

```
GPIOA->MODER |= GPIO_MODER_MODER5_1;
```

To clear the second bit to `0b0` while the first bit is unchanged,

```
GPIOA->MODER |= ~(GPIO_MODER_MODER5_1);
```

To set the bit pair to `0b01` simultaneously,

```
GPIOA->MODER |= GPIO_MODER_MODER5_0; //set first bit
GPIOA->MODER &= ~(GPIO_MODER_MODER5_1); //clear second bit
```

To set the bit pair to `0b10` simultaneously,

```
GPIOA->MODER &= ~(GPIO_MODER_MODER5_0); //clear first bit
GPIOA->MODER |= GPIO_MODER_MODER5_1; //set second bit
```

For single bit configuration registers such as `OTYPER`, we get only a position and a mask in the form of,

```
GPIO_OTYPER_OT0_Pos          (0U)
GPIO_OTYPER_OT0_Msk          (0x1U << GPIO_OTYPER_OT0_Pos)
GPIO_OTYPER_OT0              GPIO_OTYPER_OT0_Msk
```

Did you see the overhead of setting a binary pair to either `0b01` or `0b10` at the same time ? We needed two statements to do that. This is because we can not do simultaneous `AND` or `OR` operations on a single register. Also, those two statements constitute of two RMW sequences, a total 6 single instructions at the least. We are also not guaranteed that these six instructions will be executed in the order they are written in case an interrupt occurs or a thread or processor has access to the same memory locations we are manipulating. That is a problem and that is why we have "**atomic**" operations. It simplifies the RMW operations from the programmer's perspective and also ensures that the instructions will be executed in the order they are written.

There is no need to access all the registers atomically because they are usually set only once. In case of GPIO configuration registers of STM32, we can perform atomic write operations using the dedicated **BSRR** and **BRR** registers. BSRR is a 32-bit register where the lower 16-bits are used to set any of the 16 pins and the higher 16-bits to clear/reset any of the 16 pins of a particular IO port. The BRR register's higher 16-bits are reserved and the lower 16-bits reset or clear the 16 pins.

To set PA5 (5th pin of port A), we can do,

```
GPIOA->BSRR = (1U << 5);   //set the 5th bit or PA5
```

or

```
GPIOA->BSRR = GPIO_BSRR_BS5;   //set the 5th bit or PA5
```

To clear PA5,

```
GPIOA->BSRR = (1U << 21);   //clear the 5th bit or PA5
```

or

```
GPIOA->BSRR = GPIO_BSRR_BR5;   //clear the 5th bit or PA5
```

where `GPIO_BSRR_BS5` is a macro that sets the 5th bit of lower 16-bits and `GPIO_BSRR_BR5` clears the 5th bit of higher 16-bits. Similar operations can be performed on BRR register macros such as `GPIO_BRR_BR5`. Macro definitions like these are available for all of the GPIO registers.

Interestingly, some family of controllers such as STM32F446 do not have the BRR register but only the BSRR. The engineers might have thought it was redundant, I think.

Vishnu Mohanan

PREVIOUS

NEXT

Related Posts

Solving USBasp Programming Issues

6 June 2020

Interfacing R307 Optical Fingerprint Scanner with Arduino

1 July 2019

Interfacing Intersil ISL1208 RTC with Arduino

10 February 2019

Leave a Reply