

嵌入式学习 笔记

姓名

黄远灿

联系方式

18878448766

邮箱

hthousandflame@gmail.com

Github

<https://github.com/FlameKm>

目录

1. 单片机 (STM32 为主)	10
1.1. 编码器	10
1.1.1. 相关函数	10
1.2. PWM	10
1.2.1. 注意事项	10
1.2.2. 代码框架	10
1.3. 复用功能	12
1.4. 时钟	12
1.5. CAN	12
1.5.1. 标准帧	12
1.5.2. 报文	12
1.5.3. 邮箱	12
1.5.4. 波特率	12
1.5.5. 过滤器	13
1.5.6. 模式	13
1.5.7. 过滤器	13
1.6. 固件升级	14
1.7. ADC	14
1.7.1. 周期	14
1.8. BootLoad	14
1.8.1. 跳转方式	14
1.8.2. 跳转前准备	15
1.8.3. 程序烧录 XModem、YModem	15
1.9. 中断向量表	15
2. STM32Cube MX	16
2.1. 初尝试	16
2.1.1. 晶振配置	16
2.1.2. 工程	16
2.2. 串口	16
2.2.1. 配置	16
2.2.2. 中断	17

2.2.3. HAL 库 UART 函数库介绍	17
2.2.4. UART 接收中断	18
2.2.5. 串口发送/接收函数	18
2.2.6. 重新定义 printf 函数	19
2.2.7. 串口再使用的笔记	20
2.2.8. DMA 串口使用	22
2.3. PWM	23
2.4. ADC	23
2.4.1. 单通道采样 (不用 DMA)	23
2.4.2. 多通道 (非 DMA)	24
2.4.3. 多通道 (DMA)	25
2.5. IIC	26
2.6. 定时器	27
2.7. CAN	27
2.7.1. 时钟	27
2.7.2. 发送	27
2.7.3. 中断	27
3. FREERTOS	28
3.1. 调度	28
3.2. 时间	28
3.2.1. 绝对的延迟时间	28
3.2.2. pdMS_TO_TICKS(ms)	28
3.2.3. 获取当前时刻函数	28
3.3. 软件定时器	29
3.4. 多任务变量	29
3.5. 对 cubemx 创建的 FreeRtos 封装理解	30
3.6. 任务管理	31
3.6.1. 使用 FreeRTOS 源码创建任务	31
3.6.2. 任务删除	32
3.6.3. 任务暂停	32
3.6.4. 任务恢复	32
3.6.5. 注意	32
3.7. 任务优先级	32
3.7.1. 设置	32

3.7.2. 创建任务附带优先级	33
3.7.3. 资源退让	33
3.7.4. 查询当前任务优先级	33
3.8. 信号量	33
3.8.1. 二进制信号量	33
3.8.2. 计数信号量	33
3.9. 事件组	33
3.9.1. 等待	33
3.9.2. 同步 SYNC	34
3.10. 任务通知	34
3.10.1. 创建	34
3.10.2. 取代二进制信号量	34
3.10.3. 取代信号组	34
3.10.4. 模拟队列	35
3.11. 消息队列	35
4. 计算机语言	37
4.1. C 语言	37
4.1.1. 优先级	37
4.1.2. qsort	37
4.1.3. 数据类型	38
4.1.4. 编译器优化	38
4.2. JAVA	38
4.2.1. 安装	38
4.2.2. 选择版本	38
4.3. OOP FOR C++	38
4.3.1. 多态	38
5. 数据结构和算法	40
5.1. 抽象数据类型 ATD	40
5.2. PID 算法	40
5.2.1. P (比例)	40
5.2.2. I (积分)	40
5.2.3. D (微分)	40
5.2.4. 增量式和位置式	40

5.2.5. 调参口诀	41
5.3. 环形缓冲区	42
5.3.1. 裸机中利用 DMA 使用环形缓冲区实现异步发送	42
6. 其它(单片机)	45
6.1. 高级定时器注意事项	45
6.2. ROM 烧录	45
7. 嵌入式 Linux	46
7.1. 开发环境	46
7.2. 基础命令	46
7.3. VIM 快捷键	47
7.4. 远程连接 tftp	47
7.5. NFS 挂载	47
7.5.1. 安装	47
7.5.2. 挂载	47
7.5.3. 重启	47
7.6. 网络	47
7.6.1. 改变 IP	47
7.6.2. 获取 IP 赋值给变量	48
7.7. 图形化	48
7.8. Ubuntu 下载源	48
7.9. SSH 服务和公钥登录	49
7.9.1. 安装	49
7.9.2. 登录	49
7.9.3. 创建公钥	49
7.9.4. 使用公钥	49
7.10. 后台运行	49
7.11. 自启动 Service	50
7.12. 用户	51
7.12.1. 创建用户	51
7.12.2. 删除用户	51
7.12.3. 给予 sudo 权限	51
7.12.4. 切换用户	51

7.12.5. 修改用户名和密码	51
7.13. crontab 定时任务	52
7.14. screen	52
7.14.1. 命令	52
7.14.2. 快捷键	52
7.15. 路由器	52
7.15.1. 刷机前的准备工作	52
7.15.2. 刷入 openwrt	53
7.16. 交叉编译	54
7.17. 驱动	54
7.17.1. 安装卸载	54
7.18. GPIO 子系统	54
7.18.1. 查看 gpio 使用状态	54
7.18.2. 确定 GPIO 引脚的编号	54
7.18.3. shell 控制	54
7.18.4. GPIO 子系统函数	55
7.19. 中断	55
7.19.1. 流程	55
7.19.2. 获取中断号	56
7.19.3. 获取中断名称	56
7.19.4. 触发方式类型	56
7.20. zsh (oh my zsh)	56
8. DOCKER 容器	57
8.1. 添加容器	57
8.2. 进入容器	57
8.3. 退出容器	57
8.4. 查看容器	57
8.5. 镜像	57
9. QT	58
9.1. Linux 安装	58
9.2. 槽函数	58
9.2.1. 定义	58
9.2.2. 举例:	58

9.2.3. 个人理解:	58
9.3. 字符串	58
9.3.1. 数据类型转换为字符串	58
9.3.2. 字符串转数据类型	59
9.4. 打包可执行文件 (WIN)	59
9.5. 模块	59
9.5.1. 安装	59
9.5.2. 添加附加模组 (串口为例)	59
9.6. Painter 画家	60
9.6.1. 注意事项	60
9.6.2. 刷新	60
9.6.3. 美术类型	60
9.6.4. 设置零点	61
9.7. QWidget 控件类	61
9.7.1. 设置位置和大小	61
9.7.2. 显示控件	61
9.7.3. 自定义美工	61
9.8. 键盘鼠标	61
9.9. 使用自定义字体	62
9.9.1. 添加进项目	62
9.9.2. 加载字体	63
9.9.3. 使用	63
9.10. 设置屏幕大小	63
9.11. 定时器	63
9.12. CLION 配置工具	64
10. 树莓派	66
10.1. 交叉编译器	66
10.2. 交叉编译	66
10.2.1. 安装	66
10.2.2. 问题	66
10.3. 引脚图	67
10.4. 打开串口	67

11. CMake	68
11.1. 交叉编译	68
11.2. 构建静态库和动态库	68
11.3. 添加非标准库	68
12. 工具链	69
12.1. Ninja	69
12.2. MakeFile	69
12.3. GCC	70
12.3.1. 编译步骤	70
12.3.2. 动态链接库	71
12.3.3. 静态链接库	72
12.4. OpenOCD	72
12.4.1. 开启	72
12.4.2. 进入后台	73
12.4.3. 烧录	73
12.4.4. 更简单的方法（本地）	73
13. USB	75
13.1. 概况	75
13.1.1. USB 通信过程简介	75
13.1.2. USB 枚举过程简介	75
13.2. 传输速度	75
13.3. 描述符	76
13.3.1. 设备描述符	76
13.3.2. 配置描述符	77
13.3.3. 接口关联描述符（IAD）	77
13.3.4. 端点描述符	78
13.3.5. 字符串描述符	78
13.3.6. 其他杂项描述符类型	78
13.3.7. 使用多个 USB 描述符	78
13.4. 枚举	79
13.4.1. 动态检测	79
13.4.2. 枚举	79
13.4.3. 配置	80

13. 5.	获取设备 pid 和 vid	80
13. 6.	HID 人机类	81
13. 7.	MSD 大容量类	81
13. 8.	CDC 通信设备类	81
13. 9.	HIDAPI	81
13. 10.	DFU	81
14.	GIT 版本控制	82
14. 1.	常用命令	82
14. 2.	下载	82
14. 3.	上传	82
14. 4.	更新	83
14. 5.	版本回退	83
14. 6.	子模块 submodule	83
14. 7.	重新排除.gitignore	84
15.	网络	85
15. 1.	git	85
15. 2.	linux	85
16.	芯片或模块	86
16. 1.	EEPROM	86
17.	Zephyr	87
17. 1.	个人理解	87
17. 2.	Hello World	87
17. 2. 1.	安装	87
17. 2. 2.	编译烧录（支持的开发板）	87
17. 3.	调度算法	88
17. 4.	启动方式	88
17. 5.	时间	89
17. 5. 1.	定时器	89
17. 5. 2.	计时器	90
17. 5. 3.	时间转换	90
17. 6.	设备树	91
17. 6. 1.	节点	91
17. 6. 2.	特殊节点	93
17. 6. 3.	设备	93

17.6.4. 命名	94
17.6.5. 注册	94
17.6.6. 绑定	95
17.7. 信号（原子操作）	95
17.7.1. 信号量 Semaphores	95
17.7.2. 互斥量 Mutexes	95
17.7.3. 互斥量 Futex	96
17.7.4. 条件变量	96
17.8. 数据传输	97
17.8.1. Queue（底层实现）	97
17.8.2. FIFOs（底层实现）	97
17.8.3. LIFO（底层实现）	97
17.8.4. 栈 Stacks	97
17.8.5. 消息队列 MessageQueues	98
17.8.6. 邮箱 Mailboxes	98
17.8.7. 管道	99

1. 单片机（STM32 为主）

1.1. 编码器

1.1.1. 相关函数

```
void TIM_EncoderInterfaceConfig(TIM_TypeDef* TIMx, uint16_t TIM_EncoderMode, uint16_t TIM_IC1Polarity, uint16_t TIM_IC2Polarity)
```

TIMx 参数就是使用哪个定时器作为编码器接口的捕捉定时器。

TIM_EncoderMode 参数是模式,是单相计数（只能反映速度）还是两相计数（速度和方向）。

TIM_IC1Polarity 和 TIM_IC2Polarity 参数就是通道 1、2 的捕捉极性。

Table 92. Counting direction versus encoder signals

Active edge	Level on opposite signal (TI1FP1 for TI2, TI2FP2 for TI1)	TI1FP1 signal		TI2FP2 signal	
		Rising	Falling	Rising	Falling
Counting on TI1 only	High	Down	Up	No Count	No Count
	Low	Up	Down	No Count	No Count
Counting on TI2 only	High	No Count	No Count	Up	Down
	Low	No Count	No Count	Down	Up
Counting on TI1 and TI2	High	Down	Up	Up	Down
	Low	Up	Down	Down	Up

图 1-1

1.2. PWM

1.2.1. 注意事项

该函数用来初始化 TIM1_CH1 的 PWM 输出（PA8），其原理同之前介绍的 PWM 输出一模一样，只是换过一个定时器而已。不过这里 TIM1 是高级定时器，高级定时器的 PWM 输出，与普通定时器稍有区别，必须通过函数 `TIM_CtrlPWMOutputs` 来设置 BDTR 寄存器的 MOE 位为 1，才可以正常输出 PWM，这点要特别注意。

1.2.2. 代码框架

```
//TIM1 CH1 PWM 输出设置
```

```
//PWM 输出初始化
```

```
//arr: 自动重装值
```

```
//psc: 时钟预分频数
```

```
void TIM1_PWM_Init(u16 arr,u16 psc)
```

```
{
```

```
GPIO_InitTypeDef GPIO_InitStructure;
```

```
TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
```

```
TIM_OCInitTypeDef TIM_OCInitStructure;
```

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE); //使能 TIMx 外设
```

```

RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); //使能 PA 时钟
//设置该引脚为复用输出功能,输出 TIM1 CH1 的 PWM 脉冲波形
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8; //TIM1_CH1
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; //复用功能输出
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOA, &GPIO_InitStructure); //初始化 GPIO

TIM_TimeBaseStructure.TIM_Period = arr; //设置自动重装载周期值
TIM_TimeBaseStructure.TIM_Prescaler =psc; //设置预分频值 不分频
TIM_TimeBaseStructure.TIM_ClockDivision = 0; //设置时钟分割:TCTS = Tck_tim
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计数
TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure); //初始化 TIMx
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM2; //CH1 PWM2 模式
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //比较输出使能
TIM_OCInitStructure.TIM_Pulse = 0; //设置待装入捕获比较寄存器的脉冲值
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low; //OC1 低电平有效
TIM_OC1Init(TIM1, &TIM_OCInitStructure); //根据指定的参数初始化外设 TIMx
TIM_OC1PreloadConfig(TIM1, TIM_OCPreload_Enable); //CH1 预装载使能
TIM_ARRPreloadConfig(TIM1, ENABLE); //使能 TIMx 在 ARR 上的预装载寄存器
TIM_CtrlPWMOutputs(TIM1,ENABLE); //MOE 主输出使能,高级定时器必须开启
TIM_Cmd(TIM1, ENABLE); //使能 TIMx
}

```

1.3. 复用功能

中文手册 105 页

1.4. 时钟

在 STM32 中

APB1(低速外设)上的设备有：电源接口、备份接口、CAN、USB、I2C1、I2C2、UART2、UART3、SPI2、窗口看门狗、Timer2、Timer3、Timer4 。

APB2(高速外设)上的设备有：GPIO_A-E、USART1、ADC1、ADC2、ADC3、TIM1、TIM8、SPI1、ALL。

具体参考时钟树

1.5. CAN

参考链接

<https://zhuanlan.zhihu.com/p/548772223>

CAN 不能与 USB 从设备同时使用，因为他们共用一组 RAM，如果需要使用 USB 功能时，需要调用关闭 CAN 总线的初始化函数。需要使用 CAN 的时候，需要调用开启 CAN 总线的初始化函数。

在 HAL 库中，

```
HAL_CAN_MspDeInit(&hcan); //关闭
```

```
HAL_CAN_MspInit(&hcan); //初始化
```

1.5.1. 标准帧

1.5.2. 报文

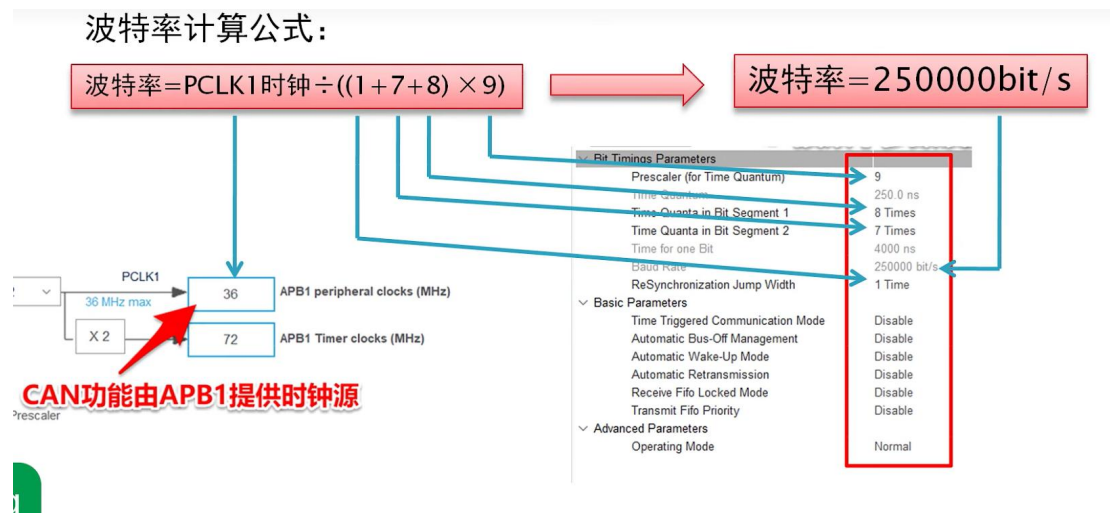
报文是短帧结构，短的传送时间使其受干扰概率低，CAN 有很好的校验机制，这些都保证了 CAN 通信的可靠性。

1.5.3. 邮箱

有两个队列，fifo0 和 fifo1，每一个队列又存在三个邮箱 BOX1-BOX3。

fifo0 和 fifo1 不同的中断

1.5.4. 波特率



波特率 = PCLK1 时钟 / ((1 + 7 + 8) *9)

1. 5. 5. 过滤器

1. 5. 6. 模式

Lookback, 用于测试代码, 无法从接收外部数据

1. 5. 7. 过滤器

```
static void CANFilter_Config(void)
{
    CAN_FilterTypeDef sFilterConfig;
    sFilterConfig.FilterBank = 0; //CAN 过滤器编号, 范围 0-27
    sFilterConfig.FilterMode = CAN_FILTERMODE_IDMASK; //CAN 过滤器模式, 掩码
    模式或列表模式
    sFilterConfig.FilterScale = CAN_FILTERSCALE_32BIT; //CAN 过滤器尺度, 16 位或 3
    2 位
    sFilterConfig.FilterIdHigh = 0x000 << 5; //32 位下, 存储要过滤 ID 的高 16 位
    sFilterConfig.FilterIdLow = 0x0000; //32 位下, 存储要过滤 ID 的低 1
    6 位
    sFilterConfig.FilterMaskIdHigh = 0x0000; //掩码模式下, 存储的是掩码
    sFilterConfig.FilterMaskIdLow = 0x0000;
    sFilterConfig.FilterFIFOAssignment = 0; //报文通过过滤器的匹配后, 存
    储到哪个 FIFO
    sFilterConfig.FilterActivation = ENABLE; //激活过滤器
    sFilterConfig.SlaveStartFilterBank = 0;

    if (HAL_CAN_ConfigFilter(&hcan1, &sFilterConfig) != HAL_OK) {
        printf("CAN Filter Config Fail!\r\n");
    }
}
```

```

        Error_Handler();
    }
    printf("CAN Filter Config Success!\r\n");
}

```

1.6. 固件升级

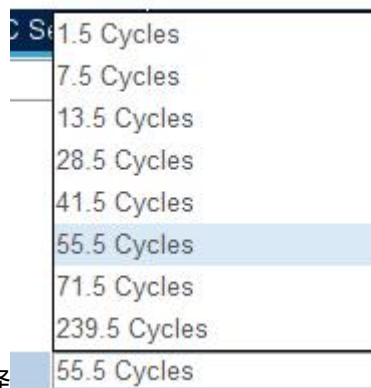
参考链接：

<https://zhuanlan.zhihu.com/p/73037424>

CMSIS-DAP 调试器是有单独的一条复位线的，但是当时没考虑它可能采用了这种方式，只考虑可能采用了内核复位

1.7. ADC

1.7.1. 周期



采集固定 12.5 个周期，间隔可以选择 55.5 Cycles，计算采样频率的时候为：ADC 时钟频率/ (55.5+12.2)

1.8. BootLoad

1.8.1. 跳转方式

```

typedef void (*fp_t)(void);
__IO uint32_t BootloaderAddr = 0x08004000; // 32k = 0x8000
void jump_app()
{
    volatile uint32_t jump_addr;
    fp_t jump;
    jump_addr = *(volatile uint32_t *) (BootloaderAddr + 4);
    jump = (fp_t) jump_addr;
    __set_MSP(*(volatile uint32_t *) BootloaderAddr);
    jump();
}

```

此时可以跳转，但是跳转后会出现问题，笔者的开发环境下中断会在 bootload 程序中，可以定位问题为中断定位错误，解决方式是将 APP 程序的中断向量表进行偏移，具体方式请查看中断向量表一节。

上面值得注意的有跳转+4 是表示前面有中断跳转，此时才是真正的 main 函数开端

1.8.2. 跳转前准备

需要清除全部中断与中断标志位。

1.8.3. 程序烧录 XModem、YModem

两种烧录协议

1.9. 中断向量表

更改中断向量表起始地址，需要取消注释，并修改偏移地址，修改的文件为 system_stm32f1xx.c。

```
configuration: */
/*!< Uncomment the following line if you need to relocate the vector table
anywhere in Flash or Sram, else the vector table is kept at the automatic
remap of boot address selected */
#define USER_VECT_TAB_ADDRESS
#if defined(USER_VECT_TAB_ADDRESS)
/*!< Uncomment the following line if you need to relocate your vector Table
in Sram else user remap will be done in Flash. */
/* #define VECT_TAB_SRAM */
#if defined(VECT_TAB_SRAM)
#define VECT_TAB_BASE_ADDRESS SRAM_BASE /*!< Vector Table base address field.
This value must be a multiple of 0x200. */
#define VECT_TAB_OFFSET 0x00000000U /*!< Vector Table base offset field.
This value must be a multiple of 0x200. */
#else
#define VECT_TAB_BASE_ADDRESS FLASH_BASE /*!< Vector Table base address field.
This value must be a multiple of 0x200. */
#define VECT_TAB_OFFSET 0x00040000 /*!< Vector Table base offset field.
This value must be a multiple of 0x200. */
#endif /* VECT_TAB_SRAM */
#endif /* USER_VECT_TAB_ADDRESS */
```

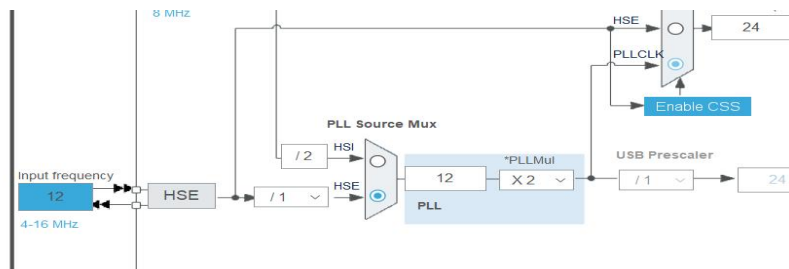
stm32cube+clion 环境中，修改.ld 文件可以修改烧录起始地址

```
system_stm32f1xx.c  STM32F103C8TX_FLASH.ld x CMakeLists_template.txt CMakeLists.txt
42 _Min_Stack_Size = 0x400; /* required amount of stack */
43
44 /* Memories definition */
45 MEMORY
46 {
47     RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 20K
48     FLASH (rx) : ORIGIN = 0x8004000, LENGTH = 48K
49 }
50
51 /* Sections */
```


2. STM32Cube MX

2.1. 初尝试

2.1.1. 晶振配置



2.1.2. 工程

STM32Cube MCU packages and embedded software packs

- ☐ Copy all used libraries into the project folder
- ☒ Copy only the necessary library files
- ☐ Add necessary library files as reference in the toolchain project configuration file

Generated files

- ☒ Generate peripheral initialization as a pair of '.c/.h' files per peripheral
- ☐ Backup previously generated files when re-generating
- ☒ Keep User Code when re-generating
- ☒ Delete previously generated files when not re-generated

HAL Settings

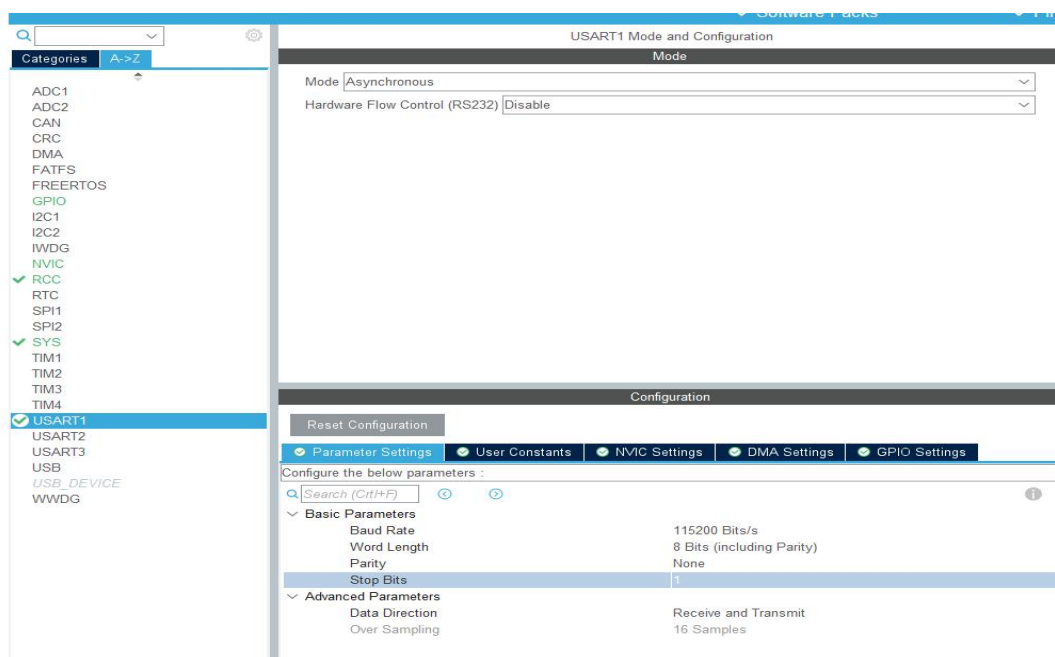
- ☐ Set all free pins as analog (to optimize the power consumption)
- ☐ Enable Full Assert

2.2. 串口

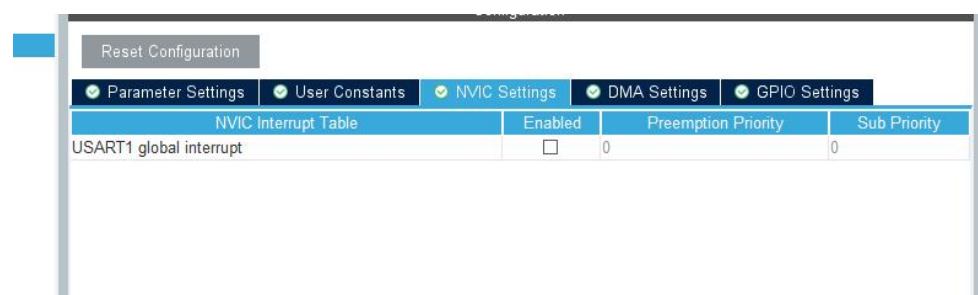
原文链接: <https://blog.csdn.net/as480133937/article/details/99073>

783

2.2.1. 配置



2. 2. 2. 中断



2. 2. 3. HAL 库 UART 函数库介绍

UART 结构体定义

```
UART_HandleTypeDef huart1;
```

UART 的名称定义，这个结构体中存放了 UART 所有用到的功能，后面的别名就是我们所用的 uart 串口的别名，默认为 huart1

```

typedef struct __UART_HandleTypeDef
{
    USART_TypeDef          *Instance;      /*!< USART寄存器基址          */
    UART_InitTypeDef       Init;          /*!< UART通信参数            */
    uint8_t                *pTxBuffPtr;   /*!< 指向UART TX传输缓冲区的指针 */
    uint16_t               TxXferSize;    /*!< UART TX传输字节          */
    __IO uint16_t          TxXferCount;    /*!< UART TX传输计数器        */
    uint8_t                *pRxBuffPtr;   /*!< 指向UART RX传输缓冲区的指针 */
    uint16_t               RxXferSize;    /*!< 指向UART RX传输缓冲区的指针 */
    __IO uint16_t          RxXferCount;    /*!< UART RX传输计数器        */
    DMA_HandleTypeDef       *hdmatx;       /*!< UART TX DMA句柄参数      */
    DMA_HandleTypeDef       *hdmarx;       /*!< UART RX DMA句柄参数      */
    HAL_LockTypeDef        Lock;           /*!< 锁定对象                  */
    __IO HAL_UART_StateTypeDef gState;     /*!< UART状态信息与全局处理管理有关，也与TX操作有关。 */
    __IO HAL_UART_StateTypeDef RxState;    /*!< 与RX操作相关的UART状态信息。 */
    __IO uint32_t          ErrorCode;      /*!< UART错误代码            */
} UART_HandleTypeDef;

```

<https://blog.csdn.net/as480133937>

2.2.4. UART 接收中断

原文链接：

<https://blog.csdn.net/as480133937/article/details/99073783>

中断函数

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
```

2.2.5. 串口发送/接收函数

```
HAL_UART_Receive_IT(&huart1, (uint8_t *)&aRxBuffer, 1);
```

HAL_UART_Transmit();//串口发送数据，使用超时管理机制

HAL_UART_Receive();//串口接收数据，使用超时管理机制

HAL_UART_Transmit_IT();//串口中断模式发送

HAL_UART_Receive_IT();//串口中断模式接收

HAL_UART_Transmit_DMA();//串口 DMA 模式发送

HAL_UART_Receive_DMA();//串口 DMA 模式接收

这几个函数的参数基本都是一样的，我们挑两个讲解一下

串口发送数据：

```
HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t Timeout)
```

功能：串口发送指定长度的数据。如果超时没发送完成，则不再发送，返回超时标志（HAL_TIMEOUT）。

参数：

UART_HandleTypeDef *huart UATR 的别名 如： UART_HandleTypeDef huart1;

别名就是 huart1

*pData 需要发送的数据

Size 发送的字节数

Timeout 最大发送时间，发送数据超过该时间退出发送

举例： HAL_UART_Transmit(&huart1, (uint8_t *)ZZX, 3, 0xffff); //串口发送三个字节数据，最大传输时间 0xffff

中断接收数据：

HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)

功能：串口中断接收，以中断方式接收指定长度数据。

大致过程是，设置数据存放位置，接收数据长度，然后使能串口接收中断。接收到数据时，会触发串口中断。

再然后，串口中断函数处理，直到接收到指定长度数据，而后关闭中断，进入中断接收回调函数，不再触发接收中断。（只触发一次中断）

参数：

UART_HandleTypeDef *huart UATR 的别名 如： UART_HandleTypeDef huart1;
 别名就是 huart1

*pData 接收到的数据存放地址

Size 接收的字节数

举例： HAL_UART_Receive_IT(&huart1,(uint8_t *)&value,1); //中断接收一个字符，存储到 value 中

2.2.6. 重新定义 printf 函数

在 stm32f4xx_hal.c 中包含#include <stdio.h>

#include "stm32f4xx_hal.h"

#include <stdio.h>

extern UART_HandleTypeDef huart1; //声明串口

在 stm32f4xx_hal.c 中重写 fgetc 和 fputc 函数

/**

* 函数功能: 重定向 c 库函数 printf 到 DEBUG_USARTx

* 输入参数: 无

* 返回值: 无

* 说明: 无

*/

int fputc(int ch, FILE *f)

{

HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xffff);

return ch;

```

}
/**
 * 函数功能: 重定向 c 库函数 getchar,scanf 到 DEBUG_USARTx
 * 输入参数: 无
 * 返回值: 无
 * 说明: 无
 */
int fgetc(FILE *f)
{
    uint8_t ch = 0;
    HAL_UART_Receive(&huart1, &ch, 1, 0xffff);
    return ch;
}

```

2.2.7. 串口再使用的笔记

配置好需主动开启中断

```

HAL_UART_Receive_IT(&huart1, (uint8_t *)&aRxBuffer, 1);
HAL_UART_Receive_IT(&huart2, (uint8_t *)&aRxBuffer, 1);

```

通过自己写的函数实现获取数据

注意最后一句

```

if (USART1_RX_STA & 0xC000)
{
    printf("999");
    printf("%s", USART1_RX_BUF);
    USART1_RX_STA = 0; // 重置接收
}

```

文件在 stm32f1xx_it.c 和 stm32f1xx_it.h 里面

```

uint8_t Res;
/* USER CODE END USART1_IRQn 0 */
HAL_UART_IRQHandler(&huart1);
/* USER CODE BEGIN USART1_IRQn 1 */

Res = aRxBuffer;
// printf("%c", Res); //把收到的数据以 a 符号变量 发送回电脑
if ((USART1_RX_STA & 0x8000) == 0) //接收未完成
{
    if (USART1_RX_STA & 0x4000) //接收到了0x0d
    {
        if (Res != 0x0a)
            USART1_RX_STA = 0; //接收错误,重新开始
        else
            USART1_RX_STA |= 0x8000; //接收完成了
    }
    else //还没收到0x0d
    {
        if (Res == 0x0d)
            USART1_RX_STA |= 0x4000;
        else
        {
            USART1_RX_BUF[USART1_RX_STA & 0x3FFF] = Res; //将收到的数据放入数组
            USART1_RX_STA++; //数据长度计数加1
            if (USART1_RX_STA > (USART1_REC_LEN - 1))
                USART1_RX_STA = 0; //接收数据错误,重新开始接收
        }
    }
}

HAL_UART_Receive_IT(&huart1, (uint8_t *)&aRxBuffer, 1); //再开启接收中断
/* USER CODE END USART1_IRQn 1 */

```

注意再调用

2. 2. 8. DMA 串口使用

配置

Configuration

Reset Configuration

Parameter Settings User Constants NVIC Settings DMA Settings GPIO Settings

DMA Request	Stream	Direction	Priority
USART6_TX	DMA2 Stream 6	Memory To Peripheral	Low

Add Delete

DMA Request Settings

Mode Normal 2

Increment Address ☐

Peripheral 3 Memory ☒

Use Fifo ☐ Threshold 1

Data Width 4 Byte Byte

Burst Size 4 1

Normal 正常模式，DMA 发送一次就停止发送；

Circular 循环模式，会一直发送数据；

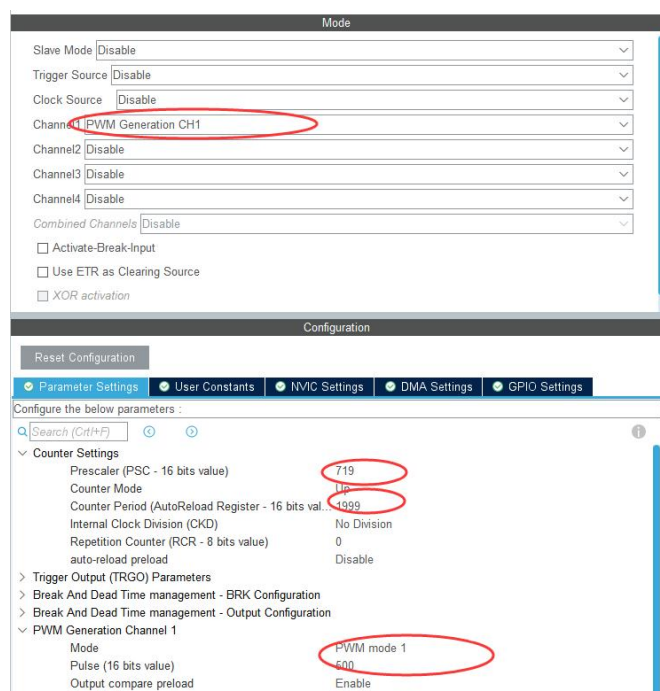
发送函数：

```
HAL_UART_Transmit_DMA( &huart6, (uint8_t *)"hello DISCO by DMA\r\n", sizeof("hello DISCO by DMA\r\n") );
```

注意事项：

打开 DMA 会自动开启 DMA 中断,此时应该打开对应串口中断。

2. 3. PWM



14.4.13 捕获/比较寄存器 1(TIMx_CCR1)

偏移地址: 0x34

复位值: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
I'W I'W I'W I'W I'W I'W I'W I'W I'W I'W I'W I'W I'W I'W I'W															
位15:0															
CCR1[15:0]: 捕获/比较1的值 (Capture/Compare 1 value) 若CC1通道配置为输出: CCR1 包含了装入当前捕获/比较1寄存器的值(预装载值)。 如果在TIMx_CCMR1寄存器(OC1PE位)中未选择预装载特性, 写入的数值会被立即传输至当前寄存器中。否则只有当更新事件发生时, 此预装载值才传输至当前捕获/比较1寄存器中。 当前捕获/比较寄存器参与同计数器TIMx_CNT的比较, 并在OC1端口上产生输出信号。 若CC1通道配置为输入: CCR1 包含了由上一次输入捕获1事件(IC1)传输的计数器值。															

```
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_1);
```

```
TIM1->CCR1 = 1500;
```

or

```
_HAL_TIM_SetCompare(&htim3, TIM_CHANNEL_1, pwmVal);
```

2. 4. ADC

2. 4. 1. 单通道采样 (不用 DMA)



使用函数

Init:

```
HAL_ADCEX_Calibration_Start(&hadc1);
```

Begin:

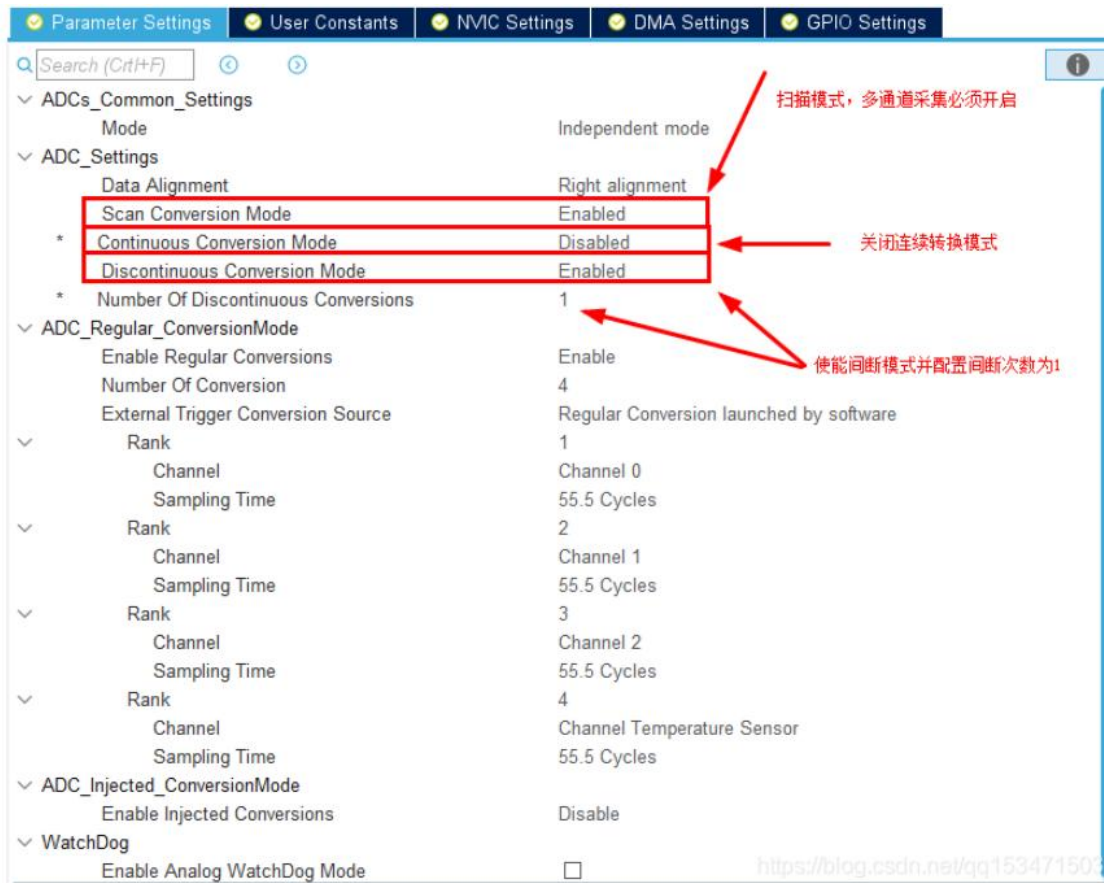
```
HAL_ADC_Start(&hadc1); //启动 ADC 转换
```

```
HAL_ADC_PollForConversion(&hadc1, 50); //等待转换完成, 50 为最大等待时间, 单位为 ms
```

Get:

```
if (HAL_IS_BIT_SET(HAL_ADC_GetState(&hadc1), HAL_ADC_STATE_REG_EOC)) {
    adcValue = HAL_ADC_GetValue(&hadc1); //获取 AD 值
    logi("ADC1 Reading : %d", adcValue);
}
```

2.4.2. 多通道 (非 DMA)



如图设置，只需要按照 1 的方法多次读取即可。

轮询读取

2.4.3. 多通道 (DMA)

2.4.3.1. 配置

需要先开启连续转换模式



1. 高级优先级
2. 循环模式，读取完成后继续重新读取。

3. 勾选寄存器模式：如果不勾选，每次都只存放在一个寄存器。勾选后，每次存放的地址增加
4. 16 位宽带寄存器，12 位的 ADC 足够。

2. 4. 3. 2. 使用

Init: HAL_ADCEx_Calibration_Start(&hadc1); //校准
 uint16_t adcValue[2];
 HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adcValue, 2);
 Get: 直接使用数据。

2. 4. 3. 3. 注意

1. 自动生成的 DMA 和 ADC 的初始化顺序不能相反。先 DMA 再 ADC

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_USART1_UART_Init();
MX_TIM3_Init();
MX_DMA_Init();
MX_ADC1_Init();
/* USER CODE BEGIN 2 */
```

2. 小心一直进入中断!
3. 传输的数值选择的是半字节的话，需要对应的内存去存储

2. 5. IIC

默认配置即可

发送函数

方法一

```
while (HAL_I2C_Mem_Write(&hi2c1, 0x78, 0x00,
I2C_MEMADD_SIZE_8BIT, &IIC_Command,
sizeof(IIC_Command), 10) != HAL_OK) {
if (HAL_I2C_GetError(&hi2c1) != HAL_I2C_ERROR_AF) {
Error_Handler();
}
}
```

方法二

发送单个数据,数据的第一个数据为内存地址

0x78 是设备地址

```
HAL_I2C_Master_Transmit(&hi2c1, 0x78, c, sizeof(c), 10);
```

方法三

需要在 cubemx 配置 slave 的地址

```
uint8_t c[2] = {0x40, IIC_Data};

HAL_I2C_Slave_Transmit(&hi2c1, c, sizeof(c), 10);
```

2.6. 定时器

定时器回调函数:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
```

中断

```
HAL_TIM_Base_Start_IT
```

PWM

```
HAL_TIM_PWM_Start
```

2.7. CAN

2.7.1. 时钟

使用 APB1 时钟

Prescaler (for Time Quantum)	3
Time Quantum	71.42857142857143 ns
Time Quanta in Bit Segment 1	9 Times
Time Quanta in Bit Segment 2	4 Times
Time for one Bit	1000 ns
Baud Rate	1000000 bit/s
ReSynchronization Jump Width	1 Time

2.7.2. 发送

添加到邮箱

```
HAL_StatusTypeDef HAL_CAN_AddTxMessage(CAN_HandleTypeDef *hcan, const CAN_TxHeaderTypeDef *pHeader, const uint8_t aData[], uint32_t *pTxMailbox)
```

2.7.3. 中断

3. FREERTOS

Clion 编译问题

```
add_compile_options(-mfloat-abi=hard -mfpu=fpv4-sp-d16)

add_link_options(-mfloat-abi=hard -mfpu=fpv4-sp-d16)
```

这两行需要反注释, 在 Cmake 中

3.1. 调度

FreeRTOS 使用的是优先级抢占式调度算法。这意味着任务的优先级决定了它们被调度的顺序, 优先级高的任务能够抢占正在执行的优先级低的任务。这种调度算法允许高优先级的任务在有需要的时候立即抢占 CPU 控制权, 从而满足实时系统的需求。

高低优先级: 抢占式

同等优先级: 时间片

3.2. 时间

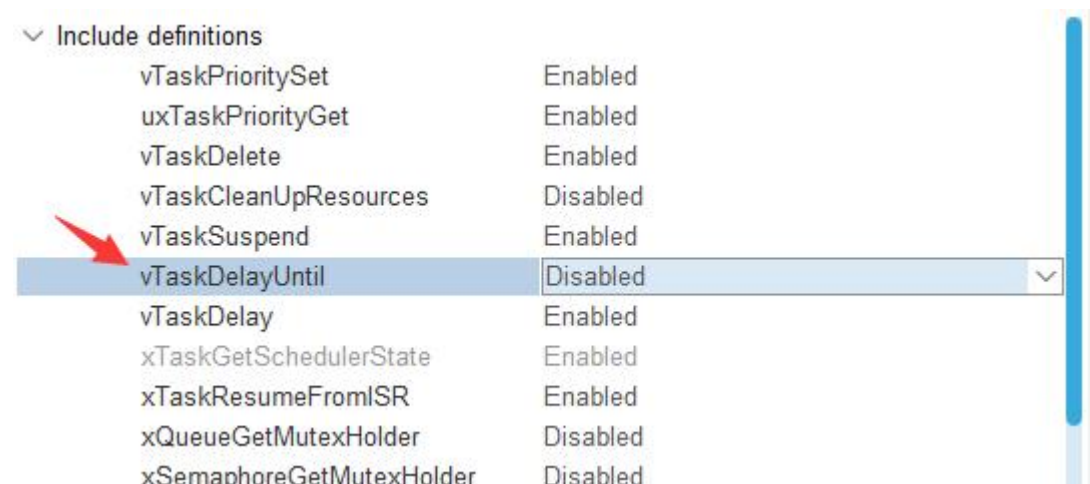
3.2.1. 绝对的延迟时间

源码中:

```
TickType_t xLastWakeTime = xTaskGetTickCount(); //获取当前时间
```

```
vTaskDelayUntil(&xLastWakeTime, 3000); //延时 3 秒
```

cubemx 中 osTask



需要开启才可使用 本大点中的 1, 2 点

3.2.2. pdMS_TO_TICKS(ms)

这个宏的意义是将 ms 时间转换为 ticks, 当然在 hal 库生成的 osdelay 已经自动转换了, 就可以不用使用这个宏。

3.2.3. 获取当前时刻函数

```
uint32_t xTaskGetTickCount();
```

3.3. 软件定时器

cubemx 配置时候需要在 Config parameters 中开启 Software timer definitions

Software timer definitions	
USE_TIMERS	Enabled
* TIMER_TASK_PRIORITY	2
* TIMER_QUEUE_LENGTH	10
* TIMER_TASK_STACK_DEPTH	256 Words

freeRtos 源码生成:

```
lockHandle = xTimerCreate("Lock Car",
2000,
pdFALSE,
(void *)0,
lockCarCallback);
checkHandle = xTimerCreate("Sensors Check",
100,
pdTRUE,
(void *)1,
checkCallback);
//必须要在 portMAX_DELAY 内开启 timer start
//portMAX_DELAY is listed as value for waiting indefinitely
//实际上 0xFFFFFFFF 2^32-1 49 天 7 周
//在此期间, 此 task 进入 Block 状态
xTimerStart(checkHandle, portMAX_DELAY);
```

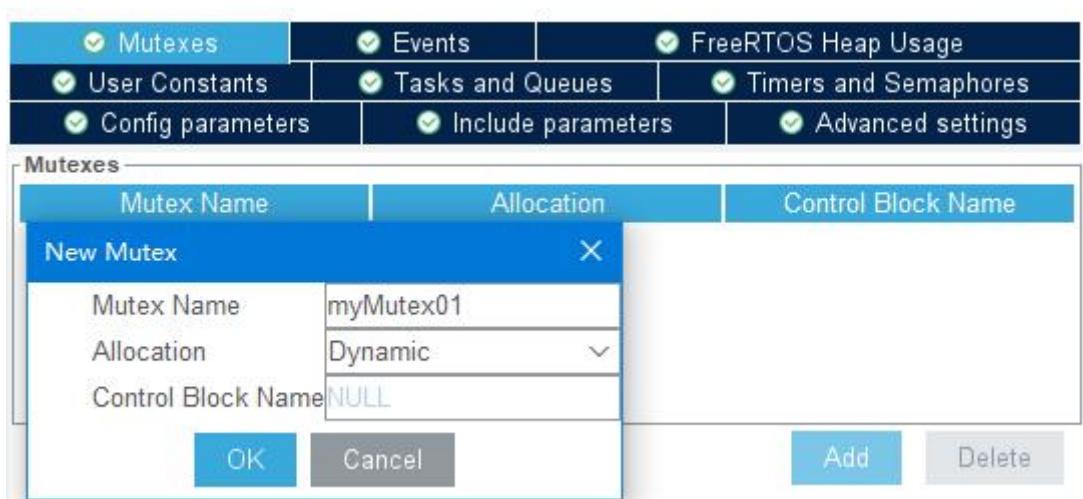
3.4. 多任务变量

1. 参数指针

Cubemx 生成时候选择指针名字(似乎不太方便)

即第三点的第四小点 pvParameters

2. 锁住 (相互排斥 Mutex)



不管是读操作还是写操作，它都各自是一个独立的 task，这样用 freertos 运行多任务就会出现某个任务因为分配的时间到了，对数据的处理被迫中断，然后另一个任务又开始对数据进行操作，而这时的数据很可能只有一半是操作完成，另一半还未完成的状态，这样的数据状态会产生很大的运算错误，非常危险。所以一个参数只要有二个或以上 task 要对其进行操作，就必须上钥匙。

```
SemaphoreHandle_t xHandler; //创建 Handler  
xHandler = xSemaphoreCreateMutex(); //创建一个 MUTEX 返回 NULL, 或者 handler  
xSemaphoreGive(xHandler); // 释放  
xSemaphoreTake(xHanlder, timeout); //指定时间内获取信号量 返回 pdPASS, 或者 pdFAIL
```

一段不完整代码举例：

```
SemaphoreHandle_t xMutexInventory = NULL;  
void retailTask(void *pvParam) {  
    while (1) {  
        if (xSemaphoreTake(xMutexInventory, timeOut) == pdPASS) {  
            //被 MUTEX 保护的内容叫做 Critical Section  
            retailSale  
            //释放钥匙  
            xSemaphoreGive(xMutexInventory);  
        }  
    }  
}  
void onlineTask(void *pvParam) {  
    while (1) {  
        if (xSemaphoreTake(xMutexInventory, timeOut) == pdPASS) {  
            onlineSale(); //等待出售  
            xSemaphoreGive(xMutexInventory);  
        }  
    }  
}  
void setup() {  
    // put your setup code here, to run once:  
    Serial.begin(115200);  
    xMutexInventory = xSemaphoreCreateMutex(); //创建 MUTEX  
    ...//创建任务  
}
```

3. 5. 对 cubemx 创建的 FreeRtos 封装理解

```
osThreadDef(task_main, Task_Main, osPriorityIdle, 0, 128);
```



```
task_mainHandle = osThreadCreate(osThread(task_main), NULL);
```

第一句的宏表示定义一个名为 task_main 的结构体，osThread(task_main) 为获取结构体名字。通过结构体在 osThreadCreate 中使用常规的函数 xTaskCreateStatic() 进行任务初始化。

3.6. 任务管理

3.6.1. 使用 FreeRTOS 源码创建任务

[点击进入代码 FREERTOS - TASK 管理 - Wokwi Arduino and ESP32 Simulator](#)

```
/*任务创建函数*/
```

```
BaseType_t xTaskCreate(  
TaskFunction_t vTaskCode, //函数指针  
const char * const pcName, //任务描述  
unsigned short usStackDepth, //堆栈大小  
void *pvParameters, //参数指针  
UBaseType_t uxPriority, //任务优先级  
TaskHandle_t *pvCreatedTask //回传句柄  
);
```

1. pvTaskCode: 函数指针, 指向任务函数的入口。任务永远不会返回 (位于死循环内)。该参数类型 TaskFunction_t 定义在文件 projdefs.h 中, 定义为: typedef void(*TaskFunction_t)(void *), 即参数为空指针类型并返回空类型。
2. pcName: 任务描述。主要用于调试。字符串的最大长度 (包括字符串结束字符) 由宏 configMAX_TASK_NAME_LEN 指定, 该宏位于 FreeRTOSConfig.h 文件中。
3. usStackDepth: 指定任务堆栈大小, 能够支持的堆栈变量数量 (堆栈深度), 而不是字节数。比如, 在 16 位宽度的堆栈下, usStackDepth 定义为 100, 则实际使用 200 字节堆栈存储空间。堆栈的宽度乘以深度必须不超过 size_t 类型所能表示的最大值。比如, size_t 为 16 位, 则可以表示堆栈的最大值是 65535 字节。这是因为堆栈在申请时是以字节为单位的, 申请的字节数就是堆栈宽度乘以深度, 如果这个乘积超出 size_t 所表示的范围, 就会溢出。
4. pvParameters: 指针, 当任务创建时, 作为一个参数传递给任务。
5. uxPriority: 任务的优先级。具有 MPU 支持的系统, 可以通过置位优先级参数的 portPRIVILEGE_BIT 位, 随意的在特权 (系统) 模式下创建任务。比如, 创建一个优先级为 2 的特权任务, 参数 uxPriority 可以设置为 2 或者 portPRIVILEGE_BIT

T。

6. pvCreatedTask: 用于回传一个句柄 (ID)，创建任务后可以使用这个句柄引用任务。

举个例子，需要创建一个任务

```
TaskHandle_t Task1_Handle;//1.定义一个句柄
void Task_1(void *arg);//2.对任务函数进行声明
xTaskCreate(Task_1, "Task1", 256, NULL, 6, &Task1_Handle);//3.创建任务
void Task_1(void *arg){
for (;;)
{
//3.函数内容
}}
```

例如

```
if (xTaskCreate(radioBilibili, "Bilibili Channel", 1024 * 8, NULL, 1, &biliHandle) == pdPASS){}
```

3.6.2. 任务删除

```
void vTaskDelete( TaskHandle_t xTask );
```

如何任务是用 xTaskCreate()创建的，那么在此任务被删除以后此任务之前申请的堆栈和控制块和控制内存会在空闲任务中被释放掉，因此当调用函数 vTaskDelete()删除任务以后必须给空闲任务一定的运行时间。

3.6.3. 任务暂停

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend );
```

3.6.4. 任务恢复

```
void vTaskResume( TaskHandle_t xTaskToResume );
```

3.6.5. 注意

以上所有任务共同配合时，可以使用 TaskHandle_t biliHandle = NULL; //Task Handler 一个 handle

注意在删除任务前，一定要确保任务是存在的

删除不存在的任务，比如连续删除两次，自动重启

创建时需要判断是否之前已经创建了 Bilibili channel task，如果没有创建，则创建该 Task

3.7. 任务优先级

[FREERTOS - 任务优先级 - Wokwi Arduino and ESP32 Simulator](#)

3.7.1. 设置

```
vTaskPrioritySet(xFirstClassHandle, 2);
```

3.7.2. 创建任务附带优先级

```
xTaskCreatePinnedToCore(ecoClass, "ecoClass", 1024 * 2, NULL, 1, NULL, 1);
```

3.7.3. 资源退让

```
taskYIELD();//资源退让给同等级或者更高级的任务
```

3.7.4. 查询当前任务优先级

```
UBaseType_t uTaskPriority = uxTaskPriorityGet(NULL);
```

3.8. 信号量

3.8.1. 二进制信号量

```
SemaphoreHandle_t xSemaName = NULL; //创建信号量 Handler
```

```
xSemaName = xSemaphoreCreateBinary(); //创建二进制信号量
```

```
if (xSemaphoreTake( xSemaLED, timeOut) == pdTRUE )//获得了信号量过来则进入
```

```
xSemaphoreGive(xSemaLED); //正常给出信号
```

```
xSemaphoreGiveFromISR(xSemaLED, NULL); //外设中断给出信号
```

二进制信号量的好处是,等待的时候会执行其它程序,相比单纯的 bool 数据,它优点很多

3.8.2. 计数信号量

```
SemaphoreHandle_t xSemaName = NULL;
```

```
xSemaName = xSemaphoreCreateCounting(3, 0); //3 表示上限,0 表示初始值
```

```
if (xSemaphoreTake(xSemaPhone, portMAX_DELAY) == pdTRUE ) //portMAX_DELAY
```

表示无限等待

```
xSemaphoreGive(xSemaPhone);
```

3.9. 事件组

能不用就不用

3.9.1. 等待

[FREERTOS - EVENTS WAIT BITS - Wokwi Arduino and ESP32 Simulator](#)

初始化

```
EventGroupHandle_t xEventPurchase = NULL; //创建 event handler(全局变量)
```

```
xEventPurchase = xEventGroupCreate(); //创建 event group
```

等待函数

```
uxBits = xEventGroupWaitBits (xEventPurchase, //Event Group Handler
```

```

    ADDTOCART_0 | PAYMENT_1 | INVENTORY_2, //等待 Event Group 中的那个 Bit(s)
    pdFALSE,      //执行后, 对应的 Bits 是否重置为 0
    pdTRUE,       //等待的 Bits 判断关系 True 为 AND, False 为 OR
    xTimeout);
    uxBits = xEventGroupSetBits(xEventPurchase, PAYMENT_1); // 将 bit1 PAYMENT_1
    设置为 1

```

3.9.2. 同步 SYNC

[FREERTOS - EVENTS SYNC - Wokwi Arduino and ESP32 Simulator](#)

```

    uxBits = xEventGroupSync (xEventPurchase, //Event Group Handler
                              0x01,          // 先将这个 bit(s)设置为 1,然后再等待
                              0x07,          //等待这些 bits 为 1
                              xTimeout);
    if ((uxBits & BOUGHT_PAID_SENT) == BOUGHT_PAID_SENT) {
        //next step;
    }

```

这个是设置某个位后等待

3.10. 任务通知

好处,可以

3.10.1. 创建

```

    TaskHandle_t xflashLED = NULL; //全局变量
    xTaskCreate(flashLED, "Flash LED", 1024 * 4, NULL, 1, &xflashLED);

```

3.10.2. 取代二进制信号量

```

//命令含义, 相当于精简化的 xTaskNotify() + eIncrement
    xTaskNotifyGive(xflashLED);
uint32_t ulNotificationValue; //用于获取数值

//命令含义: waiting for notification, then reset
    ulNotificationValue = ulTaskNotifyTake(pdTRUE, //pdTRUE 运行完后, 清零
                                           portMAX_DELAY);

    if ( ulNotificationValue > 0 ){
        //Do something
    }

```

3.10.3. 取代信号组

任务一中:

```

xTaskNotify(TaskHandle,.0b01,eSetBits);

```

任务二中:

```

xTaskNotiffy(TaskHandle,.0b10,eSetBits);

```

任务三中:

```
//xTaskNotifyWait(0x00000000, //进入前不清除
//
//          0xFFFFFFFF, //进入后全部清除
//
//          &num, //传入到 NUM
//
//          portMAX_DELAY
//);
xTaskNotifyWait(0x0, 0x0,
&num, portMAX_DELAY);
if (num == 0b11){
}
```

3. 10. 4. 模拟队列

只能模拟长度为一的队列

3. 11. 消息队列

动态创建

```
QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength,
UBaseType_t uxItemSize);
//传入参数 (队列长度, 每个数据的大小: 以字节为单位)
//返回 非零: 创建成功返回消息队列的句柄 NULL: 创建失败
```

静态创建

```
QueueHandle_t xQueueCreateStatic(UBaseType_t uxQueueLength,
UBaseType_t uxItemSize,
uint8_t *pucQueueStorageBuffer,
StaticQueue_t *pxQueueBuffer);
```

队列清空

```
BaseType_t xQueueReset( QueueHandle_t pxQueue); //传入队列句柄即可
```

删除队列

```
void vQueueDelete( QueueHandle_t xQueue ); //传入队列句柄即可
```

写队列

```
BaseType_t xQueueSend(QueueHandle_t xQueue,
const void *pvItemToQueue,
TickType_t xTicksToWait);
BaseType_t xQueueSendToBack(QueueHandle_t xQueue,
const void *pvItemToQueue,
TickType_t xTicksToWait);
```

中断写入(保护线程)

```
BaseType_t xQueueSendToBackFromISR(QueueHandle_t xQueue,
const void *pvItemToQueue,
BaseType_t
*pxHigherPriorityTaskWoken);
```

```

BaseType_t xQueueSendToFrontFromISR(QueueHandle_t xQueue,
const void *pvItemToQueue,
BaseType_t
*pxHigherPriorityTaskWoken);

```

读队列

```

BaseType_t xQueueReceive( QueueHandle_t xQueue,
void * const pvBuffer,
TickType_t xTicksToWait );
BaseType_t xQueueReceiveFromISR(QueueHandle_t xQueue, //中断读
void *pvBuffer,
BaseType_t *pxTaskWoken);

```

偷窥--获得当前数据

```

BaseType_t xQueuePeek(QueueHandle_t xQueue,
void * const pvBuffer,
TickType_t xTicksToWait);
BaseType_t xQueuePeekFromISR(QueueHandle_t xQueue, //中断
void *pvBuffer,);

```

查询

```

UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue );
//返回队列中可用数据的个数
UBaseType_t uxQueueSpacesAvailable( const QueueHandle_t xQueue );
//返回队列中可用空间的个数

```

4. 计算机语言

4.1. C 语言

4.1.1. 优先级

优先级	运算符	结合律
1	后缀运算符: [] () · -> ++ --(类型名称){列表}	从左到右
2	一元运算符: ++ -- ! ~ + - * & sizeof_alignof	从右到左
3	类型转换运算符: (类型名称)	从右到左
4	乘除法运算符: * / %	从左到右
5	加减法运算符: + -	从左到右
6	移位运算符: << >>	从左到右
7	关系运算符: < <= > >=	从左到右
8	相等运算符: == !=	从左到右
9	位运算符 AND: &	从左到右
10	位运算符 XOR: ^	从左到右
11	位运算符 OR:	从左到右
12	逻辑运算符 AND: &&	从左到右
13	逻辑运算符 OR:	从左到右
14	条件运算符: ?:	从右到左
15	赋值运算符: = += -= *= /= %= &= ^= = <<= >>=	从右到左
16	逗号运算符: ,	从左到右

4.1.2. qsort

函数用法 `qsort(buf, n, sizeof(int16_t), filter_cmp);`

`filter_cmp` 是比较函数

具体为

```
int openmv_filter_cmp(const void *a, const void *b) {  
    return (*(int16_t *) a - *(int16_t *) b);  
}
```

这个表示：

a > b 的话则替换，即从小到大排序

4.1.3. 数据类型

wchar_t 直译过来是宽字符，在 linux 系统中 4 字节，在 windows 中 2 字节

参考链接：<https://zh.wikipedia.org/wiki/%E5%AF%AC%E5%AD%97%E5%85%83>

4.1.4. 编译器优化

-O1	在不影响编译速度的前提下，尽量采用一些优化算法降低代码大小和可执行代码的运行速度。
-O2	该优化选项会牺牲部分编译速度，除了执行-O1 所执行的所有优化之外，还会采用几乎所有的目标配置支持的优化算法，用以提高目标代码的运行速度。
-O3	该选项除了执行-O2 所有的优化选项之外，一般都是采取很多向量化算法，提高代码的并行执行程度，利用现代 CPU 中的流水线，Cache 等。 这个选项会提高执行代码的大小，当然会降低目标代码的执行时间。
-Os	这个选项是在-O2 的基础之上，尽量的降低目标代码的大小，这对于存储容量很小的设备来说非常重要。
-Ofast	该选项将不会严格遵循语言标准，除了启用所有的-O3 优化选项之外，也会针对某些语言启用部分优化。
-Og	该标识会精心挑选部分与-g 选项不冲突的优化选项，当然就能提供合理的优化水平，同时产生较好的可调试信息和对语言标准的遵循程度。

4.2. JAVA

4.2.1. 安装

```
sudo apt install openjdk-xx-jdk
```

4.2.2. 选择版本

```
sudo update-alternatives --config java
```

4.3. OOP FOR C++

4.3.1. 多态

参考链接: [override final - 多态 - Wokwi Arduino and ESP32 Simulator](#)

精简做法:

父类: virtual 放在 函数 + destructor

子类: override 放在 函数

推荐:

父类: virtual 放在 函数 + destructor

子类: virtual 放在 函数 + destructor, override 放在 函数

5. 数据结构和算法

5.1. 抽象数据类型 ATD

参考链接：

<https://book.itheima.net/course/223/1276707762369208322/1276707936655122434>

抽象数据类型有两个重要特征：数据抽象和数据封装。

能够很大程度上实现代码的**移植性、通用性和拓展性**。

所谓数据抽象是指用 ADT 描述程序处理的实体时，强调的是其本质的特征，无论内部结构如何变化，只要本质特性不变，就不影响其外部使用。例如，在程序设计语言中经常使用的数据类型 `int`，它就可以理解为一个抽象数据类型，在不同的计算机或操作系统中，它的实现方式可能会有所不同，但它本质上的数学特性是保持不变的，`int` 类型的数据指的是整数，可以进行加减乘除模等一些运算，`int` 类型数据的这些数学特性保持不变，那么在编程者来看，它们都是相同的。因此数据抽象的意义在于数据类型的数学抽象特性。

而另一方面，所谓的数据封装是指用户在软件设计时从问题的数学模型中抽象出来的逻辑数据结构和逻辑数据结构上的运算，需要通过固有数据类型（高级编程语言中已实现的数据类型）来实现，它在定义时必须给出名字及其能够进行的运算操作。一旦定义了一个抽象数据类型，程序设计中就可以像使用基本数据类型那样来使用它。例如，在统计学生信息时，经常会使用姓名、学号、成绩等信息，我们可以定义一个抽象数据类型“`student`”，它封装了姓名、学号、成绩三个不同类型的变量，这样我们操作 `student` 的变量就能很方便地知道这些信息了。C 语言中的结构体以及 C++ 语言中的类等都是这种形式。

5.2. PID 算法

5.2.1. P（比例）

比例控制

5.2.2. I（积分）

误差累计

5.2.3. D（微分）

PD 控制： $U(t) = K_p * err(t) + K_d * derr(t)/dt$

5.2.4. 增量式和位置式

参考链接：(70 条消息) 位置式 PID 与增量式 PID 区别浅析_Z 小旋的博客-CSDN 博客_增量式 pid 和位置式 pid 的区别

增量式

$$u(k) = K_P e(k) + K_I \sum_{i=0} e(i) + K_D [e(k) - e(k-1)]$$

位置式

$$\Delta u(k) = u(k) - u(k-1)$$

$$= K_P [e(k) - e(k-1)] + K_I e(k) + K_D [e(k) - 2e(k-1) + e(k-2)]$$

位置式 PID 优缺点:

优点:

①位置式 PID 是一种非递推式算法,可直接控制执行机构(如平衡小车), $u(k)$ 的值和执行机构的实际位置(如小车当前角度)是一一对应的,因此在执行机构不带积分部件的对象中可以很好应用

缺点:

①每次输出均与过去的状态有关,计算时要对 $e(k)$ 进行累加,运算工作量大。

增量式 PID 优缺点:

优点:

①误动作时影响小,必要时可用逻辑判断的方法去掉出错数据。

②手动/自动切换时冲击小,便于实现无扰动切换。当计算机故障时,仍能保持原值。

③算式中不需要累加。控制增量 $\Delta u(k)$ 的确定仅与最近 3 次的采样值有关。

缺点:

①积分截断效应大,有稳态误差;

②溢出的影响大。有的被控对象用增量式则不太好;

5.2.5. 调参口诀

实际应用,进行 PID 参数调节时,一般使用试凑法, PID 参数整定口诀如下:

参数整定找最佳,从小到大顺序查,
先是比例后积分,最后再把微分加,
曲线振荡很频繁,比例度盘要放大,
曲线漂浮绕大湾,比例度盘往小扳,
曲线偏离回复慢,积分时间往下降,
曲线波动周期长,积分时间再加长,
曲线振荡频率快,先把微分降下来,

动差大来波动慢，微分时间应加长，
理想曲线两个波，前高后低 4 比 1，
一看二调多分析，调节质量不会低。

5.3. 环形缓冲区

5.3.1. 裸机中利用 DMA 使用环形缓冲区实现异步发送

定义数据

```
static uint8_t uart_buffer[UART_BUFFER_SIZE];
static uint16_t left, right;
static uint16_t pack_size; //每次出队个数
```

```
/**
 * 发送中断回调,用于维护环形缓冲区
 * @param huart
 * @param size
 */
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart == uart_handle) {
        left = (left + pack_size) % UART_BUFFER_SIZE;
        transmit_mutex = false;
        if (left != right) { //检查环形队列中是否还有数据
            uart_transmit_start();
        }
    }
}
/**
 * 发送当前需要发送的数据
 * @return
 */
static int uart_transmit_start()
{
    HAL_StatusTypeDef ret;
    if (transmit_mutex == true) {
        return -1;
    }
    transmit_mutex = true;
    // 计算发送的个数
    if (right > left) {
```

```

        pack_size = right - left;
    }
    else {
        pack_size = UART_BUFFER_SIZE - left;
    }
    ret = HAL_UART_Transmit_DMA(uart_handle, (uint8_t *) uart_buffer + left, pack_size);
e);
    if (ret != HAL_OK) {
        return -1;
    }
    return 0;
}
/**
 * 添加数据到环形缓冲区, 这个函数是 api 的接口
 * @param com
 * @param buf
 * @param size
 * @return
 */
static int uart_transmit(com_buffer_cb_t *com, uint8_t *buf, uint16_t size)
{
    UNUSED(com);
    int err;
    int next_index = (right + size) % UART_BUFFER_SIZE;

    // *保存数据
    if (next_index > right) { // 没有超限循环
        if (next_index >= left && right < left) {
            return -1;
        }
        memcpy(uart_buffer + right, buf, size);
    }
    else { // 超限循环
        if (next_index > left && right > left) {
            return -1;
        }
        memcpy(uart_buffer + right, buf, UART_BUFFER_SIZE - right);
        memcpy(uart_buffer, buf + UART_BUFFER_SIZE - right, size - (UART_BUFFER_SIZE - right));
    }
    right = next_index;

    // *开启尝试发送
    err = uart_transmit_start();
    return err;
}

```

}

6. 其它(单片机)

6.1. 高级定时器注意事项

是 APB1 时钟的 2 倍，当 APB1 的时钟不分频的时候，通用定时器 TIMx 的时钟就等于 APB1 的时钟。这里还要注意的就是高级定时器的时钟不是来自 APB1，而是来自 APB2 的。

这里顺带介绍一下 TIMx CNT 寄存器，该寄存器是定时器的计数器，该寄存器存储了当前

6.2. ROM 烧录

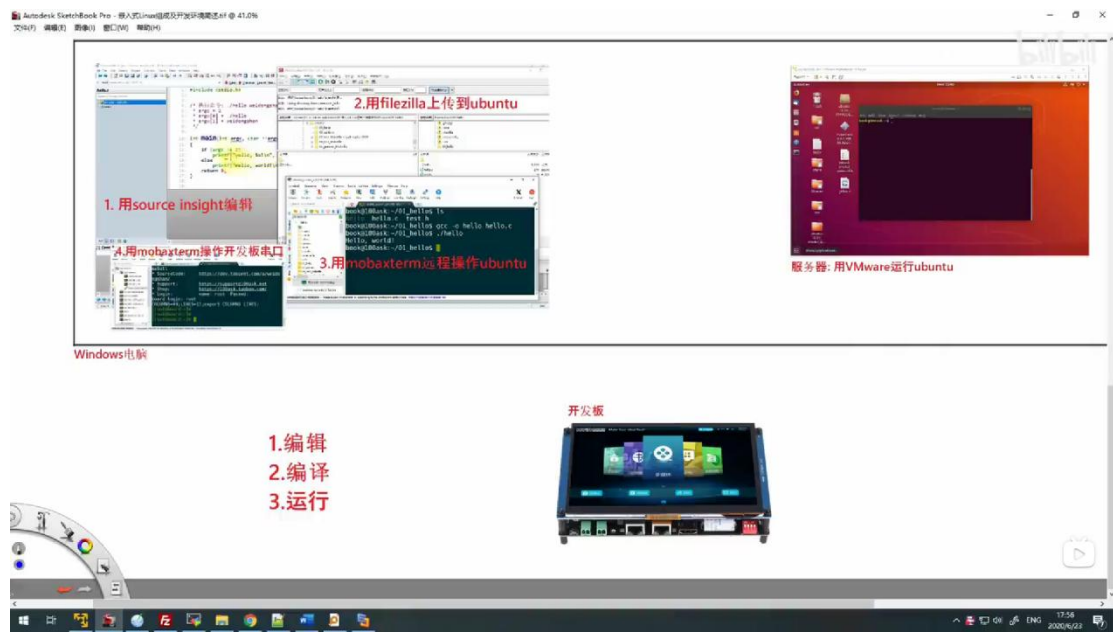
早期一般是将调试好的单片机程序写入到 ROM、EPROM 中，这种操作就像刻制光盘一样，实在高电压方式下写入，PROM 是一次性写入，存储内部发生变化，有些线路或元件就被烧断，不可再恢复，所以叫做烧写，EPROM 可以使用紫外线将原来写入的内容擦除，重新烧写，目前大量采用 EEPROM，是可以电擦写的存储器。

7. 嵌入式 Linux

因为本人学习道路比较分散，没有系统学习，所以可能很多细节的东西没办法参考，需要自行去了解。

7.1. 开发环境

参考链接: <https://zhuanlan.zhihu.com/p/100367053>



7.2. 基础命令

指的是自己用的多的命令，会随学习程度增加

cd	切换目录
ls	输出目录
ifconfig	输出当前网络状态
vim	vim 编辑器
touch	新建文件
pwd	输出当前路径的绝对路径
cp	复制
rm	删除
mv	移动
unzip	解压 zip 文件
chmod	修改权限
cat	打印文件内容
udhcpc	自动获取网络 ip

gedit	ubuntu 的文本编辑
geap	搜索

7.3. VIM 快捷键

gg 光标移动到最下

GG 光标移动到最上

v 进入 Visual 可视化

7.4. 远程连接 tftp

安装 tftp,可以联网后可以通过下载安装

```
git clone https://e.coding.net/weidongshan/DevelopmentEnvConf.git
cd DevelopmentEnvConf
sudo ./Configuring_ubuntu.sh
```

或者自行百度方法

7.5. NFS 挂载

使本地的文件夹指向网络端的数据,使本地的路径可以使用网络的资源

7.5.1. 安装

a. 下载安装

```
sudo apt-get install nfs-kernel-server nfs-common
```

b. 修改配置文件

```
sudo vim /etc/exports
```

例如: /home/hyc/nfs_share *(rw,sync,no_root_squash)

c. 增加权限 777

d. 重启

```
sudo /etc/init.d/nfs-kernel-server restart
```

7.5.2. 挂载

指网络的资源映射到本地路径

将 nfs_share 挂载到 Code/Test

即 Code/Test(后者)路径指向 nfs_share(前者)

```
sudo mount -o nolock,tcp 127.0.0.1:/home/hyc/nfs_share /mnt
```

7.5.3. 重启

```
sudo /etc/init.d/nfs-kernel-server restartins 取消
```

```
umount -a
```

7.6. 网络

7.6.1. 改变 IP

1.查看

(1). ip addr 的缩写是 ip a , 可以查看网卡的 ip、mac 等,
即使网卡处于 down 状态, 也能显示出网卡状态, 但是 ifconfig 查看就看不到。

(2).ip addr show device 查看指定网卡的信息

比如查看网卡接口的信息, 就是 ip addr show eth0

2.增加 ip

ip addr add ip/netmask dev 接口

比如给 eth0 增加一个 172.25.21.1/24 地址

ip addr add 172.25.21.1/24 dev eth0

3.删除 ip

ip addr del ip/netmask dev 接口

4.清空指定网卡的所有 ip

ip addr flush dev 接口

5. 给网卡起别名, 起别名相当于给网卡多绑定了一个 ip

用法: 比如给网卡 eth0 增加别名

ip addr add 172.25.21.1/32 dev eth0 label eth0:1

6.删除别名

ip addr del ip/netmask dev eth0

原文链接: https://blog.csdn.net/qq_43309149/article/details/104481743

7.6.2. 获取 IP 赋值给变量

IP=\$(ip a|grep -w 'eth3'|grep 'global'|sed 's/^.*inet//g'|sed 's/[0-9][0-9].*\$/g')

7.7. 图形化

ubuntu 的

开机时关闭 sudo systemctl set-default multi-user.target

开机时打开 sudo systemctl set-default graphical.target

7.8. Ubuntu 下载源

改变目录下/etc/apt/sources.list 文件

参考链接:

清华镜像源: [ubuntu | 镜像站使用帮助 | 清华大学开源软件镜像站 | Tsinghua Open Source Mirror](#)

中科大源: <https://mirrors.ustc.edu.cn/repoen/>

默认的版本不够新的话可以使用下面命令获取,

wget https://apt.kitware.com/kitware-archive.sh

sudo bash kitware-archive.sh

如果使用早于 22.04 的 Ubuntu 版本，则需要添加额外的存储库以满足上面列出的主要依赖项的最低版本要求。在这种情况下，请下载、检查并执行 Kitware 存档脚本，以将 Kitware APT 存储库添加到您的源列表中。kitware-archive.sh 的详细解释可以在[这里](#)找到 kitware 第三方 apt 存储库：

7.9. SSH 服务和公钥登录

7.9.1. 安装

```
sudo apt-get install openssh-server
```

7.9.2. 登录

```
ssh 到指定端口 ssh xx user@ip
```

如果是转发的流量 加上 -p

7.9.3. 创建公钥

- 配置

```
sudo vim /etc/ssh/sshd_config
```

PermitRootLogin yes 改为 no

- 重启

```
systemctl restart sshd.service
```

- 生成公钥

```
ssh-keygen
```

- 删除旧的连接认证,重新生成新的

```
ssh-keygen -R +输入服务器的 IP
```

一个 key 对应一个，需要连接的对应上就行

7.9.4. 使用公钥

原理：如果使用私钥 id_rsa 登录服务器时候，服务器会检查所登录的用户的”~/.ssh/authorized_keys”里搜索是否存在本地的 id_rsa.pub 文本，存在的话则允许登录。

```
mkdir .ssh
```

```
touch authorized_keys
```

```
echo id_rsa.pub >> authorized_keys
```

7.10. 后台运行

开启运行

```
nohup [command]
```

查看后台

```
ps -def | grep "runoob.sh"
```

比如，chatgptwechat 的

```
nohup python3 app.py & tail -f nohup.out
```

ps -ef | grep app.py | grep -v grep 命令可查看运行于后台的进程

7. 11. 自启动 Service

举例

clash.server 文件放在 /etc/systemd/system/下

[Unit]

Description=clash daemon

[Service]

Type=simple

User=hyc

ExecStart=/usr/local/bin/clash

Restart=on-failure

[Install]

WantedBy=multi-user.target

基本命令

1. 查看当前已经启动的服务 `systemctl list-units`
2. 查看所有服务 `systemctl list-unit-files`
3. 运行命令 `systemctl reload` 重新加载一下配置
4. 运行命令 `systemctl status clash.service` 查看状态
5. 运行命令 `systemctl start clash.service` 开启服务
6. 运行命令 `systemctl enable clash.service` 设置开机启动

重载服务

`sudo systemctl daemon-reload`

开机启动

`sudo systemctl enable clash`

启动服务

`sudo systemctl start clash`

查看服务状态

`sudo systemctl status clash`

7.12. 用户

以下命令需要管理员权限

7.12.1. 创建用户

Linux 服务器上创建账户用到 `useradd` 命名，一般常用以下命令：

创建： `useradd -m -s /bin/bash userName`

设置密码： `passwd userName`

7.12.2. 删除用户

`userdel -r userName`

7.12.3. 给予 sudo 权限

- (1) 切换到 root 用户
- (2) 打开 `/etc/sudoers` (需要 root 用户)
- (3) 再添加 `# Allow members of group sudo to execute any command` 下面一行添加 `hyc ALL=(ALL:ALL) ALL`
- (4) `:/wq!` 保存退出即可使用

7.12.4. 切换用户

`su userName`

7.12.5. 修改用户名和密码

参考链接：

<https://blog.nowcoder.net/n/525cc83df73448a0909cb2a0c286df72>

修改密码

- 1、进入 Ubuntu，打开一个终端，输入 `sudo su` 转为 root 用户。 注意，必须先转为 root 用户!!!
- 2、`sudo passwd user`(user 是对应的用户名)
- 3、输入新密码，确认密码。
- 4、修改密码成功，重启，输入新密码进入 Ubuntu。

修改用户名

- 1、进入 Ubuntu，打开一个终端，输入 `sudo su` 转为 root 用户。 注意，必须先转为 root 用户!!!
- 2、`gedit /etc/passwd` ,找到代表你的那一行，修改用户名为新的用户名。 注意：只修改用户名！后面的全名、目录等不要动！
- 3、`gedit /etc/shadow` , 找到代表你的那一行，修改用户名为新用户名
- 4、`gedit /etc/group` , 你应该发现你的用户名在很多个组中，全部修改！
- 5、修改完，保存，重启。

7.13. crontab 定时任务

#列出用户 user 的 crontab

```
crontab -l [-u user]
```

#修改用户 user 的 crontab

```
crontab -e [-u user]
```

参数还有-r(删除),-i(有提示的删除)

7.14. screen

7.14.1. 命令

screen -ls 列出桌面

screen -S 23536 -X quit 删除某终端 23536 是 id, 推荐进入终端关闭

screen -r \$name 进入某终端

7.14.2. 快捷键

所有快捷键都需要按下 ctrl + a 触发快捷键模式

k 关闭当前窗口

shift + s 上下分屏

tab 切换屏幕

c 新建终端

x 退出

7.15. 路由器

(1 条消息) 小米路由 3G 刷 openwrt 固件_奥利奥泡泡的博客-CSDN 博客_小米路由器 3

gopenwrt 固件

7.15.1. 刷机前的准备工作

7.15.1.1. 刷入开发者 ROM

1.在 http://www.miwifi.com/miwifi_download.html 上下载对应的 ROM for R3G 开发版

2.在小米路由器上选择系统升级, 然后选择 ROM 进行升级, 等重启完成后即可。

7.15.1.2. 开启路由器 ssh 登录

1.先让路由器绑定账号, 打开小米 WIFI app, 然后需要手机和路由器一个网络, 登录后即可绑定

2.http://www.miwifi.com/miwifi_open.html 在网站中找到开启 SSH 工具, 会显示 root 密码, 注意这个文件每个路由器都不一样。

3.请将下载的工具包 bin 文件复制到 U 盘 (FAT/FAT32 格式) 的根目录下, 保证文件名为 miwifi_ssh.bin。

4.断开小米路由器的电源, 将 U 盘插入 USB 接口。

5. 按住 reset 按钮之后重新接入电源，指示灯变为黄色闪烁状态即可松开 reset 键。

6. 等蓝灯亮起即可刷机完成。

7. `ssh root@192.168.1.1` 即可登录到路由器上

7. 15. 1. 3. 刷入 breed

Ps: 这一步是防止路由被刷坏

1. 在 <https://breed.hackpascal.net/> 下载 `breed-mt7621-xiaomi-r3g.bin`

2. 然后将文件重命名为 `breed.bin` 后，上传到小米路由器，可以使用 U 盘或者使用 SCP 上传，上传到 `/tmp` 下

3. 进入到小米路由开始写入 `breed mtd -r write /tmp/breed.bin Bootloader`

4. 刷入后，机器会重新启动，按住 reset 键开机，等到路由指示灯闪烁时，松开 reset 键，然后浏览器中输入 `192.168.1.1` 即可进入 breed 后台。

5. 备份后重启路由回到小米固件的，然后 `ssh` 登陆准备刷入 **openwrt**

7. 15. 2. 刷入 openwrt

1. 打开 <https://downloads.lede-project.org/snapshots/targets/ramips/mt7621/> 下载对应的文件 `mir3g-squashfs-kernel1.bin`, `mir3g-squashfs-rootfs0.bin`, `mir3g-squashfs-sysupgrade.tar`，并上传到路由器。

2. 由于我刷入了 breed，需要执行

```
mtd write openwrt-ramips-mt7621-mir3g-squashfs-kernel1.bin kernel0
mtd write openwrt-ramips-mt7621-mir3g-squashfs-kernel1.bin kernel1
mtd write openwrt-ramips-mt7621-mir3g-squashfs-rootfs0.bin rootfs0
reboot
```

2. 如果没有刷入 breed，则需要执行

```
mtd write openwrt-ramips-mt7621-mir3g-squashfs-kernel1.bin kernel1
mtd write openwrt-ramips-mt7621-mir3g-squashfs-rootfs0.bin rootfs0
nvram set flag_try_sys1_failed=1
nvram commit
reboot
```

ps: 刷入固件后请勿关闭路由器，要接着执行如下步骤，否则 `ssh` 会失效，如果 `ssh` 失效，请试着重置路由器。

3. 等路由启动后，即可进行 `ssh` 登录，`ssh root@192.168.1.1` 此时 `root` 是没有密码的，需要执行 `passwd`，设置完成密码后，然后安装 `luci`。

```
opkg update
opkg install luci
opkg install luci-il8n-base-zh-cn
```

4. 浏览器登陆 `192.168.1.1`，在界面选择系统，备份/升级，刷写新的固件，上传 `openwrt-ramips-mt7621-mir3g-squashfs-sysupgrade.tar`，升级版本，等待启动完成后执行 3 步骤，重新安装

luci。即可完成

5.如果重置路由器，等重启完成后 web 界面是被重置没有了，执行 `ssh root@192.168.1.1`，设置密码重新安装 luci。

如果报 Collected errors:

```
check_data_file_clashes: Package libubox20170601 wants to install file /lib/libubox.so
But that file is already provided by package * libubox
opkg_install_cmd: Cannot install package opkg.
opkg install luci --force-overwrite
opkg install luci-i18n-base-zh-cn
```

7.16. 交叉编译

设置环境变量

```
PATH=$PATH:/usr/lib/gcc/gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabi/hf/bi
"~/bashrc" 119L, 3852B
```

保存后,重新加载环境变量

```
source ~/.bashrc
```

7.17. 驱动

7.17.1. 安装卸载

`lsmod` 查看加载的驱动列表

`rmmod modname` 卸载已加载的驱动

`modprobe -r modname` 如果用以上命令无法卸载，先执行此命令

7.18. GPIO 子系统

7.18.1. 查看 gpio 使用状态

```
cat /sys/kernel/debug/gpio
```

7.18.2. 确定 GPIO 引脚的编号

① 先在开发板的 `/sys/class/gpio` 目录下，找到各个 `gpiochipXXX` 目录：

```
[root@imx6ull:/sys/class/gpio]# ls
export      gpio30      gpiochip128  gpiochip504  gpiochip96
gpio133     gpiochip0   gpiochip32   gpiochip64   unexport
[root@imx6ull:/sys/class/gpio]#
```

② 然后进入某个 `gpiochipXXX` 目录，查看文件 `label` 的内容，就可以知道起始号码 XXX 对于哪组 GPIO

7.18.3. shell 控制

以引脚编号为 110 为例

```
echo 110 > /sys/class/gpio/export // gpio_request
```

```
echo in > /sys/class/gpio/gpio110/direction // gpio_direction_input
```

```
cat /sys/class/gpio/gpio110/value // gpio_get_value
echo 110 > /sys/class/gpio/unexport // gpio_free
```

对于输出,以 N 为例

```
echo 104> /sys/class/gpio/export
echo out > /sys/class/gpio/gpio104/direction
echo 1 > /sys/class/gpio/gpio104/value
echo 104> /sys/class/gpio/unexport
```

7.18.4. GPIO 子系统函数

descriptor-based	**legacy**
-----	-----
获得 GPIO	
gpiod_get	gpio_request
gpiod_get_index	
gpiod_get_array	gpio_request_array
devm_gpiod_get	
devm_gpiod_get_index	
devm_gpiod_get_array	
设置方向	
gpiod_direction_input	gpio_direction_input
gpiod_direction_output	gpio_direction_output
读值、写值	
gpiod_get_value	gpio_get_value
gpiod_set_value	gpio_set_value
释放 GPIO	
gpio_free	gpio_free
gpiod_put	gpio_free_array
gpiod_put_array	
devm_gpiod_put	
devm_gpiod_put_array	

7.19. 中断

7.19.1. 流程

在驱动程序里使用中断的流程如下:

1. 确定中断号

注册中断处理函数, 函数原型如下:

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev);
```

2. 在中断处理函数里

- 分辨中断
- 处理中断

- 清除中断

7.19.2. 获取中断号

gpio 子系统中:

```
int gpio_to_irq(unsigned int gpio);  
int gpiod_to_irq(const struct gpio_desc *desc);
```

7.19.3. 获取中断名称

cat /proc/interrupts

7.19.4. 触发方式类型

```
#define IRQF_TRIGGER_NONE 0x00000000  
#define IRQF_TRIGGER_RISING 0x00000001  
#define IRQF_TRIGGER_FALLING 0x00000002  
#define IRQF_TRIGGER_HIGH 0x00000004  
#define IRQF_TRIGGER_LOW 0x00000008  
  
#define IRQF_SHARED 0x00000080
```

7.20. zsh (oh my zsh)

1. 安装 zsh: sudo apt install zsh
2. 安装 oh my zsh: sh -c "\$(curl -fsSL <https://gitee.com/mirrors/oh-my-zsh/raw/master/tools/install.sh>)"
3. 配置

8. DOCKER 容器

8.1. 添加容器

举例：

```
docker run --restart always -d --name=OpenWRT --network macnet --privileged unifreq/openwrt-aarch64 /sbin/init
```

名字：OpenWRT 使用网络模式：macnet 镜像名字：unifreq/openwrt-aarch64

8.2. 进入容器

```
doker exec -it [容器名或者 id] bash
```

8.3. 退出容器

快捷键 `ctrl+p+q`

或

`exit`

8.4. 查看容器

```
doker ps
```

8.5. 镜像

拉取镜像

```
docker pull xxx:tag
```

镜像保存为 tar

```
docker save -o xxx.tar xxx:tag
```

导入本地镜像

```
docker load --input xxx.tar
```

OR

```
cat xxx.tar.gz | docker import - openwrt/lede
```

// 查看镜像

```
docker image ls
```

9. QT

9.1. Linux 安装

sudo apt install qtcreator

sudo apt install qtbase5-dev qtchooser qt5-qmake qtbase5-dev-tools

sudo apt-get install libqt5serialport5-dev libudev-dev

9.2. 槽函数

参考资料:

[QT 信号槽机制 - melonstreet - 博客园](#)

9.2.1. 定义

public slots:

槽函数(); //接收方

signals:

信号函数(); //发送方

9.2.2. 举例:

```
connect(&gameData,&GameData::dataUpdateComplete,this,&MainWindow::updateBoard);
```

gameData: 发送槽的类

GameData::dataUpdateComplete: 发送的信号量, 是一个空函数

this: 接受槽的类

&MainWindow::updateBoard: 接受的槽函数

9.2.3. 个人理解:

槽类似回调函数, 即前者的类型调用**信号函数**后会**回调**到接受类的**槽函数**。

9.3. 字符串

参考资料:

[qt 中 double 转 QString 保留 n 位小数 qt double 保留小数位 GreenHandBruce 的博客-CSDN 博客](#)

[Qt 的 QString 数据类型转换\(整理\) qstring format 焕小谢的博客-CSDN 博客](#)

9.3.1. 数据类型转换为字符串

//整数类型转字符串, 浮点类型全转换

```
QString str = QString::number(data);
```

//浮点类型保留小数位数转换

```
QString str = QString::number(fData,'f',3);
```

//类似 printf 的方法

```
QString str = QString("hello %1 %2").arg(1).arg(2.2); //str = "hello 1 2.2"
```

```
QString str = QString("hello %1").arg(QString::number(fData,'f',3));
```

//多个数据

```
str = QString("cLapTime %1:%2:%3").arg(gameData.cLapTime[0]).arg(gameData.cLapTime[1]).arg(gameData.cLapTime[2]);
```

9.3.2. 字符串转数据类型

```
bool ret;
```

```
QString str = "15.54";
```

```
double val = str.toDouble(); //举例一 val = 15.54
```

```
float val = str.toDouble(&ok); //举例二 val = 15.54f ok = true
```

```
QString str = "FF";
```


```
float val = str.toInt(&ok,16); //举例三 val = 255 ok = true 其中 16 为原类型进制
```

前面补全零

```
.arg(hid->key, 8, 16, QLatin1Char('0'))
```

9.4. 打包可执行文件 (WIN)

打开 qt 对应编译器的命令行

比如打开  Qt 6.5.1 (MinGW 11.2.0 64-bit)

输入

```
windeployqt dir/name.exe
```

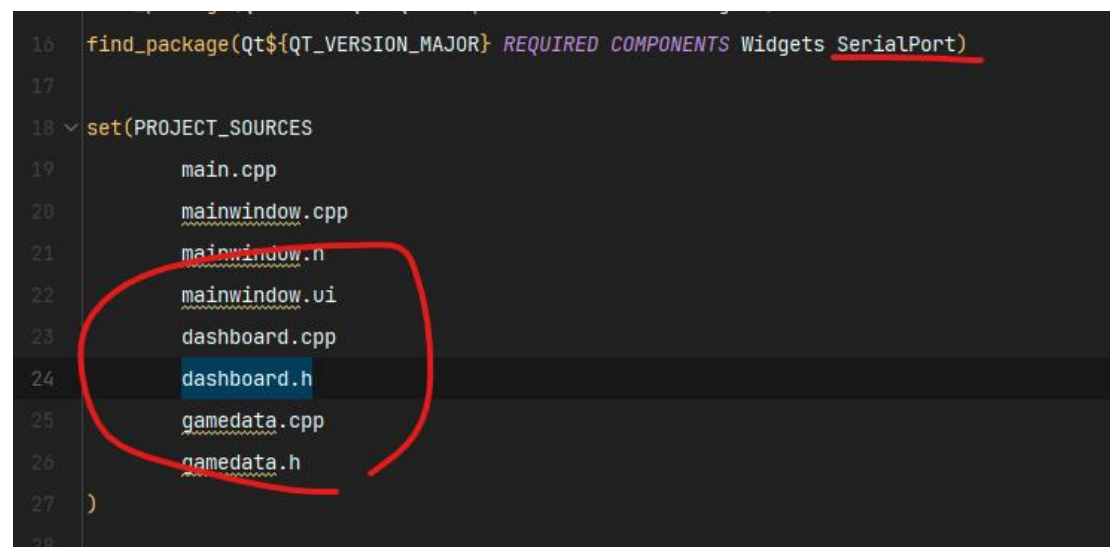
9.5. 模块

9.5.1. 安装

```
sudo apt-get install libqt5serialport5-dev libudev-dev
```

#串口模块

9.5.2. 添加附加模组 (串口为例)



```
16 find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets SerialPort)
17
18 set(PROJECT_SOURCES
19     main.cpp
20     mainwindow.cpp
21     mainwindow.n
22     mainwindow.ui
23     dashboard.cpp
24     dashboard.h
25     gamedata.cpp
26     gamedata.h
27 )
28
```

在 find_package 中添加需要拓展的模块

添加内容与 qmake 中的 QT + name 一致

```
target_link_libraries(  
    dash_board_cmake PRIVATE  
    Qt${QT_VERSION_MAJOR}::Widgets  
    Qt${QT_VERSION_MAJOR}::Core  
    Qt${QT_VERSION_MAJOR}::Gui  
    Qt${QT_VERSION_MAJOR}::Widgets  
    Qt${QT_VERSION_MAJOR}::SerialPort  
)
```

链接文件

9.6. Painter 画家

一般初始流程是：确定界面大小、确定零点、**设置画笔（颜色和类型）、求出图形、绘画。**
其中界面大小和零点不是每个子图形都需要绘制的。

9.6.1. 注意事项

使用完需要释放内存，不能够在已存在的情况下再次新建一个类。即想象为单例模式，否则画家会出问题。

9.6.2. 刷新

update()

QWidget 成员函数，每次调用的时候，会执行可继承函数 `void paintEvent(QPaintEvent *event);`

包括每次新建类的时候，会调用此函数刷新一次画面。

9.6.3. 美术类型

画刷和画笔

brush 和 pen。区别在于一个是线一个是面。使用其中一个笔会覆盖另一个笔。

颜色

颜色有纯色 `Color(r,g,b,a)`；最后一个参数 a 是透明度，四个参数取值范围都是 0-255。

除了纯色外还有渐变色，都是通过 `setColorAt (pos, color)` 设置渐变色范围的。第一个参数是 0-1 的数字，0 表示设置类的 `startAngel`，1 表示结束的位置（最）。在构建的时候有一个参数类型就是 `startAngel`。

`QConicalGradient(qreal cx, qreal cy, qreal startAngle);` //锥形渐变色

`QRadialGradient(qreal cx, qreal cy, qreal radius, qreal fx, qreal fy);` //横向渐变色

举例：

```
// 渐变色
QPainterPath gradientRing;
gradientRing.addEllipse( x: -_radius + 4, y: -_radius + 4, w: 2 * (_radius - 4), h: 2 * (_radius - 4));
QRadialGradient radialGradient( cx: 0, cy: 0, radius: _radius + 4, fx: 0, fy: 0);
radialGradient.setColorAt( pos: 1, color: QColor( r: 6, g: 148, b: 208));
radialGradient.setColorAt( pos: 0.75, color: Qt::transparent);
painter->setBrush(radialGradient);
painter->drawPath( path: outRing);

painter->restore();
```

图形形状

QPainterPath 类中的成员函数,

addEllipse(qreal x, qreal y, qreal w, qreal h); //椭圆

arcTo(qreal x, qreal y, qreal w, qreal h, qreal startAngle, qreal arcLength); //弧形扇面

9.6.4. 设置零点

原本的零点是 0,0 可以通过下面函数移动,移动后能够较为方便的通过半径绘画

painter->translate(width / 2.0, height / 2.0); //移动

9.7. QWidget 控件类

基本全部控件都是继承自此类, 有很多通用的特性

9.7.1. 设置位置和大小

坐标： setGeometry(ax,ay,aw,ah)

参数分别左上角位置 xy 轴,宽度和高度。

控件位置： setParent, QWidget(this)

两个方法, 一个是设置变量, 一个是构建的时候设置。

传入类型为指针。

9.7.2. 显示控件

show();显示命令

9.7.3. 自定义美工

见 Painter 画家项

9.8. 键盘鼠标

参考链接: <https://wizzardforcel.gitbooks.io/qt-beginning/content/12.html>

在 QWidget 类中, 已经存在了两个回调虚函数, 就是 mousePressEvent 和 keyPressEvent

只需要在控件中实现两个函数就能够使用了, 以下是举例

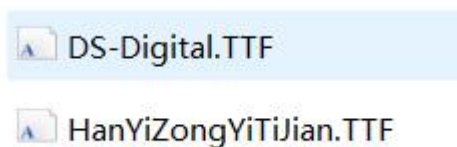
```
protected:
    void paintEvent(QPaintEvent *event) override;
    void keyPressEvent(QKeyEvent *e) override;
    void mousePressEvent(QMouseEvent *e) override;

void MainBoard::mousePressEvent(QMouseEvent *e)
{
    QWidget::mousePressEvent(event: e);
    emit pressExit();
}
low complexity (6%)
void MainBoard::keyPressEvent(QKeyEvent *e)
{
    QWidget::keyPressEvent(event: e);
    switch (e->key()) {
        case Qt::Key_Escape:
        case Qt::Key_Space:
            emit pressExit();
            break;
        default:
            break;
    }
}
```

图 11-1 键盘使用

9.9. 使用自定义字体

现有两款字体



9.9.1. 添加进项目

新建 qrc 后缀文件，比如新建 res.qrc

内容如下

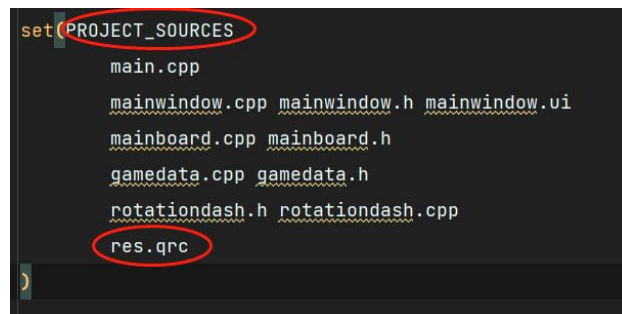
```
<RCC>
    <qresource prefix="/">
        <file>res/font/HanYiZongYiTiJian.TTF</file>
        <file>res/font/DS-Digital.TTF</file>
    </qresource>
</RCC>
```

解释:

/ 是前缀

res/font/HanYiZongYiTiJian.TTF 是路径

将项目添加进工程



```
set(PROJECT_SOURCES
    main.cpp
    mainwindow.cpp mainwindow.h mainwindow.ui
    mainboard.cpp mainboard.h
    gamedata.cpp gamedata.h
    rotationdash.h rotationdash.cpp
    res.qrc
)
```

图 11-2 添加进 Cmake

9.9.2. 加载字体

```
QFontDatabase::addApplicationFont(":/res/font/DS-Digital.TTF");
```

```
QFontDatabase::addApplicationFont(":/res/font/HanYiZongYiTiJian.TTF");
```

9.9.3. 使用



```
currentLapTimeShow = new QLabel( text: "Current Lap", parent: this);
currentLapTimeShow->setFont(QFont( family: "HanYiZongYiTiJian", pointSize: 18));
currentLapTimeShow->setStyleSheet("color:#FFFFFF;");
currentLapTimeShow->setGeometry( ax: 88, ay: 28, aw: 160, ah: 25);
currentLapTimeShow->show();|
```

注意：设置时候的字体名不是文件名，需要打开 ttf 文件查看

9.10. 设置屏幕大小

```
showFullScreen(); //全屏
```

```
setFixedSize(x,y); //设置固定大小
```

```
setGeometry(ax, ay, aw, ah); //设置初始位置和大小
```

```
setParent(); //设置附加在哪
```

9.11. 定时器

参考链接：<https://xie.infoq.cn/article/4a3c8068f8fde1105a2b72b51>

使用 QTimer 类

1.用 new 的方式创建一个 QTimer 对象。

```
QTimer *timer = new QTimer(this);
```

2.将定时器的溢出信号连接到自定义的槽函数。

```
connect(timer, &QTimer::timeout, this, &Myself::update);
```

3.启动定时器。

```
timer->start(1000);
```

函数原型为：void start(int msec);参数为定时器时间间隔，单位毫秒。

也可以调用 timer->setInterval(1000);设置定时器时间间隔，然后调用 timer->start();开启定时器。

4.停止定时器。

timer->stop();

使用 QObject 类

在需要开启定时器的地方直接调用 startTimer();

该函数的声明为: int startTimer(int interval, Qt::TimerType timerType = Qt::CoarseTimer);

该函数开启一个定时器, 返回值是定时器的编号。

参数一为时间间隔, 单位毫秒;

参数二为定时器的精确度:

Qt::PreciseTimer (精确的定时器, 尽量保持毫秒精度, 试图保持精确度在 1 毫秒);

Qt::CoarseTimer (粗略的定时器, 尽量保持精度在所需的时间间隔 5%范围内);

Qt::VeryCoarseTimer (很粗略的定时器, 只保留完整的第二精度, 大约为 500 毫秒);

2.重载 void QObject::timerEvent (QTimerEvent * event);

当定时器溢出时, 会自动响应 timerEvent()函数。

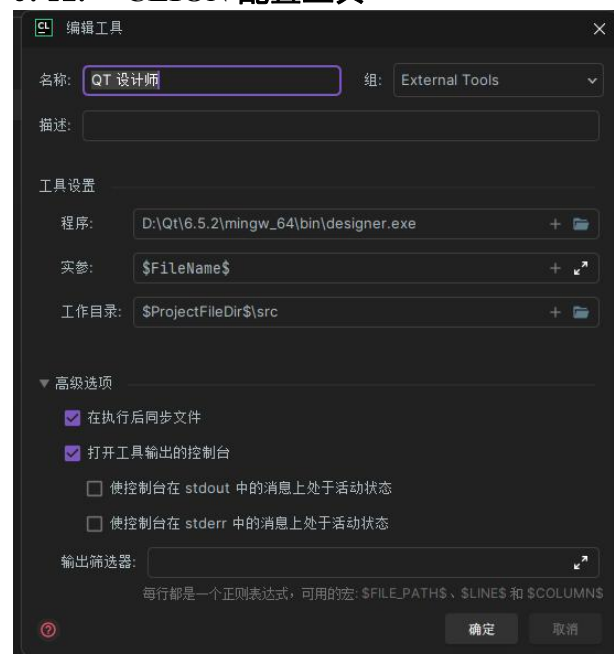
在 timerEvent()函数中, 通过 event->timerId()来确定是哪个定时器触发的;

3.在需要关闭定时器的地方调用 killTimer();

该函数的声明为: void killTimer(int Id);

该函数关闭一个定时器, 参数为定时器的编号。

9.12. CLION 配置工具



名称: 组: External Tools ▾

描述:

工具设置

程序: + 📁

实参: + ↕

工作目录: + 📁

▼ 高级选项

☒ 在执行后同步文件


☒ 打开工具输出的控制台

☐ 使控制台在 stdout 中的消息上处于活动状态

☐ 使控制台在 stderr 中的消息上处于活动状态

输出筛选器: ↕

每行都是一个正则表达式，可用的宏: \$FILE_PATH\$, \$LINE\$ 和 \$COLUMN\$



10. 树莓派

10.1. 交叉编译器

```
sudo apt-get install gcc-arm-linux-gnueabihf  
sudo apt-get install g++-arm-linux-gnueabihf
```

10.2. 交叉编译

<https://zhuanlan.zhihu.com/p/100367053>

10.2.1. 安装

```
sudo apt install gcc-arm-linux-gnueabihf
```

交叉编译

10.2.2. 问题

树莓派执行出现

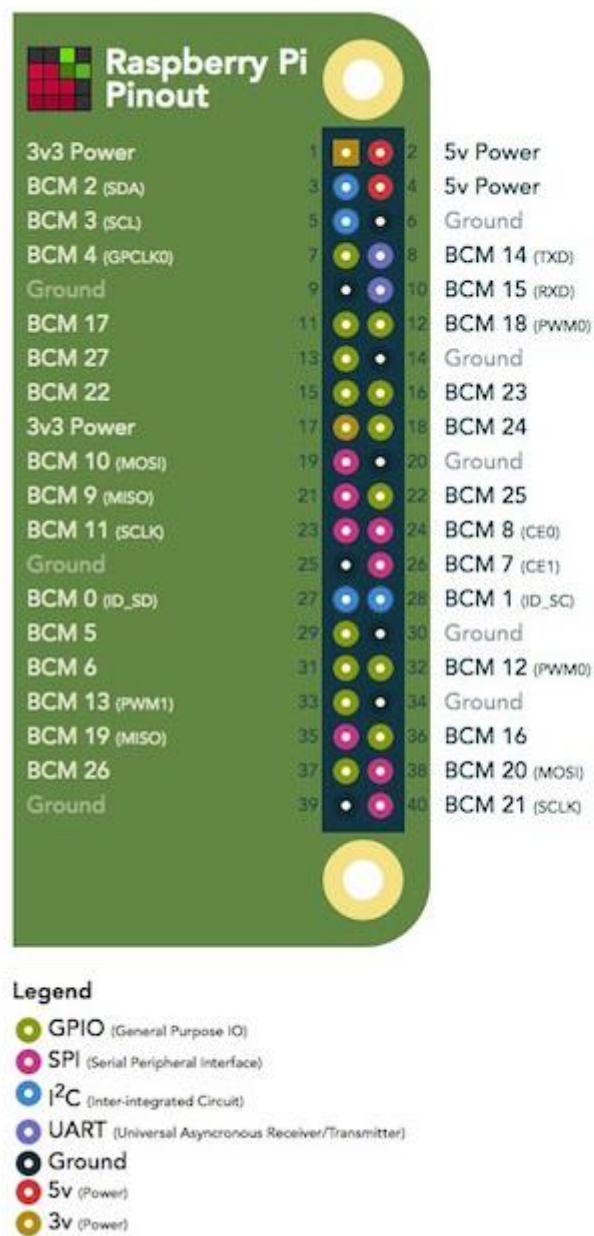
```
./hello: /lib/arm-linux-gnueabi/libc.so.6: version `GLIBC_2.34' not found (required by ./hel  
lo)
```

通过更换旧版本交叉编译器解决

旧版本下载链接

<https://releases.linaro.org/components/toolchain/binaries/>

10.3. 引脚图



YiBoard

10.4. 打开串口

<https://blog.csdn.net/playmakerDJ/article/details/105399782>

查看设备 `ls -l /dev`

设置树莓派 `sudo raspi-config`

11. CMake

参考：



从零开始详细介绍
CMake.pdf

11.1. 交叉编译

参考链接：<https://cmake.org/cmake/help/v3.25/manual/cmake.1.html#options>
<https://zhuanlan.zhihu.com/p/100367053>

11.2. 构建静态库和动态库

假设目录结构是

```
.
├── build
├── CMakeLists.txt
├── lib
│   ├── CMakeLists.txt
│   ├── hello.cpp
│   └── hello.h
```

其中 hello 为一个简单的库

外部 cmake 为

```
PROJECT(HELLO)
```

```
ADD_SUBDIRECTORY(lib bin) #lib 为包含目录, bin 为构建生成的目录
```

内部 cmake 为

```
SET(LIBHELLO_SRC hello.cpp)
```

```
ADD_LIBRARY(hello SHARED ${LIBHELLO_SRC})
```

此时直接执行 cmake 可以在 build 生成可执行程序

解析

```
ADD_LIBRARY(hello SHARED ${LIBHELLO_SRC})
```

- hello：就是正常的库名，生成的名字前面会加上 lib，最终产生的文件是 libhello.so
- SHARED，动态库 STATIC，静态库
- \${LIBHELLO_SRC}：源文件

11.3. 添加非标准库

对应 gcc 的 -L 指定目录

```
link_directories(lib)
```

```
target_link_libraries(test libshow.a)
```

12. 工具链

12.1. Ninja

使用 cmake 生成 Ninja 工程文件

```
cmake -G Ninja ..
```

使用 ninja 编译工程，会去检索当前目录下的 build.ninja 去构建
ninja

12.2. MakeFile

参考链接: <https://www.cnblogs.com/QG-whz/p/5461110.html>

命令:

原指令	原内容	翻译
-b, -m	Ignored for compatibility.	为了兼容性而忽略。
-B, --always-make	Unconditionally make all targets.	无条件地构建所有目标。
-C DIRECTORY	Change to DIRECTORY before doing anything.	在执行任何操作之前切换到 DIRECTORY 目录。
-d	Print lots of debugging information.	打印大量调试信息。
--debug[=FLAGS]	Print various types of debugging information.	打印各种类型的调试信息。
-e, --environment-override	Environment variables override makefiles.	环境变量覆盖 makefiles 中的定义。
-f FILE, --file=FILE, --makefile=FILE	Read FILE as a makefile.	以 FILE 作为 makefile 读取。
-h, --help	Print this message and exit.	打印此消息并退出。
-i, --ignore-errors	Ignore errors from commands.	忽略命令中的错误。
-I DIRECTORY	Search DIRECTORY for included makefiles.	在 DIRECTORY 中搜索包含的 makefile。
-j [N], --jobs[=N]	Allow N jobs at once; infinite jobs with no arg.	允许同时进行 N 个作业; 不带参数表示无限作业。
-k, --keep-going	Keep going when some targets can't be made.	在某些目标无法构建时继续。
-l [N], --load-average[=N], --max-load[=N]	Don't start multiple jobs unless load is below N.	除非负载低于 N, 否则不启动多个作业。

-L, --check-symlink-times	Use the latest mtime between symlinks and target.	在符号链接和目标之间使用最新的修改时间。
-n, --just-print, --dry-run, --recon	Don't actually run any commands; just print them.	不实际运行任何命令；仅打印命令。
-o FILE, --old-file=FILE, --assume-old=FILE	Consider FILE to be very old and don't remake it.	将 FILE 视为非常旧，不重新构建。
-p, --print-data-base	Print make's internal database.	打印 make 的内部数据库。
-q, --question	Run no commands; exit status says if up to date.	不执行任何命令；退出状态表示是否为最新。
-r, --no-builtin-rules	Disable the built-in implicit rules.	禁用内置的隐含规则。
-R, --no-builtin-variables	Disable the built-in variable settings.	禁用内置的变量设置。
-s, --silent, --quiet	Don't echo commands.	不回显命令。
-S, --no-keep-going, --stop	Turns off -k.	关闭-k 选项。
-t, --touch	Touch targets instead of remaking them.	触摸目标，而不是重新构建。
-v, --version	Print the version number of make and exit.	打印 make 的版本号并退出。
-w, --print-directory	Print the current directory.	打印当前目录。
--no-print-directory	Turn off -w, even if it was turned on implicitly.	关闭-w 选项，即使它是隐含开启的。
-W FILE, --what-if=FILE, --new-file=FILE, --assume-new=FILE	Consider FILE to be infinitely new.	将 FILE 视为无限新。
--warn-undefined-variables	Warn when an undefined variable is referenced.	当引用未定义的变量时发出警告。

12.3. GCC

12.3.1. 编译步骤

gcc 编译过程：

预编译：gcc -E c 源文件 -o 输出 i 目标文件；

编译阶段：gcc -S i 源文件 -o 输出 s 目标文件；

汇编阶段: gcc -c s 源文件 -o 输出 o 目标文件;

链接阶段: gcc o 源文件 -o 输出可执行文件;

12.3.2. 动态链接库

c 语言中存在静态库(.a)和动态库(.so)。

动态库也叫共享库 (share object) ,在程序链接的时候只是作些标记, 然后在程序开始启动运行的时候, 动态地加载所需库 (模块) 。

特性:

- ◆ 应用程序在运行的时候需要共享库
- ◆ 共享库链接出来的可执行文件比静态库链接出来的要小得多, 运行多个程序时占用内存空间比也比静态库方式链接少(因为内存中只有一份共享库代码的拷贝)
- ◆ 由于有一个动态加载的过程所以速度稍慢
- ◆ 更换动态库不需要重新编译程序, 只需要更换相应的库即可

相关命令

产生动态库: -shared -fPIC

使用动态库: -l name -L.

用法举例

现有两个.c 文件



```
main.c
1 #include <stdio.h>
2
3 int fun(int x);
4
5 int main() {
6     printf("hello world");
7     int val = fun(3);
8     printf("%d", val);
9     return 0;
10 }

fun.c
1
2
3 int fun(int x) {
4     return 2 * x;
5 }
```

1. 制作静态链接库

生成动态链接库, 命名必须为 libxx.so 其中, xx 为自定义名字,

(注意!!! windows 下的命名为 xx.dll)

gcc fun.c -shared -fPIC -o libxx.so

2. 编译应用

编译使用动态库的文件,下面的 xx 为上面的名字

```
gcc main.c -o main -L . -l xx
```

3. 改变环境

此时程序无法运行,根据提示可以知道程序会去 lib 目录下查找,将 libxx.so 放入到/lib 目录下,程序才能够正常运行

12.3.3. 静态链接库

静态库实际上是一些目标文件的集合,只用于链接生成可执行文件阶段。链接器会将程序中使用到函数的代码从库文件中拷贝到应用程序中,一旦链接完成生成可执行文件之后,在执行程序的时候就不需要静态库了。

特性:

- ◆ 由于每个使用静态库的应用程序都需要拷贝所用函数的代码,所以静态链接的生成的可执行文件会比较大,多个程序运行时占用内存空间比较大(每个程序在内存中都有一份重复的静态库代码)
- ◆ 由于运行的时候不用从外部动态加载额外的库了,速度会比共享库快一些
- ◆ 更换一个静态库或者修改一个静态库后,需要重新编译应用程序

静态编译

无需库的支持,即里面没有库,编译的时候加上-static

需要使用二进制代码进行生成静态链接。

下面过程以上面代码举例

静态链接库文件名称是 libxx.a, (windows 下是 libxx.lib)

1. 制作静态链接库

```
gcc -c fun.c -o fun.o      #生成目标文件 fun.o
```

```
ar -crv libfun.a fun.o     #生成静态链接文件 libfun.a
```

2. 编译应用文件

```
gcc main.c -o main -static -L . -l fun #生成静态链接的可执行文件
```

3. 执行程序

生成 main 可以在无需打包的情况下运行,无需外部添加链接文件

12.4. OpenOCD

vscode 使用 openocd 的方法链接:

<https://www.jianshu.com/p/ca26b2227a58>

12.4.1. 开启

```
openocd -f openocd.cfg
```

```

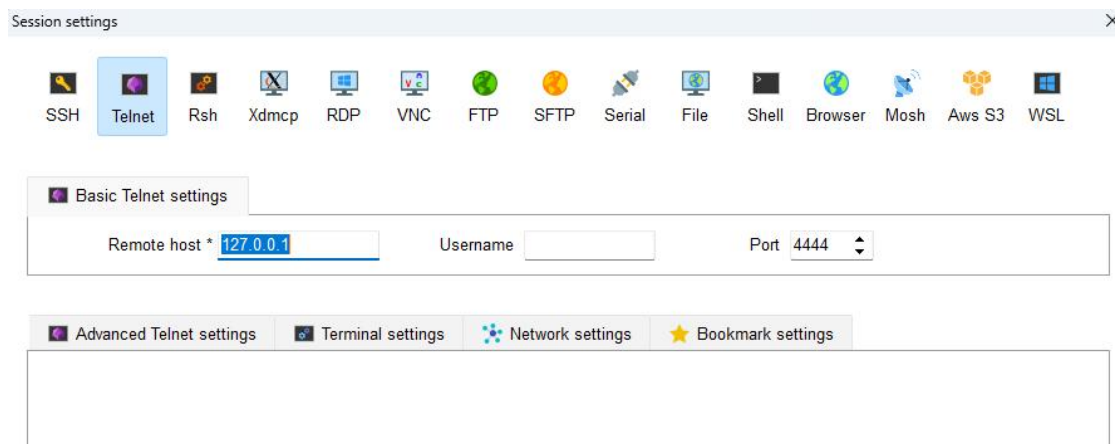
Info : SWCLK/TCK = 0 SWDIO/TMS = 1 TDI = 0 TDO = 0 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 5000 kHz
Info : SWD DPIDR 0x1ba01477
Info : [stm32f1x.cpu] Cortex-M3 r1p1 processor detected
Info : [stm32f1x.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for stm32f1x.cpu on 3333
Info : Listening on port 3333 for gdb connections
Error: [stm32f1x.cpu] clearing lockup after double fault
Polling target stm32f1x.cpu failed, trying to reexamine
Info : [stm32f1x.cpu] Cortex-M3 r1p1 processor detected
Info : [stm32f1x.cpu] target has 6 breakpoints, 4 watchpoints

```

openocd 会默认在当前路径搜索 openocd.cfg 如果有的话会执行此程序, 除非使用命令 -f 指定。

12.4.2. 进入后台

比如使用 MobaXterm



12.4.3. 烧录

在后台输入此命令即可烧录

```

program build/HELLO.elf verify reset
exit

```

或者输入

```
openocd -f openocd.cfg -c 'program build/HELLO.elf reset exit'
```

注意：这个方法需要指定 cfg，也就是前面的 “-f openocd.cfg”，这个必须要被包含，否则会导致初始化失败

12.4.4. 更简单的方法（本地）

openocd.cfg 配置文件里面能够写入执行程序，比如 reset run shutdown 等

值得注意的是，如果使用这个方法并写入 shutdown，只是烧录程序，没办法调试。

比如烧录程序的例子

```

#选择 cmsis-dap
adapter driver cmsis-dap
#swd 模式
transport select swd
source [find target/stm32f1x.cfg]
#10M xk

```

```
adapter speed 10000
```

```
program build/HELLO.elf
```

```
reset run
```

```
shutdown
```

13. USB



USB2.0中文文档.
pdf

参考 USB2.0 参考手册

13.1. 概况

参考链接欸：

https://www.cnblogs.com/shenLong1356/p/11287833.html#_label0

13.1.1. USB 通信过程简介

设备插到主机上

主机开始检测设备类型（高速/全速/低速）

复位设备

主机开始对设备枚举（根据枚举得到的各种信息加载合适的驱动程序，比如根据信息知道是一个鼠标设备，则加载鼠标的驱动程序对接下来的数据进行处理）

枚举完成后主机要发送令牌包（IN / OUT）查询有效端点是否有数据，有数据时设备自然会返还给主机

13.1.2. USB 枚举过程简介

1. 主机获取设备描述符（部分）
2. 主机对从机设置设备地址（非零，相当于我们的学号 id）
3. 主机再次获取从机设备描述符（全部）
4. 主机获取配置描述符（了解从机配置，接口，端点）情况
5. 如果有字符串描述符还有获取字符串描述符
6. 设置配置请求，就是激活配置，如果没有这一步对应的配置就不可用
7. 针对不同的类，获取它们独特的类描述符（比如 HID 报告描述符）

注意：

上面的枚举 1-4，6 步骤是必须的，

主机和从机通信时，从机时不能主动发数据给主机的，必须要等主机给从机发送令牌包后，根据主机的需求发送相应的数据

13.2. 传输速度

▪ 低速设备

例如：键盘、鼠标和游戏等外设

总线速率：1.5 Mb/s

最大的有效数据速率：800 B/s

- 全速设备

例如：手机、音频设备和压缩视频

总线速率：12 Mb/s

最大的有效数据速率：1.2 MB/s

- 高速设备

例如：视频、影像和存储设备

总线速率：480 Mb/s

最大的有效数据速率：53 MB/s

建立好了 USB 设备和主机间的连接后，需要使用 D+或 D-信号线上的上拉电阻来检测设备的速度。D+信号线上的 1.5 k Ω 大小的上拉电阻表示所连接的是一个全速设备，D-线上 1.5 k Ω 大小的上拉电阻表示所连接的是一个低速设备。

13. 3. 描述符

在 USB 中，USB HOST 是通过各种描述符来识别设备的，有设备描述符，配置描述符，接口描述符，端点描述符，字符串描述符，报告描述符等等。USB 报告描述符(Report Descriptor)是 HID 设备中的一个描述符，它是比较复杂的一个描述符。

Table 9-5. Descriptor Types

Descriptor Types	Value
DEVICE 设备描述符	1
CONFIGURATION 配置描述符	2
STRING 字符串描述符	3
INTERFACE 接口描述符	4
ENDPOINT 端点描述符	5
DEVICE_QUALIFIER 设备限定描述符	6
OTHER_SPEED_CONFIGURATION 其它速度配置描述符	7
INTERFACE_POWER ¹ 接口电源描述符	8

13. 3. 1. 设备描述符

只有一个设备描述符，设备描述符共 14 个字段，长 18 Byte。包括有 PID、VID。

偏移	字段	大小 (字节)	说明
0	bLength	1	该描述符的长度 = 18 个字节
1	bDescriptorType	1	描述符类型 = 设备 (01h)
2	bcdUSB	2	USB 规范版本 (BCD)
4	bDeviceClass	1	设备类别
5	bDeviceSubClass	1	设备子类别
6	bDeviceProtocol	1	设备协议
7	bMaxPacketSize0	1	端点 0 的最大数据包大小
8	idVendor	2	供应商 ID (VID, 由 USB-IF 分配)
10	idProduct	2	产品 ID (PID, 由制造商分配)
12	bcdDevice	2	设备释放编号 (BCD)
14	iManufacturer	1	制造商字符串索引
15	iProduct	1	产品字符串索引
16	iSerialNumber	1	序列号字符串索引
17	bNumConfigurations	1	受支持的配置数量

图 15-1

13.3.2. 配置描述符

提供特定设备配置的信息，如接口数量、设备由总线供电还是自供电、设备能否启动一个远程唤醒以及设备功耗。

偏移	字段	大小 (字节)	说明
0	bLength	1	该描述符的长度 = 9 个字节
1	bDescriptorType	1	描述符类型 = 配置 (02h)
2	wTotalLength	2	总长度包括接口和端点描述符在内
4	bNumInterfaces	1	本配置中接口的数量
5	bConfigurationValue	1	SET_CONFIGURATION 请求所使用的配置值，用于选择该配置
6	iConfiguration	1	描述该配置的字符串索引
7	bmAttributes	1	位 7: 预留 (设置为 1) 位 6: 自供电 位 5: 远程唤醒
8	bMaxPower	1	本配置所需的最大功耗 (单位为 2 mA)

13.3.3. 接口关联描述符 (IAD)

该描述符介绍两个或多个接口，这些接口与单个设备功能相关。接口关联描述符 (IAD) 会通知给主机各个接口已经连接好。例如，USB UART 具有两个与其相关的接口：控制接口和数据接口。IAD 通知主机这两个接口与同一个功能 (USB UART) 相关，并属于通信设备类别 (CDC)。

并非所有情况下都需要使用该描述符。

表 7. 接口关联描述符表

偏移	字段	大小 (字节)	说明
0	bLength	1	描述符大小 (以字节为单位)
1	bDescriptorType	1	描述符类型 = 接口关联 (0Bh)
2	bFirstInterface	1	标号与功能相关的第一个接口
3	bInterfaceCount	1	与功能相关的邻近接口的数量
4	bFunctionClass	1	类别代码
5	bFunctionSubClass	1	子类别代码
6	bFunctionProtocol	1	协议代码
7	iFunction	1	功能字符串描述符索引

13.3.4. 端点描述符

在一个设备中所使用的全部端点都有自己的描述符。该描述符会提供主机必须获取的端点信息。这些信息包括端点的方向、传输类型和数据包的最大尺寸。

13.3.5. 字符串描述符

字符串描述符是另一种可选的描述符，它为用户提供了有关设备的可读信息。该描述符中所包含的信息显示了以下内容：设备名称、生产厂家、序列号或不同接口、配置的名称。如果设备没有使用字符串，必须将前面所述的所有描述符中的字符串附加字段的值设置为 00h。

偏移	字段	大小 (字节)	说明
0	bLength	1	该描述符的长度 = 7 个字节
1	bDescriptorType	1	描述符类型 = STRING (03h)
2..n	bString 或 wLangID	变化	Unicode 编码字符串 或 LANGID 代码

图 15-2

13.3.6. 其他杂项描述符类型

13.3.7. 使用多个 USB 描述符

各个 USB 设备只有一个设备描述符。但是，一个设备可以有多种配置、接口、端点和字符串描述符。设备执行枚举时，终端阶段中有一个步是读取设备描述符，并选择需要使能的设备配置类型。每一次操作只能使能一种配置。例如，某个设计中存在两种配置：一种适用于自供电的设备，另一种适用于由总线供电的设备。这时，用于自供电设备的 USB 的总体性能会与使用于总线供电设备的不一樣。拥有多种配置和多种配置描述符可允许设备选择性实现该功能。

同时一个设备可以有多种接口，因此，它也会有多种接口描述符。具有多种接口的 USB 设备（能够执行不同功能）被称为复合设备。USB 头戴式音频耳机便是一个复合设备示例。这种音频耳机包括一个带有两个接口的 USB 设备。其中，一个接口用于音频传输，另一个接口可用于音量调整。可以同时使能多个接口。图 45 显示的是单个 USB 设备中如何分配两种接口。

13.4. 枚举

枚举指的是被识别



图 15-3

13.4.1. 动态检测

1. 设备被连接到 USB 端口上，并得到检测。此时，设备可从总线吸收 100 mA 的电流，并处于被供电状态。
2. 集线器通过监控端口的电压来检测设备。

13.4.2. 枚举

1. 主机使用中断端点获得集线器状态（包括端口状态的变化），从而了解新连接的设备。主机从集线器获得设备检测情况后，它会向集线器发送一个请求，以便询问在 GET_PORT_STATUS 请求有效时所发生状态变化的详细信息。
2. 主机收集该信息后，它通过“USB 速度”一节中所介绍的方法来检测设备的速度。最初，通过确定上拉电阻位于 D+线还是 D-线，集线器可以检测设备速度是全速还是低速。通过另一个 GET_PORT_STATUS 请求，该信息被报告给主机。
3. 主机向集线器发送 SET_PORT_FEATURE 请求，要求它复位新连接的设备。通过将 D+和 D-线下拉至 GND (0 V)，使设备进入复位状态。这些线处于低电平状态的时间长达 2.5 us，因此发生复位条件。集线器在 10 ms 内维持复位状态。

4. 复位期间发生一系列 J-State 和 K-State, 这样是为了确定设备是否支持高速传输。如果设备支持高速, 它会发出一个单一的 K-State。高速集线器检测该 K-State 并用 J 和 K 顺序 (组成 “KJKJKJ” 格式) 来回应。设备检测到该格式后, 它会移除 D+ 线上的上拉电阻。低速设备和全速设备则会忽略这一步。
5. 通过发送 GET_PORT_STATUS 请求, 主机检查设备是否仍处于复位状态。如果设备仍处于复位状态, 则主机会继续发送请求, 直到它得知设备退出复位状态为止。设备退出复位状态后, 它便进入默认状态, 如 USB 电源一节所述。现在, 设备可以回应主机的请求, 具体是对其默认地址 00h 进行控制传输。所有 USB 设备的起始地址均等于该默认地址。每次只能有一个 USB 设备使用该地址。因此, 同时将多个 USB 设备连接到同一个端口时, 它们会轮流进行枚举, 而不是同时枚举。
6. 主机开始了解有关设备的更多信息。首先, 它要知道默认管道 (端点 0) 的最大数据包大小。主机先向设备发送 GET_DESCRIPTOR 请求。设备发给主机相应应用笔记 USB 描述符一节所介绍的描述符。在设备描述符中, 第八个字节 (bMaxPacketSize0) 包含了有关 EP0 最大数据包尺寸的信息。Windows 主机要求 64 字节, 但仅在收到 8 字节设备描述符后它才转换到控制传输的状态阶段, 并要求集线器复位设备。USB 规范要求, 如果设备的默认地址为 00h, 当它得到请求时, 设备至少要返回 8 字节设备描述符。要求 64 字节是为了防止设备发生不确定行为。此外, 仅在收到 8 字节后才进行复位的操作是早期 USB 设备遗留的特性。在早期 USB 设备中, 当发送第二个请求来询问设备描述符时, 某些设备没有正确回应。为了解决该问题, 在第一个设备描述符请求后需要进行一次复位。被传输的 8 字节包含 bMaxPacketSize0 的足够信息。
7. 主机通过 SET_ADDRESS 请求为设备分配地址。在使用新分配地址前, 设备使用默认地址 00h 完成所请求的状态阶段。在该阶段后进行的所有通信均会使用新地址。如果断开与设备的连接、端口被复位或者 PC 重启, 该地址可能被更改。现在, 设备处于地址状态。

13.4.3. 配置

1. 设备退出复位状态后, 主机会发送 GET_DESCRIPTOR 命令, 以便使用新分配地址读取设备的描述符。
2. 为了让主机 PC (此情况是 Windows PC) 成功使用设备, 主机必须加载设备驱动程序。
3. 收到所有描述符后, 主机使用 SET_CONFIGURATION 请求进行特殊的设备配置。
4. 此时设备将处于配置状态。

13.5. 获取设备 pid 和 vid

使用 USB TreeView 软件, 方便直观可以看到 USB 设备的插拔和定位

下载链接: https://www.majorgeeks.com/files/details/usb_device_tree_viewer.html

13.6. HID 人机类

13.7. MSD 大容量类

13.8. CDC 通信设备类

13.9. HIDAPI

开源地址:

<https://github.com/signal11/hidapi>

注意事项:

需要链接 gcc 静态库,

cmake 添加 `target_link_libraries(${PROJECTNAME} setupapi)`

qmake 添加 `QT += -lsetupapi`

其中, `${PROJECTNAME}` 为工程名

13.10. DFU

参考链接: <https://cloud.tencent.com/developer/article/2197665>

14. GIT 版本控制

14.1. 常用命令

命令	功能
<code>git init</code>	创建为可管理仓库
<code>git status</code>	查看当前版本状态
<code>git add</code>	添加内容到仓库 一般 <code>git add .</code> 表示全部添加
<code>git commit</code>	提交到本地仓库 需要添加注释 具体为" <code>git commit -m "user commit"</code> "
<code>git remote add origin</code>	与网络仓库绑定 <code>git remote add origin git@github.com:FlameKm/test.git</code>
<code>git push</code>	提交到网上 一般举例: <code>git push origin master</code>
<code>git log</code>	日志
<code>git clone</code>	克隆到本地

14.2. 下载

有些需要绑定,即 ssh key

使用命令 `git clone`

14.3. 上传

如何使用 git 命令行上传项目到 [github_DreamMakers 的博客-CSDN 博客_git 新建仓库上传代码](#)

新建文件夹 test

在 test 内输入 `git init`

```
F:\32stm\驱动\我的驱动\test>git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in F:/32stm/驱动/我的驱动/test/.git/
F:\32stm\驱动\我的驱动\test>
```

放入代码



输入 `git add .` 添加全部内容

输入 `git commit -m "注释内容"` 添加到本地仓库

在 github 新建一个仓库,与本地关联

似乎需要 github(或其它)ssh key 绑定本地,否则不安全

绑定本地仓库

`git remote add origin git@github.com:FlameKm/test.git`

如果不为空

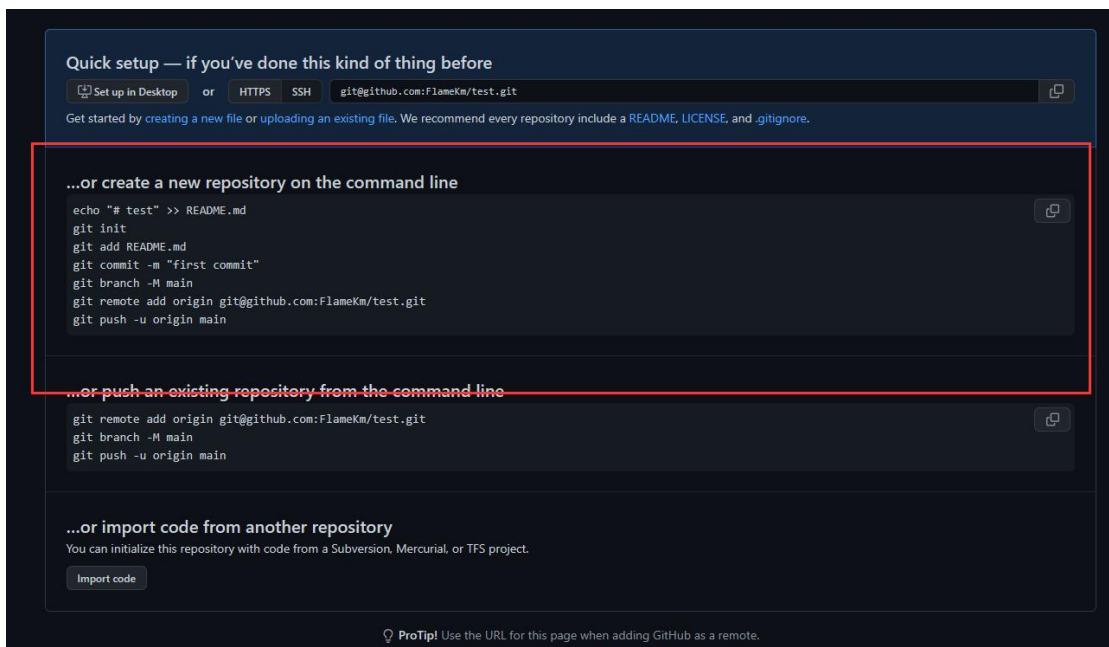
`git pull --rebase origin 分支名`

提交

`git branch -M main`

`git push -u origin main`

总结如下,GitHub 新建文件夹时也会提醒



14. 4. 更新

- 1、检查变更 `git status`
- 2、更新全部修改到本地仓库 `git add .`
- 3、提交 `git commit -m "说明信息"`
- 4、上传 `git push origin master`

14. 5. 版本回退

Git 使用教程,最详细, 最傻瓜, 最浅显, 真正手把手教 - 知乎 (zhihu.com)

14. 6. 子模块 submodule

- 添加: `git submodule add https://github.com/xxx/child.git`

- 克隆 1: 克隆大仓库后, 在大仓库路径输入 `git submodule init & git submodule uopda` `te` 即可

- 克隆 2: `git clone --recurse-submodules https://github.com/xxx/parent.git`

也可以设置 `submodule.recurse` 为 `true`, 这样每次拉去都是带`--recurse-submodules`

- 删除:

`rm -rf 子模块目录` 删除子模块目录及源码

`vi .gitmodules` 删除项目目录下`.gitmodules` 文件中子模块相关条目

`vi .git/config` 删除配置项中子模块相关条目

`rm .git/module/*` 删除模块下的子模块目录, 每个子模块对应一个目录, 注意只删除对应的子模块目录即可

执行完成后, 再执行添加子模块命令即可, 如果仍然报错, 执行如下:

```
git rm -f --cached 子模块名称
```

作者: pomelo_西

链接: <https://www.jianshu.com/p/10ae453701ed>

来源: 简书

著作权归作者所有。商业转载请联系作者获得授权, 非商业转载请注明出处。

14.7. 重新排除.gitignore

```
git rm -r --cached .
```

```
git add .
```

15. 网络

15.1. git

开启代理: `git config --global http.proxy http://127.0.0.1:7890`

取消代理: `git config --global --unset http.proxy`

查询代理: `git config --global http.proxy`

15.2. linux

`export http_proxy=http://127.0.0.1:7890`

`export https_proxy=https://127.0.0.1:7890`

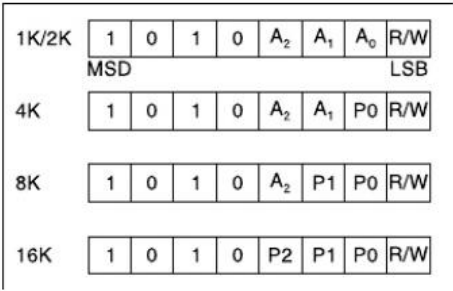
16. 芯片或模块

16. 1. EEPROM

通信：IIC

设备地址：0XA0 已经左移一位

Figure 1. Device Address



另外有关于容量的说明：

- AT24C01：一共128页，每页1字节，共需7位地址数据
- AT24C02：一共256页，每页1字节，共需8位地址数据
- AT24C04：一共256页，每页2字节，共需9位地址数据
- AT24C08：分4个块，一共256页，每页4字节，共需10位地址数据
- AT24C16：分8块，一共256页，每页8字节，共需11位地址数据

图 17-1

参考链接：<https://www.cnblogs.com/schips/p/at24cxx.html>

17. Zephyr

17.1. 个人理解

Zephyr 操作系统类似一个单片机开发 IDE，代码具有高耦合度，与硬件高解耦。能够提高代码的复用率，大概的实现方式是，将代码层面使用设备树进行管理硬件，通过 cmake（或者其它）工具将设备树文件转换为一个头文件，其中包含了不同的硬件使用的不同的宏，比如 STM32 中 GPIO 的 port、pin，从而实现代码的解耦。

优点：

- 代码高复用
- 设备树管理
- 开发方式与 linux 较为相似。

缺点：

- 门槛高
- 工具链复杂

17.2. Hello World

17.2.1. 安装

https://docs.zephyrproject.org/latest/develop/getting_started/index.html

17.2.1.1. python 虚拟环境

安装：sudo apt install python3-venv

创建：python3 -m venv ~/zephyrproject/.venv

打开：source ~/zephyrproject/.venv/bin/activate

关闭：deactivate

17.2.2. 编译烧录（支持的开发板）

cd ~/zephyrproject/zephyr/ #进入工程目录

构建

west build -p always -b stm32f103_mini samples/basic/blinky

或.

rm -rf build/* #清除上次构建

cmake -Bbuild -GNinja -DBOARD=stm32f103_mini samples/basic/blinky
ninja -Cbuild

cmake 的理解

-B 生成目录

-G 构建工具

-DBOARD 为了确定设备树

最后一项 工程目录

烧录

west flash

或

ninja flash

这个是配置在 build.ninja 的脚本，如果使用 make 的话就是 make.ninja

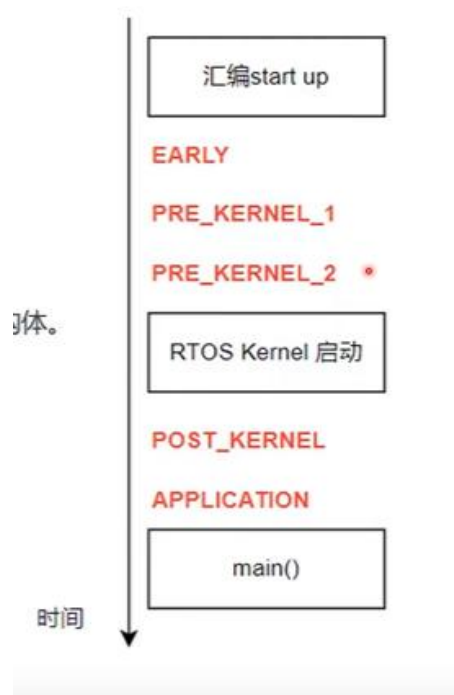
17.3. 调度算法

Zephyr 使用的调度算法主要是多级反馈队列（Multilevel Feedback Queue, MLFQ）调度算法，它是一种基于优先级的动态调度策略。这个算法允许任务在运行时改变其优先级，从而根据任务的行为和特性来动态地调整任务的执行顺序。

高低优先级：抢占

同等优先级：协作式

17.4. 启动方式



17.5. 时间

17.5.1. 定时器

地址: <https://docs.zephyrproject.org/latest/kernel/services/timing/timers.html>

使用计时器在指定时间后启动异步操作。

使用计时器来确定是否已经过了指定的时间。特别是，当需要比更简单的 `k_sleep()` 和 `k_usleep()` 调用提供的精度和/或单位控制更高的精度和/或单位控制时，应使用计时器。

在执行涉及时间限制的操作时，使用计时器来执行其他工作。

计时器是一个内核对象，它使用内核的系统时钟来测量时间的流逝。当达到计时器指定的时间限制时，它可以执行应用程序定义的操作，也可以简单地记录到期时间并等待应用程序读取其状态。

- 类型: `k_timer`
- 定义: `k_timer_init` or `K_TIMER_DEFINE`
- 启动: `k_timer_start`
- 停止: `k_timer_stop`

API

```
void k_timer_init(struct k_timer *timer, k_timer_expiry_t expiry_fn, k_timer_stop_t stop_fn)¶
```

初始化一个定时器。

该例程在第一次使用之前初始化计时器。

- 参数：**
- 计时器 - 计时器的地址。
 - `expiry_fn` - 每次计时器到期时调用的函数。
 - `stop_fn` - 计时器在运行时停止时调用的函数。

如果不需要可以指定为 `NULL`

```
void k_timer_start(struct k_timer *timer, k_timeout_t 持续时间, k_timeout_t 周期)¶
```

启动计时器。

该例程启动一个计时器，并将其状态重置为零。计时器开始使用指定的持续时间和周期值进行倒计时。

允许尝试启动已在运行的计时器。计时器的状态重置为零，并且计时器开始使用新的持续时间和周期值进行倒计时。

- 参数：**
- 计时器 - 计时器的地址。
 - 持续时间 - 初始计时器持续时间。
 - `period` - 定时器周期。

```
void k_timer_start(struct k_timer *timer, k_timeout_t duration, k_timeout_t period)
```

Start a timer.

This routine starts a timer, and resets its status to zero. The timer begins counting down using the specified duration and period values.

Attempting to start a timer that is already running is permitted. The timer's status is reset to zero and the timer begins counting down using the new duration and period values.

Parameters:

- **timer** - Address of timer.
- **duration** - Initial timer duration.
- **period** - Timer period.

duration:指定计时器第一次到期之前的时间间隔的持续时间。

period:指定第一个计时器到期后所有计时器到期之间的时间间隔的周期

例如，如果计时器以 200 的持续时间和 75 的周期启动，它将首先在 200ms 后过期，然后每 75ms 过期一次。

举例：

17.5.2. 计时器

1. 调用 `timing_init()` 来初始化定时器。
2. 调用 `timing_start()` 以发出开始收集计时信息的信号。这通常会启动计时器。
3. 调用 `timing_counter_get()` 来标记代码执行的开始。
4. 调用 `timing_counter_get()` 来标记代码执行结束。
5. 调用 `timing_cycles_get()` 来获取代码执行开始和结束之间的定时器周期数。
6. 使用总周期数调用 `timing_cycles_to_ns()`，将周期数转换为纳秒。
7. 重复步骤 3 以收集其他代码块的计时信息。
8. 调用 `timing_stop()` 以发出计时信息收集结束的信号。这通常会停止计时器。

17.5.3. 时间转换

K_SECONDS：生成秒级超时延迟

K_MSEC：生成毫秒级的超时延迟。

K_MINUTES：生成几分钟的超时延迟。

K_HOURS：生成几小时的超时延迟。

K_FOREVER：没有参数，无限长时间

17.6. 设备树

总结

- DeviceTree本身的语法只提供了一个基于总线主从关系的树形层次结构，此外每个节点可以用属性来存储信息。语法本身并没有规定硬件要如何描述。
- DeviceTree中的一些常见属性，补充了这方面的空缺。
 - reg、ranges、#address-cells、#size-cells这四个属性描述了总线上的地址分配
 - status属性描述了设备是否使能
 - compatible属性描述了设备的兼容性
- 在DeviceTree中，除了本身的树形结构以外，还具有一些逻辑上的树形结构，称为域。域具有控制器和设备节点，控制器是真正实现域的功能的硬件外设，而设备节点只是为了开发方便解耦而进行的一种抽象。
- 真正限制device tree中属性该如何写的，是device binding文件。binding文件是芯片厂商提供的。有了binding文件，就可以在VS Code中实现自动的检查与补全。Zephyr实际构建项目时，也是参考binding文件来检查dts的正确性。只有dts按照正确的规则写了，zephyr的驱动代码才能识别到硬件配置，进行自动初始化。
- zephyr中会有一些特殊的虚拟节点来为开发提供便利。

17.6.1. 节点

在C代码中访问DeviceTree —— 节点id

要想在代码中访问到DeviceTree中的信息，需要通过DeviceTree API来实现：

```
#include <zephyr/devicetree.h>
```

为了获得某个节点的属性，首先需要这个节点的id（node identifier）来作为句柄。节点id本质就是devicetree_generated.h中的宏定义。

获得节点id的方式有很多：

获取方式	示例	说明
根节点	DT_ROOT	根节点id
绝对路径	DT_PATH(soc, serial_40001000)	/soc/serial@40001000
Label	DT_NODELABEL(serial1)	根据dts中定义的label来找到节点
chosen节点	DT_CHOSEN(zephyr_console)	根据dts中chosen节点的配置： zephyr_console=&uart0

参考地址：<https://docs.zephyrproject.org/latest/build/dts/api/api.html#generic-apis>

1. DT_PATH

从路径获取节点

```
/ {
    soc {
        serial1: serial@40001000 {
            status = "okay";
            current-speed = <115200>;
            ...
        };
    };
};
```

可以使用 DT_PATH(soc, serial_40001000)获取节点

2. DT_PROP

获取节点中的某个数据，还是上例可以使用

```
DT_PROP(DT_PATH(soc, serial_40001000), current_speed)
```

获取到数值 115200

3. DT_NODELABEL

将节点标签中的非字母数字字符转换为下划线以形成有效的 C 标记，并将所有字母小写。

请注意，节点标签与标签属性不同。

上例变更为

```
serial1: serial@40001000 {
    label = "UART_0";
    status = "okay";
    current-speed = <115200>;
    ...
};
```

此时可以使用 DT_NODELABEL(serial1)获取当前节点

注意这里字符串 UART_0 不是节点标签；它是名为 label 的属性的值。

使用的话可以这样

```
DT_PROP(DT_NODELABEL(serial1), current_speed) // 115200
```

同样是获取 115200

4. DT_ALIAS(alias)

上例添加如下代码

```
aliases {
    my-serial = &serial1;
};
```

此时，可以使用 DT_ALIAS(my_serial)获取节点

5. DT_INST

这个比较复杂

示例代码

```
serial1: serial@40001000 {
    compatible = "vnd,soc-serial";
    status = "disabled";
    current-speed = <9600>;
    ...
};
```

```
serial2: serial@40002000 {
    compatible = "vnd,soc-serial";
    status = "okay";
    current-speed = <57600>;
    ...
};
```

```
serial3: serial@40003000 {
    compatible = "vnd,soc-serial";
```

```

        current-speed = <115200>;
        ...
};

```

禁用的节点分配最大的标签

DT_INST(0, vnd_soc_serial)表示的是 serial1 或者 serial2 无法确定

DT_INST(1, vnd_soc_serial)同上

但是 DT_INST(2, vnd_soc_serial) 表示的一定是 serial1

6. DT_PARENT | DT_GPARENT | DT_CHILD

获取父节点的节点标识符,获取祖父节点, 获取子节点

7. DT_NODE_PATH

获取设备树节点的完整路径作为字符串文字

```

/ {
    soc {
        node: my-node@12345678 { ... };
    };
};

DT_NODE_PATH(DT_NODELABEL(node)) // "/soc/my-node@12345678"
DT_NODE_PATH(DT_PATH(soc))       // "/soc"
DT_NODE_PATH(DT_ROOT)            // "/"

```

8. DT_NODE_FULL_NAME

上面的示例会变成

```
DT_NODE_FULL_NAME(DT_NODELABEL(node)) // "my-node@12345678"
```

17.6.2. 特殊节点

/chosen

/aliases

/pinctrl

/zephyr,user 无需写 device 就可以使用, 方便用户开发的节点

可以直接在根节点中, 不属于标准硬件节点

17.6.3. 设备

DEVICE_DT_GET(node_id)
通过node_id获取device

DEVICE_DT_GET_ANY(compat)
通过compatible获取device

DEVICE_DT_GET_ONE(compat)
通过compatible获取device

DEVICE_GET(name)
通过name (label属性) 获取device

DEVICE_DT_GET_OR_NULL(node_id)
判断节点是否okay状态

device_get_binding(name)
通过name (label属性) 获取device

device_is_ready(device)
判断device结构是否可以使用

获取配套



17.6.4. 命名

The recommended format is "vendor,device", like "avago,apds9960", or a sequence of these, like "ti,hdc", "ti,hdc1010". The vendor part is an abbreviated name of the vendor. The file [dts/bindings/vendor-prefixes.txt](#) contains a list of commonly accepted vendor names. The device part is usually taken from the datasheet.

推荐的格式是“供应商，设备”，例如“avago, apds9960”，或这些格式的序列，例如“ti, hdc”，“ti, hdc1010”。供应商部分是供应商的缩写名称。文件 dts/bindings/vendor-prefixes.txt 包含普遍接受的供应商名称列表。器件部分通常取自数据表。

It is also sometimes a value like gpio-keys, mmio-sram, or fixed-clock when the hardware's behavior is generic.

当硬件行为是通用时，有时它也是一个值，如 GPIO-Keys、MMIO-SRAM 或固定时钟。

17.6.5. 注册

reg

Information used to address the device. The value is specific to the device (i.e. is different depending on the compatible property).

The reg property is a sequence of (address, length) pairs. Each pair is called a "register block". Values are conventionally written in hex.

Here are some common patterns:

- Devices accessed via memory-mapped I/O registers (like i2c@40003000): address is usually the base address of the I/O register space, and length is the number of bytes occupied by the registers.
- I2C devices (like apds9960@39 and its siblings): address is a slave address on the I2C bus. There is no length value.
- SPI devices: address is a chip select line number; there is no length.

i2c40003000 是芯片中的 iic 地址

下面的是 iic 总线上的设备地址

一般涉及芯片的操作都是在 soc 节点下的。

17.6.6. 绑定

zephyr build system 会从以下位置寻找绑定文件

```
zephyr/dts/bindings/  
${board_dir}/dts/bindings/  
${project_dir}/dts/bindings/
```

也可以在 cmake 文件中添加 `list(APPEND DTS_ROOT /path/to/your/dts)` 用来指定目录

也可以在编译时, 增加选项 `west build -b <board_name> -- -DTS_ROOT=<path/to/your/dts>`

如果自定义设备类型, 可以把 yaml 文件添加到以上位置

文件名推荐和 compatible 一致, 但不是必须的

17.7. 信号 (原子操作)

17.7.1. 信号量 Semaphores

<https://docs.zephyrproject.org/latest/kernel/services/synchronization/semaphores.html>

使用信号量来控制多个线程对一组资源的访问。使用信号量来同步生产和消费线程或 ISR 之间的处理。

- 类型: `k_sem`
- 定义: `k_sem_init()` or `K_SEM_DEFINE(name, initial_count, count_limit)`
- 给出: `k_sem_give()`
- 获取: `k_sem_take()`
- 获取计数: `k_sem_count_get()`
- 重置: `k_sem_reset()`

举例

```
struct k_sem my_sem;  
k_sem_init(&my_sem, 0, 1);  
k_sem_give(&my_sem);  
  
if (k_sem_take(&my_sem, K_MSEC(50)) != 0) {  
    printk("Input data not available!");  
} else {  
    /* fetch available data */  
    ...  
}
```

17.7.2. 互斥量 Mutexes

<https://docs.zephyrproject.org/latest/kernel/services/synchronization/mutexes.html>

使用互斥体提供对资源 (例如物理设备) 的独占访问。

- 类型: `k_mutex`

- 定义: `k_mutex_init()` or `K_MUTEX_DEFINE(my_mutex);`
- 锁定: `k_mutex_lock()`
- 解锁: `k_mutex_unlock()`

说明: 无法锁定时则表示锁正在被使用, 使用完成后需要解锁

```
if (k_mutex_lock(&my_mutex, K_MSEC(100)) == 0) {
    /* mutex successfully locked */
} else {
    printf("Cannot lock XYZ display\n");
}
```

17.7.3. 互斥量 Futex

地址同互斥量 Mutex

`k_futex` 是一个轻量级互斥原语, 旨在最大限度地减少内核参与。无竞争操作仅依赖于对共享内存的原子访问。`k_futex` 作为内核对象进行跟踪, 并且可以驻留在用户内存中, 以便任何访问都绕过内核对象权限管理机制。

17.7.4. 条件变量

地址: <https://docs.zephyrproject.org/latest/kernel/services/synchronization/condvar.html>

使用带有互斥体的条件变量来表示从一个线程到另一线程的状态(条件)变化。条件变量不是条件本身, 也不是事件。该条件包含在周围的编程逻辑中。

互斥体本身并不是设计用作通知/同步机制的。它们旨在仅提供对共享资源的互斥访问。

- 类型: `k_condvar`
- 定义: `k_condvar_init` or `K_CONDVAR_DEFINE`
- 等待: `k_condvar_wait`
- 发出: `k_condvar_signal`
- 过程

1. 释放最后获取的互斥体。
2. 将当前线程放入条件变量队列中。

举例

```
K_MUTEX_DEFINE(mutex);
K_CONDVAR_DEFINE(condvar)
```

```
int main(void)
{
    k_mutex_lock(&mutex, K_FOREVER);
    /* block this thread until another thread signals cond. While
     * blocked, the mutex is released, then re-acquired before this
     * thread is woken up and the call returns.
     */
}
```

```

    k_condvar_wait(&condvar, &mutex, K_FOREVER);
    ...
    k_mutex_unlock(&mutex);
}

```

17.8. 数据传输

17.8.1. Queue (底层实现)

https://docs.zephyrproject.org/latest/kernel/services/data_passing/queues.html

Zephyr 中的队列是一个实现传统队列的内核对象，允许线程和 ISR 添加和删除任意大小的数据项。队列类似于 FIFO，并且充当 k_fifo 和 k_lifo 的底层实现。有关使用的更多信息，请参阅 k_fifo。

- 定义：k_queue_init or K_QUEUE_DEFINE
- 尾插入队：k_queue_append
- 头插入队：k_queue_prepend
- 中间插队：k_queue_insert
- 出队：k_queue_get
- 判空：k_queue_is_empty
- 取消等待：k_queue_cancel_wait

17.8.2. FIFOs (底层实现)

https://docs.zephyrproject.org/latest/kernel/services/data_passing/fifos.html

FIFO 是一个内核对象，它实现**传统的先进先出** (FIFO) 队列，允许线程和 ISR 添加和删除任意大小的数据项。

定义：k_fifo_init or K_FIFO_DEFINE

入队：k_fifo_put

出队：k_fifo_get (带超时)

判空：k_fifo_is_empty

17.8.3. LIFO (底层实现)

LIFO 是一个内核对象，它实现传统的**后进先出** (LIFO) 队列，允许线程和 ISR 添加和删除任意大小的数据项。

- 定义：k_lifo_init or K_LIFO_DEFINE
- 入栈：k_lifo_put
- 出栈：k_lifo_get (带超时)
- 判空：k_lifo_is_empty

17.8.4. 栈 Stacks

地址: https://docs.zephyrproject.org/latest/kernel/services/data_passing/stacks.html

当存储项的最大数量已知时, 使用堆栈以“后进先出”的方式存储和检索整数数据值。

堆栈是一个内核对象, 它实现传统的后进先出 (LIFO) 队列, 允许线程和 ISR 添加和删除有限数量的整数数据值。

- 类型: `k_stack`
- 定义: `k_stack_init`
- 入栈: `k_stack_push`
- 出栈: `k_stack_pop`
-

17.8.5. 消息队列 MessageQueues

地址: https://docs.zephyrproject.org/latest/kernel/services/data_passing/message_queues.html

使用消息队列以异步方式在线程之间传输小数据项。

是一个环形缓冲区, 消息队列的**环形缓冲区必须与 N 字节边界对齐**, 其中 N 是 2 的幂 (即 1、2、4、8、...)。为了确保存储在环形缓冲区中的消息类似地与此边界对齐, 数据项大小也必须是 N 的倍数。数据项可以通过线程或 ISR 发送到消息队列。

- 类型: `k_msgq`
- 定义: `k_msgq_init` or `K_MSGQ_DEFINE`
- 入队: `k_msgq_put`
- 出队: `k_msgq_get`
- 查看队头: `k_msgq_peek`

举例:

```
struct data_item_type {
    uint32_t field1;
    uint32_t field2;
    uint32_t field3;
};
```

```
char __aligned(4) my_msgq_buffer[10 * sizeof(struct data_item_type)];
struct k_msgq my_msgq;
```

```
k_msgq_init(&my_msgq, my_msgq_buffer, sizeof(struct data_item_type), 10);
```

or

```
K_MSGQ_DEFINE(my_msgq, sizeof(struct data_item_type), 10, 4);
```

17.8.6. 邮箱 Mailboxes

地址: https://docs.zephyrproject.org/latest/kernel/services/data_passing/mailboxes.html

当消息队列的能力不足时, 使用邮箱在线程之间传输数据项。

邮箱是一个内核对象，它提供了超出消息队列对象功能的增强消息队列功能。邮箱允许线程同步或异步发送和接收任何大小的消息。

- 类型：k_mbox
- 定义：k_mbox_init or K_MBOX_DEFINE

17.8.7. 管道

地址：https://docs.zephyrproject.org/latest/kernel/services/data_passing/pipes.html

管道是一个内核对象，允许线程将字节流发送到另一个线程。管道可用于同步传输全部或部分数据块。

管道可以配置一个**环形缓冲区**，用于保存已发送但尚未接收的数据；或者，管道可以没有环形缓冲区。

使用管道在线程之间发送数据流。