第 5 章

GPIO 编程与时钟

在嵌入式系统开发领域，外设编程是实现微控制器与外部硬件交互的核心部分。本章将深入探讨 STM32 微控制器的外设编程。

本章首先探讨 STM32G4 系列微控制器的 GPIO 的配置模式、控制函数及配置步骤，通过具体实例，说明 IO 口的综合应用方法。GPIO 端口配置模式涵盖了输入、输出、复用功能和模拟输入等多种模式，每种模式都有其特定的应用场景。通过使用 STM32 HAL 库中的函数，开发者可以轻松地设置端口模式、读取端口状态以及执行其他操作。

第 6 集
微课视频

接下来，本章介绍了 STM32G4 系列微控制器时钟源的选择与配置。时钟系统作为微控制器的核心要素，对系统的运行速度和整体性能起着决定性作用。本节重点介绍了 STM32G4 系列微控制器的时钟基本概念及时钟树结构，帮助读者理解时钟系统的工作原理。通过使用 STM32 HAL 库中的相关函数，开发者可以灵活地选择合适的时钟源并进行相应的配置，以满足不同应用场景的需求。这些函数允许开发者调整时钟设置，优化系统的性能。

第 7 集
微课视频

本章将帮助读者全面掌握 STM32G4 系列微控制器的 GPIO 编程及时钟系统配置，为后续的实际开发应用打下坚实基础。

5.1 GPIO 端口控制

STM32G431 微控制器提供了 5 组 GPIO 端口，每组包含 16 个引脚，具体数量取决于具体型号。STM32G431xB 系列芯片提供从 32 引脚到 100 引脚不等的多种规格及不同的封装，用户可根据具体需求选择。

5.1.1 GPIO 端口配置模式

STM32G431 微控制器的 GPIO 引脚支持多种配置模式。

1. 输入模式

(1) 浮空输入(GPIO_MODE_INPUT_FLOATING)。在此模式下，引脚没有连接到上拉或下拉电阻，处于未定义状态，可以用于检测外部信号的变化，但此时引脚的电平不稳定，容易受到噪声的干扰。该模式主要用于需要检测外部信号变化但无其他强信号源驱动的应

用，例如简单的开关检测，但注意需要增加去抖机制。

（2）上拉输入（GPIO_MODE_INPUT_PULLUP）。在此模式下，引脚通过内部上拉电阻连接到 VCC，在未驱动状态下引脚处于高电平。当外部信号接入并驱动该引脚到低电平时，GPIO 会读取为低电平。该模式主要用于按钮输入等场景，当按钮未按下时，引脚保持高电平；按钮按下时，引脚接地为低电平，能够有效防止信号干扰。

（3）下拉输入（GPIO_MODE_INPUT_PULLDOWN）。在此模式下，引脚通过内部下拉电阻连接到 GND，在未驱动状态下引脚处于低电平。当外部信号接入并驱动该引脚到高电平时，则 GPIO 会读取为高电平。该模式主要用于将引脚默认保持在低电平。例如，当某个外部信号源接入时，可以将引脚拉高，以便进行状态检测。

2. 输出模式

（1）推挽输出（GPIO_MODE_OUTPUT_PP）。引脚可以输出高电平或低电平。在此模式下，当引脚被设置为高电平时，它能够直接驱动负载，显示为接近 VCC 的电压；当设置为低电平时，它能够将负载接地。这是最常见的输出配置。该模式适合用于直接驱动 LED、继电器和其他低阻抗负载，或在数字电路中作为信号输出。

（2）开漏输出（GPIO_MODE_OUTPUT_OD）。引脚可以输出低电平，但高电平时需要外部上拉电阻。高电平状态时，引脚表现为高阻抗状态，因此需要外部上拉电阻将引脚拉高到 VCC。这种模式常用在多个设备共享同一信号线，该模式适合用于 I2C 通信、多个设备需要连接在同一总线的情况，或用于需要实现逻辑与操作的场景，如多个开关。

3. 复用功能模式

（1）推挽复用功能（GPIO_MODE_AF_PP）。引脚可以用于复用功能，如 SPI、I2C、USART 等。该模式主要用于高性能通信接口，例如 SPI 时钟（SCK）、USART 发送数据引脚（TX）等场景，这里要求信号在传输时能够保持稳定。

（2）开漏复用功能（GPIO_MODE_AF_OD）。引脚可以用于复用功能，需要外部上拉电阻来提供高电平。例如，I2C 接口时以开漏方式工作。具体使用场合和开漏输出一致。

4. 模拟模式

模拟模式（GPIO_MODE_ANALOG）允许引脚接收来自传感器或其他模拟信号源的模拟输入信号。在此模式下，GPIO 引脚会被配置为高阻抗状态，从而最大程度减轻对信号源的负载影响，并允许模数转换器（ADC）准确采样输入电压值。此模式不涉及任何数字信号输出，因此无须配置上下拉电阻。该模式主要适用于需要将模拟信号转换为数字信号的应用，如读取温度传感器、光传感器等模拟输出设备的信号，输入则来自电位计的信号或其他需要处理连续电压信号的模拟应用。

5.1.2 GPIO 端口控制函数

在 STM32G4 系列微控制器中，STM32 HAL 库提供了丰富的 GPIO 端口控制函数，从而使得开发人员能够轻松初始化和灵活控制 GPIO 引脚。下面详细介绍一些常用的 GPIO 控制函数及其用法。

1. 初始化 GPIO 引脚

HAL_GPIO_Init()函数是 STM32 HAL 库中用于初始化 GPIO 引脚的函数。它有两个参数：一个指向 GPIO 端口的指针，另一个是结构体指针，即设置 GPIO 端口参数。以下是关于这个函数的详细介绍，包括参数说明、返回值和使用实例。

（1）函数原型。

```
void HAL_GPIO_Init(GPIO_TypeDef  * GPIOx, GPIO_InitTypeDef  * GPIO_Init);
```

（2）参数说明。

GPIOx：用于指向待初始化的 GPIO 端口的指针。通过定义，可以获取端口 A、B、C 等对应的结构体，例如 GPIOA、GPIOB 等。

GPIO_Init：指向结构体的指针，该结构体用于配置 GPIO 的工作模式及其他属性，该结构体通常包含以下成员。

① Pin：要配置的 GPIO 引脚。

② Mode：引脚的工作模式（输入、输出、复用、模拟等）。

③ Pull：上拉或下拉电阻配置。

④ Speed：指定引脚的输出速率。

⑤ Alternate：复用时指定的功能。

2. 恢复 GPIO 引脚默认状态

HAL_GPIO_DeInit()是 STM32 HAL 库中的一个函数，用于将指定的 GPIO 引脚恢复到默认状态，即清除与引脚相关的所有配置设置，使它们回到未配置的状态。

（1）函数原型。

```
void HAL_GPIO_DeInit(GPIO_TypeDef *  GPIOx, uint32_t GPIO_Pin);
```

（2）参数说明。

① GPIOx：指向 GPIO 端口的指针，如 GPIOA、GPIOB 等。

② GPIO_Pin：要初始化的引脚编号，可以是单个引脚或多个引脚的组合（使用按位或运算符“|”连接）。

（3）使用实例。

假设将 GPIOA 上的第 0 号引脚恢复到默认状态，可以作如下调用。

```
HAL_GPIO_DeInit(GPIOA,GPIO_PIN_0);
```

如果需要同时将多个引脚恢复到默认状态，比如 GPIOA 上的第 0 号和第 1 号引脚，可以作如下调用。

```
HAL_GPIO_DeInit(GPIOA, GPIO_PIN_0 | GPIO_PIN_1);
```

（4）注意事项。

调用 HAL_GPIO_DeInit()函数后，被指定的引脚将不再有任何配置，包括输入模式、输出模式、上下拉电阻等都会被清除。如果引脚之前被锁定，则需要先解锁才能重新配置。在某些情况下，可能需要重新初始化整个 GPIO 端口，而不是单独的引脚。在这种情况下，

可以使用__HAL_RCC_GPIOx_FORCE_RESET 和__HAL_RCC_GPIOx_RELEASE_RESET 宏来复位整个 GPIO 端口。

3. 读取 GPIO 引脚状态

HAL_GPIO_ReadPin()函数是 STM32 HAL 库中的一个重要函数，用于读取特定 GPIO 引脚的当前状态。这对于检测引脚的输入状态，如开关状态、按键状态等非常有用。

（1）函数原型。

```
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin);
```

（2）参数说明。

GPIOx：指向要读取的 GPIO 端口的指针，例如 GPIOA、GPIOB 等。

GPIO_Pin：指定要读取的引脚。可以是 GPIO_PIN_0～GPIO_PIN_15 中的任意一个。

（3）返回值。

返回值是 GPIO_PinState 类型，表示引脚的状态，GPIO_PIN_SET 表示引脚处于高电平状态；GPIO_PIN_RESET 表示引脚处于低电平状态。

4. 设置 GPIO 引脚输出状态

HAL_GPIO_WritePin()函数是 STM32 HAL 库中的一个重要函数，用于设置特定 GPIO 引脚的输出状态。开发者可以通过此函数将引脚设置为高电平或低电平，通常用于控制外部设备，如 LED、继电器等。

（1）函数原型。

```
void HAL_GPIO_WritePin(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState);
```

（2）参数说明。

① GPIOx：指向要读取的 GPIO 端口的指针，例如 GPIOA、GPIOB 等。

② GPIO_Pin：指定要读取的引脚。可以是 GPIO_PIN_0～GPIO_PIN_15 中的任意一个。

③ PinState：表示引脚要设置的状态，GPIO_PIN_SET 表示将引脚设置为高电平；GPIO_PIN_RESET 表示将引脚设置为低电平。

5. 切换 GPIO 引脚状态

HAL_GPIO_TogglePin()函数是 STM32 HAL 库中的一个实用函数，旨在切换特定 GPIO 引脚的状态。如果指定的引脚当前为高电平，则此函数可将其设置为低电平；反之，若引脚当前为低电平，则此函数可将其设置为高电平。这种功能特别适用于对 LED 灯的闪烁控制以及其他需要状态切换的应用场景。

（1）函数原型。

```
void HAL_GPIO_TogglePin(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin);
```

（2）参数说明。

① GPIOx：指向要读取的 GPIO 端口的指针，例如 GPIOA、GPIOB 等。

② GPIO_Pin：指定要读取的引脚，即 GPIO_PIN_0～GPIO_PIN_15 中的任意一个。

6. 锁定 GPIO 引脚

HAL_GPIO_LockPin()函数是 STM32 HAL 库中的一个函数，用于锁定指定的 GPIO 引脚。一旦某个引脚被锁定后，就不能再通过 HAL 库或直接操作寄存器来修改该引脚的配置，直到下次复位或重新配置整个 GPIO 端口。

(1) 函数原型。

```
HAL_StatusTypeDef HAL_GPIO_LockPin(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin);
```

(2) 参数说明。

① GPIOx：指向要读取的 GPIO 端口的指针，例如 GPIOA、GPIOB 等。

② GPIO_Pin：指定要读取的引脚，即 GPIO_PIN_0～GPIO_PIN_15 中的任意一个或组合(使用按位或运算符 | 连接)。

(3) 返回值。

① HAL_OK：锁定成功。

② HAL_ERROR：错误发生。

③ HAL_BUSY：总线忙。

④ HAL_TIMEOUT：操作超时。

5.1.3 GPIO 端口配置步骤

STM32G431 微控制器的 GPIO 端口配置涉及一系列步骤，包括初始化 HAL 库、配置系统时钟、初始化 GPIO 端口和引脚等。本小节将介绍其具体步骤。

1. 初始化 HAL 库

HAL_Init()函数是 STM32 微控制器系列中硬件抽象层的一部分。HAL 是一个软件层，提供一组应用编程接口来与微控制器的硬件外设如 GPIO、ADC、UART 等进行交互。HAL_Init()函数负责初始化 HAL。它可以执行以下任务。

(1) 配置系统时钟后，根据系统时钟频率，设置 Systick 定时器的重装载值为 9000，以实现每 1ms 产生一次中断，随后使能 Systick 中断。

(2) 初始化底层硬件，配置 Systick 定时器以生成 1ms 的时间基准，并将 Flash 接口初始化，将 Systick 时钟源配置为 CPU 时钟。

(3) 初始化 HAL 驱动，复位所有外设，为选定的外设初始化底层驱动程序。通常在应用程序启动时，从 main()函数中调用 HAL_Init()函数，以设置微控制器所需的必要硬件和软件组件。在调用 HAL_Init()函数之后，用户可以继续配置和使用 HAL 提供的各种外设驱动程序，如 HAL_GPIO_Init()函数、HAL_UART_Init()函数、HAL_ADC_Init()函数等与 STM32 微控制器的硬件外设进行交互。

2. 配置系统时钟

SystemClock_Config()函数是用于配置 STM32 微控制器系统时钟的自定义函数。这个函数通常由开发人员编写，以满足应用程序的特定时钟需求。在 SystemClock_Config()函数中，会执行以下任务。

(1) 启用外部时钟源,如高速外部时钟。

(2) 配置 PLL 参数以生成所需的系统时钟频率。

(3) 选择系统时钟源。

(4) 更新系统时钟相关的变量,如 SystemCoreClock。

(5) 配置总线时钟(HCLK、PCLK1、PCLK2)。

(6) 调用 HAL 函数 HAL_RCC_ClockConfig()应用新的时钟配置。

这样做可以确保微控制器以所需的时钟频率工作,从而满足应用程序的性能和功耗要求。SystemClock_Config()函数通常在应用程序的初始化阶段调用,紧跟在 HAL_Init()函数之后。这确保在使用 HAL 驱动程序之前,系统时钟已经正确配置。SystemClock_Config()函数是一个自定义函数,是 STM32 固件开发的重要组成部分。

3. 初始化 GPIO 端口和引脚及配置 GPIO 工作模式

在 STM32 微控制器上初始化 GPIO 端口和引脚,可以使用 HAL 库提供的 HAL_GPIO_Init()函数来完成。初始化 GPIO 端口和引脚的一般步骤如下。

(1) 定义 GPIO 初始化结构体:GPIO_InitTypeDef GPIO_InitStruct。

(2) 配置 GPIO 初始化结构体的参数,即要初始化的 GPIO 引脚、GPIO 引脚的模式、引脚的上/下拉配置、引脚的输出速度、复用功能选择等。

(3) 调用 HAL_GPIO_Init()函数初始化 GPIO:HAL_GPIO_Init(GPIOA, &GPIO_InitStruct)中第一个参数是要初始化的 GPIO 端口,第二个参数是前面配置的 GPIO 初始化结构体。

4. GPIO 操作

在 STM32 微控制器上操作 GPIO 包括以下常见功能。

(1) 设置 GPIO 引脚输出状态。

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);          // 设置 PA5 为高电平
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);        // 设置 PA5 为低电平
```

(2) 读取 GPIO 引脚输入状态。

```
uint32_t pinState = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_6);     // 读取 PA6 引脚的输入状态
```

(3) 切换 GPIO 引脚输出状态。

```
HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_7);              // 切换 PA7 引脚的输出状态
```

(4) 配置 GPIO 中断。

```
GPIO_InitTypeDef GPIO_InitStruct;
GPIO_InitStruct.Pin = GPIO_PIN_3;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;          // 上升沿触发中断
GPIO_InitStruct.Pull = GPIO_PULLDOWN;                // 下拉配置
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
HAL_NVIC_SetPriority(EXTI3_IRQn, 0, 0);              // 设置中断优先级
HAL_NVIC_EnableIRQ(EXTI3_IRQn);                      // 使能中断
```

（5）在中断服务函数中处理 GPIO 中断。

```
void EXTI3_IRQHandler(void)
{
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_3);        // 调用 HAL 中断处理函数
}
```

（6）锁定 GPIO 配置。

```
HAL_GPIO_LockPin(GPIOA, GPIO_PIN_11);          // 锁定 PA11 引脚的配置
```

上述是 STM32 HAL 库中常见的 GPIO 操作函数，可以满足 GPIO 控制的基本需求。开发者可根据实际应用需求，灵活选用上述 GPIO 操作函数来实现控制。

实例 5-1　IO 口综合应用

试用 STM32G431 微控制器的端口实现按键控制 LED 小灯的基本功能。

要使用 STM32G431 微控制器实现按键控制 LED 的功能，可通过以下几个步骤。

（1）初始化 HAL 库。

（2）配置系统时钟。

（3）初始化用于控制 LED 的 GPIO 输出端口。

（4）初始化用于检测按键状态的 GPIO 输入端口。

（5）编写主循环来检测按键状态并控制 LED。

利用 NUCLEO-G431RB 开发板，将按键连接到 GPIOC 的第 PC13 号引脚，将一个 LED 连接到 GPIOA 的第 PA5 号引脚，以下是实现这一功能的一个完整实例代码。

```
1.  #include "main.h"                    //包含 HAL 头文件
2.  void SystemClock_Config(void);
3.  static void MX_GPIO_Init(void);
4.  int main(void)
5.  {
6.  HAL_Init();
7.  SystemClock_Config();
8.  MX_GPIO_Init();
```

其中，第 6 行表示初始化 HAL 库，为系统提供基础硬件抽象层功能。

第 7 行表示配置系统时钟，确保系统以正确的频率运行。

第 8 行表示初始化 GPIO 端口，为按键检测和 LED 控制做准备。

```
 1.  while (1)
 2.  {
 3.  if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_SET)
 4.  {
 5.  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
 6.  }
 7.  else
 8.  {
 9.  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);
10.  }
```

```
11.  }
12.  }
```

其中，第 3 行表示检测按键状态，PC13 引脚被配置为输入。

第 5 行表示点亮 LED，第 9 行表示熄灭 LED，PA5 被配置为输出。

配置 IO 口，主要按键和 LED 的代码如下。

```
 1.  static void MX_GPIO_Init(void)
 2.  {
 3.        GPIO_InitTypeDef GPIO_InitStruct = {0};
 4.        __HAL_RCC_GPIOC_CLK_ENABLE();
 5.        __HAL_RCC_GPIOA_CLK_ENABLE();
 6.        GPIO_InitStruct.Pin = GPIO_PIN_13;              // PC13
 7.        GPIO_InitStruct.Mode = GPIO_MODE_INPUT;         // 输入模式
 8.        GPIO_InitStruct.Pull = GPIO_PULLDOWN;           // 下拉
 9.        HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
10.         GPIO_InitStruct.Pin = GPIO_PIN_5;             // PA5
11.        GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;     // 推挽输出模式
12.        GPIO_InitStruct.Pull = GPIO_NOPULL;             // 不拉
13.        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;    // 低速
14.        HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
15.  }
```

其中，第 4 行和第 5 行表示使能 GPIOC 和 GPIOA 时钟。

第 6～9 行表示配置按键，初始化 PC13 引脚为输入模式。

第 10～14 行表示配置 LED，初始化 PA5 引脚为输出模式。

配置系统时钟代码如下。具体作用参考注释，详细含义参考 7.2 节。

```
 1.  void SystemClock_Config(void)
 2.  {
 3.  RCC_OscInitTypeDef RCC_OscInitStruct = {0};       // 定义用于初始化振荡器的结构体
 4.  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};       // 定义用于初始化时钟树的结构体
 5.  // 设置电源电压调节至 1.2V
 6.  HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1_BOOST);
 7.  // 初始化 HSI 振荡器并启用 PLL
 8.  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
 9.  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
10.  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
11.  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
12.  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
13.  RCC_OscInitStruct.PLL.PLLM = RCC_PLLM_DIV4;
14.  RCC_OscInitStruct.PLL.PLLN = 85;
15.  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
16.  RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
17.  RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
18.  // 配置时钟源
19.  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
20.  {
21.  // 如果配置失败，调用错误处理函数
22.  Error_Handler();
23.  }
24.  // 配置 APB 和 AHB 总线频率
```

```
25.  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
26.  |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
27.  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
28.  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
29.  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
30.  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;
31.  // 配置系统时钟
32.  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
33.  {
34.  // 如果配置失败,调用错误处理函数
35.  Error_Handler();
36.  }
37.  }
```

错误处理代码如下。可以根据实际情况添加代码。

```
1.  void Error_Handler(void)
2.  {
3.  while(1);
4.  }
```

5.2 时钟源的选择与配置

5.2.1 时钟基本概念及时钟树

第 8 集
微课视频

在 STM32G431RB 微控制器上选择和配置时钟源是一个关键步骤，因为它直接影响系统的性能、功耗和稳定性。系统时钟树如图 5-1 所示。

STM32G4 系列微控制器提供了多种时钟源选项，包括：高速内部振荡器（High-Speed Internal oscillator，HSI）和高速外部振荡器（High-Speed External oscillator，HSE），它们分别提供 8MHz 和 4～48MHz 的频率范围；低速内部振荡器（Low-Speed Internal oscillator，LSI）和低速外部振荡器（Low-Speed External oscillator，LSE），分别提供 40kHz 和 32.768kHz 的频率；以及锁相环（Phase-locked Loop，PLL），其输入源可选择 HSI/2、HSE 或 HSE/2，并可进行 2～16 倍的倍频，但输出频率不超过 72MHz。

1. HSE

HSE 是 STM32 微控制器中的一个时钟源，可以由石英/陶瓷谐振器或外部时钟源提供，其频率范围为 4～48MHz。当使用有源晶振时，时钟从 OSC_IN 引脚进入，OSC_OUT 引脚悬空；当选用无源晶振时，时钟从 OSC_IN 和 OSC_OUT 引脚进入，并且要配谐振电容。在使用 STM32 微控制器时，若选择 HSE 时钟或通过 PLL 倍频后的 HSE 时钟作为系统时钟（System Clock，SYSCLK），一旦检测到 HSE 故障，系统会自动关闭 HSE 和 PLL。此时，备用的高速内部时钟 HSI（16MHz）将接管，继续为系统提供时钟信号，直至 HSE 恢复正常。

2. PLL

PLL 的主要作用是对时钟进行倍频，然后把时钟输出到各个功能部件。PLL 有两个，一个是主 PLL，另外一个是专用的 PLLI2S，他们均由 HSE 或 HSI 提供时钟输入信号。

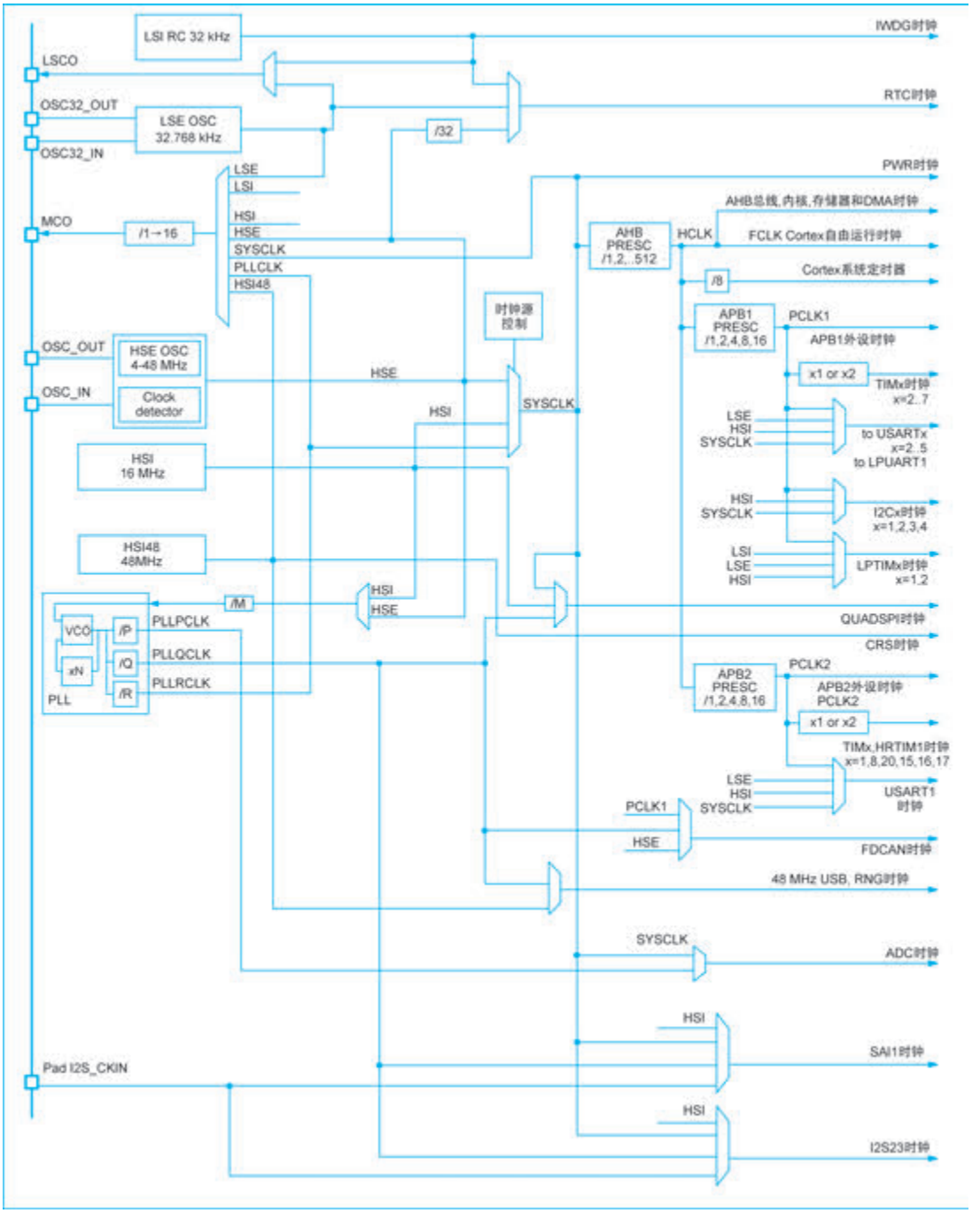

图 5-1 系统时钟树

主 PLL 有两路时钟输出,第一路用于系统时钟,在 STM32G431RB 微控制器中,其最高频率不超过 170MHz,第二路用于 USB OTG FS 时钟(48MHz)、RNG 和 SDIO 时钟(≤48MHz)。

专用的 PLLI2S 用于生成精确时钟，给 I2S 提供时钟。HSE 或 HSI 经过 PLL 时钟输入分频因子 M（范围为 1～16）分频后，作为 VCO 的时钟输入，该时钟的频率需在 2.66～16MHz。例如，选择 HSE＝25MHz 作为 PLL 的时钟输入，M 设置为 5，那么 VCO 输入时钟就等于 5MHz。

VCO 输入时钟经过 VCO 倍频因子 N 倍频之后，成为 VCO 的输出时钟，VCO 输出时钟必须在 96～344MHz。配置 N 为 50，则 VCO 的输出时钟等于 50MHz。如果要把系统时钟超频，需要修改 VCO 的倍频系数 N。

$$\text{PLLCLK\_OUTMAX}=\frac{\text{VCOCLK\_OUTMAX}}{\text{P\_MIN}}=\frac{250}{2}=125(\text{MHz})$$

注意，STM32G4 系列微控制器的最高频率不超过 170MHz。VCO 输出时钟之后有三个分频因子：PLLCLK 分频因子 P，USB OTG FS 或 RNG 或 SDIO 时钟分频因子 Q，分频因子 R。P 可以取值 2、4、6、8，如配置为 2，则得到 PLLCLK＝125MHz。Q 可以取值 2、4、6、8，但是 USB OTG FS 必须使用 48MHz，VCO 输出时钟为 250MHz，则 $Q=250/48=5.2$，跟 Q 值不符，可以选用 HSI48 时钟或重新配置参数。有关 PLL 的配置，存在一个专门的 RCC PLL 配置寄存器 RCC_PLLCFGR，具体细节请参考手册。PLL 的时钟配置过程涉及的主要公式如下。

$$\text{VCOCLK\_IN}=\frac{\text{PLLCLK\_IN}}{M}=\frac{\text{HSE}}{5}=5\text{MHz}$$

$$\text{VCOCLK\_OUT}=\text{VCOCLK\_IN}\times N=5\text{MHz}\times 50=250\text{MHz}$$

$$\text{PLLCLK\_OUT}=\frac{\text{VCOCLK\_OUT}}{P}=\frac{250}{2}=125\text{MHz}$$

3. SYSCLK

SYSCLK 来源可以是 HSI、PLLCLK、HSE，具体由时钟配置寄存器 RCC_CFGR 的 SW 位配置。将系统时钟 SYSCLK 设置为 PLLCLK，即 170MHz。若系统时钟配置为 HSE-PLL 模式，当 HSE 出现故障时，系统时钟将自动切换至 HSI，即 16MHz，直至 HSE 恢复正常。

4. 高速总线时钟

系统时钟 SYSCLK 经过 AHB（先进高性能总线，Advanced High-performance BUS）预分频器分频之后得到时钟叫高速总线时钟（AHB High-speed Clock，HCLK），分频因子可以是 1、2、4、8、16、64、128、256、512，具体的由时钟配置寄存器 RCC_CFGR 的 HPRE 位设置。片上大部分外设的时钟均通过 HCLK 分频获得。至于 AHB 上的外设时钟设置，需待实际使用该外设时再进行配置，此处仅需对 APB（高级外设总线，Advanced Peripheral BUS）时钟进行初步设置。这里设置为 1 分频，即 HCLK＝SYSCLK＝170MHz。

5. APB1 总线时钟 PCLK1

APB1 时钟 PCLK1 由 HCLK 经过低速 APB 预分频器得到，分频因子可以是 1、2、4、8、16，具体由时钟配置寄存器 RCC_CFGR 的 PPRE1 位设置。PCLK1 属于低速的总线时钟，片上低速的外设就挂载到这条总线上，比如 USART2/3/4/5、SPI2/3，I2C1/2 等。如果设

置为 4 分频，即 PCLK1＝HCLK/4＝42.5MHz。

在复位启动后，HSI16 被用作系统时钟源。这些设备还具有以下额外的时钟源。

(1) 32kHz 低速内部电阻-电容电路(Resistor-Capacitor Circuit，RC)，用于驱动独立看门狗，并可选地用于实时时钟(Real-Time Clock，RTC)以实现从停止和待机模式下的自动唤醒。

(2) 32.768kHz 的低速外部晶体可用于驱动实时时钟的时钟源(RTCCLK)。

(3) 内部 48MHz 的 RC 时钟源(HSI48)能够驱动 USB 全速接口和随机数生成器(Random Number Generator，RNG)。

6. APB2 时钟 PCLK2

APB2 时钟 PCLK2 源自 HCLK，通过高速 APB2 预分频器进行分频，分频因子可以是 1、2、4、8、16，具体设置由时钟配置寄存器 RCC_CFGR 的 PPRE2 位决定。PCLK2 是高速总线时钟，负责连接片上的高速外设，如 GPIO、USART1、SPI1 等。APB2 上外设的时钟设置，是在实际使用该外设时确定的。此处只需配置 APB2 时钟为 2 分频，即 PCLK2＝HCLK/2＝85MHz。

当不使用某个时钟源时，可以独立地将其开关关闭，以优化功耗。几个预分频器可以用来配置 AHB 频率，以及 APB1 和 APB2 域。AHB、APB1 和 APB2 域的最大频率为 170MHz。所有外设时钟都源自其总线时钟(HCLK、PCLK1 或 PCLK2)，但以下情况除外。

(1) 48 MHz 时钟，用于 USB 设备全速和 RNG。此 48 MHz 时钟源可由软件选择为以下之一。

① PLL 的“Q”输出时钟。

② 内部 HSI48 振荡器。

当可用时，HSI48 的 48MHz 时钟可以与时钟恢复系统结合使用，允许在没有外部晶体的情况下为 USB OTG FS 提供适当的时钟连接(无晶体时的解决方案)。

(2) ADC 时钟，该时钟源可由软件选择为以下之一。

① 系统时钟(SYSCLK)。

② PLL 的“P”输出时钟。

(3) U(S)ART 时钟，这些时钟源可由软件选择为以下之一。

① 系统时钟(SYSCLK)。

② HSI16 时钟。

③ LSE 时钟。

④ APB1 或 APB2 时钟(取决于 U(S)ART 映射到哪个 APB)。

从停止模式唤醒仅支持当使用 HSI16 或 LSE 作为时钟源时。

(4) I2C 时钟，这些时钟源可由软件选择为以下之一。

① 系统时钟(SYSCLK)。

② HSI16 时钟。

③ APB1 时钟(PCLK1)。

从停止模式唤醒仅支持当使用 HSI16 作为时钟源时。

(5) SAI1 时钟，该时钟源可由软件选择为以下之一。
① 映射到 I2S_CKIN 的外部时钟。
② 系统时钟。
③ PLL 的“Q”输出时钟。
④ HSI16 时钟。
(6) QUADSPI 内核时钟，该时钟源可由软件选择为以下之一。
① 系统时钟。
② PLL 的“Q”输出时钟。
③ HSI16 时钟。
(7) 低功耗定时器(Low-Power Timer，LPTIM)时钟，该时钟源可由软件选择为以下之一。
① LSI 时钟。
② LSE 时钟。
③ HSI16 时钟。
④ APB1 时钟(PCLK1)。
⑤ 映射到 LPTIMx_IN1 引脚的外部时钟。
从停止模式下的功能(包括唤醒)仅在使用 LSI、LSE 或外部时钟模式时受支持。
(8) RTC 时钟，该时钟源可由软件选择为以下之一。
① LSE 时钟。
② LSI 时钟。
③ 经过 32 分频后的 HSE 时钟。
从停止模式下的功能(包括唤醒)仅在使用 LSI 或 LSE 时受支持。
(9) 独立看门狗(Independent Watchdog，IWDG)时钟，总是使用 LSI 时钟。
(10) UCPD1 时钟，来源于 HSI16 时钟。
(11) FDCAN1 时钟，该时钟源可由软件选择为以下之一。
① HSE 时钟。
② PLL 的“Q”输出时钟。
③ PCLK 时钟。

系统定时器(System Timer，SysTick)的时钟源由 RCC 提供，有两种方式：一是经 AHB 进行 8 分频后的时钟，二是 HCLK。具体选择通过配置 SysTick 控制与状态寄存器中的 CLKSOURCE 位来实现。

FCLK 用作 Cortex-M4 带浮点单元(Floating-Point Unit，FPU)的自由运行时钟，更多详情请参阅 Cortex®-M4 编程手册。

上述内容详细阐述了 STM32 微控制器系列中的时钟分配与选择机制，并说明了部分外设在不同低功耗模式下的工作条件。

5.2.2 配置时钟相关函数

1. 配置或启用多个时钟源函数

HAL_RCC_OscConfig()函数用于配置和使能 STM32 微控制器的振荡器。通过这个

函数，可以初始化或重新配置不同的时钟源，如 HSE、HSI、LSE 和 LSI，以及 PLL。该函数以一个指向 RCC_OscInitTypeDef 结构体的指针作为参数，此结构体包含了所有需要配置的时钟源的信息。

（1）函数原型。

```
HAL_StatusTypeDef HAL_RCC_OscConfig(RCC_OscInitTypeDef * RCC_OscInitStruct)
```

（2）函数参数。

该函数主要参数是 RCC_OscInitTypeDef 结构体。

```
typedef struct
{
  uint32_t OscillatorType;
  uint32_t HSEState;
  uint32_t LSEState;
  uint32_t HSIState;
  uint32_t HSICalibrationValue;
  uint32_t LSIState;
  uint32_t HSI48State;
  RCC_PLLInitTypeDef PLL;
}RCC_OscInitTypeDef;
```

RCC_OscInitTypeDef 结构体成员 OscillatorType 是指定要配置的振荡器类型，可以是以下之一或其组合。

RCC_OSCILLATORTYPE_HSE：表示要配置 HSE。

RCC_OSCILLATORTYPE_HSI：表示要配置 HSI。

RCC_OSCILLATORTYPE_LSE：表示要配置 LSE。

RCC_OSCILLATORTYPE_LSI：表示要配置 LSI。

RCC_OscInitTypeDef 结构体成员 HSEState、HSIState、LSEState、LSIState 分别指定 HSE、HSI、LSE、LSI 的状态，可以是开启（ON）、关闭（OFF）、旁路模式等。

RCC_OscInitTypeDef 结构体成员 HSICalibrationValue 是 HSI 振荡器的校准值，该值最小为 0x00，最大为 0xFF。

RCC_OscInitTypeDef 结构体成员 PLL 是配置锁相环，也是一个结构体。

```
typedef struct
{
  uint32_t PLLState;
  uint32_t PLLSource;
  uint32_t PLLM;
  uint32_t PLLN;
  uint32_t PLLP;
  uint32_t PLLQ;
  uint32_t PLLR;
}RCC_PLLInitTypeDef;
```

其中，

PLLState 是 PLL 的状态，可以是开启（ON）或关闭（OFF）。

PLLSource 是 PLL 的输入时钟源，例如 HSE 或 HSI。

PLLM、PLLN、PLLP、PLLQ、PLLR 这些成员用于配置 PLL 的分频和倍频参数。

（3）函数工作流程。

① 根据传入的 RCC_OscInitStruct 结构体中的 OscillatorType 成员，确定哪些振荡器需要被配置。

② 按照结构体中定义的状态来配置每个选定的振荡器，并使能或禁止它们。

③ 如果启用了 PLL，则根据提供的参数配置 PLL。

④ 对于 HSE、LSE 等特定类型的振荡器，在状态更改后，可能需要等待以确保新状态生效。

（4）返回值。

① HAL_OK：操作成功完成。

② HAL_ERROR：参数错误或其他错误。

③ HAL_BUSY：总线繁忙，无法完成操作。

④ HAL_TIMEOUT：等待振荡器稳定的操作已超时。

（5）实例代码。

```
RCC_OscInitTypeDef RCC_OscInitStruct = {0};
/* 配置 HSE 为系统时钟源 */
RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
RCC_OscInitStruct.HSEState = RCC_HSE_ON;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
RCC_OscInitStruct.PLL.PLLM = 8;
RCC_OscInitStruct.PLL.PLLN = 336;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
RCC_OscInitStruct.PLL.PLLQ = 7;
```

该段代码演示了如何配置 HSE 作为 PLL 的输入，设置 PLL 的分频与倍频参数，并调用 HAL_RCC_OscConfig()函数将这些配置应用生效。

2. 系统时钟及其相关的总线时钟分频器配置函数

HAL_RCC_ClockConfig()函数专门用于配置 STM32 微控制器的系统时钟及其相关联的总线时钟分频器。这个函数可以用来选择系统时钟源、配置 AHB 和 APB 的分频器，以及根据新的时钟频率调整闪存的等待状态数量（即 FLatency 参数）。此外，当 HCLK（AHB 时钟）发生变化时，它还会更新 SysTick 定时器以确保其正常工作。

（1）函数原型。

```
HAL_StatusTypeDef HAL_RCC_ClockConfig(RCC_ClkInitTypeDef * RCC_ClkInitStruct, uint32_t
FLatency)
```

（2）函数参数。

RCC_ClkInitTypeDef 用于配置 STM32 微控制器时钟系统。这个结构体包含了几个成员，每个成员都用于指定不同方面的时钟配置。

```
typedef struct
{
```

```
  uint32_t ClockType;
  uint32_t SYSCLKSource;
  uint32_t AHBCLKDivider;
  uint32_t APB1CLKDivider;
  uint32_t APB2CLKDivider;
}RCC_ClkInitTypeDef;
```

① ClockType：指定要配置的时钟类型，可以是以下之一或其组合。

a. RCC_CLOCKTYPE_SYSCLK：系统时钟。

b. RCC_CLOCKTYPE_HCLK：AHB 时钟。

c. RCC_CLOCKTYPE_PCLK1：APB1 时钟。

d. RCC_CLOCKTYPE_PCLK2：APB2 时钟。

② SYSCLKSource：系统时钟源的选择，例如 PLLCLK、HSI、HSE 等。

③ AHBCLKDivider：AHB 时钟分频器，定义了 HCLK 相对于 SYSCLK 的分频系数。

④ APB1CLKDivider：APB1 时钟分频器，定义了 PCLK1 相对于 HCLK 的分频系数。

⑤ APB2CLKDivider：APB2 时钟分频器，定义了 PCLK2 相对于 HCLK 的分频系数。

⑥ FLatency 参数：其第二个参数定义了闪存的等待周期数。为了确保在不同的 SYSCLK 频率下正确执行指令，必须根据 SYSCLK 频率设置合适的等待周期数。这可以保证程序稳定运行而不发生错误。

(3) 返回值。

① HAL_OK：表示操作成功完成。

② HAL_ERROR：表示配置过程中出现了错误。

③ HAL_BUSY：表示当前有一个正在进行的操作阻止了此次配置。

④ HAL_TIMEOUT：表示等待某个事件(如 PLL 锁定)超时。

(4) 实例代码。

下面是一个系统时钟配置为 80MHz，并且适当地配置 AHB、APB1 和 APB2 分频器的例子。

```
RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
uint32_t FLatency = FLASH_LATENCY_4;
/* 配置 CPU, AHB 和 APB 总线时钟 */
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_SYSCLK
                            | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;     // HCLK = SYSCLK
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;      // PCLK1 = HCLK/2
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;      // PCLK2 = HCLK/2
if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLatency) != HAL_OK)
{
    Error_Handler();                                   /* 初始化错误处理 */
}
```

这段代码首先定义了一个 RCC_ClkInitTypeDef 类型的结构体变量 RCC_ClkInitStruct，并初始化它的成员来指定所需的时钟配置。在完成时钟配置后，调用 HAL_RCC_ClockConfig()函数应用这些设置，并传入适当的 Flash 等待状态值 FLASH_LATENCY_4，以确保在 80MHz

系统时钟下使用 4 个等待周期。如果配置失败，则调用 Error_Handler()函数进行错误处理。

对于不从系统时钟(如 RTC、USB)派生的一些外设时钟，其配置通过在 stm32g4xx_hal_rcc_ex.c 中定义的扩展 API 来完成。

3. 用于配置特定外设时钟函数

HAL_RCCEx_PeriphCLKConfig()函数是用于配置 STM32 微控制器中特定外设的时钟源。与 HAL_RCC_OscConfig()函数和 HAL_RCC_ClockConfig()函数专注于系统级别的时钟配置，HAL_RCCEx_PeriphCLKConfig()函数则致力于配置那些不直接从系统时钟派生的外设时钟，如 RTC、USB、SAI、ADC 及 LPTIM 等。

(1) 函数原型。

```
HAL_StatusTypeDef HAL_RCCEx_PeriphCLKConfig(RCC_PeriphCLKInitTypeDef * PeriphClkInit)
```

(2) 函数参数。

该函数以一个指向 RCC_PeriphCLKInitTypeDef 结构体的指针作为参数，这个结构体包含了所有需要配置的外设时钟信息。

```
typedef struct
{
  uint32_t PeriphClockSelection;                  // 指定要配置的外设时钟类型
  uint32_t RTCClockSelection;                     // RTC 时钟选择
  uint32_t USBClockSelection;                     // USB 时钟选择
  uint32_t ADCClockSelection;                     // ADC 时钟选择
  uint32_t I2SClockSelection;                     // I2S 时钟选择
  uint32_t SAI1ClockSelection;                    // SAI1 时钟选择
  uint32_t LPTIM1ClockSelection;                  // LPTIM1 时钟选择
  uint32_t LPUART1ClockSelection;                 // LPUART1 时钟选择
  uint32_t FDCANClockSelection;                   // FDCAN 时钟选择
  // ... 其他可能的外设时钟选择 ...
} RCC_PeriphCLKInitTypeDef;
```

① PeriphClockSelection：指定要配置的外设时钟类型，可以是以下之一或其组合。

a. RCC_PERIPHCLK_RTC：配置 RTC 时钟。

b. RCC_PERIPHCLK_USB：配置 USB 时钟。

c. RCC_PERIPHCLK_ADC：配置 ADC 时钟。

d. RCC_PERIPHCLK_I2S：配置 I2S 时钟。

e. RCC_PERIPHCLK_SAI1：配置 SAI1 时钟。

f. RCC_PERIPHCLK_LPTIM1：配置 LPTIM1 时钟。

g. RCC_PERIPHCLK_LPUART1：配置 LPUART1 时钟。

h. RCC_PERIPHCLK_FDCAN：配置 FDCAN 时钟。

② RTCClockSelection：可以选择 LSE、LSI 或 HSE 分频后的时钟作为 RTC 的时钟源。

③ USBClockSelection：可以选择 PLL 的 Q 输出、HSI48 等作为 USB 的时钟源。

④ ADCClockSelection：可以选择 SYSCLK、PLL 的 P 输出等作为 ADC 的时钟源。

类似地，其他成员也对应着各自外设的时钟源选择。

(3) 返回值。

① HAL_OK：表示操作成功完成。

② HAL_ERROR：表示配置过程中出现了错误。

③ HAL_BUSY：表示当前有一个正在进行的操作阻止了此次配置。

④ HAL_TIMEOUT：表示等待某个事件(如 PLL 锁定)超时。

(4) 使用实例。

若要配置特定外设的时钟源，需声明并初始化一个 RCC_PeriphCLKInitTypeDef 类型的变量，依据需求设置其成员变量。随后，将该结构体作为参数传递给 HAL_RCCEx_PeriphCLKConfig()函数，以实施新的时钟配置。例如，配置 USB 和 RTC 的时钟。

```
RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = {0};
// 初始化结构体成员
PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_USB | RCC_PERIPHCLK_RTC;
PeriphClkInitStruct.UsbClockSelection = RCC_USBCLKSOURCE_PLL;  // 选择 PLL 作为 USB 时钟源
PeriphClkInitStruct.RTCClockSelection = RCC_RTCCLKSOURCE_LSE;  // 选择 LSE 作为 RTC 时钟源
// 调用 HAL_RCCEx_PeriphCLKConfig 应用配置
if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct) != HAL_OK)
{
  // 错误处理
}
```

这段代码配置了 USB 时钟源为 PLL 输出，并且选择 LSE 作为 RTC 时钟源。这确保了相关外设能够按照预期的方式工作。

4. 时钟附加函数

HAL_RCC_DeInit()为时钟去初始化函数，它能将时钟配置恢复到复位状态。

一组宏定义在 stm32g4xx_hal_rcc.h 和 stm32g4xx_hal_rcc_ex.h 中。这些宏允许执行 RCC 模块寄存器上的基本操作，比如外设时钟使能/禁止、复位控制等。

__HAL_PPP_CLK_ENABLE：用于使能外设时钟。

__HAL_PPP_CLK_DISABLE：用于禁止外设时钟。

__HAL_PPP_FORCE_RESET：用于强制外设复位。

__HAL_PPP_RELEASE_RESET：用于释放外设复位。

__HAL_PPP_CLK_SLEEP_ENABLE：用于在睡眠模式期间使能外设时钟。

__HAL_PPP_CLK_SLEEP_DISABLE：用于在睡眠模式期间禁止外设时钟。

__HAL_PPP_IS_CLK_ENABLED：查询外设时钟是否处于使能状态。

__HAL_PPP_IS_CLK_DISABLED：查询外设时钟是否处于禁止状态。

__HAL_PPP_IS_CLK_SLEEP_ENABLED：查询外设时钟在睡眠模式下是否被使能。

__HAL_PPP_IS_CLK_SLEEP_DISABLED：查询外设时钟在睡眠模式下是否被禁止。

以上详细介绍了 STM32G4 系列微控制器中配置和管理时钟系统的常用方法，具体涵盖了时钟源的选择、配置步骤及代码实例中涉及的函数和宏。这些函数和宏为开发者提供了灵活且强大的工具，使其能够根据特定需求定制应用，并优化性能。

思考题

1. 在 STM32G4 系列微控制器中，如何使用 HAL 库初始化 GPIO 端口？请简述步骤。

2. 什么是复用功能(Alternate Function)？试举例说明其应用场景。

3. 在嵌入式系统中，为什么需要使用中断？列举至少两种场景，并说明如何在 STM32G4 微控制器中配置外部中断。

4. 编写一个程序：使用 HAL 库初始化 GPIOA 第 5 引脚为推挽输出模式，并使其周期性地切换该引脚的状态来控制一个 LED 闪烁。

5. 编写一个程序：使用 HAL 库初始化 GPIOB 第 6 引脚为输入模式，并读取该引脚的状态，然后通过串口打印出来。