AN4088
应用笔记
从 STM32F1 移植到 STM32F0

简介

对于 STM32 微控制器应用的设计人员而言，将一种微控制器类型轻松替换成同一产品系列的另一种微控制器非常重要。随着产品要求不断提高，对存储器大小或 I/O 数量的需求也相应增加，因此设计人员经常需要将应用程序移植到其它微控制器。另一方面，为了降低成本，用户可能被迫转换为更小的元件并缩减 PCB 面积。

本应用笔记旨在帮助您分析从现有的 STM32F1 器件移植到 STM32F0 器件所需的步骤。本文档收集了最重要的信息，并列出了需要注意的重要事项。

要将应用程序从 STM32F1 系列移植到 STM32F0 系列，用户需要分析硬件移植、外设移植和固件移植。

为了充分利用本应用笔记中的信息，用户应熟悉 STM32 微控制器系列。可以参考 www.st.com 网站提供的以下文档。

- STM32F1 系列参考手册（RM0008 和 RM0041）、STM32F1 数据手册和 STM32F1 Flash 编程手册（PM0075、PM0063 和 PM0068）。
- STM32F0 系列参考手册 (RM0091) 和 STM32F0 数据手册。

有关整个 STM32 系列的概述以及各 STM32 产品系列不同特性的对比，请参见 AN3364 *STM32 微控制器应用的移植和兼容性指南*。

表 1 列出了本应用笔记所涉及的微控制器和开发工具。

表 1. 适用产品

类型	型号
微控制器	STM32F0xxxx STM32F1xxxx

目录

表格索引

1 硬件移植

入门级 STM32F0 与通用 STM32F1xxx 系列的各引脚兼容。所有外设共用这两个产品系列的相同引脚，但二者在封装上存在微小差别。从 STM32F1 系列转换到 STM32F0 系列非常简单，因为只有少数引脚受到影响（*表 2* 中用粗体指出了受影响的引脚）。

表 2. STM32F1 系列和 STM32F0 系列引脚排列区别

STM32F1 系列			STM32F0 系列		
QFP48	**QFP64**	**引脚排列**	**QFP48**	**QFP64**	**引脚排列**
5	5	**PD0** - OSC_IN	5	5	**PF0** - OSC_IN
6	6	**PD1** - OSC_OUT	6	6	**PF1** - OSC_OUT
-	18	**VSS_4**	-	18	**PF4**
-	19	**VDD_4**	-	19	**PF5**
35	47	**VSS_2**	35	47	**PF6**
36	48	**VDD_2**	36	48	**PF7**
20	28	**BOOT1**/PB2	20	28	**PB2/NPOR**

注： *PB2 适用于 STM32F05x，而 NPOR 适用于 STM32F06x。*

除非用户为应用程序在 VSS/VDD 2 和 4 位置处使用了 2 个或 4 个 GPIO，否则从 F1 移植到 F0 时不会影响引脚排列，具体取决于使用的封装。

2 启动模式兼容性

F0 系列与 F1 系列上启动模式的选择方式不同。F0 不再使用两个引脚进行此设置，而是从存储器地址 0x1FFFF800 处的用户选项字节的选项位来获取 nBOOT1 值。利用 BOOT0 引脚，F0 系列可以选择与主 Flash、SRAM 或系统存储器对应的启动模式。*表 3* 中汇总了用于选择启动模式的不同配置。

表 3. 启动模式

F0/F1 启动模式选择		启动模式	别名使用
BOOT1	BOOT0		
x	0	主 Flash	选择主 Flash 作为启动空间
0	1	系统存储器	选择系统存储器作为启动空间
1	1	嵌入式 SRAM	选择嵌入式 SRAM 作为启动空间

注： *此 BOOT1 值与 nBOOT1 选项位的值相反。*

3 外设移植

如*表 3*所示，共有三类外设。除非外设实例不复存在，否则，无需任何修改便可通过专用固件库支持通用外设。当然，用户可以更改实例和所有相关特性（时钟配置、引脚配置、中断/DMA 请求）。

ADC、RCC 和 RTC 等已修改的外设与 F1 系列的对应外设有所不同，因此，应更新这些外设以便有效利用 F0 系列提供的增强功能和全新特性。

在 F0 系列中，上述已修改的所有外设均经过改进，有效减小了硅片面积，从而可为经济型最终产品带来先进的高端功能，并修正 F1 系列存在的某些局限。

3.1 STM32 产品交叉兼容性

STM32 系列内置一组外设，这组外设可分为三类：

- 第一类是根据定义适用于所有产品的外设。这些外设都相同，因此它们具有相同的结构、寄存器和控制位。移植后，无需进行任何固件更改，便可在应用程序级别上保持相同的功能。所有特性和行为均保持不变。
- 第二类是指由所有产品共用的外设，但其中存在微小差别（通常是对新特性的支持）。从一个产品移植到另一个产品的过程非常简单，无需进行大量新的开发工作。
- 第三类是指从一个产品移植到另一个产品后发生显著变化的外设（新架构、新特性 ...）。对于这类外设，若要进行移植，需要在应用程序级别进行全新开发。

*表 4*给出了这种分类的总览。

表 4. STM32 F1 与 F0 系列外设兼容性分析对比

外设	F1 系列	F0 系列	兼容性		
			特性	引脚排列	固件驱动程序
SPI	有	有++	提供两个 FIFO，4 位到 16 位数据大小可供选择	相同	部分兼容
WWDG	有	有	特性相同	NA	完全兼容
IWDG	有	有+	增加了窗口模式	NA	完全兼容
DBGMCU	有	有	无 JTAG，无跟踪	与 SWD 相同	部分兼容
CRC	有	有++	增加了反转功能和初始 CRC 值	NA	部分兼容
EXTI	有	有+	某些外设能够在停止模式下生成事件	相同	完全兼容
CEC	有	有++	内核时钟，仲裁丢失标志和自动发送重试，多地址配置，从停止模式唤醒	相同	部分兼容
DMA	有	有	1 个具有 5 个通道的 DMA 控制器	NA	完全兼容
TIM	有	有+	增强	相同	完全兼容

表 4. STM32 F1 与 F0 系列外设兼容性分析对比（续）

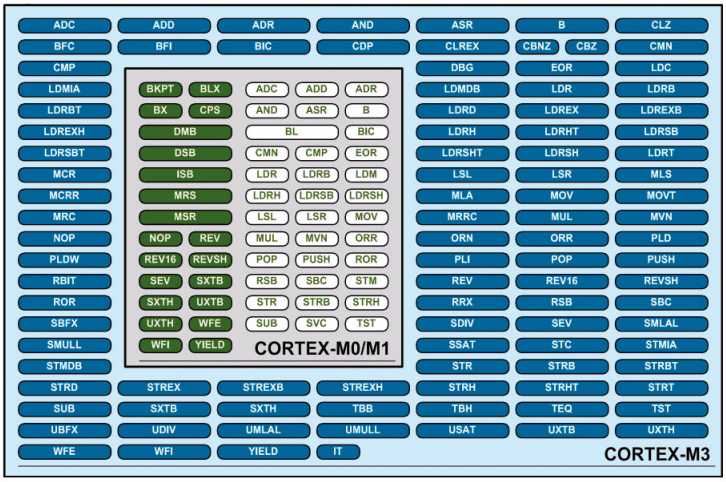
外设	F1 系列	F0 系列	兼容性		
			特性	引脚排列	固件驱动程序
PWR	有	有+	无 Vref，Vdda 可以大于 Vdd，内核采用 1.8 模式。	对于同一特性相同	部分兼容
RCC	有	有+	专用于 ADC 的新 HSI14	对于振荡器，PD0 & PD1 => PF0 & PF1	部分兼容
USART	有	有+	独立时钟源选择，超时特性，从停止模式唤醒	相同	完全兼容
I2C	有	有++	由硬件管理通信事件，FM+，从停止模式唤醒，数字滤波器	相同	新驱动程序
DAC	有	有+	DMA 下溢中断	相同	完全兼容
ADC	有	有++	模拟部分相同，但有新的数字接口	相同	部分兼容
RTC	有	有++	亚秒精度，数字校准电路，用于事件保存的时间戳功能，可编程报警	对于同一特性相同	新驱动程序
FLASH	有	有+	选项字节已修改	NA	部分兼容
GPIO	有	有++	新外设	4 个新 GPIO	部分兼容
CAN	有	NA	NA	NA	NA
USB FS 设备	有	NA	NA	NA	NA
以太网	有	NA	NA	NA	NA
SDIO	有	NA	NA	NA	NA
FSMC	有	NA	NA	NA	NA
触摸感应	NA	有	NA	NA	NA
COMP	NA	有	NA	NA	NA
SYSCFG	NA	有	NA	NA	NA

注： *有++ = 新特性或新架构*
有+ = 新特性，但技术参数发生更改或得到增强
有 = 特性可用
NA = 特性不可用

3.2 系统架构

STM32F0 MCU 系列具有低功耗和操作简单的特点，主要面向入门级市场。为实现这一目的并保留 STM32 特有的先进高端特性，这一系列产品的内核已更换为 Cortex-M0。F0 系列产品的硅片面积小并且代码量极少，能够在低成本应用中实现 32 位的高性能。*图 1* 介绍了 M3 和 M0 指令集之间的对应关系。为了避免用到无法使用的特性，从 F1 移植到 F0 时需要重新编译代码。

图 1. 系统架构



MCU 的结构也做出了重大修改，从哈佛结构切换为冯 · 诺依曼结构，这降低了系统复杂性，也就是说，更注重软件调试而降低了系统的精密程度。

3.3 存储器映射

相较 F1 系列而言，F0 系列中的外设地址映射已发生变化。主要变化体现在 GPIO 已从 APB 总线转移到 AHB 总线，这样这些 GPIO 便可在最大速度下工作。

表 5 介绍了 F0 和 F1 系列之间的外设地址映射对应关系。

表 5. STM32F0 和 STM32F1 系列之间的 IP 总线映射区别

外设	STM32 F0 系列		STM32 F1 系列	
	总线	基址	总线	基址
TSC	AHB1	0x40024000	NA	NA
CRC		0x40023000	AHB	0x40023000
FLITF		0x40022000		0x40022000
RCC		0x40021000		0x40021000
DMA1/DMA		0x40020000		0x40020000

表 5. STM32F0 和 STM32F1 系列之间的 IP 总线映射区别（续）

外设	STM32 F0 系列		STM32 F1 系列	
	总线	基址	总线	基址
GPIOF	AHB2	0x48001400	APB2	0x40011800
GPIOD		0x48000C00		0x40011400
GPIOC		0x48000800		0x40011000
GPIOB		0x48000400		0x40010C00
GPIOA		0x48000000		0x40010800
DBGMCU	APB2	0x40015800	NA	NA
TIM17		0x40014800	NA	NA
TIM16		0x40014400	NA	NA
TIM15		0x40014000	NA	NA
USART1		0x40013800	APB2	0x40013800
SPI1/I2S1		0x40013000		0x40013000
TIM1		0x40012C00		0x40012C00
ADC/ADC1		0x40012400		0x40012400
EXTI	APB2（通过 SYSCFG）	0x40010400		0x40010400
SYSCFG + COMP	APB2	0x40010000	NA	NA
CEC	APB1	0x40007800	APB1	0x40007800
DAC		0x40007400		0x40007400
PWR		0x40007000		0x40007000
I2C2		0x40005800		0x40005800
I2C1		0x40005400		0x40005400
USART2		0x40004400		0x40004400
SPI2		0x40003800		0x40003800
IWWDG/IWDG	自有时钟	0x40003000		0x40003000
WWDG	APB1	0x40002C00		0x40002C00
RTC	APB1（通过 PWR）	0x40002800（包括 BKP 寄存器）		0x40002800

表 5. STM32F0 和 STM32F1 系列之间的 IP 总线映射区别（续）

外设	STM32 F0 系列		STM32 F1 系列	
	总线	基址	总线	基址
TIM14	APB1	0x40002000	NA	NA
TIM6		0x40001000	APB1	0x40001000
TIM3		0x40000400		0x40000400
TIM2		0x40000000		0x40000000
USB 设备 FS SRAM	NA	NA	APB1	0x40006000
USB 设备 FS	NA	NA		0x40005C00
USART3	NA	NA		0x40004800
TIM7	NA	NA		0x40001400
TIM4	NA	NA		0x40000800
FSMC 寄存器	NA	NA	AHB	0xA0000000
USB OTG FS	NA	NA		0x50000000
ETHERNET MAC	NA	NA		0x40028000
DMA2	NA	NA		0x40020400
GPIOG	NA	NA	APB2	0x40012000
SDIO	NA	NA	AHB	0x40018000
TIM11	NA	NA	APB2	0x40015400
TIM10	NA	NA		0x40015000
TIM9	NA	NA		0x40014C00
ADC2	NA	NA		0x40012800
ADC3	NA	NA		0x40013C00
TIM8	NA	NA		0x40013400

表 5. STM32F0 和 STM32F1 系列之间的 IP 总线映射区别（续）

外设	STM32 F0 系列		STM32 F1 系列	
	总线	基址	总线	基址
CAN2	NA	NA	APB1	0x40006800
CAN1	NA	NA		0x40006400
UART5	NA	NA		0x40005000
UART4	NA	NA		0x40004C00
SPI3/I2S3	NA	NA		0x40003C00
TIM13	NA	NA		0x40001C00
TIM12	NA	NA		0x40001800
TIM5	NA	NA		0x40000C00
BKP 寄存器	NA	NA		0x40006C00
AFIO	NA	NA	APB2	0x40010000

注： *NA = 特性不可用。*

F0 设备只有一个 APB 总线，APB1 和 APB2 指示这些外设的时钟配置位在哪个 APB 寄存器上定义。

3.4 复位和时钟控制器 (RCC) 接口

表 6 介绍了 STM32F0 系列与 STM32F1 系列的 RCC（复位和时钟控制器）的主要区别。

表 6. STM32F0 系列与 STM32F1 系列之间的 RCC 区别

RCC	STM32 F1 系列	STM32 F0 系列
HSI 14	NA	专用于 ADC 的高速内部振荡器
HSI	经工厂调校的 8 MHz RC 振荡器	相似
LSI	40 KHz RC	相似
HSE	3 - 25 MHz，具体取决于所使用的产品线	4 - 32 MHz
LSE	32.768 KHz	相似
PLL	- 互连型：主 PLL + 2 个 PLL，用于 I2S、以太网和 OTG FS 时钟 - 其它产品线：主 PLL	主 PLL
系统时钟源	HSI、HSE 或 PLL	相似

表 6. STM32F0 系列与 STM32F1 系列之间的 RCC 区别（续）

RCC	STM32 F1 系列	STM32 F0 系列
系统时钟频率	- 高达 72 MHz，具体取决于所使用的产品线 - 使用 HSI 复位后为 8 MHz	高达 48 MHz
APB1/APB 频率	高达 36 MHz	高达 48 MHz
RTC 时钟源	LSI、LSE 或 HSE/128	LSI、LSE 或 HSE 时钟 32 分频
MCO 时钟源 MCO 引脚：(PA8)	- 互连型：HSI、HSE、PLL/2、SYSCLK、PLL2、PLL3 或 XT1 - 其它产品线：HSI、HSE、PLL/2 或 SYSCLK	SYSCLK、HSI、HSE、HSI14、PLLCLK/2、LSE、LSI
内部振荡器测量/校准	连接到 TIM5 CH4 IC 的 LSI：可相对于 HSI/HSE 时钟测量 LSI	- LSE 和 LSI 时钟由 HSI/HSE 时钟对应的定时器 TIM14 通过 MCO 间接测量 - HSI14/HSE 由 HSI 时钟对应的 TIM14 通道 1 输入捕捉通过 MCO 间接测量。

除了上表介绍的区别外，移植时还需以下附加调整步骤。

1. *系统时钟配置*：从 F1 系列移植到 F0 系列时，只需更新系统时钟配置代码中的几项设置；主要是 Flash 设置（配置适合系统频率的正确等待周期、使能/禁止预取）和/或 PLL 参数配置：
   a) 当 HSE 或 HSI 直接用作系统时钟源时，只需修改 Flash 参数。
   b) 当 PLL（由 HSE 或 HSI 提供时钟）用作系统时钟源时，需要更新 Flash 参数和 PLL 配置。

下面的*表 7* 列出了一个将系统时钟配置从 F1 移植到 F0 的示例：

- STM32F100x 超值型运行在最高性能下：系统时钟为 24 MHz（由 HSE (8 MHz) 驱动的 PLL 用作系统时钟源），Flash 等待周期为 0 且已使能 Flash 预取队列。
- F0 系列运行在最高性能下：系统时钟频率为 48 MHz（由 HSE (8 MHz) 驱动的 PLL 用作系统时钟源），Flash 等待周期为 1 且已使能 Flash 预取。

如*表 7* 所示，若要在 F0 系列上运行，只需重新写入 Flash 设置和 PLL 参数（代码用***粗斜体***表示）。不过，在 F0 系列中，HSE、AHB 预分频器和系统时钟源配置保持不变，APB 预分频器会调整到最大 APB 频率。

注： *1* *表 7 中给出的源代码已经过有意简化（等待循环中的超时已删除），这些代码基于 RCC 和 Flash 寄存器保持其复位值的假设。*

2 *对于 STM32F0xx，用户可根据应用程序要求，使用时钟配置工具 STM32F0xx_Clock_Configuration.xls 生成一个自定义 system_stm32f0xx.c 文件，其中包含系统时钟配置路径。*

表 7. 将系统时钟配置代码从 F1 移植到 F0 的示例

STM32F100x 超值型运行在 24 MHz（PLL 为时钟源）下，等待周期为 0	STM32F0xx 运行在 48 MHz（PLL 为时钟源）下，等待周期为 1

```
/* 使能 HSE ------------------------------*/
  RCC->CR |= ((uint32_t)RCC_CR_HSEON);

  /* 等待 HSE 就绪 */
  while((RCC->CR & RCC_CR_HSERDY) == 0)
  {
  }

  /* Flash 配置 --------------------*/
  /* 开启预取，Flash 0 等待周期 */
  FLASH->ACR |= FLASH_ACR_PRFTBE | FLASH_ACR_LATENCY_0;

  /* AHB 和 APB 预分频器配置 --*/
  /* HCLK = SYSCLK */
  RCC->CFGR |= (uint32_t)RCC_CFGR_HPRE_DIV1;

  /* PCLK2 = HCLK */
  RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE2_DIV1;

  /* PCLK1 = HCLK */
  RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE1_DIV1;


  /* PLL 配置 = (HSE/2) * 6 = 24 MHz */
  RCC->CFGR |= (uint32_t)(RCC_CFGR_PLLSRC_PREDIV1 |
RCC_CFGR_PLLXTPRE_PREDIV1_Div2 | RCC_CFGR_PLLMULL6);

  /* 使能 PLL */
  RCC->CR |= RCC_CR_PLLON;

  /* 等待 PLL 就绪 */
  while((RCC->CR & RCC_CR_PLLRDY) == 0)
  {
  }

  /* 选择 PLL 作为系统时钟源 ----*/
  RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_SW));
  RCC->CFGR |= (uint32_t)RCC_CFGR_SW_PLL;

  /* 等待 PLL 用作系统时钟源 */
  while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS) !=
(uint32_t)0x08)
  {
  }
```

```
/* 使能 HSE ------------------------------*/
 RCC->CR |= ((uint32_t)RCC_CR_HSEON);

 /* 等待 HSE 就绪 */
 while((RCC->CR & RCC_CR_HSERDY) == 0)
 {
 }

 /* Flash 配置 --------------------*/
 /* 开启预取，Flash 1 等待周期 */
 FLASH->ACR |= FLASH_ACR_PRFTBE | FLASH_ACR_LATENCY;

 /* AHB 和 APB 预分频器配置 --*/
 /* HCLK = SYSCLK */
 RCC->CFGR |= (uint32_t)RCC_CFGR_HPRE_DIV1;

 /* PCLK = HCLK */
 RCC->CFGR |= (uint32_t)RCC_CFGR_PPRE_DIV1;

 /* PLL 配置 = HSE * 6 = 48 MHz -*/
  RCC->CFGR |= (uint32_t)(RCC_CFGR_PLLSRC_PREDIV1 |
RCC_CFGR_PLLXTPRE_PREDIV1 | RCC_CFGR_PLLMULL6);

 /* 使能 PLL */
 RCC->CR |= RCC_CR_PLLON;

 /* 等待 PLL 就绪 */
 while((RCC->CR & RCC_CR_PLLRDY) == 0)
 {
 }

 /* 选择 PLL 作为系统时钟源 ----*/
 RCC->CFGR |= (uint32_t)RCC_CFGR_SW_PLL;

 /* 等待 PLL 用作系统时钟源 */
  while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS) !=
(uint32_t)RCC_CFGR_SWS_PLL)
 {
 }
```

2. *外设访问配置*：相对 F1 系列而言，由于 F0 系列中的某些外设地址映射已发生更改，因此用户需要使用不同的寄存器来 [使能/禁止] 外设 [时钟] 或使外设 [进入/退出] [复位模式]。

表 8. 用于外设访问配置的 RCC 寄存器

总线	寄存器	注释
AHB	RCC_AHBRSTR	用于使 AHB 外设 [进入/退出] 复位模式
	RCC_AHBENR	用于 [使能/禁止] AHB 外设时钟
APB1	RCC_APB1RSTR	用于使 APB1 外设 [进入/退出] 复位模式
	RCC_APB1ENR	用于 [使能/禁止] APB1 外设时钟
APB2	RCC_APB2RSTR	用于使 APB2 外设 [进入/退出] 复位模式
	RCC_APB2ENR	用于 [使能/禁止] APB2 外设时钟

要配置对指定外设的访问，用户首先应了解与该外设相连的总线；请参见*表 5*，然后根据所需操作对上面*表 8* 中介绍的相应寄存器进行编程。例如，如果 USART1 连接到 APB2 总线，则需要按照如下方式配置 APB2ENR 寄存器才能使能 USART1 时钟：

```
RCC->APB2ENR |= RCC_APB2ENR_USART1EN;
```

3. *外设时钟配置*：某些外设具备独立于系统时钟的专用时钟源，用于生成操作所需的时钟：
   a) *ADC：*在 STM32F0 系列中，ADC 配有两个可用时钟源：
   – 第一个时钟源基于 PCLK；在频率达到 ADC 前，用户可利用预分频器按分频系数 2 或 4 降低 ADC 输入频率。
   – 第二个时钟源是 Stingray 上的全新特性；这款芯片上集成了可用于 ADC 输入频率的专用 14 MHz 振荡器 (HSI14)。
   b) *RTC：*在 STM32F0 系列中，RTC 配有三个可用时钟源：
   – 第一个时钟源基于 HSE 时钟；在频率达到 RTC 前，用户可利用预分频器将其 32 分频。
   – 第二个时钟源是 LSE 振荡器。
   – 第三个时钟源是频率为 40 KHz 的 LSI RC。

3.5 DMA 接口

STM32F1 和 STM32F0 系列使用相同且完全兼容的 DMA 控制器。

STM32F0 系列使用一个 5 通道 DMA 控制器，STM32F1 则使用两个。每个通道专用于管理来自一个或多个外设的存储器访问请求。

下表介绍了 STM32F1 系列与 STM32F0 系列中外设的 DMA 请求之间的对应关系。

表 9. STM32F1 系列与 STM32F0 系列之间的 DMA 请求区别

外设	DMA 请求	STM32F1 系列	STM32F0 系列
ADC1/ADC	**ADC1/ADC**	DMA1_Channel1	DMA_Channel1 DMA_Channel2
ADC3	**ADC3**	DMA2_Channel5	NA
DAC	**DAC_Channel1/ DAC** **DAC_Channel2**	DMA2_Channel3/DMA1_Channel3(1) DMA2_Channel4/DMA1_Channel4(1)	DMA_Channel3
SPI1	**SPI1_Rx** **SPI1_Tx**	DMA1_Channel2 DMA1_Channel3	DMA_Channel2 DMA_Channel3
SPI2	**SPI2_Rx** **SPI2_Tx**	DMA1_Channel4 DMA1_Channel5	DMA_Channel4 DMA_Channel5
SPI3	**SPI3_Rx** **SPI3_Tx**	DMA2_Channel1 DMA2_Channel2	NA
USART1	**USART1_Rx** **USART1_Tx**	DMA1_Channel5 DMA1_Channel4	DMA_Channel3/DMA_Channel5 DMA_Channel2/DMA_Channel4
USART2	**USART2_Rx** **USART2_Tx**	DMA1_Channel6 DMA1_Channel7	DMA_Channel5 DMA_Channel4
USART3	**USART3_Rx** **USART3_Tx**	DMA1_Channel3 DMA1_Channel2	NA
UART4	**UART4_Rx** **UART4_Tx**	DMA2_Channel3 DMA2_Channel5	NA
UART5	**UART5_Rx** **UART5_Tx**	DMA2_Channel4 DMA2_Channel1	NA
I2C1	**I2C1_Rx** **I2C1_Tx**	DMA1_Channel7 DMA1_Channel6	DMA_Channel3 DMA_Channel2
I2C2	**I2C2_Rx** **I2C2_Tx**	DMA1_Channel5 DMA1_Channel4	DMA_Channel5 DMA_Channel4
SDIO	**SDIO**	DMA2_Channel4	NA
TIM1	**TIM1_UP** **TIM1_CH1** **TIM1_CH2** **TIM1_CH3** **TIM1_CH4** **TIM1_TRIG** **TIM1_COM**	DMA1_Channel5 DMA1_Channel2 DMA1_Channel3 DMA1_Channel6 DMA1_Channel4 DMA1_Channel4 DMA1_Channel4	DMA_Channel5 DMA_Channel2 DMA_Channel3 DMA_Channel5 DMA_Channel4 DMA_Channel4 DMA_Channel4

表 9. STM32F1 系列与 STM32F0 系列之间的 DMA 请求区别（续）

外设	DMA 请求	STM32F1 系列	STM32F0 系列
TIM8	**TIM8_UP** **TIM8_CH1** **TIM8_CH2** **TIM8_CH3** **TIM8_CH4** **TIM8_TRIG** **TIM8_COM**	DMA2_Channel1 DMA2_Channel3 DMA2_Channel5 DMA2_Channel1 DMA2_Channel2 DMA2_Channel2 DMA2_Channel2	NA
TIM2	**TIM2_UP** **TIM2_CH1** **TIM2_CH2** **TIM2_CH3** **TIM2_CH4**	DMA1_Channel2 DMA1_Channel5 DMA1_Channel7 DMA1_Channel1 DMA1_Channel7	DMA_Channel2 DMA_Channel5 DMA_Channel3 DMA_Channel1 DMA_Channel4
TIM3	**TIM3_UP** **TIM3_CH1** **TIM3_TRIG** **TIM3_CH3** **TIM3_CH4**	DMA1_Channel3 DMA1_Channel6 DMA1_Channel6 DMA1_Channel2 DMA1_Channel3	DMA_Channel3 DMA_Channel4 DMA_Channel4 DMA_Channel2 DMA_Channel3
TIM4	**TIM4_UP** **TIM4_CH1** **TIM4_CH2** **TIM4_CH3**	DMA1_Channel7 DMA1_Channel1 DMA1_Channel4 DMA1_Channel5	NA
TIM5	**TIM5_UP** **TIM5_CH1** **TIM5_CH2** **TIM5_CH3** **TIM5_CH4** **TIM5_TRIG**	DMA2_Channel2 DMA2_Channel5 DMA2_Channel4 DMA2_Channel2 DMA2_Channel1 DMA2_Channel1	NA
TIM6/DAC	**TIM6_UP**	DMA2_Channel3/DMA1_Channel3(1)	DMA_Channel3
TIM7	**TIM7_UP**	DMA2_Channel4/DMA1_Channel4 (1)	NA
TIM15	**TIM15_UP** **TIM15_CH1** **TIM15_TRIG** **TIM15_COM**	DMA1_Channel5 DMA1_Channel5 DMA1_Channel5 DMA1_Channel5	DMA_Channel5 DMA_Channel5 DMA_Channel5 DMA_Channel5
TIM16	**TIM16_UP** **TIM16_CH1**	DMA1_Channel6 DMA1_Channel6	DMA_Channel3/DMA_Channel4 DMA_Channel3/DMA_Channel4
TIM17	**TIM17_UP** **TIM17_CH1**	DMA1_Channel7 DMA1_Channel7	DMA_Channel1/DMA_Channel2 DMA_Channel1/DMA_Channel2

1. 对于大容量超值型器件，DAC DMA 请求分别映射到 DMA1 通道 3 和 DMA1 通道 4。

3.6 中断向量

表 10 介绍了 STM32F0 系列与 STM32F1 系列中中断向量的对应关系。

从 Cortex-M3 切换到 Cortex-M0 时会导致向量表减少。这会导致这两类器件之间出现许多差异。

表 10. STM32F1 系列与 STM32F0 系列之间的中断向量区别

位置	STM32F1 系列	STM32F0 系列
0	WWDG	WWDG
1	PVD	PVD
2	TAMPER	RTC
3	RTC	FLASH
4	FLASH	RCC
5	RCC	EXTI0_1
6	EXTI0	EXTI2_3
7	EXTI1	EXTI4_15
8	EXTI2	TSC
9	EXTI3	DMA_CH1
10	EXTI4	DMA_CH2_CH3
11	DMA1_Channel1	DMA_CH4_CH5
12	DMA1_Channel2	ADD_COMP
13	DMA1_Channel3	TIM1_BRK_UP_TRG_COM
14	DMA1_Channel4	TIM1_CC
15	DMA1_Channel5	TIM2
16	DMA1_Channel6	TIM3
17	DMA1_Channel7	TIM6_DAC
18	ADC1_2	保留
19	CAN1_TX/USB_HP_CAN_TX	TIM14
20	CAN1_RX0/USB_LP_CAN_RX0	TIM15
21	CAN1_RX1	TIM16
22	CAN1_SCE	TIM17
23	EXTI9_5	I2C1
24	TIM1_BRK/TIM1_BRK _TIM9	I2C2
25	TIM1_UP/TIM1_UP_TIM10	SPI1
26	TIM1_TRG_COM/TIM1_TRG_COM_TIM11	SPI2
27	TIM1_CC	USART1
28	TIM2	USART2

表 10. STM32F1 系列与 STM32F0 系列之间的中断向量区别（续）

位置	STM32F1 系列	STM32F0 系列
29	TIM3	保留
30	TIM4	CEC
31	I2C1_EV	保留
32	I2C1_ER	NA
33	I2C2_EV	NA
34	I2C2_ER	NA
35	SPI1	NA
36	SPI2	NA
37	USART1	NA
38	USART2	NA
39	USART3	NA
40	EXTI15_10	NA
41	RTC_Alarm	NA
42	OTG_FS_WKUP/USBWakeUp	NA
43	TIM8_BRK/TIM8_BRK_TIM12 (1)	NA
44	TIM8_UP/TIM8_UP_TIM13 (1)	NA
45	TIM8_TRG_COM/TIM8_TRG_COM_TIM14 (1)	NA
46	TIM8_CC	NA
47	ADC3	NA
48	FSMC	NA
49	SDIO	NA
50	TIM5	NA
51	SPI3	NA
52	UART4	NA
53	UART5	NA
54	TIM6	NA
55	TIM7	NA
56	DMA2_Channel1	NA
57	DMA2_Channel2	NA
58	DMA2_Channel3	NA
59	DMA2_Channel4/DMA2_Channel4_5 (1)	NA
60	DMA2_Channel5	NA
61	ETH	NA
62	ETH_WKUP	NA

表 10. STM32F1 系列与 STM32F0 系列之间的中断向量区别（续）

位置	STM32F1 系列	STM32F0 系列
63	CAN2_TX	NA
64	CAN2_RX01	NA
65	CAN2_RX1	NA
66	CAN2_SCE	NA
67	OTG_FS	NA

1. 具体取决于所使用的产品线。

Cortex M0 内核使用 2 个位来设置没有子优先级的中断优先级。用户可在嵌套向量中断控制器中定义 4 个优先级。F1 和 Cortex M3 内核使用 4 个位设置优先级，因此可以得到 16 个优先级。

3.7 GPIO 接口

与 F1 系列相比，STM32F0 GPIO 外设内置了多个新特性，主要包括：

- GPIO 映射到 AHB 总线上，可以获得更佳的性能。
- I/O 引脚复用器和映射：引脚通过多路复用器连接到片上外设 / 模块，该复用器一次只允许一个外设复用功能 (AF) 连接到 I/O 引脚。这样便可确保共用同一个 I/O 引脚的外设之间不会发生冲突。
- I/O 配置的方式和特性更加丰富。

F0 GPIO 外设是一项全新的设计，因此在结构、特性和寄存器方面均不同于 F1 系列中的 GPIO 外设。使用 GPIO 写入 F1 系列的任何代码都需要重写后才能在 F0 系列上运行。

有关 STM32F0 的 GPIO 编程和使用的详细信息，请参见 STM32F0xx 参考手册 (RM0091) 中 GPIO 一章的“I/O 引脚复用器和映射”部分。

下表介绍了 STM32F1 系列与 STM32F0 系列中 GPIO 之间的区别。

表 11. STM32F1 系列与 STM32F0 系列之间的 GPIO 区别

GPIO	STM32F1 系列	STM32F0 系列
输入模式	悬空 PU PD	悬空 PU PD
通用输出	PP OD	PP PP + PU PP + PD OD OD + PU OD + PD
复用功能输出	PP OD	PP PP + PU PP + PD OD OD + PU OD + PD

表 11. STM32F1 系列与 STM32F0 系列之间的 GPIO 区别（续）

GPIO	STM32F1 系列	STM32F0 系列
输入/输出	模拟	模拟
输出速度	2 MHz 10 MHz 50 MHz	2 MHz 10 MHz 48 MHz
复用功能选择	为针对不同器件封装优化外设 I/O 功能的数量，可以将某些复用功能重新映射到其它一些引脚上（软件重映射）。	引脚复用非常灵活，能够保证共用同一个 I/O 引脚的外设之间不会发生冲突。
最大 IO 切换频率	18 MHz	12 MHz

复用功能模式

STM32F1 系列

1. I/O 用作复用功能的配置取决于所使用的外设模式。例如，USART Tx 引脚应配置为复用功能推挽，而 USART Rx 引脚应配置为输入悬空或输入上拉。
2. 为针对不同器件封装（尤其是引脚数较少的器件）优化外设 I/O 功能的数量，可以用软件将某些复用功能重新映射到其它引脚上。例如，可将 USART2_RX 引脚映射到 PA3（默认重映射）或 PD6（软件重映射）上。

STM32F0 系列

1. 不论使用何种外设模式，都必须将 I/O 配置为复用功能，之后系统才能正确使用 I/O（输入或输出）。
2. I/O 引脚通过复用器连接到片上外设/模块，该复用器一次只允许一个外设的复用功能 (AF) 连接到 I/O 引脚。这样便可确保共用同一个 I/O 引脚的外设之间不会发生冲突。每个 I/O 引脚都有一个复用器，该复用器具有八路复用功能输入（AF0 到 AF7），可通过 GPIOx_AFRL 和 GPIOx_AFRH 寄存器对这些输入进行配置：
   – 通过配置 AF0 到 AF7 可以映射外设复用功能。
3. 除了这种灵活的 I/O 复用架构之外，各外设还具有映射到不同 I/O 引脚的复用功能，这可以针对不同器件封装优化外设 I/O 功能的数量。例如，可将 USART2_RX 引脚映射到 PA3 或 PA15 引脚上。

注： *有关系统和外设复用功能 I/O 引脚映射的详细信息，请参见 STM32F0x 数据手册中的“复用功能映射”表。*

4. 配置过程
   – 在 GPIOx_MODER 寄存器中将所需 I/O 配置为复用功能
   – 通过 GPIOx_OTYPER、GPIOx_PUPDR 和 GPIOx_OSPEEDER 寄存器，分别选择类型、上拉/下拉以及输出速度
   – 将 I/O 连接到 GPIOx_AFRL 或 GPIOx_AFRH 寄存器中所需的 AFx

3.8 EXTI 中断源选择

在 STM32F1 中，通过 AFIO_EXTICRx 寄存器的 EXTIx 位选择 EXTI 线源，而在 F0 系列中，通过 SYSCFG_EXTICRx 寄存器的 EXTIx 位完成这种选择。

只有 EXTICRx 寄存器的映射发生更改，而 EXTIx 位的含义则保持不变。不过，由于最后一个端口为 F，因此 EXTIx 位的值范围最大为 0b0101（在 F1 系列中，最大值为 0b0110）。

3.9 Flash 接口

下表介绍了 STM32F1 系列与 STM32F0 系列的 Flash 接口之间的区别，分组如下：

- 新接口，新技术
- 新架构
- 新的读保护机制，提供 3 个保护级别

因此，F0 Flash 编程过程和寄存器均不同于 F1 系列，而且写入 F1 系列 Flash 接口的任何代码都需要重写后才能在 F0 系列上运行。

表 12. STM32F0 系列与 STM32F1 系列之间的 Flash 区别

特性		STM32F1 系列	STM32F0 系列
主存储器/程序存储器	起始地址	0x0800 0000	0x0800 0000
	结束地址	最高 0x080F FFFF	最高 0x0800 FFFF
	粒度	页大小 = 2 KB 例外情况：小容量和中等容量器件的页大小 = 1 KB	64 个大小为 1 KB 的页
EEPROM 存储器	起始地址	可通过软件仿真获得	可通过软件仿真获得
	结束地址		
系统存储器	起始地址	0x1FFF F000	0x1FFF EC00
	结束地址	0x1FFF F7FF	0x1FFF F7FF
选项字节	起始地址	0x1FFF F800	0x1FFF F800
	结束地址	0x1FFF F80F	0x1FFF F80B
Flash 接口	起始地址	0x4002 2000	0x4002 2000
	编程过程	所有产品线均相同	与 F1 系列的 Flash 编程和擦除操作相同。与 F1 系列中选项字节编程的过程不同
读保护	无保护	禁止读保护 RDP = 0xA55A	级别 0，无保护 RDP = 0xAA
	保护	使能读保护 RDP != 0xA55A	级别 1，存储器保护 RDP !=（级别 2 和级别 0） 级别 2：级别 1 + 禁止调试， RDP = 0xCC

表 12. STM32F0 系列与 STM32F1 系列之间的 Flash 区别（续）

特性	STM32F1 系列	STM32F0 系列
写保护	由 4 KB 块提供保护	由 4 KB 块提供保护
用户选项字节	STOP	STOP
	STANDBY	STANDBY
	WDG	WDG
	NA	RAM_PARITY_CHECK
	NA	VDDA_MONITOR
	NA	nBOOT1
擦除粒度	页（1 或 2 KB）	页 (1 KB)
编程模式	半字（16 位）	半字（16 位）

3.10 ADC 接口

下表介绍了 STM32F1 系列与 STM32F0 系列的 ADC 接口之间的区别；具体区别如下：

- 新数字接口
- 新架构和新特性

表 13. STM32F0 系列与 STM32F1 系列之间的 ADC 区别

ADC	STM32F1 系列	STM32F0 系列
ADC 类型	SAR 结构	SAR 结构
实例	ADC1/ADC2/ADC3	ADC
最大采样频率	1 MSPS	1 MSPS
通道数	多达 21 个通道	多达 16 个通道 + 3 个内部通道
分辨率	12 位	12 位
转换模式	单一/连续/扫描/间断/双重模式	单一/连续/扫描/间断/ 双重模式/三重模式

表 13. STM32F0 系列与 STM32F1 系列之间的 ADC 区别（续）

ADC	STM32F1 系列		STM32F0 系列
DMA	有		有
外部触发器	有		有
	常规组的外部事件 对于 ADC1 和 ADC2： TIM1 CC1 TIM1 CC2 TIM1 CC3 TIM2 CC2 TIM3 TRGO TIM4 CC4 EXTI 线 11/ TIM8_TRGO 对于 ADC3： TIM3 CC1 TIM2 CC3 TIM1 CC3 TIM8 CC1 TIM8 TRGO TIM5 CC1	注入组的外部事件 对于 ADC1 和 ADC2： TIM1 TRGO TIM1 CC4 TIM2 TRGO TIM2 CC1 TIM3 CC4 TIM4 TRGO EXTI 线 15/ TIM8_CC4 对于 ADC3： TIM1 TRGO TIM1 CC4 TIM4 CC3 TIM8 CC2 TIM8 CC4 TIM5 TRGO	外部事件 TIM1_TRGO TIM1_CC4 TIM2_TRGO TIM3_TRGO TIM15_TRGO
电源要求	2.4 V 到 3.6 V		2.4 V 到 3.6 V(1)
输入范围	$V_{REF-} <= V_{IN} <= V_{REF+}$		Vdd 和 2.4 <= Vdda <= 3.6

1. ADC 由单独的 V_{DDA} 引脚供电。

3.11 PWR 接口

STM32F0 系列与 F1 系列中的 PWR 控制器存在区别，下表中概述了这些区别。但是，编程接口保持不变。

表 14. STM32F0 系列与 STM32F1 系列之间的 PWR 区别

PWR	STM32F1 系列	STM32F05x 系列	STM32F06x 系列
电源	1. V_{DD} = 2.0 V 到 3.6 V：I/O 和内部调压器的外部电源。通过 VDD 引脚从外部提供。 2. V_{SSA}、V_{DDA} = 2.0 V 到 3.6 V：ADC、复位模块、RC 和 PLL 的外部模拟电源。VDDA 和 VSSA 必须分别连接到 VDD 和 VSS。 3. V_{BAT} = 1.8 V 到 3.6 V：当 V_{DD} 不存在时，作为 RTC、32 kHz 外部时钟振荡器和备份寄存器的电源（通过电源开关供电）。	1. V_{DD} = 2.0 V 到 3.6 V：I/O 和内部调压器的外部电源。通过 VDD 引脚从外部提供。 2. V_{SSA}、V_{DDA} = 2.0 V 到 3.6 V：ADC、DAC、复位模块、RC 和 PLL 的外部模拟电源。 3. V_{BAT} = 1.65 V 到 3.6 V：当 V_{DD} 不存在时，作为 RTC、32 kHz 外部时钟振荡器和备份寄存器的电源（通过电源开关供电）。	1. V_{DD} = 1.8 V +/- 8 %：I/O 的外部电源。通过 VDD 引脚从外部提供。 2. V_{SSA}、V_{DDA} = 1.65 V 到 3.6 V：ADC、DAC、复位模块、RC 和 PLL 的外部模拟电源。 3. V_{BAT} = 1.65 V 到 3.6 V：当 V_{DD} 不存在时，作为 RTC、32 kHz 外部时钟振荡器和备份寄存器的电源（通过电源开关供电）。
电池备份域	– 备份寄存器 – RTC – LSE – PC13 到 PC15 I/O	– 备份寄存器 – RTC – LSE – RCC 备份域控制寄存器	– 备份寄存器 – RTC – LSE – RCC 备份域控制寄存器
电源监控器	集成 POR/PDR 电路 可编程电压检测器 (PVD)	集成 POR/PDR 电路 可编程电压检测器 (PVD)	通过专用 NPOR 引脚从外部进行控制。
低功耗模式	睡眠模式 停机模式 待机模式（1.8V 域断电）	睡眠模式 停机模式 待机模式（1.8V 域断电）	睡眠模式 停机模式
唤醒源	睡眠模式 – 任何外设中断/唤醒事件 停机模式 – 任何 EXTI 线事件/中断 待机模式 – WKUP 引脚上升沿 – RTC 报警 – NRST 引脚中的外部复位 – IWDG 复位	睡眠模式 – 任何外设中断/唤醒事件 停机模式 – 任何 EXTI 线事件/中断 待机模式 – WKUP0 或 WKUP1 引脚上升沿 – RTC 报警 – NRST 引脚中的外部复位 – IWDG 复位	睡眠模式 – 任何外设中断/唤醒事件 停机模式 – 任何 EXTI 线事件/中断

3.12 实时时钟 (RTC) 接口

相较 F1 系列而言，STM32F0 系列内置一个新的 RTC 外设。二者在结构、特性和编程接口方面均不同。

因此，F0 RTC 编程过程和寄存器均不同于 F1 系列，因此，使用 RTC 写入 F1 系列的任何代码都需要重写后才能在 F0 系列上运行。

F0 RTC 提供了卓越的特性：

- BCD 定时器/计数器
- 具有亚秒精度的时钟/日历，支持可编程夏令时补偿
- 可编程报警
- 数字校准电路
- 用于事件保存的时间戳功能
- 利用亚秒级移位特性与外部时钟实现精确同步。
- 5 个备份寄存器（20 字节），在发生入侵检测事件时复位

有关 STM32F0 的 RTC 特性的更多详细信息，请参见 STM32F0 参考手册 (RM0091) 的 RTC 章节。

有关 RTC 编程的深入信息，请参见应用笔记 AN3371 *使用 STM32 硬件实时时钟 (RTC)*。

3.13 SPI 接口

相较 F1 系列而言，STM32F0 系列内置一个新的 SPI 外设。其结构、特性和编程接口均经过修改，引入了全新的功能。

因此，F0 SPI 编程过程和寄存器与 F1 系列中的类似，但增加了一些新特性。如果 F0 系列不使用新功能，则只需对使用 SPI 写入 F1 系列的代码稍加改写即可在 F0 系列上运行。

F0 SPI 提供了卓越的新增特性：

- 增强的 NSS 控制 - NSS 脉冲模式 (NSSP) 和 TI 模式
- 可编程数据帧长度为 4 位到 16 位
- 两个 32 位 Tx/Rx FIFO 缓冲区，具有 DMA 功能和数据包访问功能，可访问装入到一个字节中的帧（多达 8 位）
- 针对 8 位和 16 位数据进行长度为 8 位或 16 位的 CRC 计算。

此外，F0 系列提供的 SPI 外设修正了 F1 系列产品中存在的 CRC 限制。有关 STM32F0 SPI 特性的更多信息，请参见 STM32F0 参考手册 (RM0091) 的 SPI 章节。

3.14 I2C 接口

相较 F1 系列而言，STM32F0 系列内置一个新的 I2C 外设。二者在结构、特性和编程接口方面均不同。

因此，F0 I2C 编程过程和寄存器均不同于 F1 系列，因此，使用 I2C 写入 F1 系列的任何代码都需要重写后才能在 F0 系列上运行。

F0 I2C 提供了卓越的新特性：

- 通信事件由硬件管理。
- 可编程模拟和数字噪声滤波器。
- 独立的时钟源：HSI 或 SYSCLK。
- 从停止模式唤醒。
- 快速模式 +（高达 1MHz）提供 20mA I/O 输出电流驱动能力。
- 提供 7 位和 10 位寻址模式，通过可配置掩码支持多个 7 位从地址。
- 主模式下可自动发送地址序列（7 位和 10 位）。
- 主模式下自动结束通信管理。
- 保持和设置时间可编程。
- 命令和数据确认控制。

有关 STM32F0 I2C 特性的更多信息，请参见 STM32F0 参考手册 (RM0091) 的 I2C 章节。

3.15 USART 接口

相较 F1 系列而言，STM32F0 系列内置一个新的 USART 外设。其结构、特性和编程接口均经过修改，引入了全新的功能。

因此，F0 USART 编程过程和寄存器均不同于 F1 系列，因此，使用 USART 写入 F1 系列的任何代码都需要更新后才能在 F0 系列上运行。

F0 USART 提供了卓越的新增特性：

- 允许选择独立的时钟源
  - 具备 UART 功能并且能够从低功耗模式唤醒，
  - 方便的波特率编程，与 APB 时钟重新编程无关。
- 智能卡仿真功能：T=0（支持自动重试）和 T=1
- Tx/Rx 引脚配置可交换
- 二进制数据反向
- Tx/Rx 引脚有效电平翻转
- 发送/接收使能确认标志
- 带标志的新中断源：
  - 地址/字符匹配
  - 块长度检测和超时检测
- 超时特性
- Modbus 通信
- 禁止超时标志
- 出现接收错误时禁止 DMA
- 从停止模式唤醒
- 具有波特率自动检测功能
- RS485 模式下发出驱动器使能信号 (DE)

有关 STM32F0 USART 特性的更多信息，请参见 STM32F0 参考手册 (RM0091) 的 USART 章节。

3.16 CEC 接口

相较 F1 系列而言，STM32F0 系列内置一个新的 CEC 外设。其结构、特性和编程接口均经过修改，引入了全新的功能。

因此，F0 CEC 编程过程和寄存器均不同于 F1 系列，因此，使用 CEC 写入 F1 系列的任何代码都需要重写后才能在 F0 系列上运行。

F0 CEC 提供了卓越的新增特性：

- 带双时钟的 32 KHz CEC 内核
  - LSE
  - HSI/244
- 可在监听模式下接收
- Rx 公差范围：标准或扩展
- 仲裁（信号空闲时间）：标准（通过 H/W）或积极（通过 S/W）
- 仲裁丢失检测标志/中断
- 支持仲裁丢失后自动进行传输重试
- 多地址配置
- 从停止模式唤醒
- 接收错误检测
  - 位上升沿错误（停止接收时）
  - 短位周期错误
  - 长位周期错误
- 生成可配置的错误位
  - 在位上升沿错误检测期间
  - 在长位周期错误检测期间
- 在运行检测时进行传输
- 接收溢出检测

F1 系列中存在的以下特性现由最新的 F0 CEC 特性取代，因而不再可用。

- 位定时错误模式和位周期错误模式，由新的错误处理程序取代
- 可配置预分频器，由 CEC 固定内核时钟取代

有关 STM32F0 CEC 特性的详细信息，请参见 STM32F0 参考手册 (RM0091) 的 CEC 章节。

4 使用库进行固件移植

本节介绍了如何移植基于 STM32F1xx 标准外设库的应用程序，目的是使用 STM32F0xx 标准外设库。

STM32F1xx 和 STM32F0xx 库的结构相同，并且均与 CMSIS 兼容；对于所有兼容的外设，它们都使用相同的驱动程序命名方式和相同的 API。

将应用程序从 F1 系列移植到 F0 系列产品时，只需更新几个外设驱动程序。

注： *在本节的其余部分中，术语“STM32F0xx 库”是指 STM32F0xx 标准外设库，术语“STM32F10x 库”是指 STM32F10x 标准外设库（除非另有规定）。*

4.1 移植步骤

要更新应用程序代码，以在 STM32F0xx 库上运行，用户需要按照下列步骤操作：

1. 更新工具启动文件
   a) *项目文件：*器件连接和 Flash 加载程序。这些文件随支持 STM32F0xxx 器件的最新版本工具一起提供。有关详细信息，请参见相关工具文档。
   b) *链接器配置和向量表位置文件：*这些文件根据 CMSIS 标准开发并且包含在 STM32F0xx 库安装包内，位置如下：*Libraries\CMSIS\Device\ST\STM32F0xx*。
2. 将 STM32F0xx 库源文件添加到应用程序源
   a) 用 STM32F0xx 库中提供的 *stm32f0xx_conf.h* 文件替换应用程序的 *stm32f10x_conf.h* 文件。
   b) 用 STM32F0xx 库中提供的 *stm32f0xx_it.c/Stm32f0xx_it.h* 文件替换应用程序的现有 *stm32f10x_it.c/stm32f10x_it.h* 文件。
3. 更新使用 RCC、PWR、GPIO、FLASH、ADC 和 RTC 驱动程序的部分应用程序代码。相关详细信息，请参见下一节。

注： *STM32F0xx 库随附丰富的示例（总共 67 个），具体演示了使用不同外设的方法（位于 Project\STM32F0xx_StdPeriph_Examples\ 下）。*

4.2 RCC 驱动程序

1. *系统时钟配置：*如 *3.4: 复位和时钟控制器 (RCC) 接口*一节所述，STM32 F0 和 F1 系列的时钟源配置过程相同。但是，二者产品的电压范围、PLL 配置、最大频率和 Flash 等待周期配置有所不同。利用 CMSIS 层，可从应用程序代码中隐藏这些区别；用户只需使用 *system_stm32f0xx.c* 文件替换 *system_stm32f10x.c* 文件。此文件提供了 *SystemInit()* 函数一种实现方法，利用该函数，可在启动时和转到 main() 程序之前配置微控制器系统。

注： *对于 STM32F0xx，用户可根据应用程序要求，使用时钟配置工具 STM32F0xx_Clock_Configuration.xls 生成一个自定义 SystemInit() 函数。有关详细信息，请参见 AN4055“STM32F0xx 微控制器的时钟配置工具”。*

2. *外设访问配置*：如 *3.4: 复位和时钟控制器 (RCC) 接口*一节所述，用户需要调用不同的函数来 [使能/禁止] 外设 [时钟] 或使外设 [进入/退出] [复位模式]。例如，GPIOA 映射在 F0 系列的 AHB 总线上（F1 系列的 APB2 总线）。要使能其时钟，用户必须使用 `RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);` 函数，而不是：F1 系列中的 `RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);` 函数。
   有关 F0 和 F1 系列之间的外设总线映射更改，请参见*表 5*。
3. *外设时钟配置*

某些 STM32F0xx 外设支持双时钟特性。下表概述了 STM32F0xx 外设与 STM32F10xx 外设的 IP 时钟源比较。

表 15. STM32F10x 与 STM32F0xx 时钟源 API 对应关系

外设	STM32F10xx 器件中的时钟源	STM32F0xx 器件中的时钟源
ADC	带预分频器的 APB2 时钟	- HSI14：默认 - APB2 时钟 2 分频 - APB2 时钟 4 分频
CEC	带预分频器的 APB1 时钟	- HSI/244：默认 - LSE - APB 时钟：用于数字接口的时钟（用于寄存器读/写访问）。此时钟等效于 APB2 时钟。
I2C	APB1 时钟	1. I2C1 可由以下时钟源提供时钟： - 系统时钟 - HSI 2. I2C2 只能由以下时钟源提供时钟： - HSI
SPI/I2S	系统时钟	系统时钟
USART	1. USART1 可由以下时钟源提供时钟： - PCLK2（最大 72 MHz） 2. 其它 USART 可由以下时钟源提供时钟： - PCLK1（最大 36 MHz）	1. USART1 可由以下时钟源提供时钟： - 系统时钟 - LSE 时钟 - HSI 时钟 - APB 时钟 (PCLK) 2. USART2 只能由以下时钟源提供时钟： - 系统时钟

4.3 Flash 驱动程序

下表介绍了 STM32F10x 与 STM32F0xx 库之间的 Flash 驱动程序 API 的对应关系。利用 STM32F0xx 库中的函数替换 STM32F10x 中的对应函数后，便可轻松更新应用程序代码。

表 16. **STM32F10x 与 STM32F0xx Flash 驱动程序 API 对应关系**

	STM32F10x Flash 驱动程序 API	STM32F0xx Flash 驱动程序 API
接口配置	void FLASH_SetLatency(uint32_t FLASH_Latency);	void FLASH_SetLatency(uint32_t FLASH_Latency);
	void FLASH_PrefetchBufferCmd(uint32_t FLASH_PrefetchBuffer);	void FLASH_PrefetchBufferCmd(FunctionalState NewState);
	void FLASH_HalfCycleAccessCmd(uint32_t FLASH_HalfCycleAccess);	NA
	void FLASH_ITConfig(uint32_t FLASH_IT, FunctionalState NewState);	void FLASH_ITConfig(uint32_t FLASH_IT, FunctionalState NewState);
存储器编程	void FLASH_Unlock(void);	void FLASH_Unlock(void);
	void FLASH_Lock(void);	void FLASH_Lock(void);
	FLASH_Status FLASH_ErasePage(uint32_t Page_Address);	FLASH_Status FLASH_ErasePage(uint32_t Page_Address);
	FLASH_Status FLASH_EraseAllPages(void);	FLASH_Status FLASH_EraseAllPages(void);
	FLASH_STATUS FLASH_ERASEOPTIONBYTES(void);	FLASH_STATUS FLASH_OB_ERASE(void);
	FLASH_Status FLASH_ProgramWord(uint32_t Address, uint32_t Data);	FLASH_Status FLASH_ProgramWord(uint32_t Address, uint32_t Data);
	FLASH_Status FLASH_ProgramHalfWord(uint32_t Address, uint16_t Data);	FLASH_Status FLASH_ProgramHalfWord(uint32_t Address, uint16_t Data);

表 16. STM32F10x 与 STM32F0xx Flash 驱动程序 API 对应关系（续）

	STM32F10x Flash 驱动程序 API	STM32F0xx Flash 驱动程序 API
选项字节编程	NA	void FLASH_OB_Unlock(void);
	NA	void FLASH_OB_Lock(void);
	FLASH_Status FLASH_ProgramOptionByteData(uint32_t Address, uint8_t Data);	FLASH_Status FLASH_ProgramOptionByteData(uint32_t Address, uint8_t Data);
	FLASH_Status FLASH_EnableWriteProtection(uint32_t FLASH_Pages);	FLASH_Status FLASH_OB_EnableWRP(uint32_t OB_WRP);
	FLASH_Status FLASH_ReadOutProtection(FunctionalState NewState);	FLASH_Status FLASH_OB_RDPConfig(uint8_t OB_RDP);
	FLASH_Status FLASH_UserOptionByteConfig(uint16_t OB_IWDG, uint16_t OB_STOP, uint16_t OB_STDBY);	FLASH_Status FLASH_OB_UserConfig(uint8_t OB_IWDG, uint8_t OB_STOP, uint8_t OB_STDBY);
	NA	FLASH_Status FLASH_OB_Launch(void);
	NA	FLASH_Status FLASH_OB_WriteUser(uint8_t OB_USER);
	NA	FLASH_Status FLASH_OB_BOOTConfig(uint8_t OB_BOOT1);
	NA	FLASH_Status FLASH_OB_VDDAConfig(uint8_t OB_VDDA_ANALOG);
	NA	FLASH_Status FLASH_OB_SRAMParityConfig(uint8_t OB_SRAM_Parity);
	uint32_t FLASH_GetUserOptionByte(void);	uint8_t FLASH_OB_GetUser(void);
	uint32_t FLASH_GetWriteProtectionOptionByte(void);	uint16_t FLASH_OB_GetWRP(void);
	FlagStatus FLASH_GetReadOutProtectionStatus(void);	FlagStatus FLASH_OB_GetRDP(void);

表 16. STM32F10x 与 STM32F0xx Flash 驱动程序 API 对应关系（续）

	STM32F10x Flash 驱动程序 API	STM32F0xx Flash 驱动程序 API
FLAG 管理	FlagStatus FLASH_GetFlagStatus(uint32_t FLASH_FLAG);	FlagStatus FLASH_GetFlagStatus(uint32_t FLASH_FLAG);
	void FLASH_ClearFlag(uint32_t FLASH_FLAG);	void FLASH_ClearFlag(uint32_t FLASH_FLAG);
	FLASH_Status FLASH_GetStatus(void);	FLASH_Status FLASH_GetStatus(void);
	FLASH_Status FLASH_WaitForLastOperation(uint32_t Timeout);	FLASH_Status FLASH_WaitForLastOperation(void);
	FlagStatus FLASH_GetPrefetchBufferStatus(void);	FlagStatus FLASH_GetPrefetchBufferStatus(void);

颜色说明：

= 新函数

= 函数不变，但 API 已更改

= 函数不可用 (NA)

4.4 CRC 驱动程序

下表介绍了 STM32F10x 与 STM32F0xx 库之间的 Flash 驱动程序 CRC 的对应关系。

表 17. STM32F10xx 与 STM32F0xx CRC 驱动程序 API 对应关系

	STM32F10xx CRC 驱动程序 API	STM32F0xx CRC 驱动程序 API
配置	NA	void CRC_DeInit(void);
	void CRC_ResetDR(void);	void CRC_ResetDR(void);
	NA	void CRC_ReverseInputDataSelect(uint32_t CRC_ReverseInputData);
	NA	void CRC_ReverseOutputDataCmd(FunctionalState NewState);
	NA	void CRC_SetInitRegister(uint32_t CRC_InitValue);
计算	uint32_t CRC_CalcCRC(uint32_t CRC_Data);	uint32_t CRC_CalcCRC(uint32_t CRC_Data);
	uint32_t CRC_CalcBlockCRC(uint32_t pBuffer[], uint32_t BufferLength);	uint32_t CRC_CalcBlockCRC(uint32_t pBuffer[], uint32_t BufferLength);
	uint32_t CRC_GetCRC(void);	uint32_t CRC_GetCRC(void);

表 17. STM32F10xx 与 STM32F0xx CRC 驱动程序 API 对应关系（续）

	STM32F10xx CRC 驱动程序 API	STM32F0xx CRC 驱动程序 API
IDR 访问	void CRC_SetIDRegister(uint8_t CRC_IDValue);	void CRC_SetIDRegister(uint8_t CRC_IDValue);
	uint8_t CRC_GetIDRegister(void);	uint8_t CRC_GetIDRegister(void);

颜色说明：

= 新函数

= 函数不变，但 API 已更改

= 函数不可用 (NA)

4.5 GPIO 配置更新

本节介绍了将应用程序代码从 STM32 F1 系列移植到 F0 系列时如何更新各 GPIO 模式的配置。

4.5.1 输出模式

下面的示例介绍了如何在 STM32 F1 系列中配置输出模式下的 I/O（例如，用于驱动 LED）：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_xxMHz; /* 2、10 或 50 MHz */
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(GPIOy, &GPIO_InitStructure);
```

在 F0 系列中，用户必须按下列步骤更新此代码：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; /* 推挽输出或开漏输出 */
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; /* 无、上拉或下拉 */
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_xxMHz; /* 10、2 或 50MHz */
GPIO_Init(GPIOy, &GPIO_InitStructure);
```

4.5.2 输入模式

下面的示例介绍了如何在 STM32 F1 系列中配置输入模式下的 I/O（例如，用作 EXTI 线）：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOy, &GPIO_InitStructure);
```

在 F0 系列中，用户必须按下列步骤更新此代码：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL; /* 无、上拉或下拉 */
GPIO_Init(GPIOy, &GPIO_InitStructure);
```

4.5.3 模拟模式

下面的示例介绍了如何在 STM32 F1 系列中配置模拟模式下的 I/O（例如，ADC 或 DAC 通道）：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
GPIO_Init(GPIOy, &GPIO_InitStructure);
```

在 F0 系列中，用户必须按下列步骤更新此代码：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
GPIO_Init(GPIOy, &GPIO_InitStructure);
```

4.5.4 复用功能模式

STM32 F1 系列

1. 将 I/O 用作复用功能的配置取决于所使用的外设模式；例如，USART Tx 引脚应配置为复用功能推挽，而 USART Rx 引脚应配置为悬空输入或输入上拉。
2. 为针对不同器件封装优化外设 I/O 功能的数量，可以用软件将某些复用功能重新映射到其它引脚上。例如，可将 USART2_RX 引脚映射到 PA3（默认重映射）或 PD6（软件重映射）上。

STM32 F0 系列

1. 不论使用何种外设模式，都必须将 I/O 配置为复用功能，之后系统才能正确使用 I/O（输入或输出）。
2. I/O 引脚通过复用器连接到板载外设/模块，该复用器一次只允许一个外设的复用功能 (AF) 连接到 I/O 引脚。这样便可确保共用同一个 I/O 引脚的外设之间不会发生冲突。每个 I/O 引脚都有一个复用器，该复用器具有十六路复用功能输入（AF0 到 AF15），可通过 GPIO_PinAFConfig () 函数对这些输入进行配置：
   – 完成复位后，所有 I/O 都会连接到系统的复用功能 0 (AF0)
   – 通过配置 AF1 到 AF7 可以映射外设的复用功能
3. 除了这种灵活的 I/O 复用架构之外，各外设还具有映射到不同 I/O 引脚的复用功能，这可以针对不同器件封装优化外设 I/O 功能的数量；例如，可将 USART2_RX 引脚映射到 PA3 或 PA15 引脚上。
4. 配置过程：
   – 使用 GPIO_PinAFConfig() 函数将引脚连接到所需的外设复用功能 (AF)
   – 使用 GPIO_Init() 函数配置 I/O 引脚：
     - 通过以下方式配置复用功能模式下的所需引脚
       GPIO_InitStructure->GPIO_Mode = GPIO_Mode_AF;
     - 通过以下成员选择类型、上拉/下拉和输出速度
       GPIO_PuPd、GPIO_OType 和 GPIO_Speed 成员

下面的示例介绍了如何将 USART2 Tx/Rx I/O 重映射到 STM32 F1 系列中的 PD5/PD6 引脚：

```
/* 为 GPIOD 和 AFIO 使能 APB2 接口（AFIO 外设用于
   配置 I/O 软件重映射）*/
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD | RCC_APB2Periph_AFIO, ENABLE);
```

```
/* 使能 USART2 I/O 软件重映射 [(USART2_Tx,USART2_Rx):(PD5,PD6)] */
GPIO_PinRemapConfig(GPIO_Remap_USART2, ENABLE);

/* 将 USART2_Tx 配置为复用功能推挽 */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOD, &GPIO_InitStructure);

/* 将 USART2_Rx 配置为输入悬空 */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOD, &GPIO_InitStructure);
```

在 F0 系列中，用户必须按下列步骤更新此代码：

```
/* 使能 GPIOA 的 AHB 接口时钟 */
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);

/* 选择将 USART2 I/O 映射到 PA14/15 引脚上 [(USART2_TX,USART2_RX):(PA.14,PA.15)] */
/* 将 PA14 连接到 USART2_Tx */
GPIO_PinAFConfig(GPIOA, GPIO_PinSource14, GPIO_AF_2);
/* 将 PA15 连接到 USART2_Rx*/
GPIO_PinAFConfig(GPIOA, GPIO_PinSource15, GPIO_AF_2);

/* 将 USART2_Tx 和 USART2_Rx 配置为复用功能 */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_14 | GPIO_Pin_15;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

4.6 EXTI 线 0

下面的示例介绍了如何在 STM32 F1 系列中将 PA0 引脚配置为 EXTI 线 0：

```
/* 为 GPIOA 和 AFIO 使能 APB 接口时钟 */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO, ENABLE);

/* 将 PA0 引脚配置为输入模式 */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOA, &GPIO_InitStructure);

/* 将 EXTI 线 0 连接到 PA0 引脚 */
GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0);

/*配置 EXTI 线 0 */
EXTI_InitStructure.EXTI_Line = EXTI_Line0;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);
```

在 F0 系列中，EXTI 线源引脚的配置在 SYSCFG 外设中进行（与 F1 系列中的 AFIO 不同）。因此，应按如下步骤更新源代码：

```
/* 使能 GPIOA 的 AHB 接口时钟 */
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
/* 使能 SYSCFG 的 APB 接口时钟 */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

/* 将 PA0 引脚配置为输入模式 */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
GPIO_Init(GPIOA, &GPIO_InitStructure);

/* 将 EXTI 线 0 连接到 PA0 引脚 */
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);

/*配置 EXTI 线 0 */
EXTI_InitStructure.EXTI_Line = EXTI_Line0;
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_InitStructure.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);
```

4.7 NVIC 中断配置

本节介绍了 NVIC 中断 (IRQ) 的配置。

在 F1 系列中，NVIC 支持：

- 多达 81 个中断。
- 各中断的可编程优先级为 0-15（使用 4 个中断优先级位）。级别越高，优先级越低；级别 0 表示最高中断优先级。
- 将优先级值分为组优先级和子优先级两个域。
- 优先级动态变化。

Cortex-M3 异常由 CMSIS 功能管理：

- 根据优先级分组配置，使能并配置所选 IRQ 通道的抢占式优先级和子优先级。

下面的示例介绍了如何在 STM32 F1 系列中配置 CEC 中断：

```
/* 为抢占式优先级配置两个位 */
  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
/* 使能 CEC 全局中断（优先级更高）*/
  NVIC_InitStructure.NVIC_IRQChannel = CEC_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
```

在 F0 系列中，NVIC 支持：

- 多达 32 个中断
- 4 个可编程优先级（使用 2 个中断优先级位）
- 使能中断后，不得更改其优先级

Cortex-M0 异常由 CMSIS 功能管理：

- 使能并配置所选 IRQ 通道的优先级。优先级范围介于 0 到 3 之间。优先级值越小，优先级越高。

因此，应按如下步骤更新 CEC 中断源代码的配置：

```
/* 使能 CEC 全局中断（优先级更高）*/
  NVIC_InitStructure.NVIC_IRQChannel = CEC_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
```

下表介绍了 STM32F10x 与 STM32F0xx 库之间的 MISC 驱动程序 API 的对应关系。

表 18. STM32F10x 与 STM32F0xx MISC 驱动程序 API 对应关系

STM32F10xx MISC 驱动程序 API	STM32F0xx MISC 驱动程序 API
void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct);	void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct);
void NVIC_SystemLPConfig(uint8_t LowPowerMode, FunctionalState NewState);	void NVIC_SystemLPConfig(uint8_t LowPowerMode, FunctionalState NewState);
void SysTick_CLKSourceConfig(uint32_t SysTick_CLKSource);	void SysTick_CLKSourceConfig(uint32_t SysTick_CLKSource);
NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup);	NA
void NVIC_SetVectorTable(uint32_t NVIC_VectTab, uint32_t Offset);	NA

4.8 ADC 配置

本节介绍如何将现有代码从 STM32 F1 系列移植到 F0 系列的示例。

下面的示例介绍了如何在 STM32 F1 系列中将 ADC1 配置为连续转换通道 14：

```
/* ADCCLK = PCLK2/4 */
RCC_ADCCLKConfig(RCC_PCLK2_Div4);

/* 使能 ADC 的 APB 接口时钟 */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

/* 将 ADC1 配置为连续转换通道 14 */
ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;
ADC_InitStructure.ADC_ScanConvMode = ENABLE;
ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
ADC_InitStructure.ADC_NbrOfChannel = 1;
ADC_Init(ADC1, &ADC_InitStructure);
/* ADC1 常规通道 14 配置 */
ADC_RegularChannelConfig(ADC1, ADC_Channel_14, 1, ADC_SampleTime_55Cycles5);
```

```
  /* 使能 ADC1 的 DMA 接口 */
  ADC_DMACmd(ADC1, ENABLE);

  /* 使能 ADC1 */
  ADC_Cmd(ADC1, ENABLE);

  /* 使能 ADC1 复位校准寄存器 */
  ADC_ResetCalibration(ADC1);
  /* 检查 ADC1 复位校准寄存器是否结束 */
  while(ADC_GetResetCalibrationStatus(ADC1));

  /* 启动 ADC1 校准 */
  ADC_StartCalibration(ADC1);
  /* 检查 ADC1 校准是否结束 */
  while(ADC_GetCalibrationStatus(ADC1));

  /* 启动 ADC1 软件转换 */
  ADC_SoftwareStartConvCmd(ADC1, ENABLE);
  ...
```

在 F0 系列中，用户必须按下列步骤更新此代码：

```
...
/* ADCCLK = PCLK/2 */
  RCC_ADCCLKConfig(RCC_ADCCLK_PCLK_Div2);

  /* 使能 ADC1 时钟 */
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

  /* ADC1 配置 */
  ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
  ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
  ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;

 ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T1_TRGO;;
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_ScanDirection = ADC_ScanDirection_Backward;
  ADC_Init(ADC1, &ADC_InitStructure);

/* 以 55.5 个周期作为采样时间转换 ADC1 通道 1 */
  ADC_ChannelConfig(ADC1, ADC_Channel_11 , ADC_SampleTime_55_5Cycles);

/* ADC 校准 */
  ADC_GetCalibrationFactor(ADC1);

/* 循环模式下的 ADC DMA 请求 */
  ADC_DMARequestModeConfig(ADC1, ADC_DMAMode_Circular);

/* 使能 ADC_DMA */
  ADC_DMACmd(ADC1, ENABLE);

  /* 使能 ADC1 */
  ADC_Cmd(ADC1, ENABLE);

/* 等待 ADCEN 标志 */
  while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_ADEN));

  /* ADC1 常规软件启动转换 */
  ADC_StartOfConversion(ADC1);
  ...
```

4.9 DAC 驱动程序

下表介绍了 STM32F10x 函数与 STM32F0xx 库之间的区别。

表 19. STM32F10x 与 STM32F0xx DAC 驱动程序 API 对应关系

	STM32F10x DAC 驱动程序 API	STM32F0xx DAC 驱动程序 API
配置	void DAC_DeInit(void);	void DAC_DeInit(void);
	void DAC_Init(uint32_t DAC_Channel, DAC_InitTypeDef* DAC_InitStruct);	void DAC_Init(uint32_t DAC_Channel, DAC_InitTypeDef* DAC_InitStruct);
	void DAC_StructInit(DAC_InitTypeDef* DAC_InitStruct);	void DAC_StructInit(DAC_InitTypeDef* DAC_InitStruct);
	void DAC_Cmd(uint32_t DAC_Channel, FunctionalState NewState);	void DAC_Cmd(uint32_t DAC_Channel, FunctionalState NewState);
	void DAC_SoftwareTriggerCmd(uint32_t DAC_Channel, FunctionalState NewState);	void DAC_SoftwareTriggerCmd(uint32_t DAC_Channel, FunctionalState NewState);
	void DAC_SetChannel1Data(uint32_t DAC_Align, uint16_t Data);	void DAC_SetChannel1Data(uint32_t DAC_Align, uint16_t Data);
	void DAC_SetChannel2Data(uint32_t DAC_Align, uint16_t Data);	NA
	void DAC_SetDualChannelData(uint32_t DAC_Align, uint16_t Data2, uint16_t Data1);	NA
	uint16_t DAC_GetDataOutputValue(uint32_t DAC_Channel);	uint16_t DAC_GetDataOutputValue(uint32_t DAC_Channel);
DMA 管理	void DAC_DMACmd(uint32_t DAC_Channel, FunctionalState NewState);	void DAC_DMACmd(uint32_t DAC_Channel, FunctionalState NewState);
中断和标志管理	void DAC_ITConfig(uint32_t DAC_Channel, uint32_t DAC_IT, FunctionalState NewState);(*)	void DAC_ITConfig(uint32_t DAC_Channel, uint32_t DAC_IT, FunctionalState NewState);
	FlagStatus DAC_GetFlagStatus(uint32_t DAC_Channel, uint32_t DAC_FLAG);(*)	FlagStatus DAC_GetFlagStatus(uint32_t DAC_Channel, uint32_t DAC_FLAG);
	void DAC_ClearFlag(uint32_t DAC_Channel, uint32_t DAC_FLAG);(*)	void DAC_ClearFlag(uint32_t DAC_Channel, uint32_t DAC_FLAG);
	ITStatus DAC_GetITStatus(uint32_t DAC_Channel, uint32_t DAC_IT);(*)	ITStatus DAC_GetITStatus(uint32_t DAC_Channel, uint32_t DAC_IT);
	void DAC_ClearITPendingBit(uint32_t DAC_Channel, uint32_t DAC_IT);(*)	void DAC_ClearITPendingBit(uint32_t DAC_Channel, uint32_t DAC_IT);

(*) 这些函数仅存在于 STM32F10X_LD_VL、STM32F10X_MD_VL 和 STM32F10X_HD_VL 器件上。

相较 F1 系列而言，F0 系列中源代码/流程的主要变化如下：

- DMA 通道无双重模式
- 无噪声发生器
- 无三角波发生器
- 在 DAC 结构定义中，只应初始化两个字段（外部触发器，输出缓冲区）。

下面的示例介绍了如何在 STM32 F1 系列中配置 DAC 通道 1：

```
/* DAC 通道 1 配置 */
  DAC_InitStructure.DAC_Trigger = DAC_Trigger_None;
  DAC_InitStructure.DAC_WaveGeneration = DAC_WaveGeneration_None;
  DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
  DAC_Init(DAC_Channel_1, &DAC_InitStructure);
```

在 F0 系列中，用户必须按下列步骤更新此代码：

```
/* DAC 通道 1 配置 */
  DAC_InitStructure.DAC_Trigger = DAC_Trigger_None;
  DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
  /* DAC 通道 1 初始化 */
  DAC_Init(DAC_Channel_1, &DAC_InitStructure);
```

4.10 PWR 驱动程序

下表介绍了 STM32F10x 与 STM32F0xx 库之间的 PWR 驱动程序 API 的对应关系。利用 STM32F0xx 库中的函数替换 STM32F10x 中的对应函数后，便可轻松更新应用程序代码。

表 20. STM32F10x 与 STM32F0xx PWR 驱动程序 API 对应关系

	STM32F10x PWR 驱动程序 API	STM32F0xx PWR 驱动程序 API
接口配置	void PWR_DeInit(void);	void PWR_DeInit(void);
	void PWR_BackupAccessCmd(FunctionalState NewState);	void PWR_BackupAccessCmd(FunctionalState NewState);
PVD	void PWR_PVDLevelConfig(uint32_t PWR_PVDLevel);	void PWR_PVDLevelConfig(uint32_t PWR_PVDLevel);
	void PWR_PVDCmd(FunctionalState NewState);	void PWR_PVDCmd(FunctionalState NewState);
唤醒	void PWR_WakeUpPinCmd(FunctionalState NewState);	void PWR_WakeUpPinCmd(uint32_t PWR_WakeUpPin, FunctionalState NewState);(*)
电源管理	NA	void PWR_EnterSleepMode(uint8_t PWR_SLEEPEntry);
	void PWR_EnterSTOPMode(uint32_t PWR_Regulator, uint8_t PWR_STOPEntry);	void PWR_EnterSTOPMode(uint32_t PWR_Regulator, uint8_t PWR_STOPEntry);
	void PWR_EnterSTANDBYMode(void);	void PWR_EnterSTANDBYMode(void);

表 20. STM32F10x 与 STM32F0xx PWR 驱动程序 API 对应关系（续）

	STM32F10x PWR 驱动程序 API	STM32F0xx PWR 驱动程序 API
标志管理	FlagStatus PWR_GetFlagStatus(uint32_t PWR_FLAG);	FlagStatus PWR_GetFlagStatus(uint32_t PWR_FLAG);
	void PWR_ClearFlag(uint32_t PWR_FLAG);	void PWR_ClearFlag(uint32_t PWR_FLAG);

颜色说明：

= 新函数

= 函数不变，但 API 已更改

= 函数不可用 (NA)

(*) STM32 F0 系列提供更多唤醒引脚。

4.11 备份数据寄存器

在 STM32 F1 系列中，备份数据寄存器通过 BKP 外设管理，而在 F0 系列中，这些寄存器属于 RTC 外设的一部分（不存在 BKP 外设）。

下面的示例介绍了如何在 STM32 F1 系列中对备份数据寄存器进行读/写操作：

```
uint16_t BKPdata = 0;

  ...
  /* 为 PWR 和 BKP 使能 APB2 接口时钟 */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);

  /* 使能对备份域的写访问 */
  PWR_BackupAccessCmd(ENABLE);

  /* 向备份数据寄存器 1 写入数据 */
  BKP_WriteBackupRegister(BKP_DR1, 0x3210);

  /* 从备份数据寄存器 1 中读取数据 */
  BKPdata = BKP_ReadBackupRegister(BKP_DR1);
```

在 F0 系列中，用户必须按下列步骤更新此代码：

```
uint16_t BKPdata = 0;

  ...
  /* 使能 PWR 时钟 */
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);

  /* 使能对 RTC 域的写访问 */
  PWR_RTCAccessCmd(ENABLE);

  /* 向备份数据寄存器 1 写入数据 */
  RTC_WriteBackupRegister(RTC_BKP_DR1, 0x3220);

  /* 从备份数据寄存器 1 中读取数据 */
  BKPdata = RTC_ReadBackupRegister(RTC_BKP_DR1);
```

相较 F1 系列而言，F0 系列中源代码的主要变化如下：

1. 无 BKP 外设。
2. 备份数据寄存器的读/写操作通过 RTC 驱动程序完成。
3. 备份数据寄存器命名从 BKP_DRx 改为 RTC_BKP_DRx，编号从 0 而不是 1 开始。

4.12 CEC 应用程序代码

利用 STM32F0xx 库中的函数替换 STM32F10x 中的对应函数后，便可轻松更新 CEC 应用程序代码。下表介绍了 STM32F10x 与 STM32F0xx 库之间的 CEC 驱动程序 API 的对应关系。

表 21. STM32F10xx 与 STM32F0xx CEC 驱动程序 API 对应关系

	STM32F10xx CEC 驱动程序 API	STM32F0xx CEC 驱动程序 API
接口配置	void CEC_DeInit(void);	void CEC_DeInit(void);
	void CEC_Init(CEC_InitTypeDef* CEC_InitStruct);	void CEC_Init(CEC_InitTypeDef* CEC_InitStruct);
	NA	void CEC_StructInit(CEC_InitTypeDef* CEC_InitStruct);
	void CEC_Cmd(FunctionalState NewState);	void CEC_Cmd(FunctionalState NewState);
	NA	void CEC_ListenModeCmd(FunctionalState NewState);
	void CEC_OwnAddressConfig(uint8_t CEC_OwnAddress);	void CEC_OwnAddressConfig(uint8_t CEC_OwnAddress);
	NA	void CEC_OwnAddressClear(void);
	void CEC_SetPrescaler(uint16_t CEC_Prescaler);	NA
数据传输	void CEC_SendDataByte(uint8_t Data);	void CEC_SendData(uint8_t Data);
	uint8_t CEC_ReceiveDataByte(void);	uint8_t CEC_ReceiveData(void);
	void CEC_StartOfMessage(void);	void CEC_StartOfMessage(void);
	void CEC_EndOfMessageCmd(FunctionalState NewState);	void CEC_EndOfMessage(void);
中断和标志管理	void CEC_ITConfig(FunctionalState NewState)	void CEC_ITConfig(uint16_t CEC_IT, FunctionalState NewState);
	FlagStatus CEC_GetFlagStatus(uint32_t CEC_FLAG);	FlagStatus CEC_GetFlagStatus(uint16_t CEC_FLAG);
	void CEC_ClearFlag(uint32_t CEC_FLAG)	void CEC_ClearFlag(uint32_t CEC_FLAG);
	ITStatus CEC_GetITStatus(uint8_t CEC_IT)	ITStatus CEC_GetITStatus(uint16_t CEC_IT);
	void CEC_ClearITPendingBit(uint16_t CEC_IT)	void CEC_ClearITPendingBit(uint16_t CEC_IT);

颜色说明：

= 新函数

= 函数不变，但 API 已更改

= 函数不可用 (NA)

相较 F1 系列而言，F0 系列中源代码/流程的主要变化如下：

- 双时钟源（详细信息请参见 RCC 部分）。
- 无预分频器特性配置。
- 支持多个地址（多地址寻址）。
- 每个事件标志都有相应的使能控制位，用于生成适当的中断。
- 在 CEC 结构定义中，应初始化七个字段。

下列示例介绍了如何配置 CEC STM32 F1 系列：

```
/* 配置 CEC 外设 */
  CEC_InitStructure.CEC_BitTimingMode = CEC_BitTimingStdMode;
  CEC_InitStructure.CEC_BitPeriodMode = CEC_BitPeriodStdMode;
  CEC_Init(&CEC_InitStructure);
```

在 F0 系列中，用户必须按下列步骤更新此代码：

```
  /* 配置 CEC */
  CEC_InitStructure.CEC_SignalFreeTime = CEC_SignalFreeTime_Standard;
  CEC_InitStructure.CEC_RxTolerance = CEC_RxTolerance_Standard;
  CEC_InitStructure.CEC_StopReception = CEC_StopReception_Off;
  CEC_InitStructure.CEC_BitRisingError = CEC_BitRisingError_Off;
  CEC_InitStructure.CEC_LongBitPeriodError = CEC_LongBitPeriodError_Off;
  CEC_InitStructure.CEC_BRDNoGen = CEC_BRDNoGen_Off;
  CEC_InitStructure.CEC_SFTOption = CEC_SFTOption_Off;
  CEC_Init(&CEC_InitStructure);
```

4.13 I2C 驱动程序

STM32F0xx 器件中整合了全新的 I2C 特性。下表介绍了 STM32F10x 与 STM32F0xx 库之间的 I2C 驱动程序 CRC 的对应关系。利用 STM32F0xx 库中的函数替换 STM32F10x 中的对应函数后，便可更新应用程序代码。

表 22. **STM32F10xx 与 STM32F0xx I2C 驱动程序 API 对应关系**

	STM32F10xx I2C 驱动程序 API	STM32F0xx I2C 驱动程序 API
初始化和配置	void I2C_DeInit(I2C_TypeDef* I2Cx);	void I2C_DeInit(I2C_TypeDef* I2Cx);
	void I2C_Init(I2C_TypeDef* I2Cx, I2C_InitTypeDef* I2C_InitStruct);	void I2C_Init(I2C_TypeDef* I2Cx, I2C_InitTypeDef* I2C_InitStruct);
	void I2C_StructInit(I2C_InitTypeDef* I2C_InitStruct);	void I2C_StructInit(I2C_InitTypeDef* I2C_InitStruct);
	void I2C_Cmd(I2C_TypeDef* I2Cx, FunctionalState NewState);	void I2C_Cmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	void I2C_SoftwareResetCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);	void I2C_SoftwareResetCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	void I2C_ITConfig(I2C_TypeDef* I2Cx, uint16_t I2C_IT, FunctionalState NewState);	void I2C_ITConfig(I2C_TypeDef* I2Cx, uint16_t I2C_IT, FunctionalState NewState);
	void I2C_StretchClockCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);	void I2C_StretchClockCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	void I2C_StopModeCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	void I2C_DualAddressCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);	void I2C_DualAddressCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	void I2C_OwnAddress2Config(I2C_TypeDef* I2Cx, uint8_t Address);	void I2C_OwnAddress2Config(I2C_TypeDef* I2Cx, uint16_t Address, uint8_t Mask);
	void I2C_GeneralCallCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);	void I2C_GeneralCallCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	void I2C_SlaveByteControlCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	void I2C_SlaveAddressConfig(I2C_TypeDef* I2Cx, uint16_t Address);
	NA	void I2C_10BitAddressingModeCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	void I2C_NACKPositionConfig(I2C_TypeDef* I2Cx, uint16_t I2C_NACKPosition);	NA
	void I2C_ARPCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);	NA

表 22. STM32F10xx 与 STM32F0xx I2C 驱动程序 API 对应关系（续）

	STM32F10xx I2C 驱动程序 API	STM32F0xx I2C 驱动程序 API
通信处理	NA	void I2C_AutoEndCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	void I2C_ReloadCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	void I2C_NumberOfBytesConfig(I2C_TypeDef* I2Cx, uint8_t Number_Bytes);
	NA	void I2C_MasterRequestConfig(I2C_TypeDef* I2Cx, uint16_t I2C_Direction);
	void I2C_GenerateSTART(I2C_TypeDef* I2Cx, FunctionalState NewState);	void I2C_GenerateSTART(I2C_TypeDef* I2Cx, FunctionalState NewState);
	void I2C_GenerateSTOP(I2C_TypeDef* I2Cx, FunctionalState NewState);	void I2C_GenerateSTOP(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	void I2C_10BitAddressHeaderCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	void I2C_AcknowledgeConfig(I2C_TypeDef* I2Cx, FunctionalState NewState);	void I2C_AcknowledgeConfig(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	uint8_t I2C_GetAddressMatched(I2C_TypeDef* I2Cx);
	NA	uint16_t I2C_GetTransferDirection(I2C_TypeDef* I2Cx);
	NA	void I2C_TransferHandling(I2C_TypeDef* I2Cx, uint16_t Address, uint8_t Number_Bytes, uint32_t ReloadEndMode, uint32_t StartStopMode);
	ErrorStatus I2C_CheckEvent(I2C_TypeDef* I2Cx, uint32_t I2C_EVENT)	NA
	void I2C_Send7bitAddress(I2C_TypeDef* I2Cx, uint8_t Address, uint8_t I2C_Direction)	NA

表 22. STM32F10xx 与 STM32F0xx I2C 驱动程序 API 对应关系（续）

	STM32F10xx I2C 驱动程序 API	STM32F0xx I2C 驱动程序 API
SMBUS 管理	void I2C_SMBusAlertConfig(I2C_TypeDef* I2Cx, uint16_t I2C_SMBusAlert);	void I2C_SMBusAlertCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	void I2C_ClockTimeoutCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	void I2C_ExtendedClockTimeoutCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	void I2C_IdleClockTimeoutCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	void I2C_TimeoutAConfig(I2C_TypeDef* I2Cx, uint16_t Timeout);
	NA	void I2C_TimeoutBConfig(I2C_TypeDef* I2Cx, uint16_t Timeout);
	void I2C_CalculatePEC(I2C_TypeDef* I2Cx, FunctionalState NewState);	void I2C_CalculatePEC(I2C_TypeDef* I2Cx, FunctionalState NewState);
	NA	void I2C_PECRequestCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);
	uint8_t I2C_GetPEC(I2C_TypeDef* I2Cx);	uint8_t I2C_GetPEC(I2C_TypeDef* I2Cx);
数据传输	uint32_t I2C_ReadRegister(I2C_TypeDef* I2Cx, uint8_t I2C_Register);	uint32_t I2C_ReadRegister(I2C_TypeDef* I2Cx, uint8_t I2C_Register);
	void I2C_SendData(I2C_TypeDef* I2Cx, uint8_t Data);	void I2C_SendData(I2C_TypeDef* I2Cx, uint8_t Data);
	uint8_t I2C_ReceiveData(I2C_TypeDef* I2Cx);	uint8_t I2C_ReceiveData(I2C_TypeDef* I2Cx);
DMA 管理	void I2C_DMACmd(I2C_TypeDef* I2Cx, FunctionalState NewState);	void I2C_DMACmd(I2C_TypeDef* I2Cx, uint32_t I2C_DMAReq, FunctionalState NewState);
	void I2C_DMALastTransferCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);	NA

表 22. STM32F10xx 与 STM32F0xx I2C 驱动程序 API 对应关系（续）

	STM32F10xx I2C 驱动程序 API	STM32F0xx I2C 驱动程序 API
中断和标志管理	FlagStatus I2C_GetFlagStatus(I2C_TypeDef* I2Cx, uint32_t I2C_FLAG);	FlagStatus I2C_GetFlagStatus(I2C_TypeDef* I2Cx, uint32_t I2C_FLAG);
	void I2C_ClearFlag(I2C_TypeDef* I2Cx, uint32_t I2C_FLAG);	void I2C_ClearFlag(I2C_TypeDef* I2Cx, uint32_t I2C_FLAG);
	ITStatus I2C_GetITStatus(I2C_TypeDef* I2Cx, uint32_t I2C_IT);	ITStatus I2C_GetITStatus(I2C_TypeDef* I2Cx, uint32_t I2C_IT);
	void I2C_ClearITPendingBit(I2C_TypeDef* I2Cx, uint32_t I2C_IT);	void I2C_ClearITPendingBit(I2C_TypeDef* I2Cx, uint32_t I2C_IT);

颜色说明：

= 新函数

= 函数不变，但 API 已更改

= 函数不可用 (NA)

尽管 STM32F1 和 STM32F0 中的某些 API 函数相同，但在大多数情况下，从 STM32F1 移植到 STM32F0 时需要重新写入应用程序代码。不过，利用意法半导体提供的“I2C 通信外设应用程序库 (CPAL)”，即可从 STM32F1 无缝移植到 STM32F0：用户只需修改几项设置，而无需对应用程序代码进行任何更改。有关 STM32F1 I2C CPAL 的详细信息，请参见 UM1029。对于 STM32F0，标准外设库程序包内提供了 I2C CPAL。

4.14 SPI 驱动程序

与 STM32F10xx SPI 相比，STM32F0xx SPI 中包括一些新特性。*表 23* 介绍了 STM32F10x 与 STM32F0xx 库之间的 SPI 驱动程序 API 的对应关系。

表 23. STM32F10xx 与 STM32F0xx SPI 驱动程序 API 对应关系

	STM32F10xx SPI 驱动程序 API	STM32F0xx SPI 驱动程序 API
初始化和配置	void SPI_I2S_DeInit(SPI_TypeDef* SPIx);	void SPI_I2S_DeInit(SPI_TypeDef* SPIx);
	void SPI_Init(SPI_TypeDef* SPIx, SPI_InitTypeDef* SPI_InitStruct);	void SPI_Init(SPI_TypeDef* SPIx, SPI_InitTypeDef* SPI_InitStruct);
	void I2S_Init(SPI_TypeDef* SPIx, I2S_InitTypeDef* I2S_InitStruct);	void I2S_Init(SPI_TypeDef* SPIx, I2S_InitTypeDef* I2S_InitStruct);
	void SPI_StructInit(SPI_InitTypeDef* SPI_InitStruct);	void SPI_StructInit(SPI_InitTypeDef* SPI_InitStruct);
	void I2S_StructInit(I2S_InitTypeDef* I2S_InitStruct);	void I2S_StructInit(I2S_InitTypeDef* I2S_InitStruct);
	NA	void SPI_TIModeCmd(SPI_TypeDef* SPIx, FunctionalState NewState);
	NA	void SPI_NSSPulseModeCmd(SPI_TypeDef* SPIx, FunctionalState NewState);
	void SPI_Cmd(SPI_TypeDef* SPIx, FunctionalState NewState);	void SPI_Cmd(SPI_TypeDef* SPIx, FunctionalState NewState);
	void I2S_Cmd(SPI_TypeDef* SPIx, FunctionalState NewState);	void I2S_Cmd(SPI_TypeDef* SPIx, FunctionalState NewState);
	void SPI_DataSizeConfig(SPI_TypeDef* SPIx, uint16_t SPI_DataSize);	void SPI_DataSizeConfig(SPI_TypeDef* SPIx, uint16_t SPI_DataSize);
	NA	void SPI_RxFIFOThresholdConfig(SPI_TypeDef* SPIx, uint16_t SPI_RxFIFOThreshold);
	NA	void SPI_BiDirectionalLineConfig(SPI_TypeDef* SPIx, uint16_t SPI_Direction);
	void SPI_NSSInternalSoftwareConfig(SPI_TypeDef* SPIx, uint16_t SPI_NSSInternalSoft);	void SPI_NSSInternalSoftwareConfig(SPI_TypeDef* SPIx, uint16_t SPI_NSSInternalSoft);
	void SPI_SSOutputCmd(SPI_TypeDef* SPIx, FunctionalState NewState);	void SPI_SSOutputCmd(SPI_TypeDef* SPIx, FunctionalState NewState);
数据传输	void SPI_I2S_SendData(SPI_TypeDef* SPIx, uint16_t Data);	void SPI_SendData8(SPI_TypeDef* SPIx, uint8_t Data); void SPI_I2S_SendData16(SPI_TypeDef* SPIx, uint16_t Data);
	uint16_t SPI_I2S_ReceiveData(SPI_TypeDef* SPIx);	uint8_t SPI_ReceiveData8(SPI_TypeDef* SPIx); uint16_t SPI_I2S_ReceiveData16(SPI_TypeDef* SPIx);

表 23. STM32F10xx 与 STM32F0xx SPI 驱动程序 API 对应关系（续）

	STM32F10xx SPI 驱动程序 API	STM32F0xx SPI 驱动程序 API
硬件 CRC 计算函数	NA	void SPI_CRCLengthConfig(SPI_TypeDef* SPIx, uint16_t SPI_CRCLength);
	void SPI_TransmitCRC(SPI_TypeDef* SPIx);	void SPI_TransmitCRC(SPI_TypeDef* SPIx);
	void SPI_CalculateCRC(SPI_TypeDef* SPIx, FunctionalState NewState);	void SPI_CalculateCRC(SPI_TypeDef* SPIx, FunctionalState NewState);
	uint16_t SPI_GetCRC(SPI_TypeDef* SPIx, uint8_t SPI_CRC);	uint16_t SPI_GetCRC(SPI_TypeDef* SPIx, uint8_t SPI_CRC);
	uint16_t SPI_GetCRCPolynomial(SPI_TypeDef* SPIx);	uint16_t SPI_GetCRCPolynomial(SPI_TypeDef* SPIx);
DMA 传输	void SPI_I2S_DMACmd(SPI_TypeDef* SPIx, uint16_t SPI_I2S_DMAReq, FunctionalState NewState);	void SPI_I2S_DMACmd(SPI_TypeDef* SPIx, uint16_t SPI_I2S_DMAReq, FunctionalState NewState);
	NA	void SPI_LastDMATransferCmd(SPI_TypeDef* SPIx, uint16_t SPI_LastDMATransfer);
中断和标志管理	void SPI_I2S_ITConfig(SPI_TypeDef* SPIx, uint8_t SPI_I2S_IT, FunctionalState NewState);	void SPI_I2S_ITConfig(SPI_TypeDef* SPIx, uint8_t SPI_I2S_IT, FunctionalState NewState);
	NA	uint16_t SPI_GetTransmissionFIFOStatus(SPI_TypeDef* SPIx);
	NA	uint16_t SPI_GetReceptionFIFOStatus(SPI_TypeDef* SPIx);
	FlagStatus SPI_I2S_GetFlagStatus(SPI_TypeDef* SPIx, uint16_t SPI_I2S_FLAG);	FlagStatus SPI_I2S_GetFlagStatus(SPI_TypeDef* SPIx, uint16_t SPI_I2S_FLAG);(*)
	void SPI_I2S_ClearFlag(SPI_TypeDef* SPIx, uint16_t SPI_I2S_FLAG); void SPI_I2S_ClearITPendingBit(SPI_TypeDef* SPIx, uint8_t SPI_I2S_IT);	void SPI_I2S_ClearFlag(SPI_TypeDef* SPIx, uint16_t SPI_I2S_FLAG);(*)
	ITStatus SPI_I2S_GetITStatus(SPI_TypeDef* SPIx, uint8_t SPI_I2S_IT);	ITStatus SPI_I2S_GetITStatus(SPI_TypeDef* SPIx, uint8_t SPI_I2S_IT);(*)

颜色说明：

= 新函数

= 函数不变，但 API 已更改

= 函数不可用 (NA)

(*) 相较 STM32F10xx 驱动程序 API 而言，STM32F0xx 中有多个标志（TI 帧格式错误）可以生成事件。

4.15 USART 驱动程序

与 STM32F10xx USART 相比，STM32F0xx USART 中包括一些增强功能。*表 24* 介绍了 STM32F10x 与 STM32F0xx 库之间的 USART 驱动程序 API 的对应关系。

表 24. STM32F10x 与 STM32F0xx USART 驱动程序 API 对应关系

	STM32F10xx USART 驱动程序 API	STM32F0xx USART 驱动程序 API
初始化和配置	void USART_DeInit(USART_TypeDef* USARTx);	void USART_DeInit(USART_TypeDef* USARTx);
	void USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct);	void USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct);
	void USART_StructInit(USART_InitTypeDef* USART_InitStruct);	void USART_StructInit(USART_InitTypeDef* USART_InitStruct);
	void USART_ClockInit(USART_TypeDef* USARTx, USART_ClockInitTypeDef* USART_ClockInitStruct);	void USART_ClockInit(USART_TypeDef* USARTx, USART_ClockInitTypeDef* USART_ClockInitStruct);
	void USART_ClockStructInit(USART_ClockInitTypeDef* USART_ClockInitStruct);	void USART_ClockStructInit(USART_ClockInitTypeDef* USART_ClockInitStruct);
	void USART_Cmd(USART_TypeDef* USARTx, FunctionalState NewState);	void USART_Cmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_DirectionModeCmd(USART_TypeDef* USARTx, uint32_t USART_DirectionMode, FunctionalState NewState);
	void USART_SetPrescaler(USART_TypeDef* USARTx, uint8_t USART_Prescaler);	void USART_SetPrescaler(USART_TypeDef* USARTx, uint8_t USART_Prescaler);
	void USART_OverSampling8Cmd(USART_TypeDef* USARTx, FunctionalState NewState);	void USART_OverSampling8Cmd(USART_TypeDef* USARTx, FunctionalState NewState);
	void USART_OneBitMethodCmd(USART_TypeDef* USARTx, FunctionalState NewState);	void USART_OneBitMethodCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_MSBFirstCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_DataInvCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_InvPinCmd(USART_TypeDef* USARTx, uint32_t USART_InvPin, FunctionalState NewState);
	NA	void USART_SWAPPinCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_ReceiverTimeOutCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_SetReceiverTimeOut(USART_TypeDef* USARTx, uint32_t USART_ReceiverTimeOut);

表 24. STM32F10x 与 STM32F0xx USART 驱动程序 API 对应关系（续）

	STM32F10xx USART 驱动程序 API	STM32F0xx USART 驱动程序 API
停机模式	NA	void USART_STOPModeCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_StopModeWakeUpSourceConfig(USART_TypeDef* USARTx, uint32_t USART_WakeUpSource);
波特率自动检测	NA	void USART_AutoBaudRateCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_AutoBaudRateConfig(USART_TypeDef* USARTx, uint32_t USART_AutoBaudRate);
数据传输	void USART_SendData(USART_TypeDef* USARTx, uint16_t Data);	void USART_SendData(USART_TypeDef* USARTx, uint16_t Data);
	uint16_t USART_ReceiveData(USART_TypeDef* USARTx);	uint16_t USART_ReceiveData(USART_TypeDef* USARTx);
多处理器通信	void USART_SetAddress(USART_TypeDef* USARTx, uint8_t USART_Address);	void USART_SetAddress(USART_TypeDef* USARTx, uint8_t USART_Address);
	NA	void USART_MuteModeWakeUpConfig(USART_TypeDef* USARTx, uint32_t USART_WakeUp);
	NA	void USART_MuteModeCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_AddressDetectionConfig(USART_TypeDef * USARTx, uint32_t USART_AddressLength);
LIN 模式	void USART_LINBreakDetectLengthConfig(USART_TypeDef* USARTx, uint32_t USART_LINBreakDetectLength);	void USART_LINBreakDetectLengthConfig(USART_TypeDef* USARTx, uint32_t USART_LINBreakDetectLength);
	void USART_LINCmd(USART_TypeDef* USARTx, FunctionalState NewState);	void USART_LINCmd(USART_TypeDef* USARTx, FunctionalState NewState);
半双工模式	void USART_HalfDuplexCmd(USART_TypeDef* USARTx, FunctionalState NewState);	void USART_HalfDuplexCmd(USART_TypeDef* USARTx, FunctionalState NewState);

表 24. STM32F10x 与 STM32F0xx USART 驱动程序 API 对应关系（续）

	STM32F10xx USART 驱动程序 API	STM32F0xx USART 驱动程序 API
智能卡模式	void USART_SmartCardCmd(USART_TypeDef* USARTx, FunctionalState NewState);	void USART_SmartCardCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	void USART_SmartCardNACKCmd(USART_TypeDef* USARTx, FunctionalState NewState);	void USART_SmartCardNACKCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	void USART_SetGuardTime(USART_TypeDef* USARTx, uint8_t USART_GuardTime);	void USART_SetGuardTime(USART_TypeDef* USARTx, uint8_t USART_GuardTime);
	NA	void USART_SetAutoRetryCount(USART_TypeDef* USARTx, uint8_t USART_AutoCount);
	NA	void USART_SetBlockLength(USART_TypeDef* USARTx, uint8_t USART_BlockLength);
IrDA 模式	void USART_IrDAConfig(USART_TypeDef* USARTx, uint32_t USART_IrDAMode);	void USART_IrDAConfig(USART_TypeDef* USARTx, uint32_t USART_IrDAMode);
	void USART_IrDACmd(USART_TypeDef* USARTx, FunctionalState NewState);	void USART_IrDACmd(USART_TypeDef* USARTx, FunctionalState NewState);
RS485 模式	NA	void USART_DECmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_DEPolarityConfig(USART_TypeDef* USARTx, uint32_t USART_DEPolarity);
	NA	void USART_SetDEAssertionTime(USART_TypeDef* USARTx, uint32_t USART_DEAssertionTime);
	NA	void USART_SetDEDeassertionTime(USART_TypeDef * USARTx, uint32_t USART_DEDeassertionTime);
DMA 传输	void USART_DMACmd(USART_TypeDef* USARTx, uint32_t USART_DMAReq, FunctionalState NewState);	void USART_DMACmd(USART_TypeDef* USARTx, uint32_t USART_DMAReq, FunctionalState NewState);
	void USART_DMAReceptionErrorConfig(USART_TypeDef * USARTx, uint32_t USART_DMAOnError);	void USART_DMAReceptionErrorConfig(USART_TypeDef* USARTx, uint32_t USART_DMAOnError);

表 24. STM32F10x 与 STM32F0xx USART 驱动程序 API 对应关系（续）

	STM32F10xx USART 驱动程序 API	STM32F0xx USART 驱动程序 API
中断和标志管理	void USART_ITConfig(USART_TypeDef* USARTx, uint16_t USART_IT, FunctionalState NewState);	void USART_ITConfig(USART_TypeDef* USARTx, uint32_t USART_IT, FunctionalState NewState);
	NA	void USART_RequestCmd(USART_TypeDef* USARTx, uint32_t USART_Request, FunctionalState NewState);
	NA	void USART_OverrunDetectionConfig(USART_TypeDef * USARTx, uint32_t USART_OVRDetection);
	FlagStatus USART_GetFlagStatus(USART_TypeDef* USARTx, uint16_t USART_FLAG);	FlagStatus USART_GetFlagStatus(USART_TypeDef* USARTx, uint32_t USART_FLAG);
	void USART_ClearFlag(USART_TypeDef* USARTx, uint16_t USART_FLAG);	void USART_ClearFlag(USART_TypeDef* USARTx, uint32_t USART_FLAG);
	ITStatus USART_GetITStatus(USART_TypeDef* USARTx, uint32_t USART_IT);	ITStatus USART_GetITStatus(USART_TypeDef* USARTx, uint32_t USART_IT);
	void USART_ClearITPendingBit(USART_TypeDef* USARTx, uint32_t USART_IT);	void USART_ClearITPendingBit(USART_TypeDef* USARTx, uint32_t USART_IT);

颜色说明：

= 新函数

= 函数不变，但 API 已更改

= 函数不可用 (NA)

4.16 IWDG 驱动程序

STM32F10xx 和 STM32F0xx 器件上的现有 IWDG 具有相同规格，不过，F0 系列中新增了窗口功能特性，可用于检测外部振荡器上的过频率。下表列出了 IWDG 驱动程序 API。

表 25. STM32F10xx 与 STM32F0xx IWDG 驱动程序 API 对应关系

	STM32F10xx IWDG 驱动程序 API	STM32F0xx IWDG 驱动程序 API
预分频器和计数器配置	void IWDG_WriteAccessCmd(uint16_t IWDG_WriteAccess);	void IWDG_WriteAccessCmd(uint16_t IWDG_WriteAccess);
	void IWDG_SetPrescaler(uint8_t IWDG_Prescaler);	void IWDG_SetPrescaler(uint8_t IWDG_Prescaler);
	void IWDG_SetReload(uint16_t Reload);	void IWDG_SetReload(uint16_t Reload);
	void IWDG_ReloadCounter(void);	void IWDG_ReloadCounter(void);
	NA	void IWDG_SetWindowValue(uint16_t WindowValue);
IWDG 激活	void IWDG_Enable(void);	void IWDG_Enable(void);
标志管理	FlagStatus IWDG_GetFlagStatus(uint16_t IWDG_FLAG);	FlagStatus IWDG_GetFlagStatus(uint16_t IWDG_FLAG);

颜色说明：

= 新函数

= 函数不变，但 API 已更改

= 函数不可用 (NA)

5 版本历史

表 26. 文档版本历史

日期	版本	变更
2012 年 07 月 10 日	1	初始版本
2013 年 01 月 24 日	2	修改了*表 2： STM32F1 系列和 STM32F0 系列引脚排列区别*。 在*表 5* 下增加了注释。 修改了*表 12* 中 STM32F0 系列的*读保护*。 修改了*表 13* 中 STM32F0 系列的*电源要求*。 修改了*表 14： STM32F0 系列与 STM32F1 系列之间的 PWR 区别*（增加了关于 STM32F06x 系列）。

请仔细阅读：

中文翻译仅为方便阅读之目的。该翻译也许不是对本文档最新版本的翻译，如有任何不同，以最新版本的英文原版文档为准。

本文档中信息的提供仅与ST产品有关。意法半导体公司及其子公司（“ST”）保留随时对本文档及本文所述产品与服务进行变更、更正、修改或改进的权利，恕不另行通知。

所有ST产品均根据ST的销售条款出售。

买方自行负责对本文所述ST产品和服务的选择和使用，ST概不承担与选择或使用本文所述ST产品和服务相关的任何责任。

无论之前是否有过任何形式的表示，本文档不以任何方式对任何知识产权进行任何明示或默示的授权或许可。如果本文档任何部分涉及任何第三方产品或服务，不应被视为 ST 授权使用此类第三方产品或服务，或许可其中的任何知识产权，或者被视为涉及以任何方式使用任何此类第三方产品或服务或其中任何知识产权的保证。

除非在ST的销售条款中另有说明，否则，ST对ST产品的使用和/或销售不做任何明示或默示的保证，包括但不限于有关适销性、适合特定用途（及其依据任何司法管辖区的法律的对应情况），或侵犯任何专利、版权或其他知识产权的默示保证。

意法半导体的产品不得应用于武器。此外，意法半导体产品也不是为下列用途而设计并不得应用于下列用途：（A）对安全性有特别要求的应用，例如，生命支持、主动植入设备或对产品功能安全有要求的系统；（B）航空应用；（C）汽车应用或汽车环境，且/或（D）航天应用或航天环境。如果意法半导体产品不是为前述应用设计的，而采购商擅自将其用于前述应用，即使采购商向意法半导体发出了书面通知，采购商仍将独自承担因此而导致的任何风险，意法半导体的产品设计规格明确指定的汽车、汽车安全或医疗工业领域专用产品除外。根据相关政府主管部门的规定，ESCC、QML或JAN正式认证产品适用于航天应用。

经销的ST产品如有不同于本文档中提出的声明和/或技术特点的规定，将立即导致ST针对本文所述ST产品或服务授予的任何保证失效，并且不应以任何形式造成或扩大ST的任何责任。

意法半导体集团公司

澳大利亚 - 比利时 - 巴西 - 加拿大 - 中国 - 捷克共和国 - 芬兰 - 法国 - 德国 - 中国香港 - 印度 - 以色列 - 意大利 - 日本 - 马来西亚 - 马耳他 - 摩洛哥 – 菲律宾 – 新加坡 - 西班牙 - 瑞典 - 瑞士 - 英国 - 美国

www.st.com