Embedded System
Southern University of Science and Technology
Mengxuan Wu
12212006

Assignment 1

Mengxuan Wu

1 Brief Introduction to HAL

HAL stands for Hardware Abstraction Layer. Its main purpose is to provide a unified interface to the hardware. With HAL, code can be written in a way that is independent of the hardware. By using functions provided by HAL, instead of directly accessing the hardware, the code can be easily ported to different hardware platforms.
Take the example of using a GPIO pin to control an LED. Without HAL, we need to know the specific register address of the GPIO pin, which varies from one microcontroller to another. With HAL, we can use a function like `HAL_GPIO_TogglePin()` to control the pin, without worrying about the specific register address.

2 Analysis of the GPIO HAL Module

2.1 Flow of the GPIO HAL Function Calls

Generally, the flow of the GPIO HAL function calls is as follows:

Step	Function	Description
Initialization	`HAL_GPIO_Init()`	Initialize the GPIO pin.

Modification	`HAL_GPIO_ReadPin()`
	`HAL_GPIO_WritePin()`
	`HAL_GPIO_TogglePin()`
	`HAL_GPIO_LockPin()`
		Read from the GPIO pin.
		Write to the GPIO pin.
		Toggle the GPIO pin.
		Lock the GPIO pin.

Deinitialization	`HAL_GPIO_DeInit()`	Deinitialize the GPIO pin.

2.2 Code Analysis

2.2.1 Initialization

Function Prototype

```
void  HAL_GPIO_Init(GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_Init);
```

The `HAL_GPIO_Init()` function initializes the GPIO pin(s) according to the specified parameters in the `GPIO_Init`.
It takes two arguments. The first argument specifies the GPIO peripheral (e.g. `GPIOA`, `GPIOB`, etc.). The second argument is a pointer to a structure of type `GPIO_InitTypeDef` that contains the configuration information for the specified GPIO pin(s).

```
typedef struct
{
  uint32_t Pin;   // any value of @ref GPIO_pins_define
  uint32_t Mode;  // a value of @ref GPIO_mode_define
  uint32_t Pull;  // a value of @ref GPIO_pull_define
  uint32_t Speed; // a value of @ref GPIO_speed_define
} GPIO_InitTypeDef;
```

In this structure, `Pin` specifies the GPIO pins to be configured. Its first 16 bits are used to specify the pins, and can be a combination of multiple pins. The other three fields specify the mode, pull, and speed of the GPIO pins.

Argument Checking

```
assert_param(IS_GPIO_ALL_INSTANCE(GPIOx));
assert_param(IS_GPIO_PIN(GPIO_Init->Pin));
assert_param(IS_GPIO_MODE(GPIO_Init->Mode));
```

Into the body of the function, the first thing that is done is to check the validity of the arguments. Three macros are used to check the validity of the GPIO peripheral, the GPIO pins, and the GPIO mode.

Loop Through Pins

```
uint32_t position = 0x00u;
uint32_t ioposition;
uint32_t iocurrent;
while (((GPIO_Init->Pin) >> position) != 0x00u)
{
  /* Get the IO position */
  ioposition = (0x01uL << position);
  /* Get the current IO position */
  iocurrent = (uint32_t)(GPIO_Init->Pin) & ioposition;

  if (iocurrent == ioposition)
  {
    /*
      code to configure the GPIO pin at the current position
    */
  }
  position++;
}
```

In this part of the code, the function loops through each bit of the `Pin` field in the `GPIO_Init` structure. It first creates a mask `ioposition` with a single bit set at the current position. Then it extracts the bit from the `Pin` field to `iocurrent`. If the bit is set, then the corresponding GPIO pin will be configured. Finally, the position is incremented to move to the next bit.

Configure the GPIO Pin

```
uint32_t config = 0x00u;
__IO uint32_t *configregister; /* Store the address of CRL or CRH
↪  register based on pin number */
uint32_t registeroffset;       /* offset used during computation of CNF
↪  and MODE bits placement inside CRL or CRH register */
/* Based on the required mode, filling config variable with MODEy[1:0]
↪  and CNFy[3:2] corresponding bits */
switch (GPIO_Init->Mode)
{
  /* If we are configuring the pin in OUTPUT push-pull mode */
  case GPIO_MODE_OUTPUT_PP:
    /* Check the GPIO speed parameter */
    assert_param(IS_GPIO_SPEED(GPIO_Init->Speed));
    config = GPIO_Init->Speed + GPIO_CR_CNF_GP_OUTPUT_PP;
    break;

  /* If we are configuring the pin in OUTPUT open-drain mode */
  case GPIO_MODE_OUTPUT_OD:
    /* Check the GPIO speed parameter */
    assert_param(IS_GPIO_SPEED(GPIO_Init->Speed));
    config = GPIO_Init->Speed + GPIO_CR_CNF_GP_OUTPUT_OD;
    break;

  // other cases are similar
}

/* Check if the current bit belongs to first half or last half of the
↪  pin count number
 in order to address CRH or CRL register*/
configregister = (iocurrent < GPIO_PIN_8) ? &GPIOx->CRL     :
↪  &GPIOx->CRH;
registeroffset = (iocurrent < GPIO_PIN_8) ? (position << 2u) : ((position
↪  - 8u) << 2u);

/* Apply the new configuration of the pin to the register */
MODIFY_REG((*configregister), ((GPIO_CRL_MODE0 | GPIO_CRL_CNF0) <<
↪  registeroffset), (config << registeroffset));
```

In this part of the code, the function first uses a switch statement for the GPIO mode. For each mode, it sets a variable `config`. The `config` variable is a 4-bit value, with higher 2 bits for CNF and lower 2 bits for MODE, which is used in CRH or CRL register to configure the GPIO pin.

Then, it determines whether the current pin belongs to the first half or the last half of the pin count number. Based on this, it selects the CRH or CRL register and the offset to modify the register.
Finally, it modifies the register by applying the new configuration of the pin to the register.

2.3 Modification

In this section, I will analyze the functions `HAL_GPIO_TogglePin()` as an example.

```
void HAL_GPIO_TogglePin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)
{
  uint32_t odr;

  /* Check the parameters */
  assert_param(IS_GPIO_PIN(GPIO_Pin));

  /* get current Output Data Register value */
  odr = GPIOx->ODR;

  /* Set selected pins that were at low level, and reset ones that were
  ↪  high */
  GPIOx->BSRR = ((odr & GPIO_Pin) « GPIO_NUMBER) | (~odr & GPIO_Pin);
}
```

The function checks the validity of the GPIO pin. Then it reads the current output data register value. It toggles the selected pins by setting the BSRR register as the opposite of current ODR value: if the pin was at low level, it will write the set bit to BSRR; if the pin was at high level, it will write the reset bit to BSRR.

3 Troubleshooting

3.1 BSRR Register

When analyzing code in the GPIO HAL module, I found that the BSRR register confusing: When we operate on registers like ODR, we carefully use mask to only modify the bits we want to change. However, in the BSRR register, we write the bits directly without any mask operations.

```
// An example code that set the 3rd to 4th bits of ODR to 0b10

// Directly write to ODR
GPIOA -> ODR &= ~(0b11 « 2); // clear the 3rd and 4th bits
GPIOA -> ODR |= (0b10 « 2); // set the 3rd and 4th bits to 0b10

// Write to BSRR
GPIOA -> BSRR = (0b10 « 2); // set the 3rd and 4th bits to 0b10
```

It seems that we don't care about the previous state of the BSRR register. We just write the bits we want to change directly to the BSRR register. This is different from the way

we operate on other registers. After looking into the reference manual, I found that the BSRR register is a write-only register. The way it works more like a trigger: if we write a 1 to a bit, it triggers the set/reset operation on the corresponding bit in the ODR register. And it does not hold any state, meaning that any previous operation will not persist and has no effect on the next operation.

3.2 `__weak` and `__IO` Keywords

When analyzing the code, I found two keywords that I am not familiar with: `__weak` and `__IO`. After looking into the code, I found that these are compiler-specific keywords.
`__weak` is used to declare a weak function, which means it can be overridden by a strong function with the same name. This is useful when we want to provide a default implementation of a function, but allow the user to override it if needed.
`__IO` is used to declare a volatile variable. This tells the compiler that the variable may be changed by external sources, so it should not optimize the access to this variable. For example, we may use a volatile variable as condition in a while loop. If we assign 1 to the variable, it might be optimized by compiler and the loop will never check the condition. By declaring the variable as `__IO`, we tell the compiler that the variable may be changed by external sources, so it will leave a check in the loop.