解析 STM32 的库函数

意法半导体在推出 STM32 微控制器之初，也同时提供了一套完整细致的固件开发包，里面包含了在 STM32 开发过程中所涉及到的所有底层操作。通过在程序开发中引入这样的固件开发包，可以使开发人员从复杂冗余的底层寄存器操作中解放出来，将精力专注应用程序的开发上，这便是 ST 推出这样一个开发包的初衷。

但这对于许多从 51/AVR 这类单片机的开发转到 STM32 平台的开发人员来说，势必有一个不适应的过程。因为程序开发不再是从寄存器层次起始，而要首先去熟悉 STM32 所提供的固件库。那是否一定要使用固件库呢？当然不是。但 STM32 微控制器的寄存器规模可不是常见的 8 位单片机可以比拟，若自己细细琢磨各个寄存器的意义，必然会消耗相当的时间，并且对于程序后续的维护，升级来说也会增加资源的消耗。对于当前“时间就是金钱”的行业竞争环境，无疑使用库函数进行 STM32 的产品开发是更好的选择。本文将通过一个简单的例子对 STM32 的库函数做一个简单的剖析。

以最常用的 GPIO 设备的初始化函数为例，如下程序段一：

```
GPIO_InitTypeDef GPIO_InitStructure;                          ①
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;                     ②
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;             ③
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;              ④
GPIO_Init(GPIOA , &GPIO_InitStructure);                       ⑤
```

这是一个在 STM32 的程序开发中经常使用到的 GPIO 初始化程序段，其功能是将 GPIOA.4 口初始化为推挽输出状态，并最大翻转速率为 50MHz。下面逐一分解：

- 首先是①，该语句显然定义了一个 GPIO_InitTypeDef 类型的变量，名为 GPIO_InitStructure，则找出 GPIO_InitTypeDef 的原型位于“stm32f10x_gpio.h”文件，原型如下：

```
typedef struct
{
    u16 GPIO_Pin;
    GPIOSpeed_TypeDef GPIO_Speed;
    GPIOMode_TypeDef GPIO_Mode;
}GPIO_InitTypeDef;
```

由此可知 GPIO_InitTypeDef 是一个结构体类型同义字，其功能是定义一个结构体，该结构体有三个成员分别是 u16 类型的 GPIO_Pin、GPIOSpeed_TypeDef 类型的 GPIO_Speed 和 GPIOMode_TypeDef 类型的 GPIO_Mode。继续探查 GPIOSpeed_TypeDef 和 GPIOMode_TypeDef 类型，在“stm32f10x_gpio.h”文件中找到对 GPIOSpeed_TypeDef 的定义：

```
typedef enum
{
    GPIO_Speed_10MHz = 1,
    GPIO_Speed_2MHz,
    GPIO_Speed_50MHz
}GPIOSpeed_TypeDef;
```

则可知 GPIOSpeed_TypeDef 枚举类型同一只，其功能是定义一个枚举类型变量，该变量可表示 GPIO_Speed_10MHz、GPIO_Speed_2MHz 和 GPIO_Speed_50MHz 三个含义（其中 GPIO_Speed_10MHz 已经定义为 1，读者必须知道 GPIO_Speed_2MHz 则依次被编译器赋予 2，而 GPIO_Speed_50MHz 为 3）。

同样也在“stm32f10x_gpio.h”文件中找到对 GPIOMode_TypeDef 的定义：

```
typedef enum
{
    GPIO_Mode_AIN = 0x0,
    GPIO_Mode_IN_FLOATING = 0x04,
    GPIO_Mode_IPD = 0x28,
    GPIO_Mode_IPU = 0x48,
    GPIO_Mode_Out_OD = 0x14,
    GPIO_Mode_Out_PP = 0x10,
    GPIO_Mode_AF_OD = 0x1C,
    GPIO_Mode_AF_PP = 0x18
}GPIOMode_TypeDef;
```

这同样是一个枚举类型同义字，其成员有 GPIO_Mode_AIN、GPIO_Mode_AF_OD 等（也可以轻易判断出这表示 GPIO 设备的工作模式）。

至此对程序段一的①解析可以做一个总结：

该行定义一个结构体类型的变量 GPIO_InitStructure，并且该结构体有 3 个成员，分别为 GPIO_Pin、GPIO_Speed 和 GPIO_Mode，并且 GPIO_Pin 表示 GPIO 设备引脚 GPIO_Speed 表示 GPIO 设备速率和 GPIO_Mode 表示 GPIO 设备工作模式。

- 接下来是②，此句是一个赋值语句，把 GPIO_Pin_4 赋给 GPIO_InitStructure 结构体中的成员 GPIO_Pin，可以在“stm32f10x_gpio.h”文件中找到对 GPIO_Pin_4 做的宏定义：

  ```
  #define GPIO_Pin_4          ((u16)0x0010)
  ```

  因此②的本质是将 16 位数 0x0010 赋给 GPIO_InitStructure 结构体中的成员 GPIO_Pin。
- ③语句和②相似将 GPIO_Speed_50MHz 赋给 GPIO_InitStructure 结构体中的成员 GPIO_Speed，但注意到此处 GPIO_Speed_50MHz 只是一个枚举变量，并非具体的某个值。
- ④语句亦和②语句类似，把 GPIO_Mode_Out_PP 赋给 GPIO_InitStructure 结构体中的成员 GPIO_Mode，从上文可知 GPIO_Mode_Out_PP 的值为 0x10。
- ⑤是一个函数调用，即调用 GPIO_Init 函数，并提供给该函数 2 个参数，分别为 GPIOA 和&GPIO_InitStructure，其中&GPIO_InitStructure 表示结构体变量 GPIO_InitStructure 的地址，而 GPIOA 则在“stm32f10x_map.h”文件中找到定义：

  ```
  #ifdef _GPIOA
      #define GPIOA                ((GPIO_TypeDef *) GPIOA_BASE)
  #endif
  ```

  此三行代码是一个预编译结构，首先判断是否定义了宏_GPIOA。可以在“stm32f10x_conf.h”中发现对_GPIOA 的定义为：

  ```
  #define _GPIOA
  ```

  这表示编译器会将代码中出现的 GPIOA 全部替换为((GPIO_TypeDef *) GPIOA_BASE)。从该句的 C 语言语法可以判断出((GPIO_TypeDef *) GPIOA_BASE)的功能为将 GPIOA_BASE 强制类型转换为指向 GPIO_TypeDef 类型的结构体变量。如此则需要找出 GPIOA_BASE 的含义，依次在“stm32f10x_map.h”文件中找到：

  ```
  #define GPIOA_BASE              (APB2PERIPH_BASE + 0x0800)
  ```

  和：

  ```
  #define APB2PERIPH_BASE         (PERIPH_BASE + 0x10000)
  ```

  还有：

  ```
  #define PERIPH_BASE             ((u32)0x40000000)
  ```

  明显 GPIOA_BASE 表示一个地址，通过将以上 3 个宏展开可以得到：

GPIOA_BASE = 0x40000000 + 0x10000 + 0x0800

此处的关键便在于 0x40000000、0x10000 和 0x0800 这三个数值的来历。读者应该通过宏名猜到了，这就是 STM32 微控制器的 GPIOA 的设备地址。通过查阅 STM32 微控制器开发手册可以得知，STM32 的外设起始基地址为 0x40000000，而 APB2 总线设备起始地址相对于外设基地址的偏移量为 0x10000，GPIOA 设备相对于 APB2 总线设备起始地址偏移量为 0x0800。

对⑤句代码进行一个总结：调用 GPIO_Init 函数，并将 STM32 微控制器的 GPIOA 设备地址和所定义的结构体变量 GPIO_InitStructure 的地址传入。

以上是对 GPIOA 初始化库函数的剖析，现继续转移到函数内部分析，GPIO_Init 函数原型如程序段二：

```
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
{
    u32 currentmode = 0x00, currentpin = 0x00, pinpos = 0x00, pos = 0x00;
    u32 tmpreg = 0x00, pinmask = 0x00;

    /* 检查参数是否正确 */
    assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
    assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
    assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));

    /* 将工作模式暂存至 currentmode 变量中 */
    currentmode = ((u32)GPIO_InitStruct->GPIO_Mode) & ((u32)0x0F);
    /* 如果欲设置为任意一种输出模式，则再检查”翻转速率“参数是否正确 */
    if ((((u32)GPIO_InitStruct->GPIO_Mode) & ((u32)0x10)) != 0x00)
    {
        assert_param(IS_GPIO_SPEED(GPIO_InitStruct->GPIO_Speed));
        currentmode |= (u32)GPIO_InitStruct->GPIO_Speed;
    }

    /* 设置低八位引脚（即 pin0 ~ pin7） */
    if (((u32)GPIO_InitStruct->GPIO_Pin & ((u32)0x00FF)) != 0x00)
    {
        /* 读出当前配置字 */
        tmpreg = GPIOx->CRL;
        for (pinpos = 0x00; pinpos < 0x08; pinpos++)
        {
            /* 获取将要配置的引脚号 */
            pos = ((u32)0x01) << pinpos;
            currentpin = (GPIO_InitStruct->GPIO_Pin) & pos;
            if (currentpin == pos)
            {
                /* 先清除对应引脚的配置字 */
                pos = pinpos << 2;
                pinmask = ((u32)0x0F) << pos;
```

```
                tmpreg &= ~pinmask;
                /* 写入新的配置字 */
                tmpreg |= (currentmode << pos);
                /* 若欲配置为上拉 / 下拉输入，则需要配置 BRR 和 BSRR 寄存器 */
                if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPD)
                {
                    GPIOx->BRR = (((u32)0x01) << pinpos);
                }
                else
                {
                    if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPU)
                    {
                        GPIOx->BSRR = (((u32)0x01) << pinpos);
                    }
                }
            }
        }
        /* 写入低八位引脚配置字 */
        GPIOx->CRL = tmpreg;
}

/* 设置高八位引脚（即 pin8 ~ pin15），流程和第八位引脚配置流程一致，不再作解析 */
if (GPIO_InitStruct->GPIO_Pin > 0x00FF)
{
        tmpreg = GPIOx->CRH;
        for (pinpos = 0x00; pinpos < 0x08; pinpos++)
        {
            pos = (((u32)0x01) << (pinpos + 0x08));
            currentpin = ((GPIO_InitStruct->GPIO_Pin) & pos);
            if (currentpin == pos)
            {
                pos = pinpos << 2;
                pinmask = ((u32)0x0F) << pos;
                tmpreg &= ~pinmask;
                tmpreg |= (currentmode << pos);
                if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPD)
                {
                    GPIOx->BRR = (((u32)0x01) << (pinpos + 0x08));
                }
                if (GPIO_InitStruct->GPIO_Mode == GPIO_Mode_IPU)
                {
                    GPIOx->BSRR = (((u32)0x01) << (pinpos + 0x08));
                }
            }
```

```
        }
        GPIOx->CRH = tmpreg;
    }
}
```

这段程序的流程是：首先检查由结构体变量 GPIO_InitStructure 所传入的参数是否正确，然后对 GPIO 寄存器进行“保存——修改——写入”的操作，完成对 GPIO 设备的设置工作。显然，结构体变量 GPIO_InitStructure 所传入参数的目的是设置对应 GPIO 设备的寄存器。而 STM32 的参考手册对关于 GPIO 设备的设置寄存器的描述如下图 1 和表 1（仅列出低八位引脚寄存器描述，高八位引脚类同）：

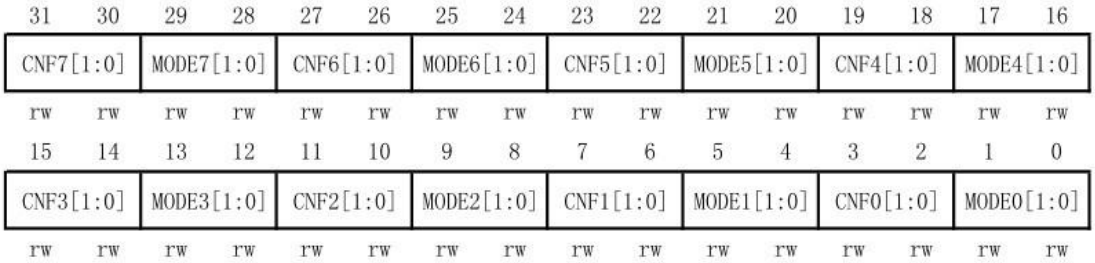

图 1　GPIO 设备控制寄存器 GPIOx_CRL

位 31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	在输入模式(MODE[1:0]=00)： 00：模拟输入模式 01：浮空输入模式(复位后的状态) 10：上拉/下拉输入模式 11：保留 在输出模式(MODE[1:0]>00)： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位 29:28 25:24 21:20 17:16 13:12 9:8 5:4 1:0	MODEy[1:0] ：端口 x 的模式位(y = 0…7) (Port x mode bits) 软件通过这些位配置相应的 I/O 端口，请参考表 17 端口位配置表。 00：输入模式(复位后的状态) 01：输出模式，最大速度 10MHz 10：输出模式，最大速度 2MHz 11：输出模式，最大速度 50MHz

表 1　GPIO 设备控制寄存器 GPIOx_CRL 描述

该寄存器为 32 位，其中分为 8 份，每份 4 位，对应低八位引脚的设置。每一个引脚的设置字分为两部分，分别为 CNF 和 MODE，各占两位空间。当 MODE 的设置字为 0 时，表示将对应引脚配置为输入模式，反之设置为输出模式，并有最大翻转速率限制。而当引脚配置为输出模式时，CNF 配置字则决定引脚以哪种输出方式工作（通用推挽输出、通用开漏输出等）。通过对程序的阅读和分析不难发现，本文最初程序段中 GPIO_InitStructure 所传入参数的对寄存器的作用如下：

- GPIO_Pin_4 被宏替换为 0x0010，对应图 1 可看出为用于选择配置 GPIOx_CRL 的[19:16]

位，分别为 CNF4[1:0]、MODE4[1:0]。

- GPIO_Speed_50MHz 为枚举类型，包含值 0x03，被用于将 GPIOx_CRL 位中的 MODE4[1:0]配置为 b11（此处 b 意指二进制）。
- GPIO_Mode 亦为枚举类型，包含值 0x10，被用于将 GPIOx_CRL 位中的 MODE4[1:0]配置为 b00。事实上 GPIO_Mode 的值直接影响寄存器的只有低四位，而高四位的作用可以从程序段二中看出，是用于判断此参数是否用于 GPIO 引脚输出模式的配置。

至此应不难知道 STM32 的固件库最后是怎样影响最底层的寄存器的。总结起来就是：固件库首先将各个设备所有寄存器的配置字进行预先定义，然后封装在结构或枚举变量中，待用户调用对应的固件库函数时，会根据用户传入的参数从这些封装好的结构或枚举变量中取出对应的配置字，最后写入寄存器中，完成对底层寄存器的配置。

可以看到，STM32 的固件库函数对于程序开发人员来说是十分便利的存在，只需要填写言简意赅的参数就可以在完全不关心底层寄存器的前提下完成相关寄存器的配置，具有相当不错的通用性和易用性，也采取了一定措施保证库函数的安全性（主要引入了参数检查函数 assert_param）。但同时也应该知道，通用性、易用性和安全性的代价是加大了代码量，同时增加了一些逻辑判断代码造成了一定的时间消耗，在对时间要求比较苛刻的应用场合需要评估使用固件库函数对程序运行时间所带来的影响。读者在使用 STM32 的固件库函数进行程序开发时，应该意识到这些问题。