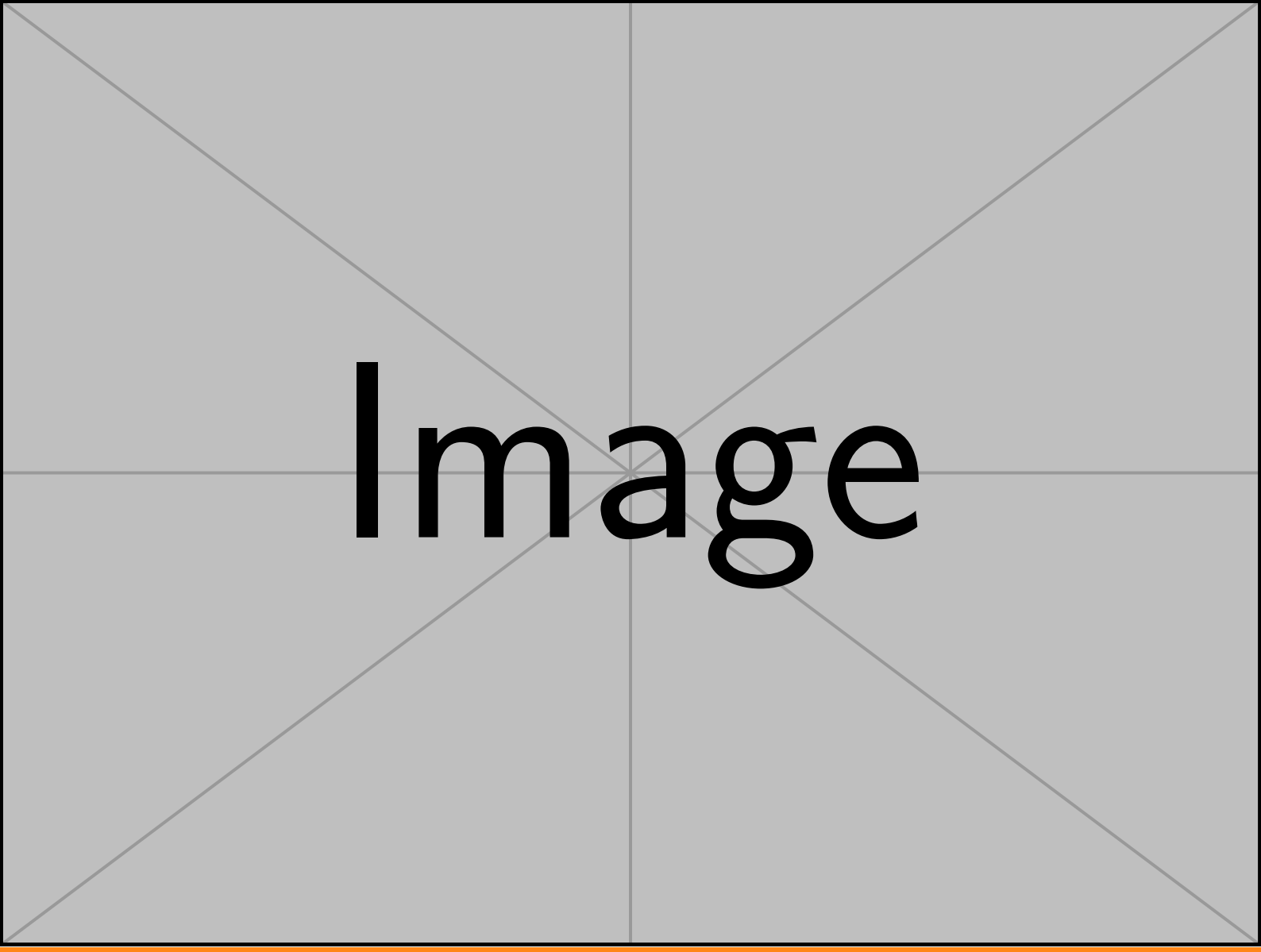

嵌入式培训参考资料

2024 科协用

作者：Yuhang Gu

组织：Southeast University, School of Electronics Engineering

时间：2024

版本：Rev 1.0

目录

序

我在大一学习嵌入式开发的过程中曾经无数次陷入迷茫. 网上的资料多而杂, 要么是花费了太多时间在这个阶段不必要的细节, 要么是太过简略以至于完全不明所以. 另一个痛苦的来源其实是你校的计算机课程教学. 几乎可以暴论, 嵌入式开发学习的上限完全取决于**对计算机系统的认识程度**; 我在大一留下的诸多疑问一直到我学习了**南京大学计算机系统实验**课程之后才得到解答.

所以我决定做一份学习资料 (不如说更像是实验导引, 就像是国外学校 EECS 课程常有的 Lab). 在这份文档中, 我会尽量做到**只讲述当前有必要的细节**, 应当交给视频去讲明白的就会交给视频去做; 应当由你自己 RTFM, RTFSC(这两个缩写是什么意思? 马上就会告诉你了) 弄懂的就交由你自己去慢慢理解.

提示 0.1

本文档目前的版本目前由科协内部使用 (当然你分享出去了大家也不会说什么).

试用的一个重要目的是, 希望能收集大家在自由探索过程中真实遇到的困惑和改进建议, 这些建议会在此后 SEU-Project R 项目讲义的编写中发挥重要作用.

提示 0.2

”我们都是活生生的人, 从小就被不由自主地教导用最小的付出获得最大的得到, 经常会忘记我们究竟要的是什么. 我承认我完美主义, 但我想每个人心中都有那一份求知的渴望和对真理的向往, ”大学”的灵魂也就在于超越世俗, 超越时代的纯真和理想 – 我们不是要讨好企业的毕业生, 而是要寻找改变世界的力量.”

— jyy

”教育除了知识的记忆之外, 更本质的是能力的训练, 即所谓的 training. 而但凡 training 就必须克服一定的难度, 否则你就是在做重复劳动, 能力也不会有改变. 如果遇到难度就选择退缩, 或者让别人来替你克服本该由你自己克服的难度, 等于是自动放弃了获得 training 的机会, 而这其实是大学专业教育最宝贵的部分.”

— etone

计算机的所有东西都是人做出来的，别人能想得出来的，我也一定能想得出来，在计算机里头，没有任何黑魔法，所有的东西，只不过是我现在不知道而已，总有一天，我会把所有的细节，所有的内部的东西全都搞明白的。

— 翁恺

0.1 正确提问和解决问题

0.1.1 如何求助

在碰到问题求助之前 — 哦不, 在开始实验之旅之前, 请先简单阅读一下这两篇文章: 提问的智慧和别像弱智一样提问.

提示 0.3

是的, 别再问”为什么板子上的灯亮起来了但是电脑检测不到”这种问题了 - 仔细检查一下你连接开发板用的是数据线还是电源线 (—它们之间有什么区别?)

注意 0.2

一定要记住: 机器 (除非芯片烧了) 和编译器永远是对的; 未测试的代码一定是错的; 杜邦线很有可能是连错的; 时钟有可能是忘记开了的.

不可否认的是, 嵌入式开发的软硬件结合特性和底层性注定了错误调试的难度, 但错误一定不是不可排除的黑魔法. 你有万用表, 有调试器 (在单片机开发时, 你同样可以通过 gdb 打断点, 看内存等), 已经足够排除 90% 的问题了. 剩下的交给玄学吧.

♡

0.1.2 用正确的手段解决问题

一个重要的任务是, 希望能在实验的过程中培养大家用正确的手段解决问题的能力. 具体的内容在此后的讲义中将会一再涉及.

0.2 C 语言基础?

毫无疑问嵌入式开发的主力是 C 语言. 尽管需要强调 **C++ 和 C 基本上是两种语言**, 但大家大一学的 C++ 仍然足够帮你应付嵌入式开发的入门需要了. 然而 C 中有两件事物仍然值得强调:

- 指针, 指针的本质, 数据类型的本质.
- 结构体, 结构体的内存分布, 结构体指针, 结构体指针和各种其他事物的互转

在此, 非常推荐访问**笨方法学 C**进行 C 语言的学习 (同样会涉及一些命令行和编译器的使用小方法, 很实用). 推荐完成练习 1-18, 31(调试器), 44(环形缓冲区) 的学习.

如果你没有太多时间的话, 也请一定要重看其**练习 15: 可怕的指针**和**练习 16: 结构体和指向它们的指针**和**练习 18: 函数指针**的内容. 不然对于之后的内容给你带来的, 可能的心灵创伤. 作者概不负责.

提示 0.4

一个有趣的问题: 能不能用 C++ 写单片机程序?

答案是肯定的. 不过要完全理解这个过程还需要一些对编译过程的理解. 而且你不能直接在 C 里引用 C++ 声明的函数 — 为什么?

这个问题就交给两千年后的你来 STFW 解决吧.

♣

0.3 实验方案

Part 1. 从零开始的单片机之旅

0.3.1 实验一: 点灯工程师

- 流水灯
- 读取按键
- RTFM(一)
- 发生了什么?

0.3.2 实验二: Hello World!

- UART 和发送消息
- printf 在做什么?
- 接受号令吧!

0.3.3 实验三: 穿越时空的旅程

- 正片开始: 中断
- 执行流的切换: 状态机的视角
- GPIO 中断和 UART 中断
- 缓冲区, 消息队列

Part 2. 连结世界

0.3.4 实验四: IIC 初见

- EEPROM 读写
- SCL 与 SDA: 了解协议
- 从单片机到现代计算机系统: ” 外设” 究竟是什么?
- Hello World (二): 屏幕, 启动!

0.3.5 实验五: One Last Kiss

- 尝试 MPU6050
- SD 卡
- One Last Kiss

0.3.6 实验六: 跳动的方块

- 姿态解算
- 对它使用线性代数吧!
- 数学运算: 性能与空间的 trade-off
- * Kalman Filter

Part 3. 我逐渐开始理解一切

0.3.7 实验六: SPI 和图形库

- SPI 协议和屏幕驱动
- 使用 LVGL

0.3.8 RTOS

0.4 实验环境

- 操作系统: Windows / GNU/Linux
- 编程语言: C 语言
- IDE 环境: Keil / STM32-CubeIDE / CLion

0.5 如何获得帮助

0.6 其他说明

欢迎加入科协嵌入式培训教材编写组 / Project-R 文档编写和维护组.

关于本讲义内容的问题和建议请联系 gyh: 213221544@seu.edu.cn / 127941818 (QQ)

第 1 章 开始之前: 准备工作与相关注意事项

在本章中, 你将会完成嵌入式开发相关的准备工作 — 包括相关软件和环境的安装.

1.1 开发环境配置

注意 1.1 (中文路径与用户名)

在嵌入式开发的过程中, 一定要避免中文路径和 Windows 用户名 (C://Users 下的文件夹) 的使用!
否则你在本节的安装过程中就会遇到令人费解的问题.
如果你的 Windows 的用户名已经设置成了中文, 请参考网上的资料对 C://Users 下的文件夹名称进行更改.

1.1.1 什么是开发环境?

所谓开发环境就是用于开发的一系列工具 — 代码编辑器, 代码分析器, 编译器 (当然, 还有嵌入式开发的下载器).

提示 1.1 (Visual Studio 的本质)

在大一的课程中, 只需要安装好 VS, 点两下鼠标, 就可以创建工程, 编写代码, 构建项目, 运行程序. 我们把 Visual Studio 称为"IDE", 就是因为它的本质是一套集成开发环境.
现在让我们对 Visual Studio 怯魅 — 仅仅使用文本编辑器和系统命令, 应该怎样编写, 编译和运行一个 C/C++ 程序?
你可以去之前提到的笨方法学 C 中寻找答案.

1.1.2 配置 CubeIDE 作为开发环境

如果你习惯于使用 Keil, 那你可以忽略本节. 尽管 Keil 作为嵌入式开发 IDE 方面的权威备受工业界推崇, 但它无论是从界面还是代码编写体验上讲都实在不是一个合格的现代 IDE.

这里我们推荐使用的是 ST 公司专为 stm32 开发定制的 IDE, **STM32CubeIDE**.

请按照安装开发环境 STM32CubeIDE | keysking 的 stm32 教程-哔哩哔哩视频中的指引安装好 CubeIDE.

1.1.3 * 配置 CLion 作为开发环境

CLion 是 Jetbrains 推出的 C/C++ IDE, 以其友好的界面, 强大的代码提示, 高效的代码重构工具而闻名. 如果你愿意为更好的开发体验而折腾, 不妨尝试为 CLion 添加 STM32 开发支持. CLion 在 2019 年起就开放了对 STM32 开发的官方支持.

配置的方法可以寻找稚晖君的知乎文章, 以及一些其他的博客文章.

注意 1.2

STM32CubeMX 在 6.5 版本之后无法生成 CLion 所需要的 SW4STM32 类型项目, 在安装时可以到官网选择旧版本, 或是参考本文章.

1.2 熟悉全新的世界

在配置完开发环境后, 不要呆在那里, 先试着熟悉一下环境吧.

- 生成一个新项目 (选择什么型号/开发板?)

- 生成的项目十分庞大, 它的文件组织结构大致是什么样的?
- 之后需要在这里编写代码, 那么代码应该写在哪里? (程序设计课的时候代码是写在哪里的?)
- 激动人心的时刻: 点击构建 (Build), 编译项目试试看吧!

1.3 开发板型号以及相关参考

第 2 章 点灯工程师

点灯之于嵌入式开发的学习就如同 Hello World 之于编程的学习, 几乎是每位学习者迈出的第一步. 在这个过程中, 你将会掌握对 MCU 上最基本的外设 — GPIO 的相关操作, 延时函数的使用, 以及整个 C 代码工程的整体情况.

2.1 点亮你的 LED

2.1.1 创建工程

我们所使用的 Nucleo-144 开发板是 ST 官方推出的, 基于 STM32F413ZH 的开发板. 因此 STM32CubeMX 软件中可以直接选用这个板子, 数据库会帮我们初始化一些引脚信息而不用自己手动配置.

关于创建工程, 初始化板子型号的步骤可以参见 **STM32 实验指导书**中**第四章跑马灯实验**的内容.

2.1.2 GPIO

2.1.3 编写代码

请先观看视频: 点亮第一颗小灯和闪烁的小灯. 完成视频里所演示的代码的内容. 注意: 由于型号和开发板不同, 创建工程的过程请先参考实验指导书第四章.

提示 2.1

视频中展示的点灯所用到的 GPIO 引脚和我们使用的开发板并不一致. 在使用相应 API 的时候示例如下:

```
HAL_GPIO_WritePin ( LD1_GPIO_Port, LD1_Pin, GPIO_PIN_SET ); // 将LD1对应的GPIO设为高电平
HAL_GPIO_WritePin ( LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET ); // 将LD2对应的GPIO设为高电平
```

2.2 RTFSC (1)

2.2.1 引脚和宏的使用

让我们把目光首先放在 LD1_Pin 这几个类似的定义上. 很明显, LD1_GPIO_Port, LD1_Pin, GPIO_PIN_SET 都不是本来就属于 C 的东西, 那么它们是如何出现的呢?

使用 IDE 的代码跳转功能可以轻松地帮助我们解答这个疑问. 哦! 原来它们的奥妙在 CubeMX 为我们生成的源码里:

```
// in main.h...
/* Private defines -----------------------------------------------------------*/
#define USER_Btn_Pin GPIO_PIN_13
#define USER_Btn_GPIO_Port GPIOC
#define LD1_Pin GPIO_PIN_0
#define LD1_GPIO_Port GPIOB
// ......
```

接着, main.c 引用了 main.h, 于是我们可以直接在代码里写 LD1_Pin; 在编译期间, 这些定义的宏会被自动展开成本应该有的样子.

问题 2.1 如果你对**预编译指令, #define, #include** 这些概念仍不熟悉的话, 是时候 STFW 了解一下了.

以及: #include 的本质其实是” 复制粘贴”. 怎么理解这个概念?

它将有助于你解释你可能经常遇到的 undeclared identifier, multiple declaration 之类的问题, 以及为什么每一个头文件都需要有一个令人费解的 #ifndef...#define...#endif.

通过 C 中的宏, 我们实现了一种软件的封装. 我们当然可以把之前的代码写成:

```
HAL_GPIO_WritePin ( GPIOB, GPIO_PIN_0, 1 ); // 将LD1对应的GPIO设为高电平
HAL_GPIO_WritePin ( GPIOB, GPIO_PIN_2, 1 ); // 将LD2对应的GPIO设为高电平
```

但这种写法很大程度上降低了代码可读性, 更重要的是破坏了**可维护性**: 如果在未来这个引脚被改变了, 这是否代表着我们未来需要一个一个改引脚?

而我们的写法则没有这种顾虑: 只需要更改 #define 的内容, 剩下的任务编译器会在启动编译时的预编译期间解决一切问题.

2.2.2 引脚的配置

接下来我们会在硬件, CubeMX, 代码三个层面理解引脚和配置. 阅读开发板原理图, 可以发现 LD1 被接到了 PB0 这个 GPIO 引脚.

那么代码中呢? 在上一节, 我们已经看到了:

```
#define LD1_Pin GPIO_PIN_0
#define LD1_GPIO_Port GPIOB
```

我们定义的 LD1 对应的 GPIO 刚好是 GPIOB 端口的 PB0! 同样地, 请你寻找开发板原理图上的 LD2, 3, 寻找它们对应的引脚和宏声明.

那么 CubeMX 是如何知道这一点的呢? 别忘了我们生成项目的时候勾选了”default initialize peripheral”, 由于官方的开发板在数据库中已经有了 LED 端口的信息, 它们在一开始就已经被配置好了. 你可以打开 CubeMX, 在 Pinout&configuration 一栏中寻找到这几个 GPIO, 看看它们是如何被定义好的.

注意 2.1

了解了引脚的配置, 接下来请你使用 **MCU selector** 而不是 Board selector 重新生成一个工程, 自己为 LD0, 1, 2 进行引脚的定义, 并点亮这三颗 LED. 指定引脚的过程可以看 2.1.3 部分中的视频.

2.3 流水灯

到目前为止, 你已经学会了基础的点灯, 以及结合视频完成了动态变换的 LED 效果. 接下来请你自己完成一个流水灯: 使 LD1, LD2, LD3 按顺序交替点亮.

2.3.1 copy-paste

到这一步, 估计你的代码已经变成了这样:

```
HAL_GPIO_WritePin ( LD1_GPIO_Port, LD1_Pin, GPIO_PIN_SET ); // 将LD1对应的GPIO设为高电平
HAL_GPIO_WritePin ( LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET ); // 将LD2对应的GPIO设为低电平
HAL_GPIO_WritePin ( LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET ); // 将LD3对应的GPIO设为低电平
HAL_Delay(500); // 延时, 点亮LD2
HAL_GPIO_WritePin ( LD1_GPIO_Port, LD1_Pin, GPIO_PIN_RESET ); // 将LD1对应的GPIO设为低电平
HAL_GPIO_WritePin ( LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET ); // 将LD2对应的GPIO设为高电平
HAL_GPIO_WritePin ( LD3_GPIO_Port, LD3_Pin, GPIO_PIN_RESET ); // 将LD3对应的GPIO设为低电平
```

```
// ......
```

— 丑到家了, 对不对? 那有没有什么办法让我们要写的东西少一点, 代码好看一点呢?

当然是有的 — 那就是我们之前提到的**宏**.

再加上一点点小技巧 — (**宏拼接和宏参数**) — 我们就可以把" 点灯" 的操作**封装**起来.

```
#define LTON(n) HAL_GPIO_WritePin ( LD##n##_GPIO_Port, LD##n##_Pin, GPIO_PIN_SET ) // 点亮LDn
#define LTOFF(n) HAL_GPIO_WritePin ( LD##n##_GPIO_Port, LD##n##_Pin, GPIO_PIN_RESET ) // 熄灭LDn
```

特别注意其中 ## 的**拼接宏**用法 (如果难以理解, 可以 STFW). 于是原本的代码就可以写成:

```
LDON(1);
LDOFF(2);
LDOFF(3);
HAL_Delay(500); // 延时, 点亮LD2
LDOFF(1);
LDON(2);
LDOFF(3);
```

好看很多了, 对吧? 不过每次点亮一个灯就要写三行疑似还是有点太臭了, 让我们在此基础上再套一层宏 (使用**反斜杠可以定义多行的宏**):

```
#ONELT(n)   \     // 点亮LDn, 熄灭其他LED
  LDOFF(1); \
  LDOFF(2); \
  LDOFF(3); \
  LDON(n);
```

于是:

```
ONELT(1);
HAL_Delay(500);
ONELT(2);
```

注意 2.2

请用宏定义等方法重构自己的代码, 创造一个属于你的炫酷点灯程序吧!

问题 2.2 为什么要费尽心思设计这些宏? 请看本节的小标题: Copy-paste — 指的就是本节一开始, 我们写出来的代码的样子.

经过这一番代码层面的优化, 相信聪明的你已经 get 到了为什么 Copy-paste 行为是写代码的大忌, 以及我们有哪些规避写出 Copy-paste 代码的手段.

当然, 解决 copy-paste 问题的方案不仅仅有宏, 更复杂的任务也可以交给函数去完成 — 不过由于宏仅仅是文本展开, 而不涉及代码跳转, 毫无疑问它是不会造成多余的计算资源消耗的. (不过宏会导致最终编译的代码所需的空间更大; 但是相比起嵌入式开发中金贵的计算资源而言, 这点空间的使用大部分时候算不上什么)

提示 2.2

还有一种有趣的宏用法: 可变参数宏. 比如:

```
#define CASE(n, ...) \
    case n:        \
        __VA_ARGS__; \
        break;
```

于是, 我们可以把 switch 写成这样:

```
switch (state) {
    CASE(0, ONELT(0));
    CASE(3, ONELT(1));
    CASE(4, ONELT(2));
    default:
        break;
}
```

你可以 STFW, 来弄明白发生了什么; 或者还有一种好方法 — 为什么不问问神奇的 ChatGPT 呢?

♣

提示 2.3

复杂的宏也许会让人一头雾水. CLion 之类的现代 IDE 可以给出宏展开的结果; 但是一旦宏复杂起来, 就连强大的 IDE 也无法告诉你它本来是什么样的, 还会影响它正常的代码提示和跳转. 在这种情况下应当如何理解复杂的宏?

想想看宏是如何被编译器所理解的 — 它会在编译的第一步, 预处理阶段, 被编译器展开.

那么如果我们可以让编译器 (如 gcc) 输出源代码预处理的结果, 不就可以弄明白发生了什么了吗?

♣

2.4 RTFM

2.4.1 GPIO: 外设, 时钟和基本情况

2.4.2 GPIO: 寄存器控制 — 外设即访存与抽象

2.5 RTFSC (2)