



北京理工大學珠海學院
BEIJING INSTITUTE OF TECHNOLOGY, ZHUHAI

學以精之
德以明理

嵌入式系统原理与设计 课程作品设计答辩—— 基于STM32的显示自适应万年历

BEIJING INSTITUTE OF TECHNOLOGY



北京理工大學珠海學院
BEIJING INSTITUTE OF TECHNOLOGY, ZHUHAI

目 录

CONTENTS

- 1 项目背景
Project Background
- 2 系统架构
System Architecture
- 3 硬件构成
Hardware Composition
- 4 软件代码
Software Code
- 5 总结
Summary



第一部分 ▷▷

项目背景

- 项目简介与环境
- RTC背景回顾
- HAL_RTC库的缺陷
- 重写RTC库

德以明理 学以精工



本项目是基于正点原子精英板 V2 开发的智能万年历。具备 LCD 时间显示、时间设置及自动深浅色模式切换功能。核心亮点是自研 RTC 库，解决了 HAL 库掉电丢失和重启延迟问题。项目采用模块化、解耦设计，运用时间片轮询与任务驱动思想，代码规范且注释完善。



HAL 库



STM32CubeMX6.14.0

STM32Cube MCU Package for
STM32F1 Series 1.8.6

集成开发环境



Keil5 V5.38.0.0

MDK-ARM 5.41.0.0

编辑器



Visual Studio Code

Keil Assistant 1.7.0

字符集



统一使用 GB 2312 字符集



“

RTC

RTC，即实时时钟，是一个能够独立于主系统电源，持续保持和更新当前日期与时间信息的硬件模块或芯片。它的核心功能就是“计时”。它通常包含一个高精度的晶体振荡器（Crystal Oscillator）作为时间基准，以及一些寄存器来存储年、月、日、时、分、秒等信息。它有低功耗的特性。RTC通常连接一个独立小容量的后备电池（Backup Battery）保障其在主系统断电后继续工作，保持时间的准确。



- RTC 内部的预分频器负责将时钟源（如 LSE 的 32.768 KHz 信号）进行分频，最终产生一个 1 Hz 的脉冲信号供给计数器。这个计数器是一个 32 位的寄存器，其计数值范围是 0 到 $2^{32} - 1$ 。每当 1 Hz 的信号产生一个脉冲（即每经过一秒钟），计数器的值就会加 1。
- 通常，我们会将时间以 Unix 时间戳的形式存储在这个 32 位的 RTC 计数器中。这种时间戳表示自 1970 年 1 月 1 日以来经过的秒数。基于 32 位计数器（无符号）的限制，这种表示方法的有效时间范围大约是从 1976 年到 2106 年，之后会发生溢出。



- 系统采用 3.3V 电源为 VDD 供电。当 VDD 掉电时，后备电池（VBAT）会接管，为 RTC 所在的后备区域提供持续供电。由于 RTC 核心模块位于这个后备区域，因此 VDD 掉电不会影响其运行，VBAT 能确保时间持续更新。
- 在时钟源的选择上，只有低速外部时钟（LSE，通常为 32.768 KHz）能够在 VDD 掉电后，依然由 VBAT 供电继续工作，因此我们选用 LSE 作为 RTC 的时钟源。
- 此外，后备区域还包含备份寄存器。这些寄存器同样由 VBAT 维持供电，因此存储的数据在系统掉电后不会丢失。它们还具备“入侵检测”功能，我们可以利用这一特性来判断 RTC 时钟是否曾被手动设置过。



HAL 库的 RTC 时钟部分将时间与日期分开处理，导致其并未将日期数据记录在 RTC 计数器中，从而使得日期数据会在 VDD 掉电后丢失。

默认生成的代码每次都会初始化 RTC，但初始化过程中会使得 RTC 停止运行一小会，导致每次程序重启时 RTC 时间变慢一点。

```

543 /* RTC Time and Date functions *****/
544 /**
545 * @{
546 */
547 HAL_StatusTypeDef HAL_RTC_SetTime(RTC_HandleTypeDef *hrtc, RTC_TimeTypeDef *sTime, uint32_t Format);
548 HAL_StatusTypeDef HAL_RTC_GetTime(RTC_HandleTypeDef *hrtc, RTC_TimeTypeDef *sTime, uint32_t Format);
549 HAL_StatusTypeDef HAL_RTC_SetDate(RTC_HandleTypeDef *hrtc, RTC_DateTypeDef *sDate, uint32_t Format);
550 HAL_StatusTypeDef HAL_RTC_GetDate(RTC_HandleTypeDef *hrtc, RTC_DateTypeDef *sDate, uint32_t Format);

```

```

49 /**
50  * @}
51 hrtc.Instance = RTC;
52 hrtc.Init.AsynchPrediv = RTC_AUTO_1_SECOND;
53 hrtc.Init.OutPut = RTC_OUTPUTSOURCE_ALARM;
54 if (HAL_RTC_Init(&hrtc) != HAL_OK)
55 {
56     Error_Handler();
57 }

```



- 在 RTC 相关函数中，除了HAL_RTC_SetTime（用于设置时间），还有一个名为 HAL_RTC_SetDate 的函数。由此可推断，HAL_RTC_SetTime 仅处理时间相关的数据，即时、分、秒，而不涉及年、月、日。
- 查看 HAL_RTC_SetTime 函数的具体实现，可以验证上述推测：函数内部将时、分、秒统一转换为以“秒”为单位的总秒数，然后调用了 RTC_WriteTimeCounter 函数。顾名思义，该函数用于将时间信息写入 RTC 的计数器中。

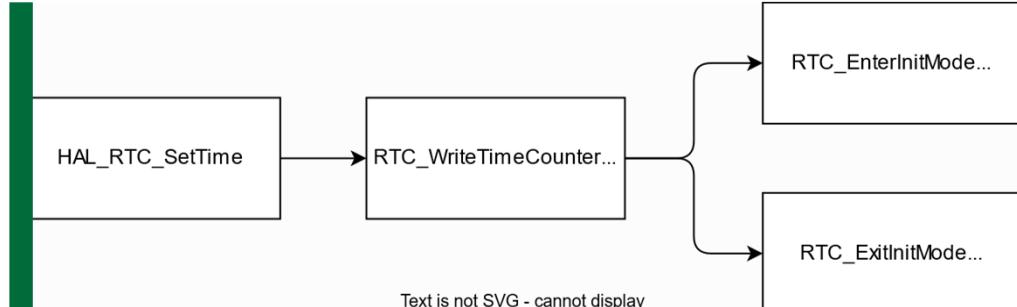


`RTC_WriteTimeCounter` 函数实现了 RTC 计数器的写入操作，其流程如下：

1. 调用 `RTC_EnterInitMode` 函数，使 RTC 进入初始化模式；
2. 向 RTC 计数器写入数据。

值得注意的是，虽然 RTC 计数器是 32 位的，但实际上它由两个 16 位的寄存器组成。因此，代码中分两次分别写入高 16 位和低 16 位的数据。

完成写入后，调用 `RTC_ExitInitMode` 函数退出初始化模式。



```

1613 /**
1614 * @brief Write the time counter in RTC_CNT registers.
1615 * @param hrtc pointer to a RTC_HandleTypeDef structure that contains
1616 *              the configuration information for RTC.
1617 * @param TimeCounter: Counter to write in RTC_CNT registers
1618 * @retval HAL status
1619 */
1620 static HAL_StatusTypeDef RTC_WriteTimeCounter(RTC_HandleTypeDef *hrtc, uint32_t TimeCounter)
1621 {
1622     HAL_StatusTypeDef status = HAL_OK;
1623
1624     /* Set Initialization mode */
1625     if (RTC_EnterInitMode(hrtc) != HAL_OK)
1626     {
1627         status = HAL_ERROR;
1628     }
1629     else
1630     {
1631         /* Set RTC COUNTER MSB word */
1632         WRITE_REG(hrtc->Instance->CNTH, (TimeCounter >> 16U));
1633         /* Set RTC COUNTER LSB word */
1634         WRITE_REG(hrtc->Instance->CNTL, (TimeCounter & RTC_CNTL_RTC_CNT));
1635
1636         /* Wait for synchro */
1637         if (RTC_ExitInitMode(hrtc) != HAL_OK)
1638         {
1639             status = HAL_ERROR;
1640         }
1641     }
1642
1643     return status;
1644 }
  
```



为什么要进入和退出初始模式？

根据 STM32 提供的中文参考手册，为了对 RTC 的计数器进行写操作，必须设置 **RTC_CRL 寄存器中的 CNF 位**。此外，为确保前一次写操作完成，也可以通过查询 RTC_CRL 中的 RTOFF 位来判断是否可以继续操作。对应的步骤如下：

- 第一步：进入初始化模式（设置 CNF 位）；
- 第二步：完成计数器写入；
- 第三步：退出初始化模式（清除 CNF 位）；
- 每一步都必须等待 RTOFF 位变为 1，表示上一次操作已完成。

实时时钟(RTC)

16.3.4 配置RTC寄存器

STM32F10xxx参考手册

必须设置 RTC_CRL 寄存器中的 CNF 位，使 RTC 进入配置模式后，才能写入 RTC_PRL、RTC_CNT、RTC_ALR 寄存器。

另外，对 RTC 任何寄存器的写操作，都必须在前一次写操作结束后进行。可以通过查询 RTC_CR 寄存器中的 RTOFF 状态位，判断 RTC 寄存器是否处于更新中。仅当 RTOFF 状态位是'1'时，才可以写入 RTC 寄存器。

配置过程：

1. 查询 RTOFF 位，直到 RTOFF 的值变为'1'
2. 置 CNF 值为 1，进入配置模式
3. 对一个或多个 RTC 寄存器进行写操作
4. 清除 CNF 标志位，退出配置模式
5. 查询 RTOFF，直至 RTOFF 位变为'1'以确认写操作已经完成。

仅当 CNF 标志位被清除时，写操作才能进行，这个过程至少需要 3 个 RTCCLK 周期。



配置过程：

1. 查询RTOFF位，直到RTOFF的值变为'1'
2. 置CNF值为1，进入配置模式
3. 对一个或多个RTC寄存器进行写操作
4. 清除CNF标志位，退出配置模式
5. 查询RTOFF，直至RTOFF位变为'1'以确认写操作已经完成。

仅当CNF标志位被清除时，写操作才能进行，这个过程至少需要3个RTCCLK周期。

```
static HAL_StatusTypeDef RTC_ExitInitMode(RTC_HandleTypeDef *hrtc)
{
    uint32_t tickstart = 0U;

    /* Disable the write protection for RTC registers */
    __HAL_RTC_WRITEPROTECTION_ENABLE(hrtc);

    tickstart = HAL_GetTick();
    /* Wait till RTC is in INIT state and if Time out is reached exit */
    while ((hrtc->Instance->CRL & RTC_CRL_RTOFF) == (uint32_t)RESET)
    {
        if ((HAL_GetTick() - tickstart) > RTC_TIMEOUT_VALUE)
        {
            return HAL_TIMEOUT;
        }
    }

    return HAL_OK;
}

/** @brief Enable the write protection for RTC registers.
 * @param __HANDLE__: specifies the RTC handle.
 * @retval None
 */
#define __HAL_RTC_WRITEPROTECTION_ENABLE(__HANDLE__)

```

类似地，RTC_ExitInitMode
函数会：

1. 清除 CNF 位；
2. 重新开启写保护；
3. 等待 RTOFF 位置为 1。

这两个函数通过 HAL 库的宏定义，实现了对 RTC 写保护机制的控制。

```
16 static HAL_StatusTypeDef RTC_EnterInitMode(RTC_HandleTypeDef *hrtc)
17 {
18     uint32_t tickstart = 0U;
19
20     tickstart = HAL_GetTick(); // 获取当前系统tick值
21     /* 循环等待RTOFF标志置位(表示可操作寄存器) */
22     while (((hrtc->Instance->CRL & RTC_CRL_RTOFF) == (uint32_t)RESET)
23     {
24         /* 超时检测(RTC_TIMEOUT_VALUE需在头文件中定义) */
25         if ((HAL_GetTick() - tickstart) > RTC_TIMEOUT_VALUE)
26         {
27             return HAL_TIMEOUT; // 超时返回错误
28         }
29     }
30
31     /* 解除RTC寄存器写保护(使能寄存器修改) */
32     __HAL_RTC_WRITEPROTECTION_DISABLE(hrtc);
33
34     return HAL_OK; // 成功进入初始化模式
35 }

348 /**
349  * @brief Disable the write protection for RTC registers.
350  * @param __HANDLE__: specifies the RTC handle.
351  * @retval None
352  */
353 #define __HAL_RTC_WRITEPROTECTION_DISABLE(__HANDLE__)

```

打开RTC_EnterInitMode
函数源码，可以发现：

1. 首先等待 RTOFF 位置为 1；
2. 然后通过宏定义关闭写保护；
3. 最后将 CNF 位置 1。

```
1728 static HAL_StatusTypeDef RTC_ExitInitMode(RTC_HandleTypeDef *hrtc)
1729 {
1730     uint32_t tickstart = 0U;
1731
1732     /* Disable the write protection for RTC registers */
1733     __HAL_RTC_WRITEPROTECTION_ENABLE(hrtc);
1734
1735     tickstart = HAL_GetTick();
1736     /* wait till RTC is in INIT state and if Time out is reached exit */
1737     while ((hrtc->Instance->CRL & RTC_CRL_RTOFF) == (uint32_t)RESET)
1738     {
1739         if ((HAL_GetTick() - tickstart) > RTC_TIMEOUT_VALUE)
1740         {
1741             return HAL_TIMEOUT;
1742         }
1743     }
1744
1745     return HAL_OK;
1746 }
```



关键步骤

如果我们需要在自定义的 RTC 库中实现设置时间的功能，关键步骤如下：

1. 将年月日、时分秒等时间信息转换为 Unix 时间戳（即从 1970 年 1 月 1 日起的总秒数）；
2. 调用 RTC 写计数器函数，将时间戳写入 RTC 计数器。

复制函数

此处的问题在于 RTC_WriteTimeCounter 及其相关的初始化 在头文件中加入以下引用：

控制函数均为 static 函数，无法在外 部文件中直接调用。因此，我们将以下函数**复制**到自定义库中：

- RTC_EnterInitMode
- RTC_ExitInitMode
- RTC_WriteTimeCounter

此外，为了后续可以读取 RTC 当前时间，也一并复制 RTC_ReadTimeCounter 函数。

头文件处理

#include "stm32f1xx_hal.h" // HAL 操作句柄定义
#include "stm32f1xx_hal_rtc.h" // RTC 相关操作函数和句柄定义
尤其是 hrtc (RTC 的句柄结构体) 在库函数中频繁出现，必须确保其类型已正确定义。



```

141 /**
142 * @brief 设置RTC时间 (用户接口)
143 * @param time 指向tm结构体的指针
144 * @retval HAL状态值
145 *
146 * 转换流程:
147 * 1. 将日历时间转换为Unix时间戳
148 * 2. 调用内部函数写入计数器
149 */
150 HAL_StatusTypeDef KK_RTC_SetTime(struct tm *time)
151 {
152     /* 将tm结构转换为Unix时间戳 */
153     uint32_t unixTime = mktime(time);
154     /* 写入RTC计数器 */
155     return RTC_WriteTimeCounter(&hrtc, unixTime);
156 }

```

KK_RTC_SetTime 的实现

KK_RTC_SetTime 函数签名

- 返回值类型：HAL_StatusTypeDef，是 HAL 库中通用的函数执行状态类型，表示函数执行是否成功（例如 HAL_OK, HAL_ERROR, HAL_BUSY, HAL_TIMEOUT 等）。
- 参数一：struct tm *time，标准 C 库 <time.h> 中定义的结构体，用于表示时间信息。

时间转换逻辑

函数的主要逻辑是将传入的 struct tm 类型的时间信息，转换为 Unix 时间戳，然后将时间戳写入 RTC 计数器中。

标准 C 库中提供了 mktime() 函数，可直接将 struct tm 类型转换为 time_t 类型（Unix 时间戳）。因此，这一转换无需手动计算。

然后调用前文从 HAL 库中借鉴过来的计数器写入函数



```

158  /**
159  * @brief 获取RTC时间 (用户接口)
160  * @retval 指向静态tm结构体的指针
161  *
162  * 注意:
163  * - 返回的指针指向静态内存, 不可长期保存
164  * - 需确保系统时区配置正确
165  */
166 struct tm *KK_RTC_GetTime(void)
167 {
168     /* 读取计数器获取Unix时间戳 */
169     time_t unixTime = RTC_ReadTimeCounter(&hrtc);
170     /* 转换为日历时间 (Keil MDK使用localtime()) */
171     return localtime(&unixTime);
172 }

```

KK_RTC_GetTime 的实现

KK_RTC_GetTime 函数签名

该函数无输入参数, 仅接受 RTC 的句柄, 并返回一个指向 `struct tm` 类型的指针, 表示当前时间。

函数逻辑

函数逻辑如下:

1. 调用我们已有的 `RTC_ReadTimeCounter` 函数, 从 RTC 计数器中读取当前的 Unix 时间戳;
2. 使用 `localtime()` 或 `gmtime()` 函数将时间戳转换为 `struct tm` 结构体;
3. 返回结构体指针。

需要特别注意的是, `time_t` 在 C 标准中通常定义为 64 位整型, 而 RTC 内部使用的是 32 位的 `uint32_t`。为了避免因类型不一致而导致的数据精度或范围错误, 应确保读取时使用 `time_t` 类型进行接收。



基本思路

当前程序中，RTC 时间在每次启动时都会重新设置。这意味着，只要设备重新启动，RTC 时间就会回到默认设置的初始值。为了避免这一现象，我们可以引入“**初始化标志**”的概念：

- **首次设置时间时**，向 RTC 的备份寄存器中写入一个特定的标志值（如 0x2333）；
- **后续启动时**，先读取该寄存器，如果发现值已存在，则跳过时间设置过程；
- 否则，执行时间初始化，并写入该标志值。

备份寄存器使用

STM32F103ZET6 提供了 10 个备份寄存器（编号 RTC_BKP_DR1 到 RTC_BKP_DR10）。我们选用第一个寄存器（RTC_BKP_DR1）用于存储初始化标志。



```

174 /**
175 * @brief RTC初始化函数
176 *
177 * 初始化流程：
178 * 1. 检查备份寄存器中的初始化标志
179 * 2. 若未初始化则进行HAL层初始化
180 * 3. 设置默认时间 (2025-04-30 23:59:55)
181 * 4. 写入初始化标志到备份寄存器
182 */
183 void KK_RTC_Init(void)
184 {
185     /* 读取备份寄存器DR1的初始化标志 */
186     uint32_t initFlag = HAL_RTCEx_BKUPRead(&hrtc, RTC_BKP_DR1);
187
188     /* 已初始化则直接返回 */
189     if (initFlag == RTC_INIT_FLAG)
190         return;
191
192     /* 初始化RTC外设 (HAL层配置) */
193     if (HAL_RTC_Init(&hrtc) != HAL_OK)
194     {
195         Error_Handler(); // 错误处理 (需用户实现)
196     }
197
198     /* 设置默认时间结构体 */
199     struct tm time = {
200         .tm_year = 2025 - 1900, // 年份偏移 (1900基准)
201         .tm_mon = 4 - 1, // 月份范围0~11
202         .tm_mday = 30, // 日
203         .tm_hour = 23, // 时
204         .tm_min = 59, // 分
205         .tm_sec = 55, // 秒
206     };
207
208     /* 设置RTC时间 */
209     KK_RTC_SetTime(&time);
210
211     /* 写入初始化标志到备份寄存器DR1 */
212     HAL_RTCEx_BKUPWrite(&hrtc, RTC_BKP_DR1, RTC_INIT_FLAG);
213 }

```

KK_RTC_Init 的实现

KK_RTC_Init 使用

在 main.c 的主函数初始化过程中，应在调用其他 RTC 操作前先调用该初始化函数。

设备在第一次启动时会设置时间并写入标志值；而后续重启过程中，程序会检测到备份寄存器中已存在标志，从而跳过重复设置。

时间偏移

根据tm类型的定义，成员变量年要存储的是年份与1900年的差值；月份的取值是0~11，代表1~12月份，这里写实际的月份-1。



```

174 /**
175 * @brief RTC初始化函数
176 *
177 * 初始化流程:
178 * 1. 检查备份寄存器中的初始化标志
179 * 2. 若未初始化则进行HAL层初始化
180 * 3. 设置默认时间 (2025-04-30 23:59:55)
181 * 4. 写入初始化标志到备份寄存器
182 */
183 void KK_RTC_Init(void)
184 {
185     /* 读取备份寄存器DR1的初始化标志 */
186     uint32_t initFlag = HAL_RTCEx_BKUPRead(&hrtc, RTC_BKP_DR1);
187
188     /* 已初始化则直接返回 */
189     if (initFlag == RTC_INIT_FLAG)
190         return;
191
192     /* 初始化RTC外设 (HAL层配置) */
193     if (HAL_RTC_Init(&hrtc) != HAL_OK)
194     {
195         Error_Handler(); // 错误处理 (需用户实现)
196     }
197
198     /* 设置默认时间结构体 */
199     struct tm time = {
200         .tm_year = 2025 - 1900, // 年份偏移 (1900基准)
201         .tm_mon = 4 - 1, // 月份范围0-11
202         .tm_mday = 30, // 日
203         .tm_hour = 23, // 时
204         .tm_min = 59, // 分
205         .tm_sec = 55, // 秒
206     };
207
208     /* 设置RTC时间 */
209     KK_RTC_SetTime(&time);
210
211     /* 写入初始化标志到备份寄存器DR1 */
212     HAL_RTCEx_BKUPWrite(&hrtc, RTC_BKP_DR1, RTC_INIT_FLAG);
213 }

```

KK_RTC_Init 的实现

RTC 初始化过程导致时间暂停的问题

即使在备份寄存器机制生效之后，在连续按下复位键时，RTC 时间仍会出现滞后或暂停。其根本原因在于 HAL 库自动生成的 `MX_RTC_Init` 函数中包含了对 `HAL_RTC_Init()` 的调用，而该函数在初始化过程中会短暂停止 RTC 的运行，导致计时中断。

优化策略

只有在首次初始化时才执行 `HAL_RTC_Init()`，在已初始化的情况下应跳过其调用。这一点与前述“只初始化一次”的思路一致。

我们将 `KK_RTC_Init()` 中对备份寄存器的判断前置，并在 `MX_RTC_Init` 函数内包裹一段判断逻辑。

这种处理方式可有效避免 HAL 库中的初始化操作再次干扰 RTC 的计时过程。

```

29 /* RTC init function */
30 void MX_RTC_Init(void)
31 {
32
33     /* USER CODE BEGIN RTC_Init 0 */
34     //Clear "statement is unreachable"
35     if(1 == 1){
36         hrtc.Instance = RTC;
37         hrtc.Init.AsynchPrediv = RTC_AUTO_1_SECOND;
38         hrtc.Init.OutPut = RTC_OUTPUTSOURCE_ALARM;
39         //KK_RTC_Init();
40         return; //return 0000HAL_RTC_Init00
41     }
42     /* USER CODE END RTC_Init 0 */
43
44     /* USER CODE BEGIN RTC_Init 1 */
45
46     /* USER CODE END RTC_Init 1 */
47
48     /** Initialize RTC Only
49     */
50
51     hrtc.Instance = RTC;
52     hrtc.Init.AsynchPrediv = RTC_AUTO_1_SECOND;
53     hrtc.Init.OutPut = RTC_OUTPUTSOURCE_ALARM;
54     if (HAL_RTC_Init(&hrtc) != HAL_OK)
55     {
56         Error_Handler();
57     }
58     /* USER CODE BEGIN RTC_Init 2 */
59
60     /* USER CODE END RTC_Init 2 */
61
62 }

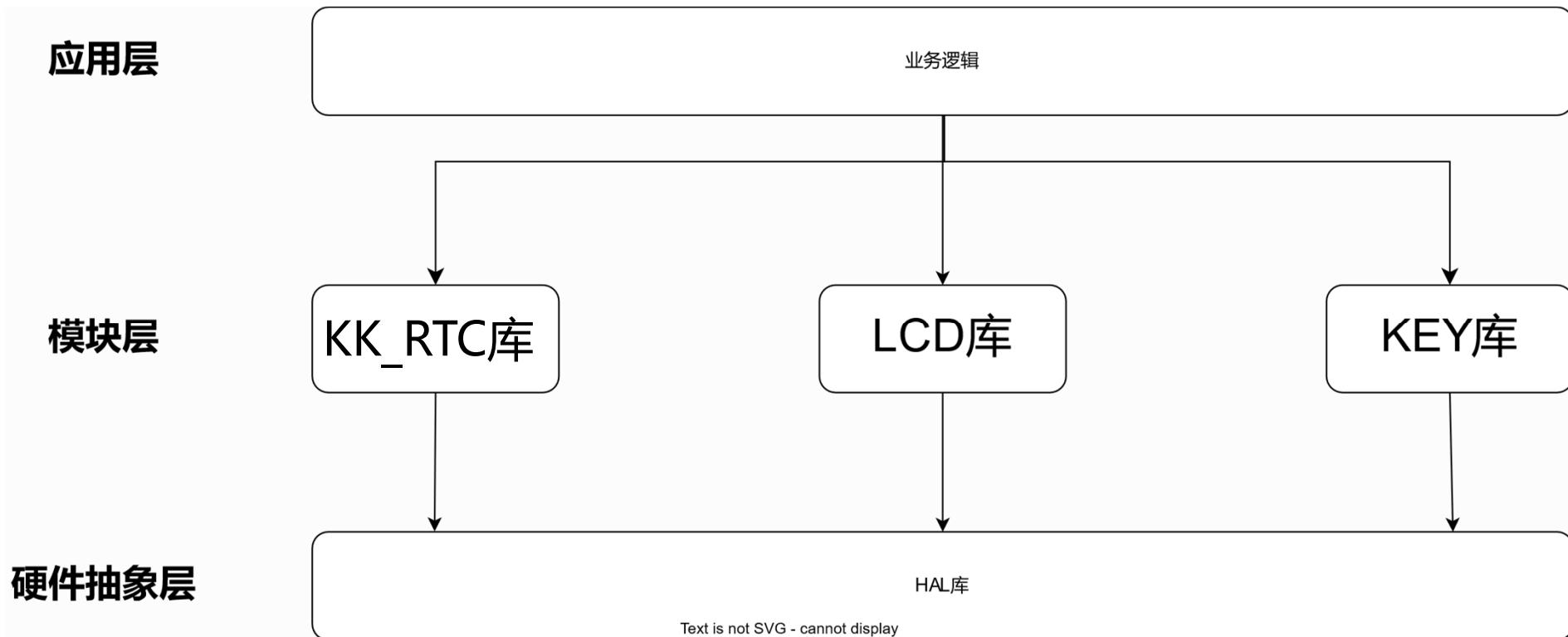
```

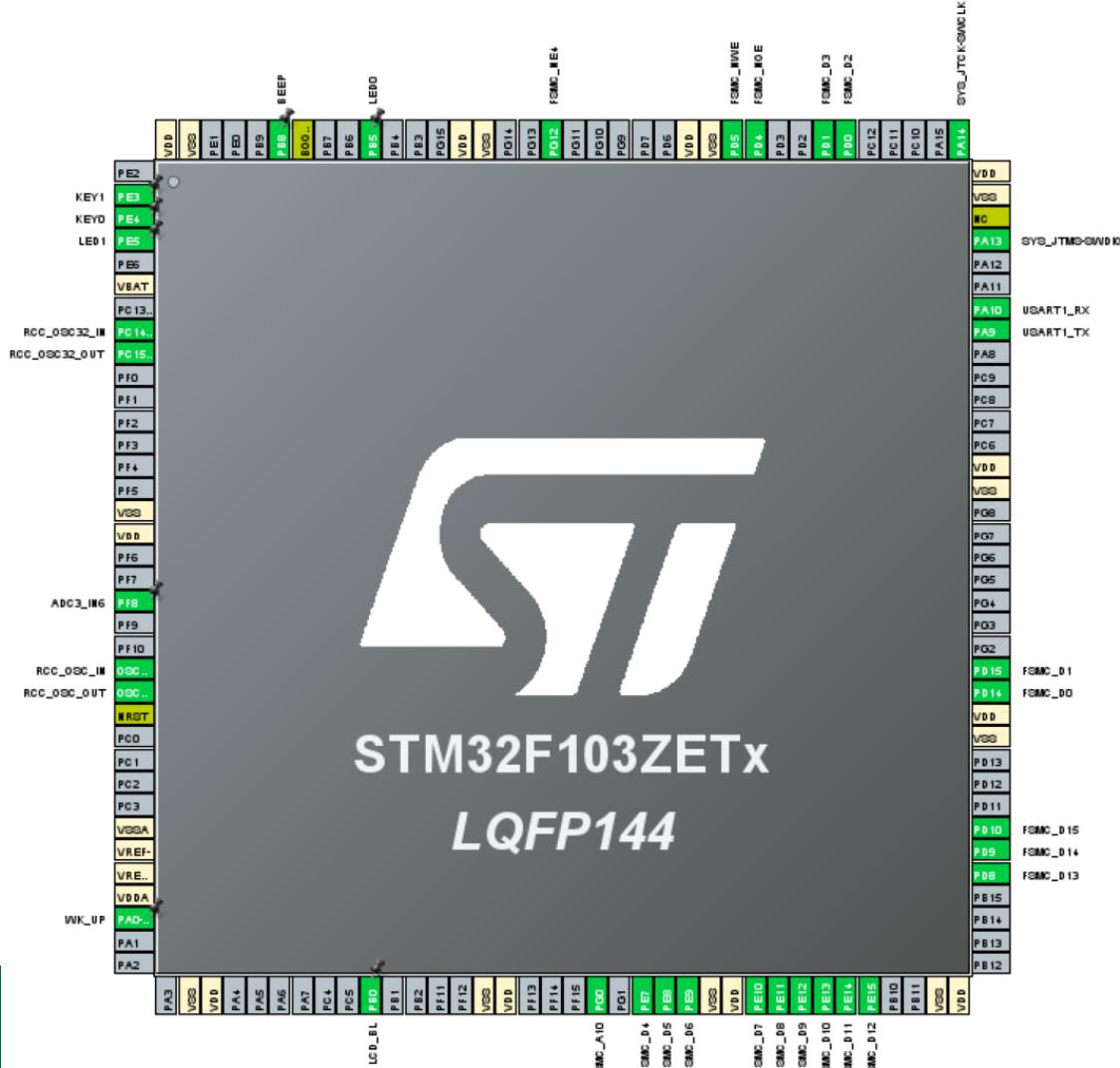
第二部分 ▷▷

系统架构

- 系统架构图
- CubeMX相关设置

德以明理 学以精工







RCC Mode and Configuration

Mode

High Speed Clock (HSE) Crystal/Ceramic Resonator

Low Speed Clock (LSE) Crystal/Ceramic Resonator

Master Clock Output

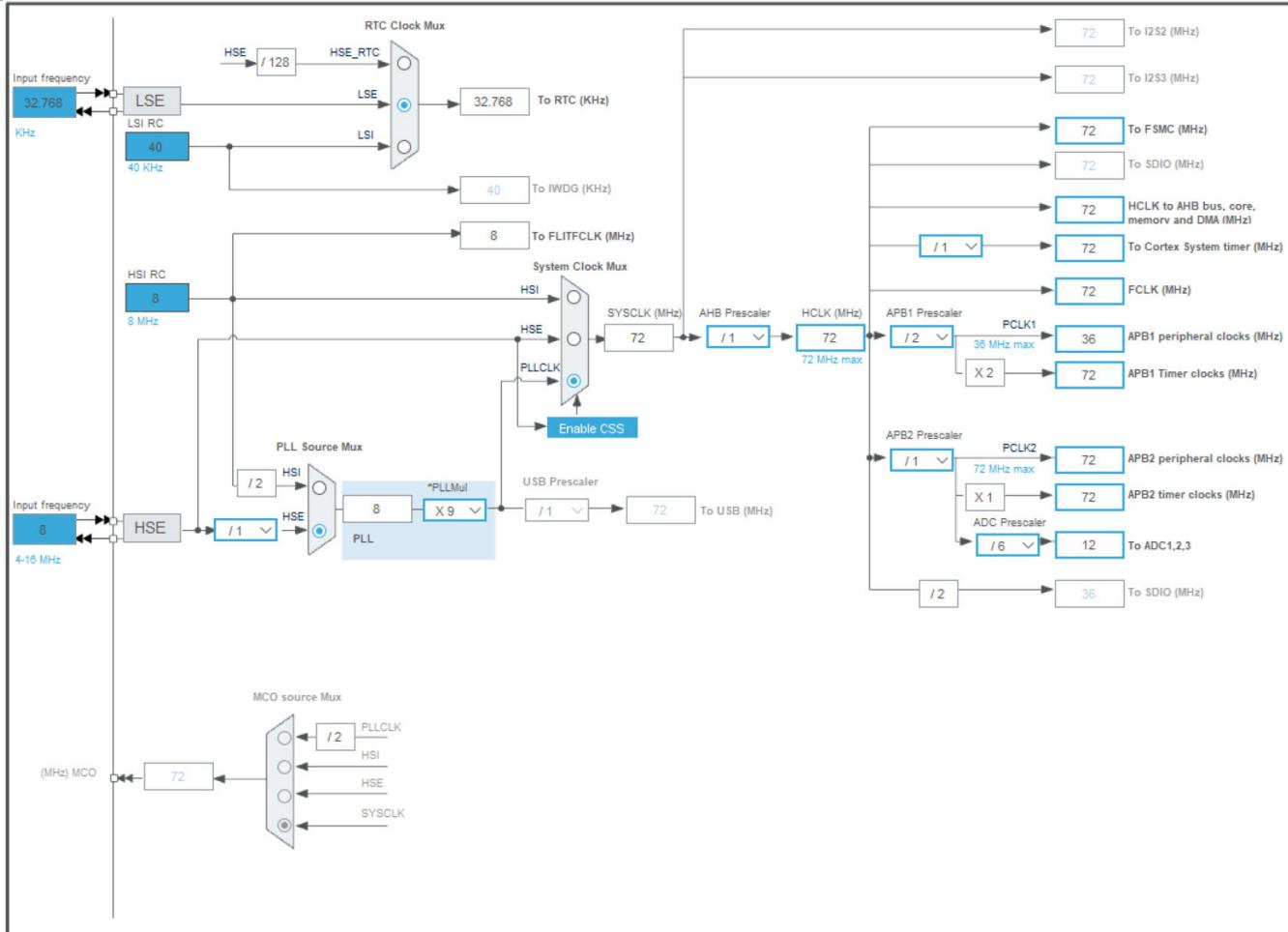
SYS Mode and Configuration

Mode

Debug Serial Wire

System Wake-Up

Timebase Source SysTick





The screenshot shows the GPIO configuration table in the CubeMX software. The table lists six pins: PA0-WKUP, PB0, PB5, PE3, PE4, and PE5. Each row contains information about the signal on the pin, the GPIO output type, the mode, pull-up/pull-down settings, maximum voltage, user label, and a modified status indicator.

Pin Name	Signal on Pin	GPIO output	GPIO mode	GPIO Pull-up	Maximum ...	User Label	Modified
PA0-WKUP	n/a	n/a	Input mode	Pull-down	n/a	WK_UP	<input checked="" type="checkbox"/>
PB0	n/a	Low	Output Pu...	No pull-up	Low	LCD_BL	<input checked="" type="checkbox"/>
PB5	n/a	Low	Output Pu...	No pull-up	Low	LED0	<input checked="" type="checkbox"/>
PE3	n/a	n/a	Input mode	Pull-up	n/a	KEY1	<input checked="" type="checkbox"/>
PE4	n/a	n/a	Input mode	Pull-up	n/a	KEY0	<input checked="" type="checkbox"/>
PE5	n/a	Low	Output Pu...	No pull-up	Low	LED1	<input checked="" type="checkbox"/>

CubeMX相关设置-USART



Parameter Settings		User Constants	
Configure the below parameters :			
<input type="text" value="Search (Ctrl+F)"/> <input type="button" value="..."/> <input type="button" value="..."/>			
Basic Parameters Baud Rate: 115200 Bits/s Word Length: 8 Bits (including Parity) Parity: None Stop Bits: 1			
Advanced Parameters Data Direction: Receive and Transmit Over Sampling: 16 Samples			

Parameter Settings		User Constants	
NVIC Interrupt Table		Enabled	Preemption Priority
DMA Settings		Sub Priority	
DMA1 channel4	global interrupt	<input checked="" type="checkbox"/>	0
DMA1 channel5	global interrupt	<input checked="" type="checkbox"/>	0
USART1	global interrupt	<input checked="" type="checkbox"/>	0

Parameter Settings		User Constants									
DMA Request Channel Direction Priority											
USART1_RX	DMA1 Channel 5	Peripheral To Memory	Low								
USART1_TX	DMA1 Channel 4	Memory To Peripheral	Low								
<input type="button" value="Add"/> <input type="button" value="Delete"/>											
DMA Request Settings <table border="1"> <tr> <td>Mode: Normal</td> <td>Increment Address: <input type="checkbox"/></td> <td>Peripheral</td> <td>Memory</td> </tr> <tr> <td>Data Width: Byte</td> <td>Byte</td> <td>Byte</td> <td></td> </tr> </table>				Mode: Normal	Increment Address: <input type="checkbox"/>	Peripheral	Memory	Data Width: Byte	Byte	Byte	
Mode: Normal	Increment Address: <input type="checkbox"/>	Peripheral	Memory								
Data Width: Byte	Byte	Byte									

Parameter Settings		User Constants	
Search Signals		<input type="checkbox"/> Show only ▾	
<input type="text"/> <input type="checkbox"/>			
Pin Name	Signal on Pin	GPIO output...	GPIO mode
PA9	USART1_TX	n/a	Alternate Fu... n/a
PA10	USART1_RX	n/a	Input mode No pull-up a... n/a
		User Label	



ADC3 Mode and Configuration

Mode	DMA Request	Channel	Direction	Priority
<input type="checkbox"/> IN6 <input checked="" type="checkbox"/> IN7 <input type="checkbox"/> IN8	ADC3	DMA2 Channel 5	Peripheral To Memory	Low

Configuration

Reset Configuration

NVIC Settings DMA Settings GPIO Settings
 Parameter Settings User Constants

Configure the below parameters :

Search (Ctrl+F) (i)

ADC_Settings

Data Alignment	Right alignment
Scan Conversion Mode	Disabled
Continuous Conversion Mode	Enabled
Discontinuous Conversion Mode	Disabled

ADC-Regular_ConversionMode

Enable Regular Conversions	Enable
Number Of Conversion	1
External Trigger Conversion Source	Regular Conversion launched by software

Rank

Channel	Channel 6
Sampling Time	71.5 Cycles

ADC_Injected_ConversionMode

Enable Injected Conversions	Disable
-----------------------------	---------

WatchDog

Enable Analog WatchDog Mode	<input type="checkbox"/>
-----------------------------	--------------------------



RTC Mode and Configuration

Mode

Activate Clock Source

Activate Calendar

RTC OUT

Tamper

Configuration

Reset Configuration

Parameter Settings User Constants NVIC Settings

Configure the below parameters :

Search (Ctrl+F)

Calendar Time Data Format BCD data format

General Auto Predivider Calculation Enabled

Asynchronous Predivider value Automatic Predivider Calculation Enabled

Output Alarm pulse signal on the TAMPER pin

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
RTC global interrupt	<input checked="" type="checkbox"/>	0	0
RTC alarm interrupt through EXTI line 17	<input checked="" type="checkbox"/>	0	0



FSMC Mode and Configuration

Mode	
<input checked="" type="checkbox"/> NOR Flash/PSRAM/SRAM/ROM/LCD 3	
<input checked="" type="checkbox"/> NOR Flash/PSRAM/SRAM/ROM/LCD 4	
Chip Select	NE4
Memory type	LCD Interface
Address	Disable
LCD Register Select	A10
Data	16 bits
Data/Address	Disable
Clock	Disable
<input type="checkbox"/> Address valid	
<input type="checkbox"/> Wait	Disable
<input type="checkbox"/> Byte enable	
> NAND Flash 1	

Configuration

Reset Configuration	
<input checked="" type="radio"/> NOR/PSRAM 4	<input type="radio"/> User Constants
<input type="radio"/> GPIO Settings	
Configure the below parameters :	
<input type="text"/> Search (Ctrl+F)	<input type="button"/>
<input checked="" type="checkbox"/> NOR/PSRAM control	
Memory type	LCD Interface
Bank	Bank 1 NOR/PSRAM 4
Write operation	Enabled
Extended mode	Enabled
<input checked="" type="checkbox"/> NOR/PSRAM timing	
Address setup time in HCLK clock cycles	6
Data setup time in HCLK clock cycles	26
Bus turn around time in HCLK clock cyc...	0
Access mode	A
<input checked="" type="checkbox"/> NOR/PSRAM timing for write accesses	
Extended address setup time	3
Extended data setup time	6
Extended bus turn around time	15
Extended access mode	A



Configuration

NVIC Code generation

Priority Group .. Sort by Preemption Priority and Sub Priority Sort by interrupts names

Search Show available interrupts Force DMA channels Interrupts

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
Non maskable interrupt	<input checked="" type="checkbox"/>	0	0
Hard fault interrupt	<input checked="" type="checkbox"/>	0	0
Memory management fault	<input checked="" type="checkbox"/>	0	0
Prefetch fault, memory access fault	<input checked="" type="checkbox"/>	0	0
Undefined instruction or illegal state	<input checked="" type="checkbox"/>	0	0
System service call via SWI instruction	<input checked="" type="checkbox"/>	0	0
Debug monitor	<input checked="" type="checkbox"/>	0	0
Pendable request for system service	<input checked="" type="checkbox"/>	0	0
Time base: System tick timer	<input checked="" type="checkbox"/>	15	0
PVD interrupt through EXTI line 16	<input type="checkbox"/>	0	0
RTC global interrupt	<input checked="" type="checkbox"/>	0	0
Flash global interrupt	<input type="checkbox"/>	0	0
RCC global interrupt	<input type="checkbox"/>	0	0
DMA1 channel4 global interrupt	<input checked="" type="checkbox"/>	0	0
DMA1 channel5 global interrupt	<input checked="" type="checkbox"/>	0	0
TIM2 global interrupt	<input checked="" type="checkbox"/>	3	0
TIM4 global interrupt	<input type="checkbox"/>	0	0
USART1 global interrupt	<input checked="" type="checkbox"/>	0	0
RTC alarm interrupt through EXTI line 17	<input checked="" type="checkbox"/>	0	0
ADC3 global interrupt	<input checked="" type="checkbox"/>	0	0
TIM7 global interrupt	<input type="checkbox"/>	0	0
DMA2 channel4 and channel5 global interrupts	<input checked="" type="checkbox"/>	0	0

Enabled Preemption Priority Sub Priority

时基系统滴答设为最低
避免与HAL_Delay()冲突

CubeMX相关设置-Project Manager



Project Settings

Project Name: LVGL_TP_mx_zd

Project Location: D:\STM32_Project\LVGL_TP\LVGL_TP_mx_zd

Application Structure: Advanced Do not generate the main()

Toolchain Folder Location: D:\STM32_Project\LVGL_TP\LVGL_TP_mx_zd\LVGL_TP_mx_zd

Toolchain / IDE: MDK-ARM Min Version: V5 Generate Under Root

Linker Settings

Minimum Heap Size: 0x200

Minimum Stack Size: 0x1000

Thread-safe Settings

Cortex-M3NS

Enable multi-threaded support

Thread-safe Locking Strategy: Default – Mapping suitable strategy depending on RTOS selection.

Mcu and Firmware Package

Mcu Reference: STM32F103ZETx

Firmware Package Name and Version: STM32Cube_FW_F1_V1.8.6 Use latest available version

Use Default Firmware Location

Firmware Relative Path: D:/STM32CMXpack/STM32Cube_FW_F1_V1.8.6

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

第三部分 ▷▷

硬件构成

- 硬件资源列表
- 硬件电路图
- TFT_LCD液晶显示
- 硬件测试

德以明理 学以精工

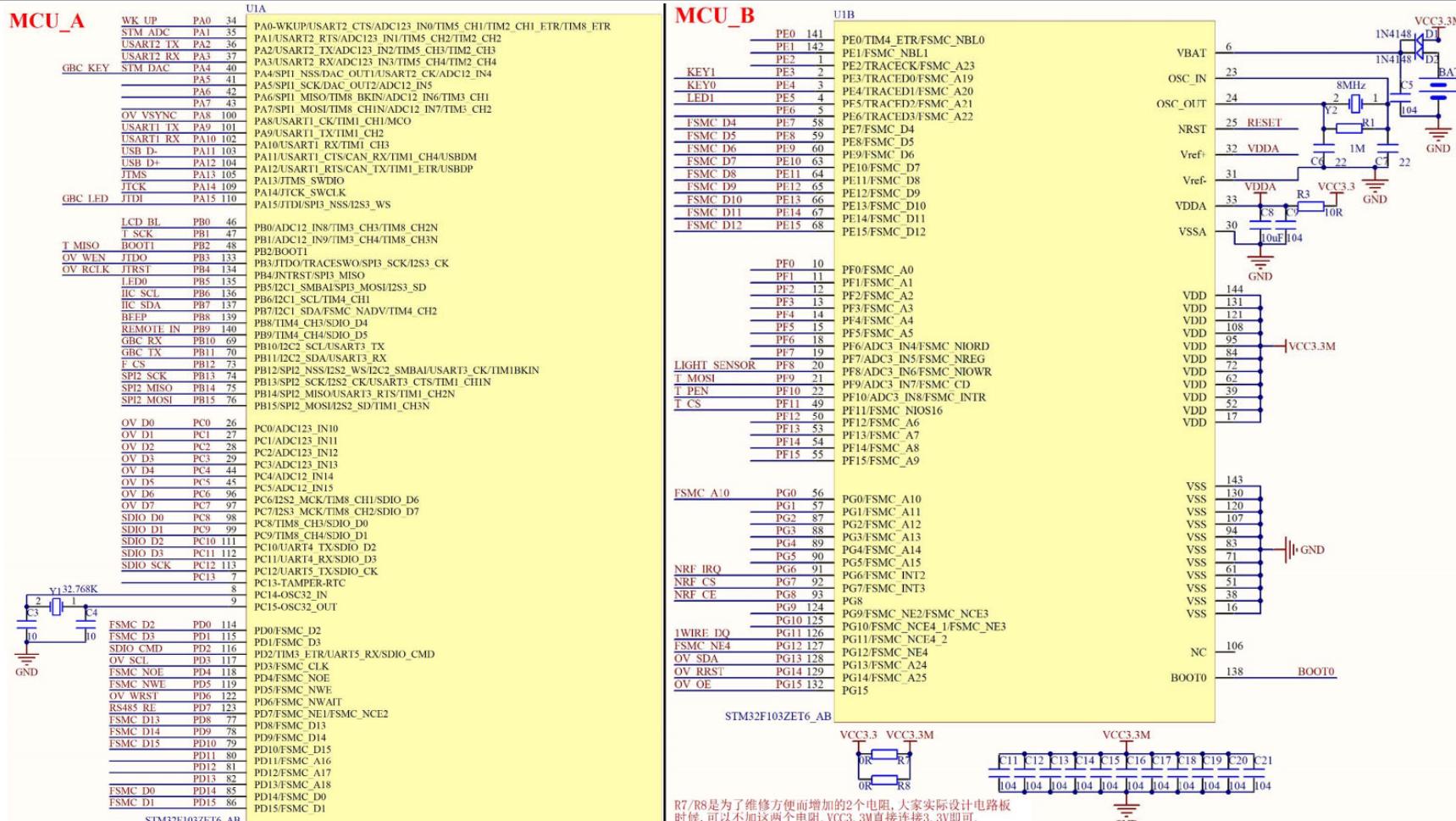


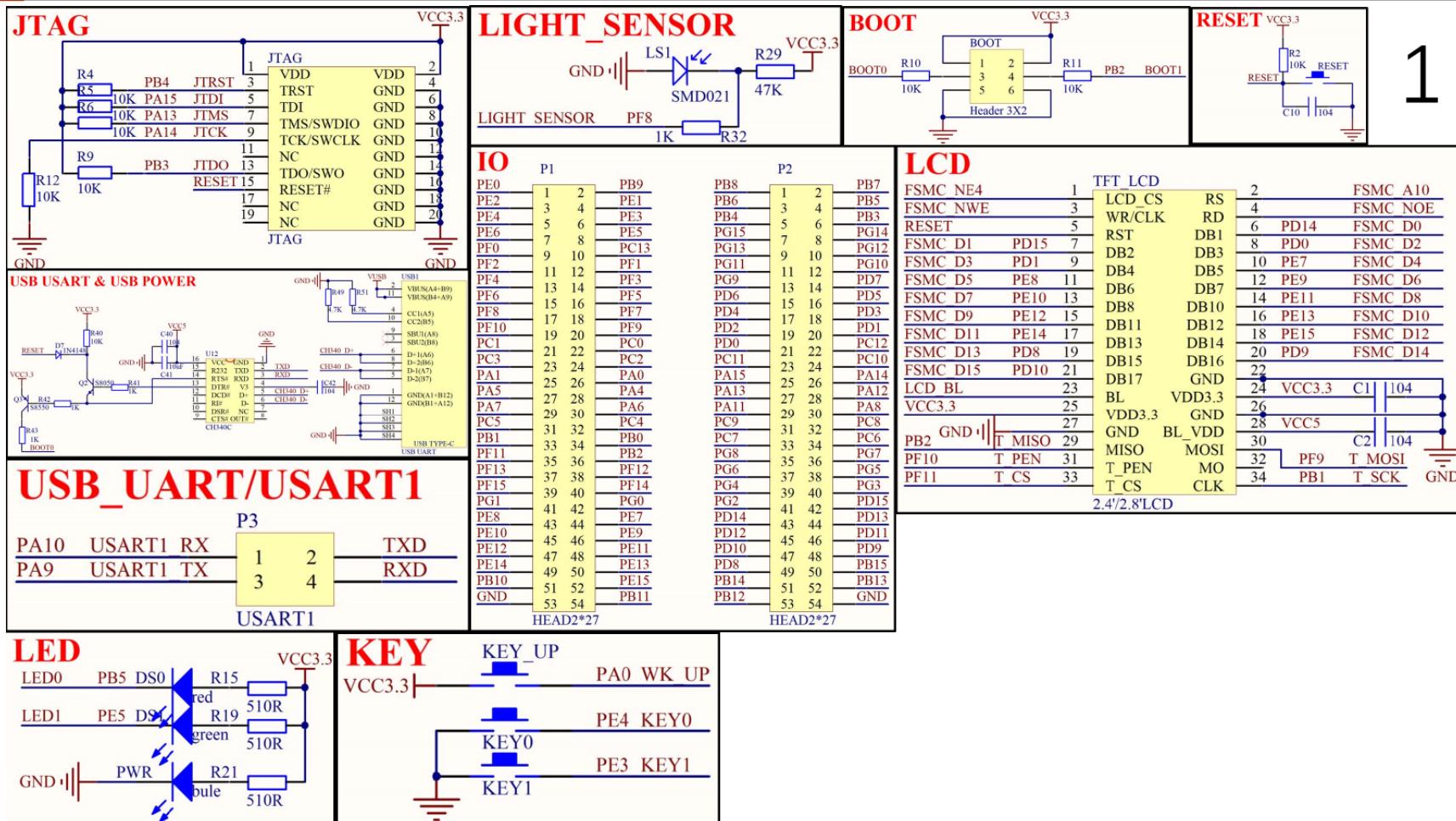
资源	数量	说明
CPU	1 个	STM32F103ZET6； FLASH: 512KB； SRAM: 64KB；
电源指示灯	1 个	蓝色
状态指示灯	2 个	红色 (DS0)； 绿色 (DS1)；
功能按键	3 个	KEY0、KEY1、KEY_UP (具备唤醒功能)
电源开关	1 个	控制整个板子供电
光敏传感器	1 个	用于感应环境光照强度
LCD 接口	1 个	支持正点原子 3.5 寸 TFTLCD 模块
USB 转串口	1 个	用于 USB 转 TTL 串口通信



资源	数量	说明
JTAG/SWD 调试口	1 个	用于仿真调试、下载代码等
后备电池接口	1 个	用于 RTC 后备电池
一键下载电路	1 个	方便使用串口下载代码

硬件电路图







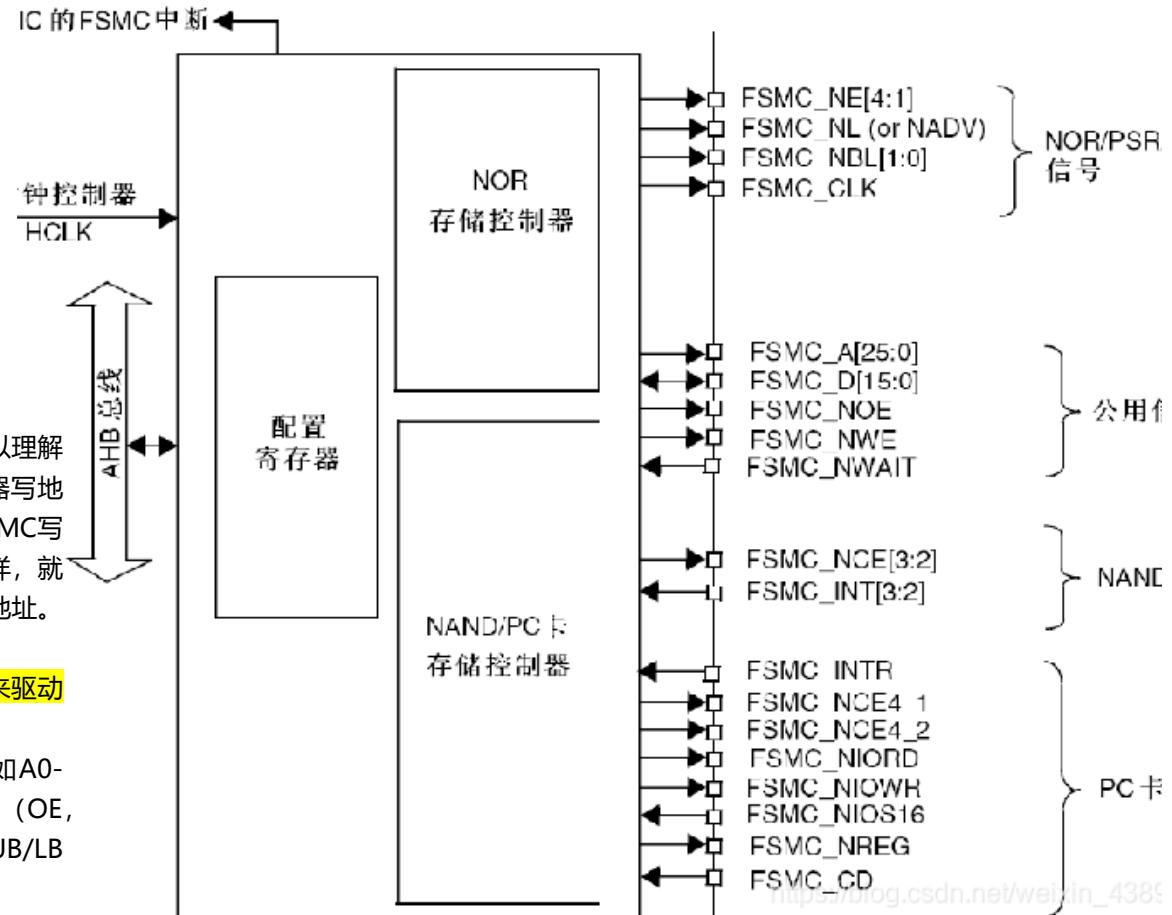
FSMC接口

FSMC，即灵活的静态存储控制器，能够与同步或异步存储器和16位PC存储器卡连接，STM32的FSMC接口支持包括SRAM、NAND FLASH、NOR FLASH和PSRAM等存储器。

TFTLCD通过RS信号来决定传送的数据是数据还是命令，本质上可以理解为一个地址信号，比如我们把RS接在A0上面，那么当FSMC控制器写地址0的时候，会使得A0变为0，对TFTLCD来说，就是写命令。而FSMC写地址1的时候，A0将会变为1，对TFTLCD来说，就是写数据了。这样，就把数据和命令区分开了，他们其实就是对应SRAM操作的两个连续地址。当然RS也可以接在其他地址线上，而这个板子是把RS接在A6上面。

注意：FSMC接口驱动LCD时，其实是将LCD当作一个外部的SRAM来驱动的，唯一不同就是TFTLCD有RS信号，但是没有地址信号

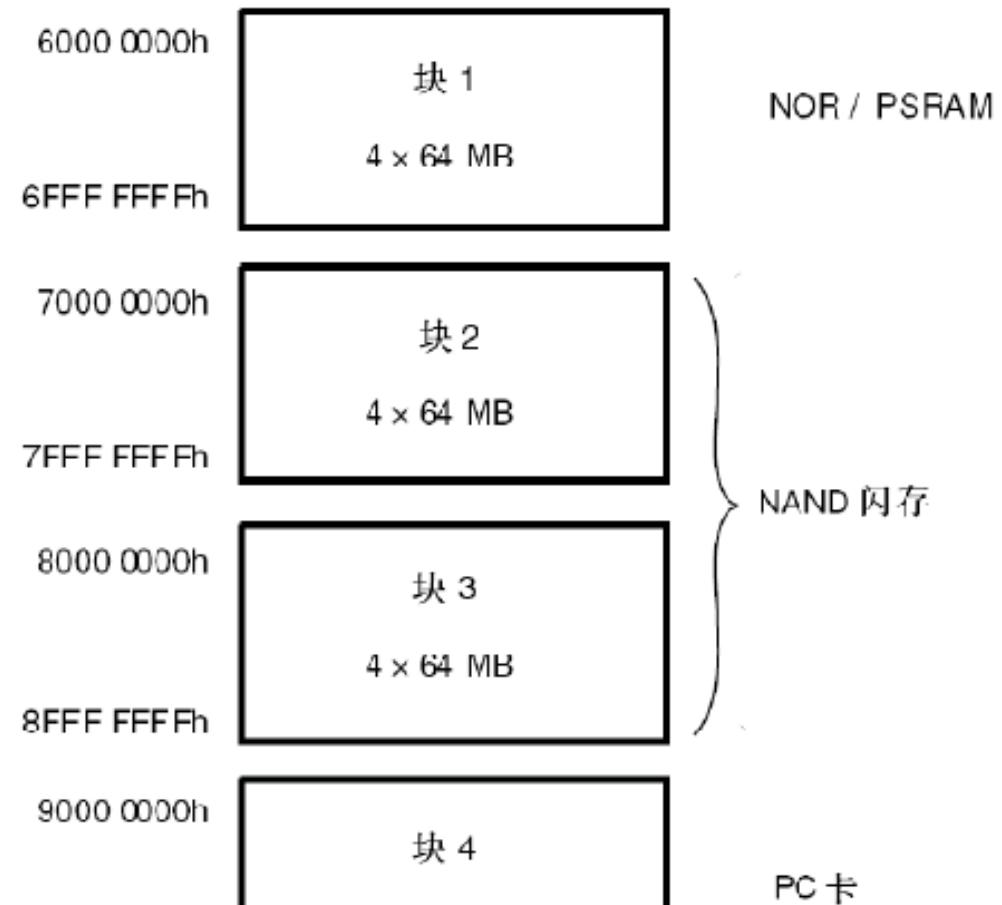
FSMC驱动外部SRAM时，外部SRAM的控制一般有：地址线（如A0-A25）、数据线（如D0-D15）、写信号（WE，即WR）、读信号（OE，即RD）、片选信号（CS），如果SRAM支持字节控制，那么还有UB/LB信号。





FSMC存储块

STM32的FSMC支持**8/16/32**位数据宽度，我们用到的LCD是16位宽度的，所以在设置时选择16位宽。FSMC的外部设备地址映像，STM32的FSMC将外部存储器划分为固定大小为256M字节的四个存储块（右图）。





FSMC地址

STM32的FSMC存储块1（Bank1）用于驱动 **NOR FLASH/SRAM/PSRAM**，被分为4个区，每个区管理64M字节空间，每个区都有独立的寄存器对所连接的存储器进行配置。Bank1的256M字节空间由28根地址线（HADDR[27:0]）寻址。这里HADDR，是内部**AHB**地址总线，其中，**HADDR[25:0]**来自外部存储器地址 **FSMC_A[25:0]**，而**HADDR[26:27]**对4个区进行寻址。

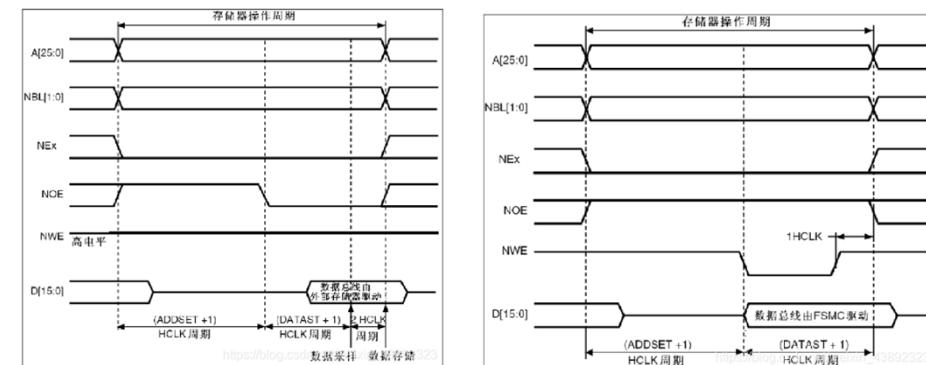
STM32的FSMC存储块1 支持的异步突发访问模式包括：模式1、模式A~D等多种时序模型，**驱动SRAM时一般使用模式1或者模式 A，这里使用模式A来驱动LCD（当SRAM用）**

Bank1 所选区	片选信号	地址范围	HADDR	
			[27:26]	[25:0]
第 1 区	FSMC_NE1	0X6000, 0000~63FF, FFFF	00	FSMC_A[25:0] https://blog.csdn.net/weixin_43892323
第 2 区	FSMC_NE2	0X6400, 0000~67FF, FFFF	01	
第 3 区	FSMC_NE3	0X6800, 0000~6BFF, FFFF	10	
第 4 区	FSMC_NE4	0X6C00, 0000~6FFF, FFFF	11	

当Bank1接的是16位宽度存储器的时候HADDR[25:1]->FSMC_A[24:0]

当Bank1接的是8位宽度存储器的时候HADDR[25:0]->FSMC_A[25:0]

不论外部接8位/16位宽设备，FSMC_A[0]永远接在外部设备地址A[0]

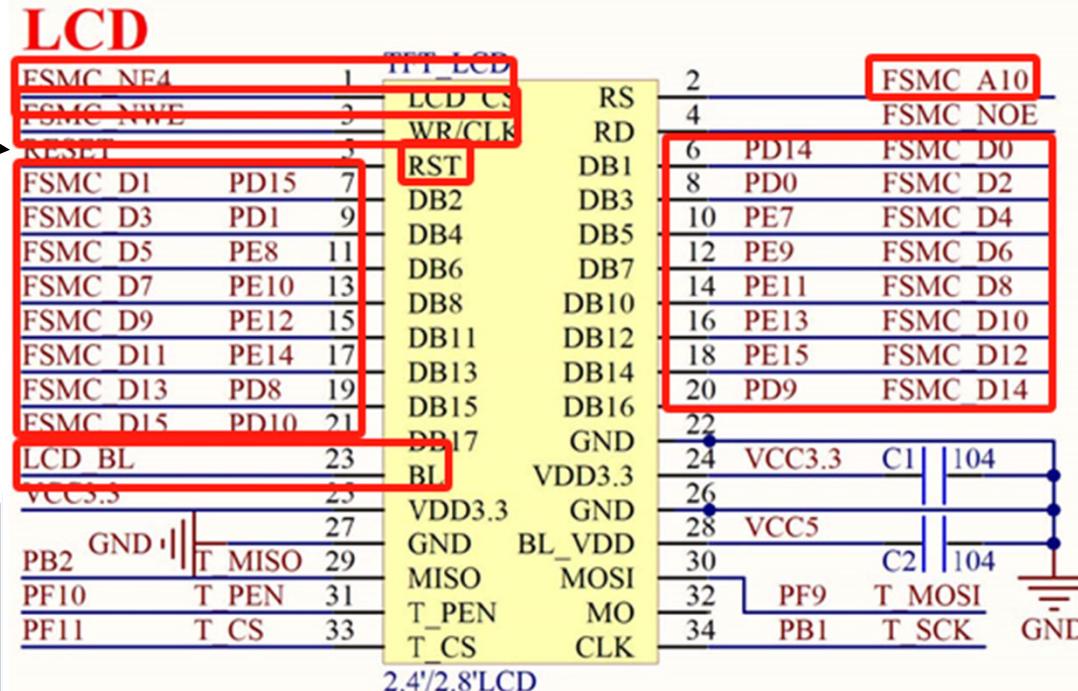




复位脚和单片机的复位是接到一起的，也就是整个系统的复位，在程序中并没有额外操作

背光电源脚

PB0 Configuration :	
GPIO output level	Low
GPIO mode	Output Push Pull
GPIO Pull-up/Pull-down	No pull-up and no pull-down
Maximum output speed	Low
User Label	LCD_BL



先使能背光电源脚



FSMC配置

- NOR Flash/PSRAM/SRAM/ROM/LCD 1**, 这里选择 STM32的FSMC存储块4 (Bank4)。
- Chip Select**, 选择Bank1的第四区, 是根据原理图的映射管脚进行选择的, 这里选择不同区对应的引脚是不同的。
- Memory Type**, 存储类型, 选择LCD接口。
- LCD Register Select**, 这里是选择RS脚, 也就是命令/数据选择位, 同样是根据原理图得知这里应该选择A10。
- Data**, 数据位, 很明显从原理图看出有16个数据引脚, 这里选择16bits.

NOR Flash/PSRAM/SRAM/ROM/LCD 4

Chip Select	NE4
Memory type	LCD Interface
Address	Disable
LCD Register Select	A10
Data	16 bits
Data/Address	Disable
Clock	Disable
<input type="checkbox"/> Address valid	
Wait	Disable
<input type="checkbox"/> Rote enable	



- 地址建立的时钟周期
- 数据建立的时钟周期
- 总线转阶段持续时间
- 扩展地址建立时间
- 扩展数据建立时间
- 扩展总线建立时间

▼ NOR/PSRAM control		
Memory type	LCD Interface	
Bank	Bank 1 NOR/PSRAM 4	
Write operation	Enabled	
Extended mode	Enabled	
▼ NOR/PSRAM timing		
Address setup time in HCLK clock cycles	6	
Data setup time in HCLK clock cycles	26	
Bus turn around time in HCLK clock cycles	0	
Access mode	A	
▼ NOR/PSRAM timing for write accesses		
Extended address setup time	3	
Extended data setup time	6	
Extended bus turn around time	15	
Extended access mode	A	

参考手册计算得出



LCD驱动程序

- 移植正点原子驱动程序
- 删去delay_ms改用HAL_Delay()
- 重新梳理头文件顺序

```
#include "lcd.h"
#include "stdlib.h"
#include "main.h"
#include "font.h"

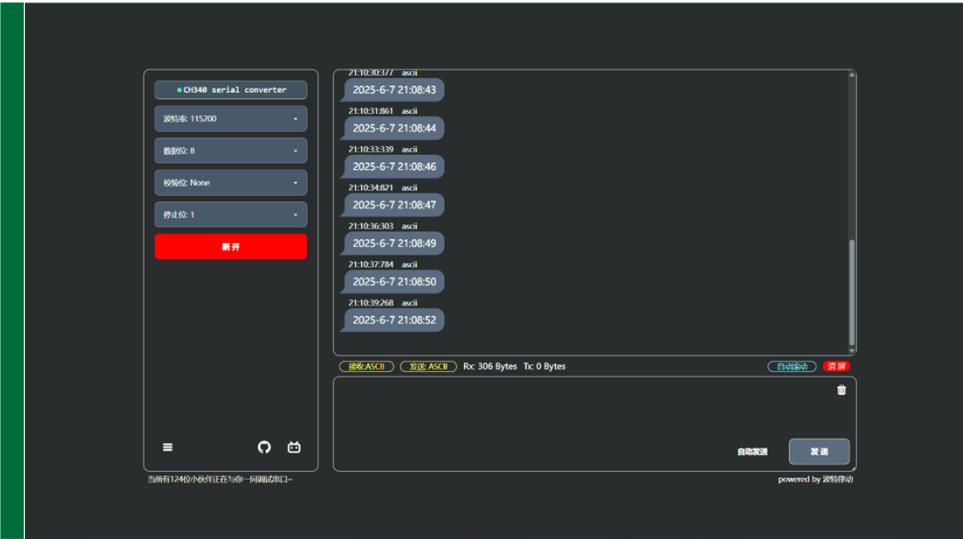
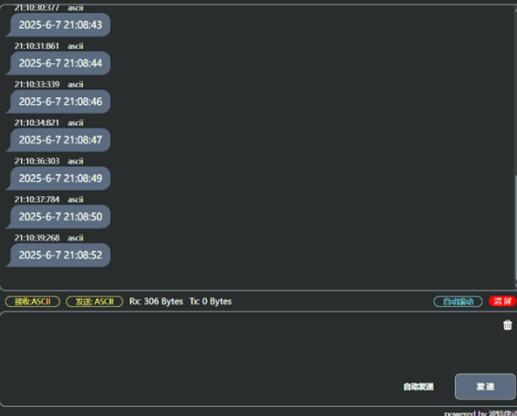
/* lcd_ex.c存放各个LCD驱动IC的寄存器初始化部分代码,以简化lcd.c,该.c文件
 * 不直接加入到工程里面,只有lcd.c会用到,所以通过include的形式添加.(不要在
 * 其他文件再包含该.c文件!!否则会报错!)
 */
#include "lcd_ex.c"

SRAM_HandleTypeDef g_sram_handle; /* SRAM句柄(用于控制LCD) */

/* LCD的画笔颜色和背景色 */
uint32_t g_point_color = 0xF800; /* 画笔颜色 */
uint32_t g_back_color = 0xFFFF; /* 背景色 */

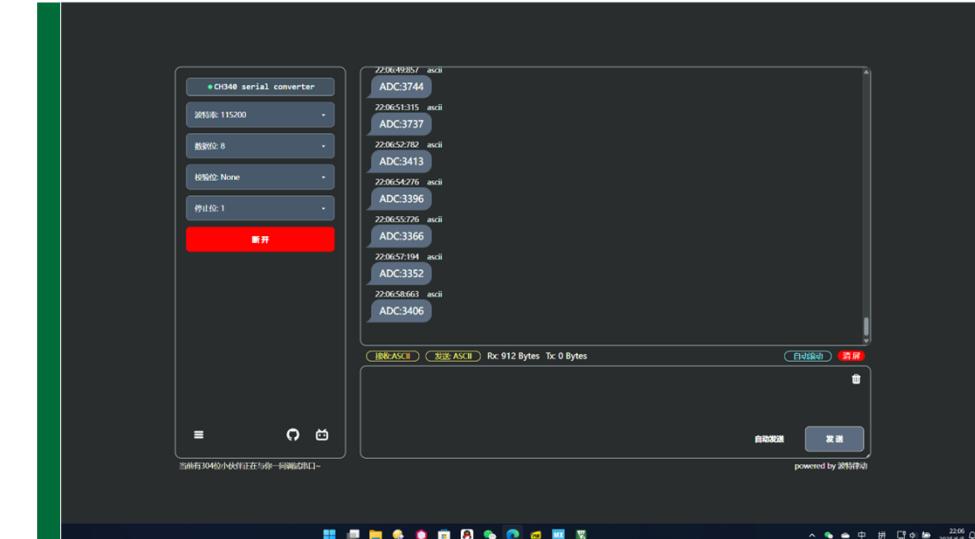
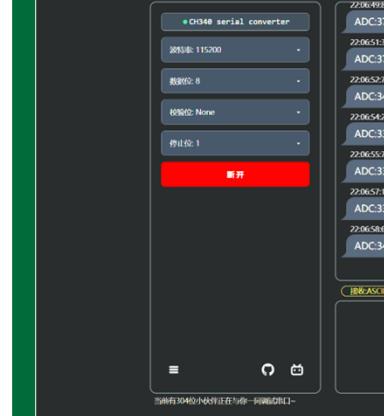
/* 管理LCD重要参数 */
_lcd_dev lcddev;

/**
 * @brief      LCD写数据
 * @param      data: 要写入的数据
 * @retval     无
 */
void lcd_wr_data(volatile uint16_t data)
{
    data = data; /* 使用-O2优化的时候,必须插入的延时 */
    LCD->LCD_RAM = data;
}
```



USART串口测试RTC

德以明理 学以精工



USART串口测试ADC

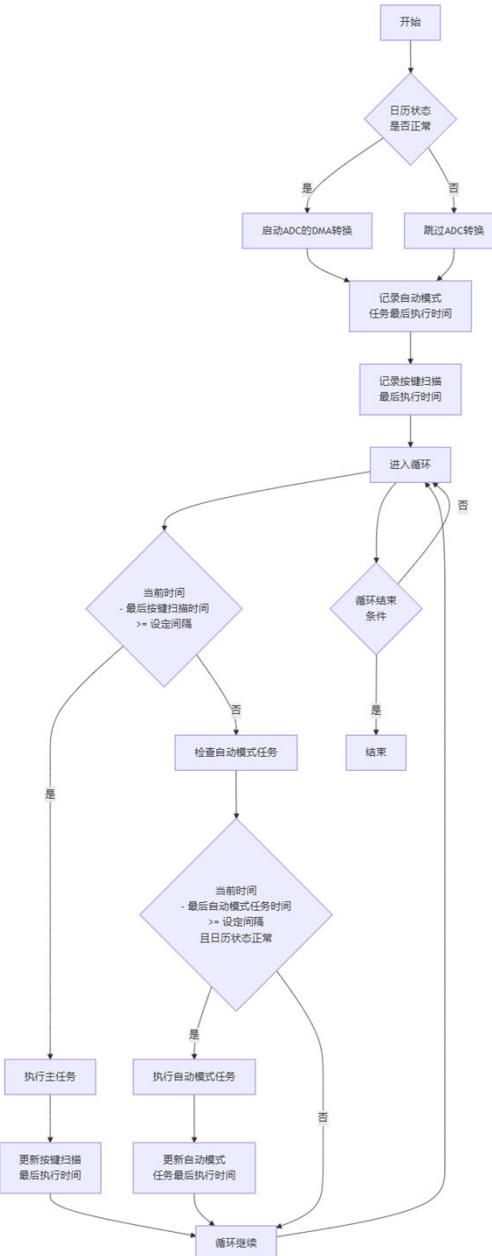
第四部分 ▷▷

软件代码

- 软件流程
- 时间片与任务驱动
- 按键状态机
- 自动深色/浅色变化
- 软件测试

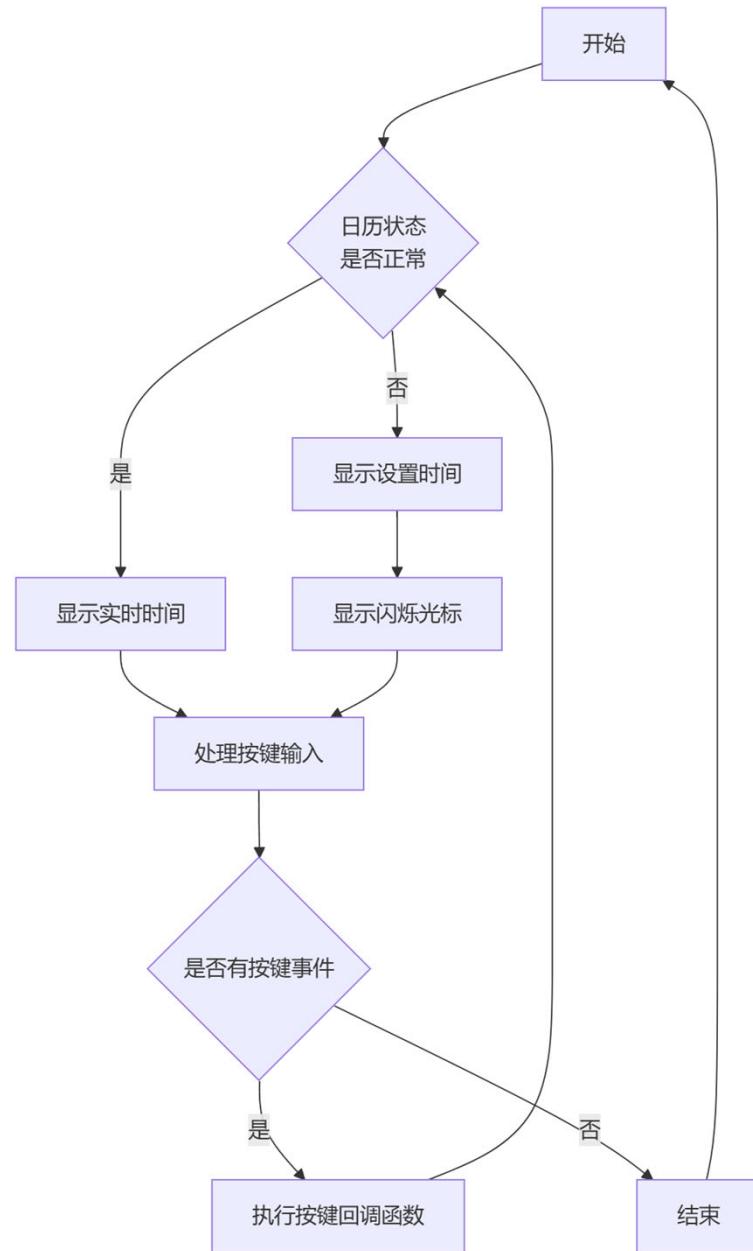
德以明理 学以精工

main.c流程图



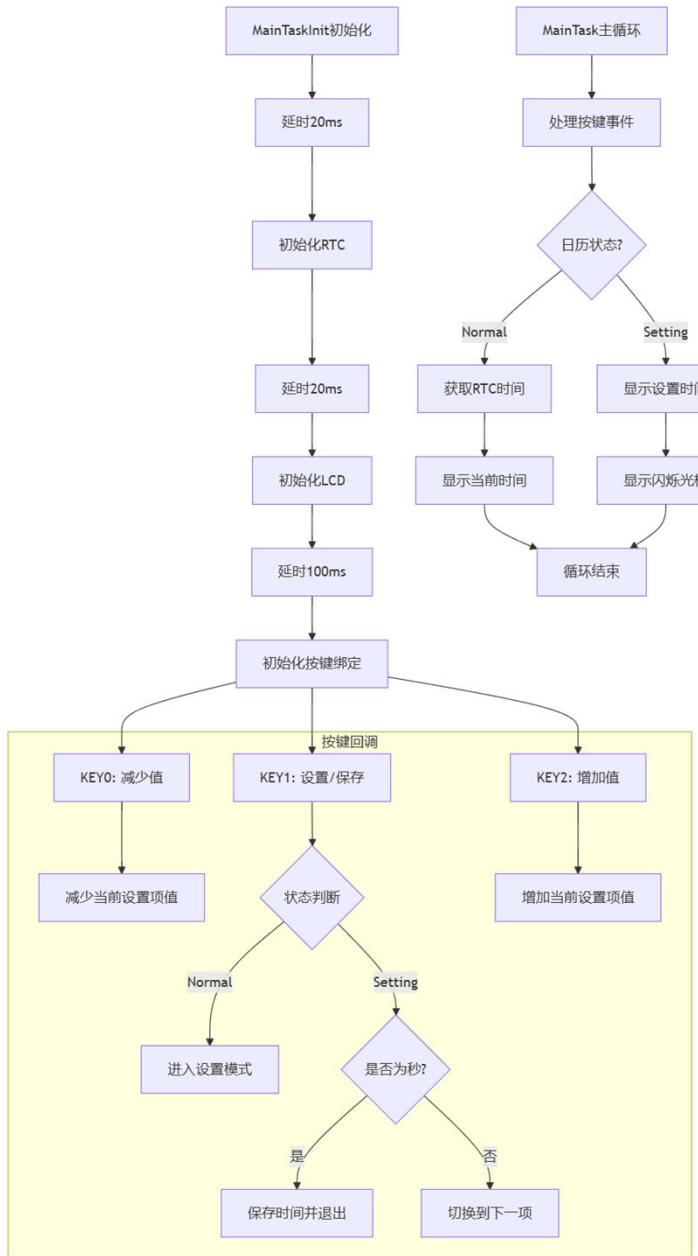


task_main.c流程图

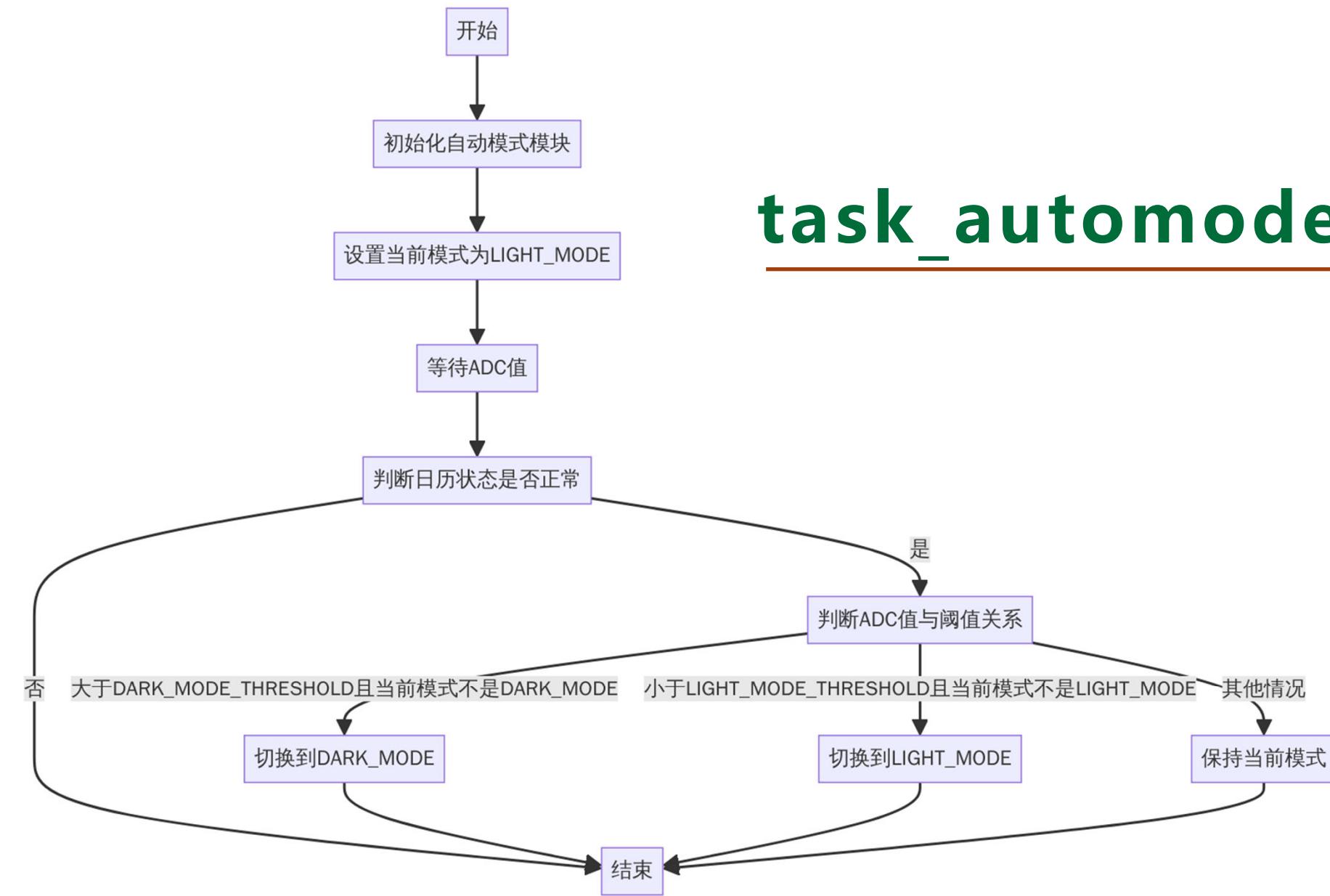




task main.c流程图

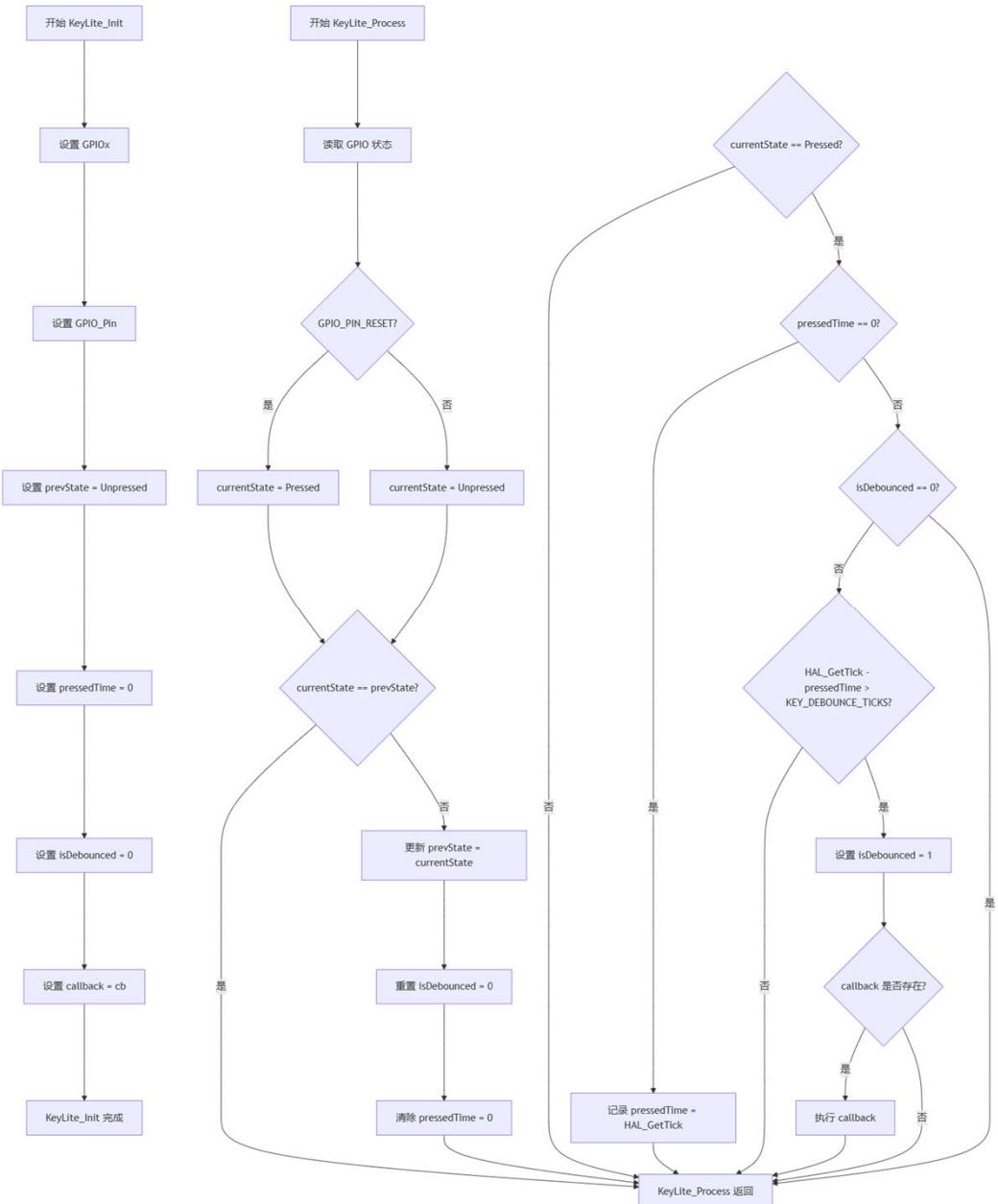


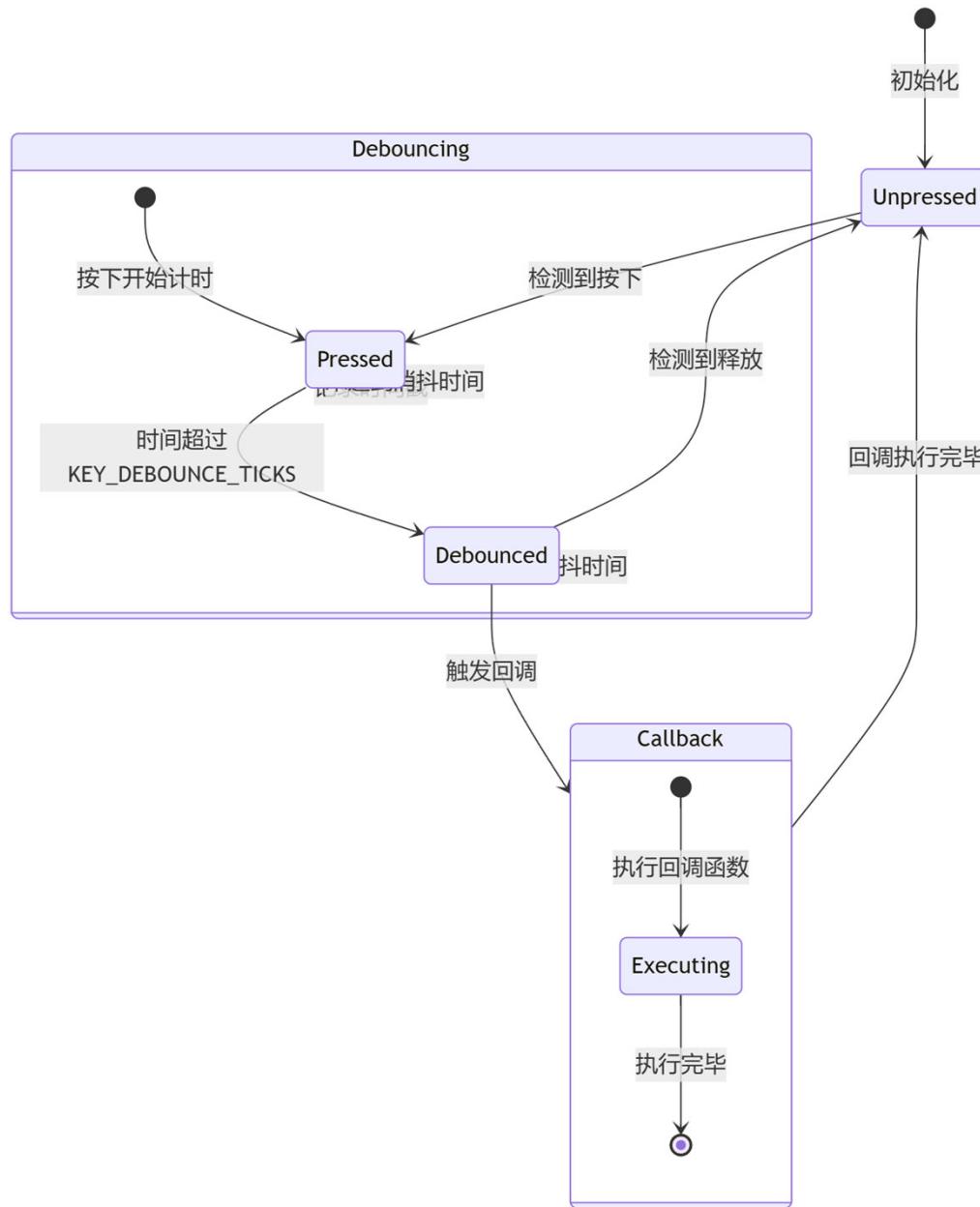
task_automode.c流程图



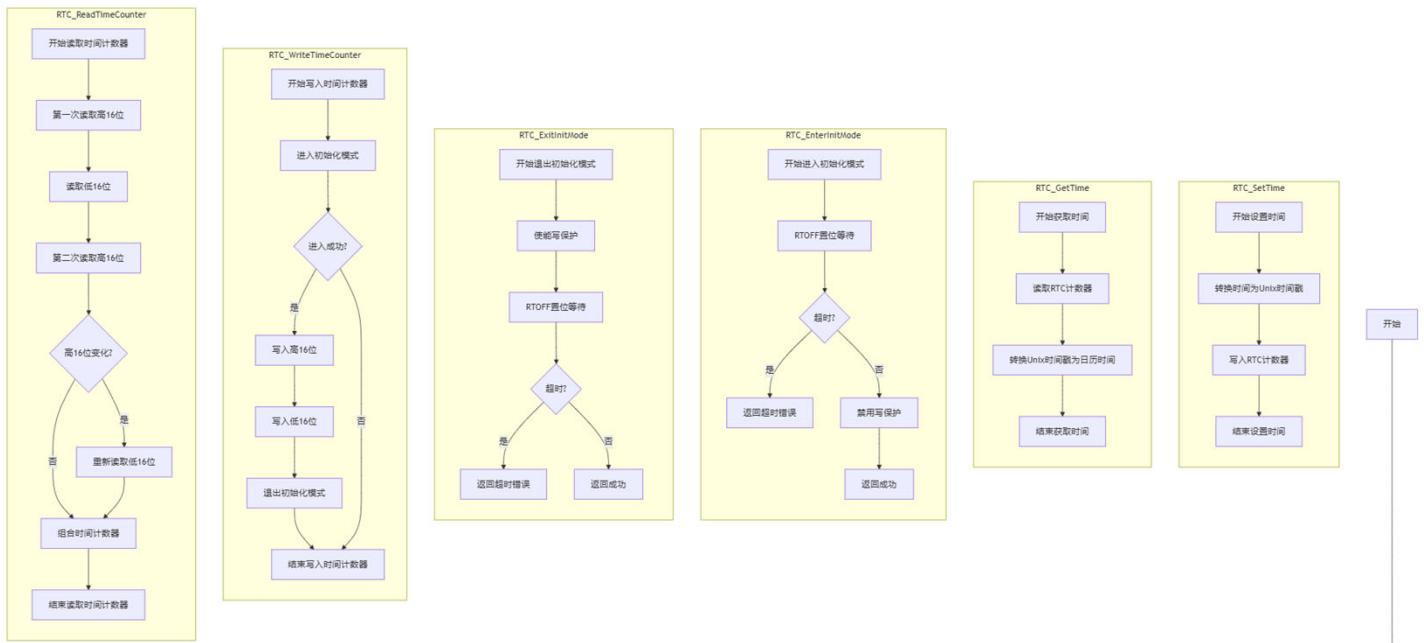


KEY.c流程图

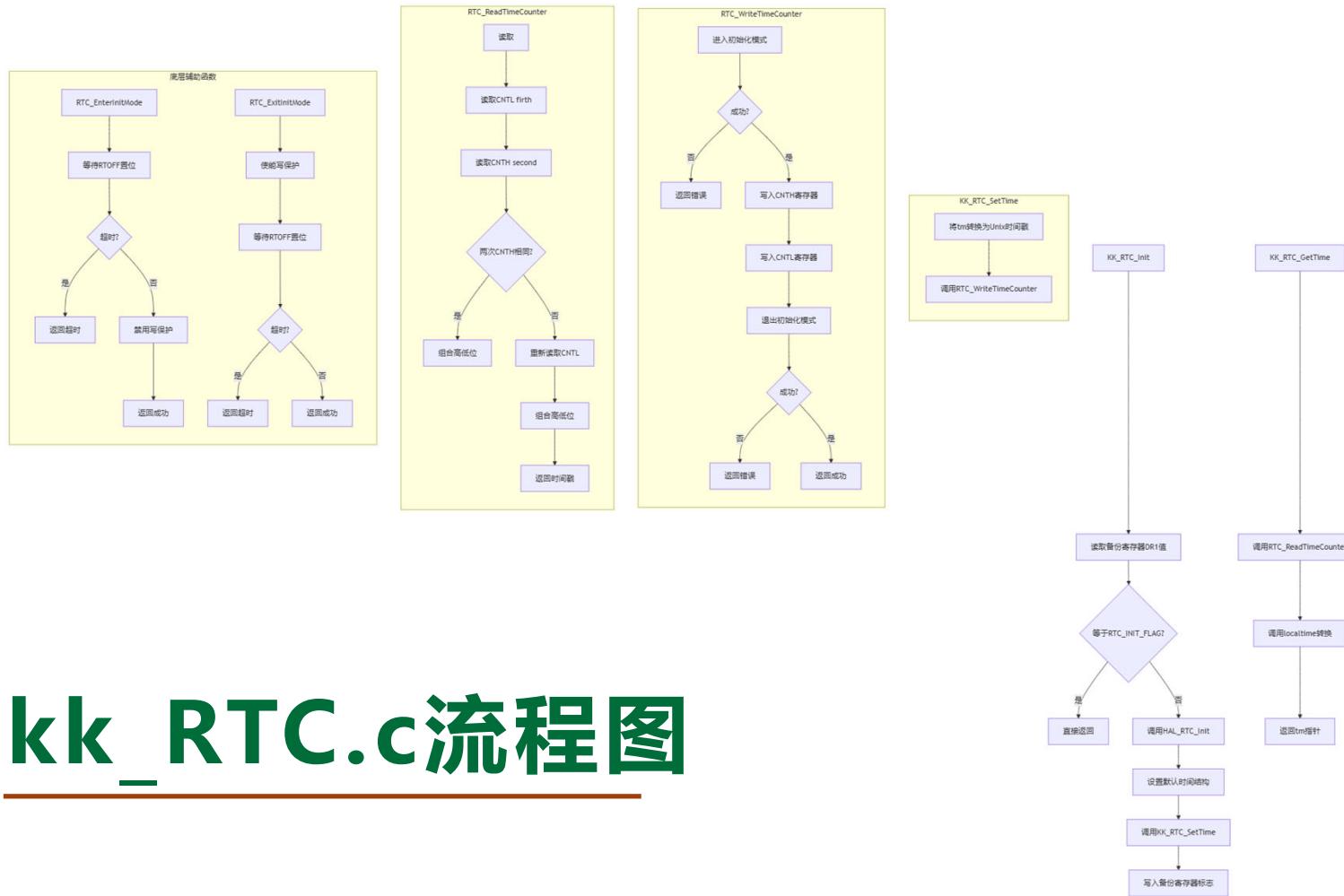




KEY.c状态机



kk_RTC.c流程图



kk RTC.c流程图



任务调度的创新

本项目的一个核心创新在于，即使在裸机平台上，也构建了一个基于时间片和任务驱动的清晰架构。每个功能模块被封装成独立的任务，通过明确的任务切换点（时间片结束或事件触发）进行协作。这种设计使得代码结构更清晰，任务间的耦合度更低，极大地提升了代码的可读性、可维护性和可扩展性。

这对于复杂功能的万年历项目尤为重要，这种结构化的方法让开发过程更加高效，也便于后续功能的增加或修改。

```

// 记录自动模式任务的最后执行时间
last_automode_tick = HAL_GetTick();
// 记录按键扫描的最后执行时间
last_key_scan_tick = HAL_GetTick();
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /*串口测试RTC所需变量
    len = sprintf(str,"ADC:%hd\n",adc_val);
    HAL_UART_Transmit_IT(&huart1,(uint8_t *)str,len);
    HAL_Delay(1000);
    */
    // 如果距离上次按键扫描的时间间隔达到设定值，则{
    if (HAL_GetTick() - last_key_scan_tick
    {
        MainTask();
        // 更新按键扫描的最后执行时间
        last_key_scan_tick = HAL_GetTick();
    }

    // 如果距离上次自动模式任务执行的时间间隔达到设
    // 并且日历状态为正常模式，则执行自动模式任务
    if (HAL_GetTick() - last_automode_tick
        && calendarState == CalendarState_Normal
    {
        automode_task(adc_val);
        // 更新自动模式任务的最后执行时间
        last_automode_tick = HAL_GetTick();
    }
}

void MainTask(void)
{
    // 处理三个按键对象的状态更新与事件检测
    KeyLite_Process(&key0);
    KeyLite_Process(&key1);
    KeyLite_Process(&key2);

    // 判断当前是否处于正常模式（非设置模式）
    if (calendarState == CalendarState_Normal)
    {
        // 获取当前实时时间（来自RTC模块）
        struct tm *now = KK_RTC_GetTime();

        // 显示当前时间到LCD屏幕上
        showTime(now);

        /* 串口输出时间信息（调试用途）注意：此函数内可能包含 Hal_Delay(),
        * 可能会阻塞主任务执行。如需使用，请确保不影响系统实时性。
        *
        * usartTime(now);
        */
    }
    else
    {
        // 如果处于设置模式，则显示正在设置的时间
        showTime(&settingTime);

        // 在设置界面显示闪烁光标，提示当前编辑位置
        showCursor();
    }
}

```



```
// 记录自动模式任务的最后执行时间
last_automode_tick = HAL_GetTick();
// 记录按键扫描的最后执行时间
last_key_scan_tick = HAL_GetTick();
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /*串口测试RTC所需变量
    len = sprintf(str,"ADC:%hd\n",adc_val);
    HAL_UART_Transmit_IT(&huart1,(uint8_t *)str,len);
    HAL_Delay(1000);
    */

    // 如果距离上次按键扫描的时间间隔达到设定值，则执行主任务
    if (HAL_GetTick() - last_key_scan_tick >= KEY_SCAN_INTERVAL)
    {
        MainTask();
        // 更新按键扫描的最后执行时间
        last_key_scan_tick = HAL_GetTick();
    }

    // 如果距离上次自动模式任务执行的时间间隔达到设定值
    //，并且日历状态为正常模式，则执行自动模式任务
    if (HAL_GetTick() - last_automode_tick >= TASK_AUTOMODE_INTERVAL
        && calendarState == CalendarState_Normal)
    {
        automode_task(adc_val);
        // 更新自动模式任务的最后执行时间
        last_automode_tick = HAL_GetTick();
    }
}
```

时间片

时间片轮询（非抢占式调度），利用 HAL_GetTick 来产生一个差值进行比较。在主循环中，不再是简单地依次调用所有任务，而是根据时间来判断哪些任务“到点”该执行了。

```
void MainTask(void)
{
    // 处理三个按键对象的状态更新与事件检测
    KeyLite_Process(&key0);
    KeyLite_Process(&key1);
    KeyLite_Process(&key2);

    // 判断当前是否处于正常模式（非设置模式）
    if (calendarState == CalendarState_Normal)
    {
        // 获取当前实时时间（来自RTC模块）
        struct tm *now = KK_RTC_GetTime();

        // 显示当前时间到LCD屏幕上
        showTime(now);

        /* 串口输出时间信息（调试用途）注意：此函数内可能包含 Hal_Delay()，
        * 可能会阻塞主任务执行。如需使用，请确保不影响系统实时性。
        *
        * usartTime(now);
        */
    }
    else
    {
        // 如果处于设置模式，则显示正在设置的时间
        showTime(&settingTime);

        // 在设置界面显示闪烁光标，提示当前编辑位置
        showCursor();
    }
}
```

任务驱动

在裸机环境下，任务驱动机制将业务逻辑解耦后分为一个个的 Task 任务，提升了系统的并发性和实时性



```

void KeyLite_Process(KeyLiteHandle *handle)
{
    /* 读取当前物理状态 (GPIO_RESET表示按下)
     *   currentState = (HAL_GPIO_ReadPin(handle->GPIOx, handle->GPIO_Pin) == GPIO_PIN_RESET)
     *       ? Pressed
     *       : Unpressed;

    /* 状态变化检测 (前沿/后沿检测) */
    if (currentState != handle->prevState)
    {
        handle->prevState = currentState; // 更新历史状态
        handle->isDebounced = 0;          // 重置消抖标志
        handle->pressedTime = 0;          // 清除按下时间戳
    }

    /* 按下状态处理 */
    if (currentState == Pressed)
    {
        // 初次检测到按下时记录时间戳
        if (handle->pressedTime == 0)
        {
            handle->pressedTime = HAL_GetTick();
        }
        // 消抖时间检查 (防抖周期: KEY_DEBOUNCE_TICKS, 通常10-20ms)
        else if (!handle->isDebounced && (HAL_GetTick() - handle->pressedTime) > KEY_DEBOUNCE_TICKS)
        {
            handle->isDebounced = 1; // 标记消抖完成
            // 触发回调函数 (如果已注册)
            if (handle->callback)
            {
                handle->callback(); // 执行用户定义的操作
            }
        }
    }
}

```

状态机相关代码

状态机的创新

本项目在按键输入处理上进行了精心设计，其创新之处在于构建了一个基于状态机、融合前沿触发、软件消抖和非阻塞机制的高效可靠输入子系统。

我们设计的按键状态机不仅追求底层的高效与可靠，还注重上层的易用性。通过引入回调函数机制（如 Key0KeyLiteTask），按键状态机在检测到有效按下事件后，会将具体的任务逻辑（如调整时间、切换设置项）完全交给用户自定义的回调函数去处理。状态机本身只负责事件检测和触发，这种解耦设计使得上层应用开发者可以专注于业务逻辑的实现，而不必关心底层复杂的按键状态管理和消抖细节。

这种模块化的设计极大地提高了代码的可维护性和可扩展性。开发者可以轻松地为不同的按键绑定不同的功能，修改或增加按键功能时无需改动状态机核心代码，降低了开发难度和维护成本。



```

void KeyLite_Process(KeyLiteHandle *handle)
{
    // 读取当前物理状态 (GPIO_RESET表示按下)
    KeyLiteState currentState = (HAL_GPIO_ReadPin(handle->GPIOx, handle->GPIO_Pin) == GPIO_PIN_RESET)
        ? Pressed
        : Unpressed;

    /* 状态变化检测 (前沿/后沿检测) */
    if (currentState != handle->prevState)
    {
        handle->prevState = currentState; // 更新历史状态
        handle->isDebounced = 0;          // 重置消抖标志
        handle->pressedTime = 0;          // 清除按下时间戳
    }

    /* 按下状态处理 */
    if (currentState == Pressed)
    {
        // 初次检测到按下时记录时间戳
        if (handle->pressedTime == 0)
        {
            handle->pressedTime = HAL_GetTick();
        }
        // 消抖时间检查 (防抖周期: KEY_DEBOUNCE_TICKS, 通常10~20ms)
        else if (!handle->isDebounced && (HAL_GetTick() - handle->pressedTime) > KEY_DEBOUNCE_TICKS)
        {
            handle->isDebounced = 1; // 标记消抖完成
            // 触发回调函数 (如果已注册)
            if (handle->callback)
            {
                handle->callback(); // 执行用户定义的操作
            }
        }
    }
}

```

状态机相关代码

状态机的特点

前沿触发 (Edge Detection)

状态机仅在按键状态变化时（从释放到按下）触发回调，避免重复响应长按事件。通过 `handle->prevState` 和 `currentState` 的比较实现前沿检测。

消抖处理 (Debouncing)

- 通过 `KEY_DEBOUNCE_TICKS` (消抖时间阈值) 过滤机械按键的抖动噪声。
- 只有按键稳定按下超过消抖时间后，才会触发回调。

(3) 非阻塞设计

使用 `HAL_GetTick()` 实现软件延时，而非阻塞式延时（如 `delay_ms()`），确保系统可以同时处理其他任务。

状态机与其他模块的交互

(1) 回调函数 (Key0KeyLiteTask 等)

- 当按键状态机检测到有效按下事件后，调用用户注册的回调函数。
- 回调函数负责具体的任务逻辑（如调整时间、切换设置项）。

(2) 主任务循环 (MainTask())

- `MainTask()` 通过不断调用 `KeyLite_Process()` 检测按键事件，并更新显示 `showTime()` 和 `showCursor()`。
- 这是一种事件驱动的任务调度方式。



```

38 /**
39 * @brief 自动模式任务处理函数
40 *
41 * 该函数根据ADC值判断当前环境亮度，并相应地切换LCD显示模式。
42 * 仅在正常日历状态下处理自动模式。
43 *
44 * @param adc_val ADC读取的亮度值
45 */
46 void automode_task(uint16_t adc_val)
47 {
48     // 仅在正常日历状态下处理自动模式
49     if (calendarState != CalendarState_Normal)
50     {
51         return;
52     }
53
54     // 处理模式切换
55     if (adc_val > DARK_MODE_THRESHOLD && current_mode != DARK_MODE)
56     {
57         // 如果ADC值大于暗模式阈值且当前模式不是暗模式，则清屏为黑色，并设置当前模式为DARK_MODE
58         lcd_clear(BLACK);
59         current_mode = DARK_MODE;
60     }
61     else if (adc_val < LIGHT_MODE_THRESHOLD && current_mode != LIGHT_MODE)
62     {
63         // 如果ADC值小于亮模式阈值且当前模式不是亮模式，则清屏为白色，并设置当前模式为LIGHT_MODE
64         lcd_clear(WHITE);
65         current_mode = LIGHT_MODE;
66     }
67
68     // 中间区域 (LIGHT_THRESHOLD-DARK_THRESHOLD) 保持当前模式不变
69 }

```

自动深色/浅色变化相关代码

显示的创新

本项目在显示交互层面的一大创新，是实现了基于环境光的自适应LCD显示调节。

我们创新性地将光敏传感器与LCD显示模块相结合，通过ADC精确采集环境光照强度。系统并非简单地设置固定亮度，而是能够根据环境光线的变化，智能地自动调整LCD的显示对比度，使其从深色到浅色平滑过渡。例如，在光线昏暗的环境下，LCD会自动切换到深色模式，降低亮度，既保护用户视力，减少眩光，又能在一定程度上模拟夜光显示，提升夜间可读性；而在光线充足的环境下，则自动调整为浅色模式，确保显示内容清晰可见。这种自适应能力极大地提升了设备在不同光线条件下的可用性和用户体验。

这种设计让设备更加智能，无需用户手动调节，自动适应环境，尤其对于需要长时间观察显示信息的设备（如万年历）来说，能显著提升舒适度和便利性。



```

38 /**
39 * @brief 自动模式任务处理函数
40 *
41 * 该函数根据ADC值判断当前环境亮度，并相应地切换LCD显示模式。
42 * 仅在正常日历状态下处理自动模式。
43 *
44 * @param adc_val ADC读取的亮度值
45 */
46 void automode_task(uint16_t adc_val)
47 {
48     // 仅在正常日历状态下处理自动模式
49     if (calendarState != CalendarState_Normal)
50     {
51         return;
52     }
53
54     // 处理模式切换
55     if (adc_val > DARK_MODE_THRESHOLD && current_mode != DARK_MODE)
56     {
57         // 如果ADC值大于暗模式阈值且当前模式不是暗模式，则清屏为黑色，并设置当前模式为DARK_MODE
58         lcd_clear(BLACK);
59         current_mode = DARK_MODE;
60     }
61     else if (adc_val < LIGHT_MODE_THRESHOLD && current_mode != LIGHT_MODE)
62     {
63         // 如果ADC值小于亮模式阈值且当前模式不是亮模式，则清屏为白色，并设置当前模式为LIGHT_MODE
64         lcd_clear(WHITE);
65         current_mode = LIGHT_MODE;
66     }
67
68     // 中间区域 (LIGHT_THRESHOLD-DARK_THRESHOLD) 保持当前模式不变
69 }

```

自动深色/浅色变化相关代码

实现

采用了基于环境光强和时间的动态亮度调节算法

(1) ADC 值读取：

函数通过参数 `adc_val` 接收 ADC 读取的环境亮度值

(2) 阈值比较：

如果 `adc_val > DARK_MODE_THRESHOLD`，说明环境过暗，切换到 `DARK_MODE` (黑底白字)。

如果 `adc_val < LIGHT_MODE_THRESHOLD`，说明环境过亮，切换到 `LIGHT_MODE` (白底黑字)。

在两个阈值之间时，保持当前模式不变，避免频繁切换。

(3) 模式切换：

通过 `lcd_clear(BLACK)` 或 `lcd_clear(WHITE)` 直接清屏为纯色，实现深浅模式的切换。

扩展性

可通过调整 `DARK_MODE_THRESHOLD` 和 `LIGHT_MODE_THRESHOLD` 改变切换灵敏度，或增加渐变过渡效果提升用户体验。



测试异常情况处理▶

利用C语言库<time.h>排除大部分错误。



闰月处理



跨年处理



自动深浅色变化



第五部分 ▷▷

总结

- 总结与不足
- 参考文献

德以明理 学以精工



利用正点原子精英板 V2，本项目完成了一款功能完善的智能万年历开发。该产品不仅能清晰显示时间、支持用户设置，还能根据环境自动切换深浅色模式，提升用户体验。项目的最大突破在于自研 RTC 库，攻克了 HAL 库的固有缺陷。整体设计遵循模块化、解耦原则，采用时间片轮询与任务驱动方法，确保了代码的规范性、完整性及良好的注释，为后续功能扩展奠定了坚实基础。



厂 不足：

1. LCD界面可以继续美化；
2. 按键可以增加长按快速设置日期；
3. 按下按键仍有一定延迟，与HAL_ADC发生冲突，需解决。



[1] 正点原子. STM32F103开发指南V1.2 *EB/OL*. 引用日期: **2025-06-08**.

[正点原子资料下载中心 — 正点原子资料下载中心 1.0.0 文档](#).

[2] 正点原子. 精英V2硬件参考手册V1.0 *EB/OL*. 引用日期: **2025-06-08**.

[正点原子资料下载中心 — 正点原子资料下载中心 1.0.0 文档](#).

[3] 正点原子. ATK-MD0350模块用户手册 *EB/OL*. 引用日期: **2025-06-08**.

[正点原子资料下载中心 — 正点原子资料下载中心 1.0.0 文档](#).

[4] STMicroelectronics. STM32F10xxx参考手册 *EB/OL*. 引用日期: **2025-06-08**.

<https://www.st.com/>.

[5] weixin_43892323. STM32CubeMX实战教程 (七) ——TFT_LCD液晶显示 (附驱动代码)

EB/OL. CSDN, 2020-07-10

引用日期: **2025-06-08**.

https://blog.csdn.net/weixin_43892323/article/details/107305536.

[6] keysking. 【STM32入门教程】RTC实时时钟，超清晰动画讲解 *EB/OL*. bilibili, 2024-12-21

引用日期: **2025-06-08**.

https://www.bilibili.com/video/BV1u7kbYVEeh/?share_source=copy_web&vd_source=641d4ebbe1afea8b6c0683d08afe9b87



北京理工大學珠海學院
BEIJING INSTITUTE OF TECHNOLOGY, ZHUHAI

感谢观看
请您批评指正

德以明
学以精