Assignment 1

STM32 HAL 库源码分析

仇琨元
11913019@mail.sustech.edu.cn

Oct 15, 2024

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit.

目录

1. HAL 库简介

HAL (Hardware Abstraction Layer) 库是由 STMicroelectronics 官方提供的、一套符合 CMSIS 标准 [1] 的硬件抽象层, 可以有效简化基于 STM32 微控制器系列产品的应用程序开发.

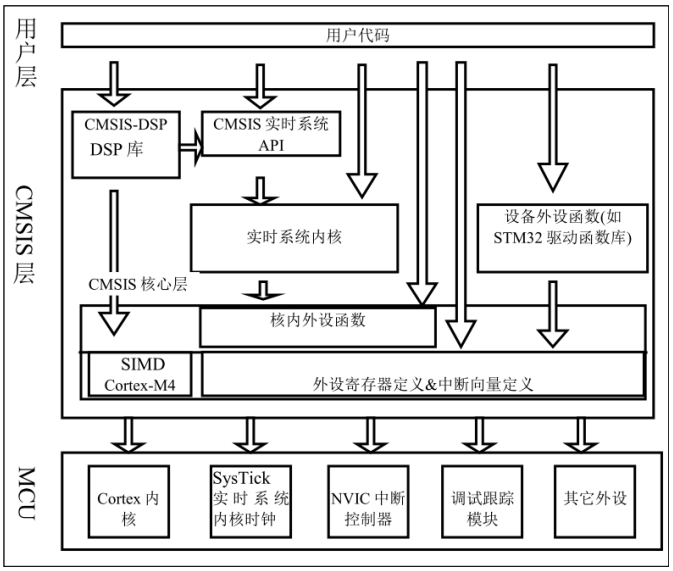


图 1　CMSIS 层架构简图

HAL 库将各个 Cortex 内核寄存器、片上外设控制字和内存映射表等手册内容 [2], [3] 抽象为一组库函数和便于记忆的宏定义, 使得开发者能够脱离随着单片机具体型号变化的细枝末节, 专注于应用程序的核心逻辑, 从而大大提升开发效率.

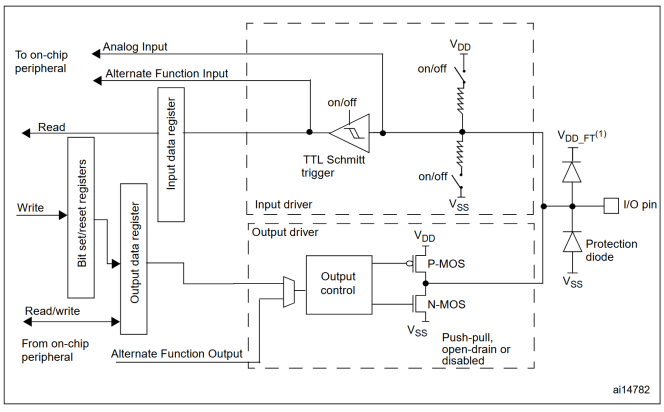
由于微控制器等嵌入式设备的片上资源较为紧张, 最终生成的机器代码需要尽可能压缩体积、提高执行效率, 因此 HAL 库的主要组成部分是宏定义和内联函数而不是函数调用. 宏定义在预处理阶段便会被编译器展开到具体的寄存器和内存映射, 内联函数在预处理和编译阶段中也会被内联到调用语句, 从而最大程度上避免了函数调用过程带来的额外开销和栈溢出风险.

HAL 库具有下面几个主要特点:

- 简单易用: 通过统一 API 访问硬件资源, 最大程度减小代码量和复杂度.
- 可移植性: 在不同单片机型号和系列之间具有良好的可移植性.
- 良好性能: 编译器优化后能够生成近似于基于寄存器开发的高效代码.

2. 调用过程分析和代码分析

以 GPIO 端口这一最常用的片上外设为例, 分析 HAL 库的调用过程和源码实现. 对于 STM32 系列的微控制器, 其 GPIO 端口具有如 图 2 所示的片上实现:



1. V_{DD_FT} is a potential specific to 5-Volt tolerant I/Os, and different from V_{DD}.

图 2　GPIO 端口片上实现框图

每个 GPIO 端口可接入数字 IO、模拟输入和外设 IO 共计 5 种信号源, 可配置为 3 种输入/输出方向和 1 个锁定状态, 总共需要 7 个寄存器来管理. 每 16 个 GPIO 端口分为一组, 直接与以下 7 组寄存器相关联:

- 32 位配置寄存器 GPIOx_CRL, GPIOx_CRH
- 32 位输出寄存器 GPIOx_ODR
- 32 位数据寄存器 GPIOx_IDR
- 32 位置位/复位寄存器 GPIOx_BSRR
  - 16 位复位寄存器 GPIOx_BRR
- 32 位锁定寄存器 GPIOx_LCKR

2.1. 寄存器与 HAL 库函数的关系

对于寻址位宽 32 位的 STM32 单片机来说, 除了映射到真实内存的 512K 地址空间之外, 还有大量的物理地址通过 图 3 所示的总线矩阵 [4] 映射到各种外设寄存器, 其中 `0x40010800-0x400127FF` 被映射到 GPIO 的各个控制寄存器 [5], [3], 如 图 4 所示.

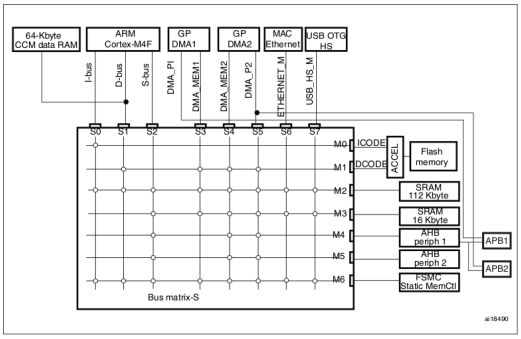


图 3　Cortex-M3 总线矩阵

ADC2	0x4001 2800 - 0x4001 2BFF
ADC1	0x4001 2400 - 0x4001 27FF
Port G	0x4001 2000 - 0x4001 23FF
Port F	0x4001 1C00 - 0x4001 1FFF
Port E	0x4001 1800 - 0x4001 1BFF
Port D	0x4001 1400 - 0x4001 17FF
Port C	0x4001 1000 - 0x4001 13FF
Port B	0x4001 0C00 - 0x4001 0FFF
Port A	0x4001 0800 - 0x4001 0BFF
EXTI	0x4001 0400 - 0x4001 07FF
AFIO	0x4001 0000 - 0x4001 03FF

图 4　GPIO 端口在 STM32F1 上的内存映射

通过访问上述的物理地址映射, 便可以直接访问对应的控制寄存器, 例如以下的代码将 `GPIOA->PA15` 引脚设置为下拉推挽输出模式:

```
uint32_t APB2_BASE=0x40000000UL; /* 查手册得到 APB2 地址=0x40000000 */
uint32_t GPIOA_BASE=0x800UL; /* 同理得到 GPIOA 的基地址=0x800 */
uint32_t PA15=0x8000UL; /* PA15 控制寄存器组基地址=0x8000=0x1<<15 */
uint32_t gpioa_pa15=APB2_BASE+GPIOA_BASE+PA15; /*手动填写偏移量相加的结果*/
uint32_t dCNF=0x4; /*GPIOA_CRH*/
uint32_t dODR=0xC; /*GPIOA_ODR*/
uint32_t dBSRR=0x10;

(uint32_t*)(gpioa_pa15+dCNF) |= (0x3) << ((15-8)*4); /*推挽输出, 50MHz*/
(uint32_t*)(gpioa_pa15+dODR) &= ~(0x1) << 15; /*下拉*/
```

代码 1　手工填写所有基地址和偏移量进行寄存器操作

代码 1 中展示的手填地址写法只要各个基地址和偏移量填写正确, 便能正确操作对应的寄存器, 但是稍有常识的开发者便会感受到这种充斥着难以记忆的 magic number 的写法既需要不停在 IDE 和 MCU 手册之间来回切换, 又容易记错、填错地址数值, 从而产生难以发现的错误.

对于同一型号的器件, 地址映射和片上控制字在设计时便已经唯一确定, 因此只要将这些由版图确定的基地址和偏移量数值固定为人类可读的宏和常量, 便可以用宏展开、函数调用等不依赖人工的自动化操作来代替容易出错的手工填写步骤, 从而大大增强代码的可读性与开发的便利性, 而适配不同型号的器件则只需要对照硬件设计更改对应的基地址和偏移量. 将这些统一的、标准化的 API 汇总到一起, 便形成了 HAL 库、正点原子库等硬件库.

2.2. GPIO 相关调用过程分析

- main()
  - ▸ MX_GPIO_Init()
    - – GPIO_InitStruct
      - Pin
      - Mode
      - Speed
    - – HAL_GPIO_Init()

在主程序初始化时，若 HAL_MspInit() 函数未被重新定义，首先执行的 GPIO 相关函数便是 MX_GPIO_Init() 函数，一般位于 main.c 或 gpio.c 文件中. 进入 MX_GPIO_Init() 后，首先执行展开的 __HAL_RCC_GPIOx_CLK_ENABLE() 宏，使能挂载 GPIO 的 APB2 时钟. 然后，在函数体开头定义的结构体 GPIO_InitStruct 得到具体需要初始化的引脚编号 GPIO_PIN_y、输入输出模式 GPIO_MODE_XX_X 和边沿速度 GPIO_SPEED_FREQ_XXX 赋值，准备传入实际初始化 GPIO 引脚的函数 HAL_GPIO_Init().

HAL_GPIO_Init() 函数是实际对 GPIO 引脚进行配置的函数，它为 16 端口的 GPIO 组配置每个引脚对应的输入/输出模式和上下拉/开漏模式. 在预处理过程中，编译器首先依据宏 -DSTM32F103xE 找到包含对应芯片型号控制字和地址映射的头文件 stm32f103xe.h，并选中合适的条件编译节，然后将传入的结构体和该函数中指定的各个配置宏转换为手册中对应的寄存器值；函数运行时则利用一个指针从低到高扫描 GPIOx->Piny 指定的 GPIO 引脚，将正确的控制字写入到位于正确物理地址的寄存器中完成配置.

2.3. 代码分析

2.3.1. MX_GPIO_Init() 函数

```
GPIO_InitTypeDef GPIO_InitStruct;
//...

/*Configure GPIO pin : PB11 */
GPIO_InitStruct.Pin = GPIO_PIN_11;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Spee d =GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init_(GPIOB,&GPIO_Initstruct);
```

在 MX_GPIO_Init() 函数中，PB11 引脚的位置和模式配置被赋值给结构体 GPIO_InitStruct 中的 Pin、Mode 和 Speed 字段. 利用结构体指针，将这些参数传入实际初始化 GPIO 引脚的函数中. 其中 GPIO_PIN_y、GPIO_MODE_XX_X 和 GPIO_SPEED_FREQ_XXX 为 GPIO 引脚编号、输入输出模式和边沿速度的宏定义，展开顺序依次为文件 stm32f1xx_hal_gpio.h 中

```
#define  GPIO_SPEED_FREQ_LOW             (GPIO_CRL_MODE0_1) /*!< Low speed */
#define  GPIO_SPEED_FREQ_MEDIUM          (GPIO_CRL_MODE0_0) /*!< Medium speed */
#define  GPIO_SPEED_FREQ_HIGH            (GPIO_CRL_MODE0)   /*!< High speed */

#define  GPIO_NOPULL        0x00000000u   /*!< No Pull-up or Pull-down activation  */
#define  GPIO_PULLUP        0x00000001u   /*!< Pull-up activation                  */
#define  GPIO_PULLDOWN      0x00000002u   /*!< Pull-down activation                */
```

和 stm32f103xe.h 中

```
#define GPIO_CRL_MODE_Pos                   (0U)
#define GPIO_CRL_MODE_Msk                   (0x33333333UL << GPIO_CRL_MODE_Pos) /*!< 0x33333333 */
#define GPIO_CRL_MODE                       GPIO_CRL_MODE_Msk                   /*!< Port x mode bits */
```

```
#define GPIO_CRL_MODE0_Pos                  (0U)
#define GPIO_CRL_MODE0_Msk                  (0x3UL << GPIO_CRL_MODE0_Pos)      /*!< 0x00000003 */
#define GPIO_CRL_MODE0                      GPIO_CRL_MODE0_Msk                 /*!< MODE0[1:0] bits (Port
x mode bits, pin 0) */
#define GPIO_CRL_MODE0_0                    (0x1UL << GPIO_CRL_MODE0_Pos)      /*!< 0x00000001 */
#define GPIO_CRL_MODE0_1                    (0x2UL << GPIO_CRL_MODE0_Pos)      /*!< 0x00000002 */
```

因此, 该结构体最终展开为

```
GPIO_InitStruct = HAL_GPIOInitStruct {
  0x0800,
  0x0001,
  0x0001,
  0x0002
} GPIO_InitStruct;
```

2.3.2. HAL_GPIO_Init() 函数

未触发断言的情况下, 该函数可以简化为以下的形式:

```
HAL_GPIO_Init_(GPIOx, &GPIO_InitStruct) {

  /* temp variable declaration */
  uint32_t position = 0x00u;
  ...

  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)
    {
      /* Set the corresponding IO */
      ...
    }
    position++;
  }
}
```

由 `GPIOx->Piny` 的宏定义可知, 指定某个引脚需要初始化等价于该成员变量上对应位的值为 1, 如果高位全为 0, 则不再有需要进行设置的引脚. 因此, 使用一个 while 循环来扫描所有需要初始化的引脚. 同理, `ioposition = (0x01uL << position)` 语句便获得了当前引脚的位置, `iocurrent = (uint32_t)(GPIO_Init->Pin) & ioposition` 语句则获得了 `GPIOx->Piny` 变量中当前位是否确实为 1. 当 `ioposition==iocurrent` 时, 即表明 `iocurrent` 指向的位确实对应 `MX_GPIO_Init()`中指定需要设置的引脚.

参考文献

[1] 《CMSIS: Introduction》. 见于: 2024 年 10 月 8 日. [在线]. 载于: https://arm-software.github.io/CMSIS_6/latest/General/index.html#License

[2] 《STM32F10xxx/20xxx/21xxx/L1xxxx Cortex®-M3 Programming Manual》.

[3]《High-Density Performance Line ARM-Based 32bit MCU with 256 to 512 KB Flash, USB, CAN, 11 Timers, 3 ADCs, 13 Communication Interfaces》.

[4] STMicroelectronics, ST STM32F4 SERIES PROGRAMMING MANUAL Pdf Download. STMicroelectronics. 见于: 2024 年 10 月 8 日. [在线]. 载于: https://www.manualslib.com/manual/1375966/St-Stm32f4-Series.html

[5] ST RM0008 Reference Manual STM32F101xx STM32F102xx STM32F103xx STM32F105xx STM32F107xx Advanced ARM-Based 32-Bit MCUs Manual. 见于: 2024 年 10 月 8 日. [在线]. 载于: http://archive.org/details/manuallib-id-2384681