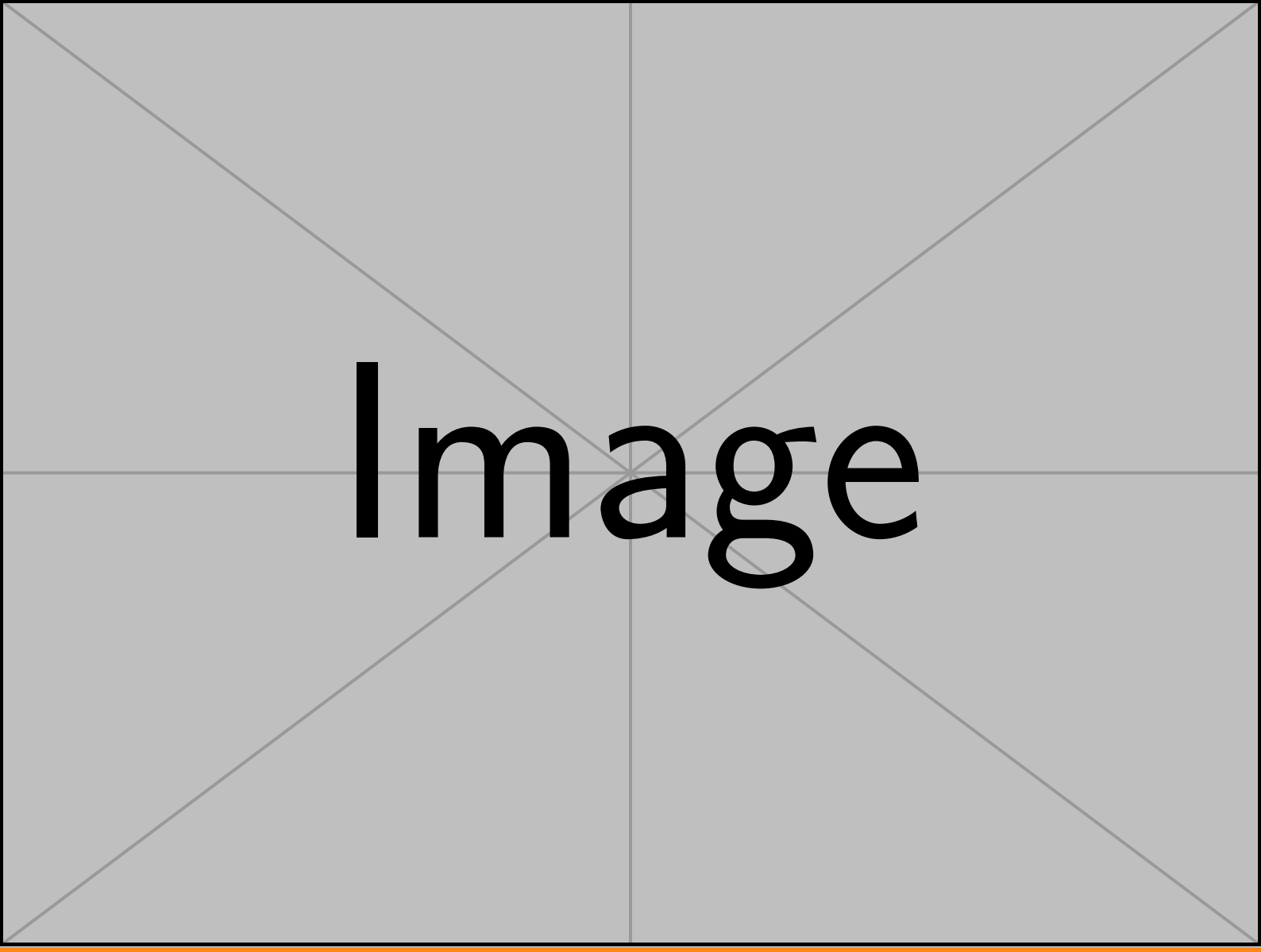

嵌入式培训参考资料

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)

第 3 章 系统中断

在这一章节，我们将会学到嵌入式中最重要的事件处理机制——中断机制，它能够实现系统异步事件处理并提高系统效率

3.1 什么是中断？

什么叫做中断？这要从处理器的工作方式说起。在嵌入式系统中，处理器不会专门等待某一个特定事件发生然后再执行对应的操作，因为这样会浪费大量的资源 (当事件没有发生的时候，处理器就白白等待了)。与此相反，它会保持不断工作或者处于休眠的状态。当特定事件发生时，相关外设会给处理器发送处理该事件的信号，处理器收到信号后，会暂时停下手中的活，先处理特定事件，完成后再回过来继续做之前没做完的事情。这就像是特定事件打断 (Interrupt) 了处理器的工作，因此这种工作机制被称为中断机制，这样的特定事件被称为中断事件。而中断事件发生，并且给处理器传递了相关信号的过程，就叫做产生了中断。

和中断相反的机制叫做轮询 (Polling)。在轮询机制中，系统会循环地检查特定的状态或条件，来判断是否要进行相对应的操作。然而在嵌入式系统这种极致追求资源利用的地方，如果没有检测到任何事件，检测过程中所消耗的资源就被浪费掉了。因此一般不采用轮询的机制。

根据触发方式的不同，中断又可以分为上升沿中断和

3.2 执行流的切换

处理器响应并处理中断事件实际上就是执行流的切换。我们不妨捋一捋中断是怎么一步步被处理器响应并处理的。

1. **外部事件触发**：外部事件可以来自硬件设备 (比如定时器到期，引脚状态发生变化)，也可以来自软件 (比如软件中断请求)。
2. **中断请求发起**：当检测到事件发生后，外设会向处理器发出中断请求信号，表示需要处理器立刻响应并处理相应的事件。
3. **中断检测和响应**：处理器在执行当前任务的过程中，会周期性地检测中断请求信号。当处理器检测到外设发来的信号之后，就会暂停当前任务的执行，保存当前任务的执行状态，并开始执行对应的中断处理程序。
4. **中断服务例程执行**：中断服务例程就是中断处理程序，决定了处理器应该怎么处理对应的中断事件。
5. **中断结束**：中断服务例程结束后，处理器会返回到原先被打断的任务的执行点，继续执行原来的任务，直到下一次中断产生。

着重看一看处理器在收到中断信号之后是怎么开始执行中断服务例程的。

1. **保存当前执行状态**：比如程序计数器，堆栈指针，寄存器状态等等关键信息。
2. **选择新的执行流**：处理器找到对应的中断服务例程。
3. **恢复新的执行状态**：处理器会根据新执行流的信息，设置好程序计数器，堆栈指针，寄存器等等，准备进入新的执行流。
4. **执行新的指令流**：处理器开始新的执行流，执行相应的中断服务例程中的指令。

通过以上步骤，处理器就实现了从原先的执行流跳转到新执行流的完整过程。这就称为“执行流的切换”。当中断服务流程结束后，处理器再次切换执行流，回到原先的任务中。

3.3 处理器去哪找中断服务例程？——中断向量表

在上面切换执行流的过程中，最重要的就是该切换到哪一个执行流？处理器去哪里找一个中断事件对应的中断服务例程？这就需要中断向量表了。

首先什么是向量表？向量可以表示方向，指向想去的地方。在计算机中，向量可以用地址值来表示，这样就能指明方向，该去哪找某个值或函数等等。向量表就是一组地址值组成的集合，它把某一类值或者函数的地址集中放在一起。

而中断向量表储存的就是中断服务例程的地址。在 STM32 的启动文件中有这样的汇编代码：

```
; Vector Table Mapped to Address 0 at Reset
                AREA    RESET, DATA, READONLY
                EXPORT __Vectors
                EXPORT __Vectors_End
                EXPORT __Vectors_Size

__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
; 中间部分省略
__Vectors_End
```

我们不妨试试手撕汇编，看看这部分到底在干啥。首先AREA指令定义了一个名为“RESET”的区域，并告诉编译器这部分区域用来储存数据 (DATA)，而且是只读的 (READONLY)。接着导出了三个值 (EXPORT)，以便在外部调用，这三个值会在接下来看到。然后开始向量表的主要部分。

指令DCD的作用是定义一个 32 位的常量数据，并分配储存空间储存这个数据。首先定义并分配了两个字节储存一个名为__initial_sp 的值 (这个名称代表了栈顶指针的地址)；接着继续DCD，定义并分配了两个字节储存Reset_Handler代表的值 (这是复位中断服务例程的地址)；再继续DCD，定义并分配了两个字节储存NMI_Handler的值 (这是不可屏蔽中断服务例程的地址)。依次类推，不断DCD下去，不断地分配空间储存了各个中断服务例程的地址。请注意，连续DCD会连续分配空间，因此这一列DCD下来，我们就得到了一块连续的空间，连续地储存了各个中断服务例程的地址，这块空间就是中断向量表。

为了方便我们表示或调用这块空间的地址，我们在第一次DCD的前面加上__Vectors，用它来标记中断向量表的起始地址。同理，我们在最后用__Vectors_End来表示向量表的结束地址。而两个地址相减，就得到了向量表的大小，用__Vectors_Size表示 (它的定义在向量表定义的下面)。

```
    __Vectors_Size EQU __Vectors_End - __Vectors
```

其中汇编指令EQU用来定义符号常量，也就是给在EQU后面的值起一个名称，而名称就是EQU前面的表达式。

于是我们知道了，中断向量表在启动文件中被定义了，它是一块连续的区域，储存了每个中断服务例程的地址。当处理器需要处理中断事件时，只需来这里这里找对应服务例程的地址，然后直接跳转到对应的位置开始执行指令流即可。

3.4 中断服务例程会做些什么？

找到并跳转至中断服务例程后，接下来的问题是，中断服务例程会做些什么？答案是，现在它会让系统陷入死循环。所有的中断服务例程被统一放在stm32f4xx_it.h/.c两个文件里，前者存放函数接口和相关的宏定义，后者存放函数的实现。打开源文件，可以发现所有的中断服务例程都是这样的 (ppp是中断的名称)

```
void ppp_Handler(void)
{
```

```
    while (1) {}
}
```

当中断发生后,处理器跳转到对应函数ppp_Handler所在地址,然后执行这个函数。但在函数里只有一个while(1),因此处理器将陷入死循环。而因为所有的中断函数都是这样的死循环，所以只要发生了中断事件，处理器将永远无法返回先前的执行流，直到系统被复位。

显然这不是我们想要的结果,我们希望处理器处理完中断事件后回到原先的执行流。事实上,stm32f4xx_it.c仅仅是给你提供了一个模板。中断服务例程要干什么事情，是需要自己编写的。这里全部写成死循环，是为了防止某些未编写服务例程的中断事件发生，进而导致系统异常。

至此，我们已经知道了当中断事件发生后，处理器会去哪找中断服务例程，中断服务例程会做些什么。最重要的是，我们知道了该去哪个文件编写满足我们自己需要的中断服务例程。接下来我们将上手试试看。

3.5 不一样的点灯方法——GPIO 中断

在 STM32 中，几乎每一个外设都能产生中断，我们先从最基础的 GPIO 中断开始，研究如何利用中断机制编程。

3.5.1 实验操作

为了使用 GPIO 中断,还需要补充 EXTI(External Interrupt) 外部中断控制器的知识。请观看视频初识中断和深入中断，尝试自己实现 GPIO 中断点灯，并解释一遍 GPIO 中断信号进入到中断控制器的过程。

3.6 RTFSC(1)

3.6.1 中断标志位

跳转到 CubeMX 自动帮我们生成的HAL_GPIO_EXTI_IRQHandler函数的定义

```
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
  /* EXTI line interrupt detected */
  if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET) // 检测中断标志位
  {
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin); // 清除中断标志位
    HAL_GPIO_EXTI_Callback(GPIO_Pin); // 回调函数
  }
}
```

各种中断发生时，都会提醒处理器这里有一个中断需要处理。提醒的方法就是设置一个对应的中断标志位，并将其置 1。当处理器处理完中断后，就要将标志位置 0，否则会一直提醒处理器这里有中断需要被处理。有一些外设能够在中断服务例程开始时自动清除中断标志位 (硬件设计)，但遗憾的是 EXTI 没有这样的设计，只能手动清除中断标志位。我们不妨试试不清除中断标志位。将清除中断标志位的代码注释掉，然后再编译并烧录。我们会发现如果一直按着按键，小灯将会不停闪烁，说明中断服务例程正在不断被触发。试试自行理顺逻辑，并讲给自己听，为什么会发生这种情况？

3.6.2 关于检测中断信号的问题

回到主文件main.c，主函数是这样的

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    while (1)
    {
        /* 内容省略 */
    }
}
```

你也许会有些疑问，我们说处理器是要检测到中断信号才会响应并处理中断，那它不是要一直检测中断信号吗？那检测的代码不是应该放在`while`循环里反复执行吗？但我们写的循环里明显没有检测中断信号的代码，那检测中断的代码哪去了呢？这要从处理器执行指令的方式说起。简单来讲，处理器执行指令的流水线是：从内存读取指令-> 解释指令-> 执行指令-> **检测中断**。处理器不是单纯翻译并执行指令，还会在每个指令周期的最后检测一次中断。也就是说，处理器的工作方式决定了检测中断这事，就是自动进行的，不需要你自己编写。回到主程序中的代码，处理器每执行一条指令都会自动附加一次中断检查。主循环不断循环，就能不断地检测中断。因此，我们只需要初始化好相应外设，就能静静等待中断被触发并被处理器响应，进而执行中断服务例程 (反转灯的亮灭)。

当然，即使现在你已经知道了处理器确实在自动检测中断，但可能还有一个问题，处理器是怎么检测中断的呢？在 STM32 中，各个外设都会将中断信息传到 NVIC 中断控制器处 (这是通过电路设计实现的)。处理器只需扫描 NVIC 中断控制器就能获得中断信息。而 NVIC 就是接下来将要讲到的内容。

3.7 NVIC 嵌套中断向量控制器

NVIC(Nested Vectored Interrupt Controller)，译为嵌套中断向量控制器。它与内核紧密耦合，属于内核里面的一个外设，是 ARM Cortex-M 微控制器用于管理中断的核心组件。除了上面实验中见到的使能中断，它还能失能中断，控制中断优先级，处理中断服务例程的嵌套等等。

常见的与 NVIC 相关的库函数有

```
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn); // 使能中断
void HAL_NVIC_DisableIRQ(IRQn_Type IRQn); // 失能中断
void HAL_NVIC_SetPendingIRQ(IRQn_Type IRQn); // 设置中断标志位
void HAL_NVIC_ClearPendingIRQ(IRQn_Type IRQn); // 清除中断标志位
void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority); // 设置中断
    优先级
```

前两个函数我们已经见过了，一部分模块产生中断的功能默认情况下是禁用（失能）的，想要使用需要先使能，如果不想再使用则可以将其失能。而第三个和第四个函数是用来检测和清除中断标志位的，而在上面的实验里我们用另两个函数实现了同样的功能。感兴趣的话可以试试用这两个函数再实现一次上面的功能。

我们的重点在于最后一个函数，中断优先级的设置。

3.7.1 中断优先级

在 GPIO 中断里，我们只需要用到一种中断。但如果需要用到多种中断，或者有多个中断事件同时发生会怎样呢？显然处理器是不能同时处理多个中断事件的。

为了解决这个问题，我们引入中断优先级的概念。当多个中断事件同时发生时，处理器会根据中断事件的重要性从高到低，按顺序处理这些事件。这个重要性排序，就是中断优先级，越重要的中断事件优先级就越高。当然你可以问，你怎么知道对我来讲哪些事件更重要？的确，不同情况下对优先级的要求是不一样的，正因此，HAL 库为我们提供了设置中断优先级的函数`HAL_NVIC_SetPriority`，除了一些不可变的中断事件（比如总线错误，内核错误等），其他的外设中断事件的优先级都是可以调整的。

但这个函数需要提供三个参数，其中`IRQn`是中断号，另外两个是什么呢？

在 NVIC 里有一个专门的寄存器：中断优先级寄存器`NVIC_IPRx`用来配置外部中断的优先级。在 Cortex-M4 内核中，该寄存器的定义如下

```
volatile uint8_t IP[240U];
```

数组中的每个元素都是 8 位无符号整型，储存一个中断的优先级信息，因此理论上最多可以储存 240 个中断的优先级。但 CM4 芯片做了精简设计，导致每个 8 位无符号整型实际只有最高的四位用来表达优先级，最低的四位则保留。而用来表达优先级的这四位，又被分为抢占优先级 (PreemptPriority) 和子优先级 (SubPriority)。如果有多个中断同时响应，抢占优先级高的中断就会比抢占优先级低的中断优先被处理，如果抢占优先级相同，就比较子优先级。如果子优先级还一样，则比较中断号，中断号越小，优先级越高。

在这表达优先级的四位里，有几位表示抢占优先级，剩下几位表示子优先级，是可以自行设置的。根据分配位数的不同，可以给优先级分组编号

```
NVIC_PriorityGroup_0 // 0bit抢占优先级 4bit子优先级
NVIC_PriorityGroup_1 // 1bit抢占优先级 3bit子优先级
NVIC_PriorityGroup_2 // 2bit抢占优先级 2bit子优先级
NVIC_PriorityGroup_3 // 3bit抢占优先级 1bit子优先级
NVIC_PriorityGroup_4 // 4bit抢占优先级 0bit子优先级

void NVIC_PriorityGroup(uint32_t PriorityGroup) {
    NVIC_SetPriorityGrouping(PriorityGroup);
}
```

上面展示了每个分组编号对应的分配情况，以及 HAL 库中设置优先级分组的函数。这里抢占优先级有`n`位，是从最高位向下数`n`位；子优先级有`n`位，是从最低位向上数`n`位。

默认情况下 (即不做任何优先级设置)，各个中断的优先级是按照中断号排序的，中断号越小，优先级越高，但一般情况下我们都会根据所需调整优先级顺序。

3.7.2 关于中断号

在`stm32f411xx.h`中有名为`IRQn_Type`的枚举类型，里面枚举了所有中断类型对应的中断号

```
typedef enum
{
    NonMaskableInt_IRQn     = -14,
    /* 中间部分省略 */
    SysTick_IRQn            = -1,

    WWDG_IRQn               = 0,
    /* 中间部分省略 */
    FPU_IRQn                = 81
} IRQn_type;
```

中断类型可以分为两种，第一种的中断号小于 0，表示中断类型是内核中断；第二种的中断号大于 0，表示中断类型是外设中断。内核中断的优先级是不可编程的 (从对应的中断事件也可以看得出来它们的重要性的确很高)，而外设中断的优先级则可以根据需要改变。

在NVIC_IPRx里，每个中断的优先级信息根据其中断号储存在IP数组中对应的位置上。比如中断号为 6 的 EXTI0 中断，就储存在IP[6]处。对于中断号小于零的中断类型，不妨在__NVIC_SetPriority函数的定义中探索一下对应关系。

3.8 让小灯定时闪烁——定时器中断

这一节我们将尝试另一种中断——定时器中断。定时器 (Timer) 是实现定时功能最基础的外设，在 STM32F4xx 系列中定时器分为几种，但这里只需要用到基本定时器，其余则留到相应章节讲解。

3.8.1 定时器原理

请观看视频基本定时功能，尝试自行复述一遍定时器的计时原理，并在 CubeMX 中正确配置基本定时器。与串口相关的内容可自行略过。

3.8.2 代码编写

我们尝试利用定时器中断实现小灯的定时闪烁。首先在 CubeMX 中设置好预分频和自动重装载值，使得产生中断的间隔为 1s，并开启对应的 NVIC 中断。生成代码后，首先要在主函数中开启基本定时器中断

```
HAL_TIM_Base_Start_IT(&htim6);
```

接着编写中断服务例程。

```
void TIM6_DAC_IRQHandler(void)
{
    HAL_TIM_IRQHandler(&htim6); // 已自动生成
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) // 计数到达自动重装载值时的回调函数
{
    if (htim == (&htim6)) // 检测是否是TIM6到达计数周期
    {
        HAL_GPIO_TogglePin(LD1_GPIO_Port, LD1_Pin); // 反转灯的亮灭
    }
}
```

最后主函数中应该会是这个样子的

```
int main(void) {
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_TIM6_Init();

    HAL_TIM_Base_Start_IT(&htim6);

    while (1) {}
```

```
}
```

编译并烧录，可以发现单片机上的 LED 灯以 1s 的间隔闪烁。

3.9 RTFSC(2)

这里的中断服务例程和 GPIO 中断不同，除了TIM6_DAC_IRQHandler以外，还使用了另一个回调函数，这是为什么呢？事实上，定时器中断可以由很多种中断事件触发，显然需要不一样的处理方法，自然也就需要不同的函数。这里我们用到的 PeriodElapsedCallback函数，就是专门用来处理计时器更新事件的。

我们不妨跳转到HAL_TIM_IRQHandler的定义。这个函数中有很多判断条件，简单来讲，就是根据计数器状态寄存器中的信息，依次判断产生定时器中断的原因 (感兴趣的话可以对照参考手册中关于定时器寄存器的说明，研究一下这些判断语句是怎么读取寄存器中的信息的)。在这个实验中，当定时器中断产生时，处理器进入TIM6_DAC_IRQHandler函数，接着进入HAL_TIM_IRQHandler函数，逐个判断中断事件类型；判断发生的事件是计时器更新事件后，进入PeriodElapsedCallback函数，执行反转灯的亮灭的指令。

```
if ((itflag & (TIM_FLAG_UPDATE)) == (TIM_FLAG_UPDATE))
{
   if ((itsource & (TIM_IT_UPDATE)) == (TIM_IT_UPDATE)) // 判断事件是否为计时器更新事件
   {
     __HAL_TIM_CLEAR_FLAG(htim, TIM_FLAG_UPDATE); // 清除计时器更新事件中断标志位
#if (USE_HAL_TIM_REGISTER_CALLBACKS == 1) // 开启手动注册回调函数机制(不做说明)
     htim->PeriodElapsedCallback(htim);
#else
     HAL_TIM_PeriodElapsedCallback(htim); // 调用处理计时器更新事件的函数
#endif /* USE_HAL_TIM_REGISTER_CALLBACKS */
   }
 }
```

在HAL_TIM_IRQHandler中还有大量的条件编译，但我们在实验中显然没有操作过任何操作过任何一个编译条件，因此暂时可以不管，只要看#else后面的语句就好。有条件的话也可以研究一下，这个编译条件是什么意思，开启之后会发生什么。

3.10 用键盘控制小灯——UART 中断

在这个实验中，我们将利用前面讲到的 UART 串口通讯，让小灯“按着我们的意思”来闪烁。请观看视频串口中断，尝试利用串口通讯实现通过在键盘上输入不同数字，实现改变灯的亮灭。

3.11 缓冲区与消息队列

在中断机制中，缓冲区和消息队列是两种常用的数据结构，它们用于管理和存储中断相关的数据。

3.11.1 缓冲区

缓冲区是内存中的一块区域，用于临时存储数据。在中断上下文中，缓冲区通常用于存储从硬件设备 (如储存介质，键盘等) 传入的数据，或者存储准备发送到硬件设备的数据。当硬件设备准备好接收或发送数据时，它会发出中断信号，操作系统将数据从设备的缓冲区复制到内核的缓冲区，或者从内核的缓冲区复制到设备的缓冲区。

一种特殊的缓冲区叫环形缓冲区，它在数据结构上首尾相连，形成一个环状，因此得名 (实际储存在内存中当然还是线性的)。环形缓冲区环形缓冲区在创建时被分配为一个固定大小的数组，即它有一定的容量限制。它用两个指针来跟踪数据的写入和读取位置。写指针指示下一个数据应该被写入的位置，而读指针指示下一个数据应该被读取的位置。如果写指针追上读指针，新的数据将覆盖旧的数据，或者也可以定义一些其他的处理策略。但普通的线性缓冲区写满后就无法再写入了。相比普通线性缓冲区，环形缓冲区一个很大的优势就是数据被写入和读取时，不需要移动其他数据，大大减少了内存操作的次数。

3.11.2 消息队列

消息队列是一种数据结构，它允许消息按顺序存储和检索。在中断机制中，消息队列用于管理中断请求和中断处理程序的通信。当一个中断发生时，它可以向消息队列中添加一个消息，该消息包含了中断的详细信息。操作系统或中断服务例程可以按顺序处理消息队列中的消息，确保每个中断都得到适当的响应。

消息队列的好处是它提供了一种有序的方式来处理多个中断，可以确保高优先级的中断得到优先处理，并且不会因为低优先级的中断而延迟。