

STM32F1 到 STM32F0 的变化

简介:

对于 STM32 微控制器的设计者来说,使一种类型的微控制器能够方便的被同一系列中的其它型号的微控制器代替是非常重要的。尤其当产品需求量增加,内存任务量增大,I/O 口数量升高的时候,更加需要不同微控制器的间移植。另一方面,为了降低成本,可能会被迫使用更小的组件以及缩小 PCB 板的面积。

写这篇应用注释的目的是帮助读者分析从现有的 STM32F1 移植到 STM32F0 的所需步骤。它收集了最重要的信息以及你需要处理的重要方面。

为了将应用程序从 STM32F1 系列移植到 STM32F0 系列,需要分析硬件移植、外设移植以及固件移植。

为了充分理解本应用注释的内容,读者应对 STM32 微控制器家族有一定的了解。可以参考官网提供的以下文档:

- STM32F1 系列参考手册 (RM0008 和 RM0041)、STM32F1 数据手册、STM32F1 闪存程序设计手册 (PM0075, PM0063 和 PM0068)。
- STM32F0 系列参考手册、STM32F1 数据手册。

为了概要了解整个 STM32 系列中各个系列的对比以及不同特性,请参考:《AN3364 STM32 微控制器移植及兼容性指导手册》。

表一:可适用产品

Table 1. Applicable products

Type	Product
Microcontroller	STM32 F0 Entry-level
	STM32 F1 Mainstream

目录

- 1 硬件移植
- 2 启动方式兼容性
- 3 外部设备移植
 - 3.1 STM32 产品的移植亲和性
 - 3.2 系统架构
 - 3.3 内存映射
 - 3.4 重置和时钟控制器（RCC）接口
 - 3.5 DMA 接口
 - 3.6 中断向量
 - 3.7 GPIO 接口
 - 3.8 外部中断/事件来源选择
 - 3.9 FLASH 接口
 - 3.10 ADC 接口
 - 3.11 电源接口
 - 3.12 实时时钟（RTC）接口
 - 3.13 SPI 接口
 - 3.14 I2C 接口
 - 3.15 USART 接口
 - 3.16 CEC 接口
- 4 固件库移植
 - 4.1 移植步骤
 - 4.2 时钟控制器驱动
 - 4.3 FLASH 驱动
 - 4.4 循环冗余校验驱动
 - 4.5 GPIO 配置更新
 - 4.5.1 输出模式
 - 4.5.2 输入模式
 - 4.5.3 模拟模式
 - 4.5.4 复用模式
 - 4.6 外部中断线路 0
 - 4.7 嵌套向量中断控制器中断配置
 - 4.8 ADC 配置
 - 4.9 DAC 驱动
 - 4.10 电源驱动
 - 4.11 备份数据寄存器
 - 4.12 CEC 应用代码
 - 4.13 I2C 驱动

4.14 SPI 驱动

4.15 串口驱动

4.16 独立看门狗驱动

5 修订记录

表格汇总

表 1	适用产品
表 2	STM32F1 系列与 STM32F0 系列引脚区别
表 3	启动模式
表 4	STM32F1 及 F0 系列外设兼容性分析
表 5	STM32F0 及 F1 系列网际协议总线区别
表 6	F0 与 F1 时钟控制器区别
表 7	F1 移植到 F0 的系统时钟配置代码示例
表 8	时钟控制寄存器用于外设通道配置
表 9	F1 与 F0 的 DMA 请求区别
表 10	F0 与 F1 中断向量区别
表 11	F0 与 F1 的 GPIO 区别
表 12	F0 与 F1 的 FLASH 区别
表 13	F0 与 F1 的 ADC 区别
表 14	F0 与 F1 电源模块区别
表 15	F0 与 F1 的时钟源应用接口匹配
表 16	F0 与 F1 的 FLASH 驱动程序接口匹配
表 17	F0 与 F1 的 CRC 驱动程序接口匹配
表 18	F0 与 F1 的 MISC 驱动程序接口匹配
表 19	F0 与 F1 的 DAC 驱动程序接口匹配
表 20	F0 与 F1 的电源驱动程序接口匹配
表 21	F0 与 F1 的 CEC 驱动程序接口匹配
表 22	F0 与 F1 的 I2C 驱动程序接口匹配
表 23	F0 与 F1 的 SPI 驱动程序接口匹配
表 24	F0 与 F1 的串口驱动程序接口匹配
表 25	F0 与 F1 的独立看门狗驱动程序接口匹配
表 26	文件修订记录

1. 硬件移植

适用于初学者的 STM32F0 和多用途的 STM32F1 系列芯片的引脚是一一对应兼容的。对于这两个系列，所有的外设使用相同的引脚，但在程序包的封装上略有不同。从 F1 到 F0 的转变只是一些引脚被压紧了（被压紧的引脚为表二中粗体表示的部分）。

Table 2. STM32F1 series and STM32F0 series pinout differences

STM32F1 series			STM32F0 series		
QFP48	QFP64	Pino	QFP48	QFP64	Pino
5	5	PD0 - OSC_IN	5	5	PH0 - OSC_IN
6	6	PD1 - OSC_OUT	6	6	PH1 - OSC_OUT
-	1	VSS_4	-	1	PF4
-	1	VDD_4	-	1	PF5
35	4	VSS_2	3	4	PF6
36	4	VDD_2	3	4	PF7
20	2	Boot1/PB2	2	2	PB2

从 F1 到 F0 的移植对引脚输出没有任何影响，除了使用者可以在 VSS/VDD 的使用中节省 2 或 4 个 GPIO 引脚（具体取决于所用的程序包）。

2. 启动模式兼容性

F0 系列启动模式的选择有别于 F1 系列器件。F0 通过内存地址为 0X1FFFF800 的用户选项字节中的选择位来获取 nBOOT1 的值，而不是利用两个引脚进行设置。连同 BOOT0 引脚，选择启动模式到 FLASH 存储、静态随机存储或系统内存。表三概要说明了用于选择启动模式的几种不同配置。

Table 3. Boot modes

F0/F1 Boot mode selection		Boot mode	Aliasing
BOOT1	BOOT0		
x	0	Main Flash memory	Main Flash memory is selected as boot space
0	1	System memory	System memory is selected as boot space
1	1	Embedded SRAM	Embedded SRAM is selected as boot space

注意：BOOT1 的值与 nBOOT1 选项位的值相反。

3. 外设移植

如表3 所列，外设共分三类。常见的外设由固件库支持，不需要作任何改变，除了某些现在已用不到的实例。你可以改变这些例程，当然也可以改变其相关部分（如时钟配置、引脚配置、中断及DMA 请求等）。

一些改进的外设在F0 与F1 上有所不同，例如ADC、RCC、RTC 等，对这些外设应该进行更新移植，以便利用它的增强功能。

所有F0 系列的改进外设都是为了制作更小的硅晶片以便在最后更加经济的产品中获得更高端的性能，以及修复目前F1 系列产品的某些局限性。

3.1 STM32 产品的移植亲和性

STM32 系列产品嵌入了一套可分成三类的外设：

- 第一类外设，很显然，是适用于所有产品的一类。这一类外设是完全相同的，所以它们拥有结构、寄存器和控制位。要移植后的应用程序层保持相同的功能，固件无需做任何改变。所有的特性均保持不变。
- 第二类外设也是适用于所有产品但有小的调整的类型（一般是为了支持新的特性）。对这一类外设，移植也是非常简单的，不需要任何重大的新改进。
- 第三类外设从一类产品移植到另一类产品时有相当大的改变（新的结构、新的特性）。对于这一类的外设，应用程序层的移植需要新的改进。

表 4 给出了上述分类的综述：

Table 4. STM32 peripheral compatibility analysis F1 versus F0 series

Peripheral	F1 series	F0 series	Compatibility		
			Feature	Pinout	FW driver
SPI	Yes	Yes++	Two FIFO available, 4-bit to 16-bit data size selection	Identical	Partial compatibility
WWDG	Yes	Yes	Same features	NA	Full compatibility
IWDG	Yes	Yes+	Added a Window mode	NA	Full compatibility
DBGMCU	Yes	Yes	No JTAG, No Trace	Identical for the SWD	Partial compatibility
CRC	Yes	Yes++	Added reverse capability and initial CRC value	NA	Partial compatibility
EXTI	Yes	Yes+	Some peripherals are able to generate event in stop mode	Identical	Full compatibility
CEC	Yes	Yes++	Kernel clock, arbitration lost flag and automatic transmission retry, multi-address config, wakeup from stop mode	Identical	Partial compatibility
DMA	Yes	Yes	1 DMA controller with 5 channels	NA	Full compatibility
TIM	Yes	Yes+	Enhancement	Identical	Full compatibility
PWR	Yes	Yes+	No Vref, Vdda can be greater than Vdd, 1.8 mode for core.	Identical for the same	Partial compatibility
RCC	Yes	Yes+	New HSI14 dedicated to ADC	PD0 & PD1 => PF0 & PF1 for the osc	Partial compatibility
USART	Yes	Yes+	Choice for independent clock sources, timeout feature, wakeup from stop mode	Identical	Full compatibility
I2C	Yes	Yes++	Communication events managed by HW, FM+, wakeup from stop mode, digital filter	Identical	New driver
DAC	Yes	Yes+	DMA underrun interrupt	Identical	Full compatibility

ADC	Yes	Yes++	Same analogic part, but new digital interface	Identical	Partial compatibility
RTC	Yes	Yes++	Subsecond precision, digital calibration circuit, time-stamp function for event saving, programmable alarm	Identical for the same feature	New driver
FLASH	Yes	Yes+	Option byte modified	NA	Partial compatibility
GPIO	Yes	Yes++	New peripheral	4 new GPIOs	Partial compatibility

Table 4. STM32 peripheral compatibility analysis F1 versus F0 series (continued)

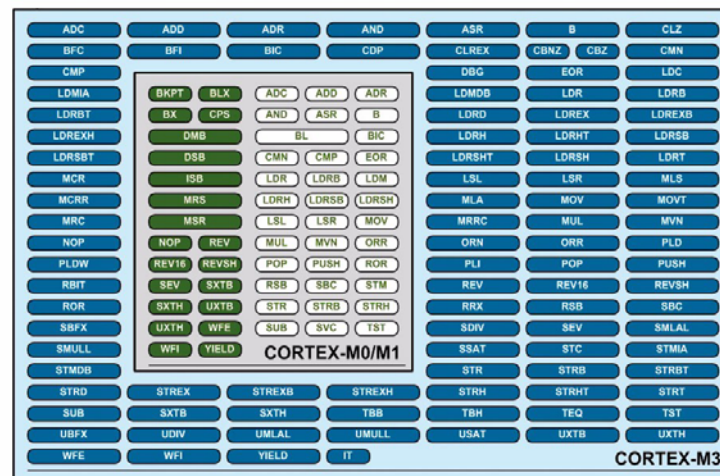
Peripheral	F1 series	F0 series	Compatibility		
			Feature	Pinout	FW driver
CAN	Yes	NA	NA	NA	NA
USB FS Device	Yes	NA	NA	NA	NA
Ethernet	Yes	NA	NA	NA	NA
SDIO	Yes	NA	NA	NA	NA
FSMC	Yes	NA	NA	NA	NA
Touch Sensing	NA	Yes	NA	NA	NA
COMP	NA	Yes	NA	NA	NA
SYSCFG	NA	Yes	NA	NA	NA

注意： Yes++=新特性或新架构
Yes+=相同的特性，但在技术参数上有改变或提升
Yes=有效特性
NA=无效特性

3.2 系统架构

STM32F0 系列微控制器的设计目标是面向初学者市场，功耗低且易于操作。为了实现这个目标同时保持 STM32 的高端性能，处理器换成了 Cortex-M0。他将小面积硅片和最小限度的代码封装连接在一起，实现了低价的 32 位应用程序运行。图 1 列出了 M3 与 M0 的一系列对应指令。从 F1 到 F0 的移植需要重新编译这些代码以免使用了不可用的特性。

Figure 1. System architecture



在微处理器的而组织结构上也做了重要修改，从哈佛结构改为了冯·诺依曼结构、降低了系统复杂度、集中处理 SW 调试模式以简化这种精确特性。

3.3 内存映射

F0 系列的外设地址映射相比于 F1 系列已有了改变。最主要的改变集中于 GPIO 从 APB 总线移动到了 AHB 总线以达到最高的运行速度。

表 5 提供了 F0 系列与 F1 系列的外设地址映像对比。

Table 5. IP bus mapping differences between STM32F0 and STM32F1 series

Peripheral	STM32 F0 series		STM32 F1 series	
	Bus	Base address	Bus	Base address
TSC	AHB1	0x40024000	NA	NA
CRC		0x40023000	AHB	0x40023000
FLITF		0x40022000		0x40022000
RCC		0x40021000		0x40021000
DMA1/DMA		0x40020000		0x40020000

Table 5. IP bus mapping differences between STM32F0 and STM32F1 series

Peripheral	STM32 F0 series		STM32 F1 series	
	Bus	Base address	Bus	Base address
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 (through 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	Own Clock	0x40003000		0x40003000
WWDG	APB1	0x40002C00		0x40002C00
RTC	APB1 (through PWR)	0x40002800 (inc. BKP registers)		0x40002800

Table 5. IP bus mapping differences between STM32F0 and STM32F1 series

Peripheral	STM32 F0 series		STM32 F1 series	
	Bus	Base address	Bus	Base address
TIM14	APB1	0x40002000	NA	NA
TIM6		0x40001000	APB1	0x40001000
TIM3		0x40000400		0x40000400
TIM2		0x40000000		0x40000000
USB device FS SRAM	NA	NA	APB1	0x40006000
USB device FS	NA	NA		0x40005C00
USART3	NA	NA		0x40004800
TIM7	NA	NA		0x40001400
TIM4	NA	NA		0x40000800
FSMC Registers	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

Table 5. IP bus mapping differences between STM32F0 and STM32F1 series

Peripheral	STM32 F0 series		STM32 F1 series	
	Bus	Base address	Bus	Base address
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 registers	NA	NA		0x40006C00
AFIO	NA	NA	APB2	0x40010000

注意：NA=不可用特性

3.4 重置和时钟控制器接口

F0 系列与 F1 有关 RCC 模块的不同之处在表六中列出：

Table 6. RCC differences between STM32F1 and STM32F0 series

RCC	STM32 F1 series	STM32 F0 series
HSI 14	NA	High speed internal oscillator dedicated to ADC
HSI	8 MHz RC factory-trimmed	Similar
LSI	40 KHz RC	Similar
HSE	3 - 25 MHz depending on the product line used	4 - 32 MHz
LSE	32.768 KHz	Similar
PLL	<ul style="list-style-type: none"> - Connectivity line: main PLL + 2 PLLs for I2S, Ethernet and OTG FS clock - Other product lines: main PLL 	Main PLL

System clock source	HSI, HSE or PLL	Similar
---------------------	-----------------	---------

Table 6. RCC differences between STM32F1 and STM32F0 series (continued)

RCC	STM32 F1 series	STM32 F0 series
System clock frequency	- Up to 72 MHz depending on the product line used	Up to 48 MHz
APB1/APB frequency	Up to 36 MHz	Up to 48 MHz
RTC clock source	LSI, LSE or HSE/128	LSI, LSE or HSE clock divided by 32
MCO clock source	- MCO pin (PA8) - Connectivity line: HSI, HSE, PLL/2, SYSClk, PLL2, PLL3 or XT1	MCO(PA8): SYSClk, HSI, HSE, HSI14, PLLCLK/2, LSE, LSI
Internal oscillator measurement / calibration	LSI connected to TIM5 CH4 IC: can measure LSI with respect to HSI/HSE clock	- LSE & LSI clocks are indirectly measured through MCO by the timer TIM14 with respect to HSI/HSE clock - HSI14/HSE are indirectly measured through MCO by means of the TIM14 channel 1 input capture with respect to HSI clock.

除了上表中所描述的不同点以外，在移植的过程中还需要注意以下几点改变。

1. 系统时钟配置：从 F1 到 F0 的移植过程中，有关系统时钟配置代码只有几处设置需要校正；主要是 FLASH 配置（为系统频率配置正确的等待模式，使能或失能预取功能）、锁相环参数配置：

- a) 假如 HSE 或 HSI 被直接用作系统时钟源，只需对 FLASH 参数进行修改。
- b) 如果锁相环被用作系统时钟源，FLASH 参数和锁相环参数都要进行校正。

下面的表 7 提供了一个从 F1 到 F0 系统时钟配置移植的例子：

- STM32F1 系列芯片运行在最高性能模式：系统时钟为 24Hz（锁相环用作系统时钟源），FLASH 设置在等待模式 0 及使能预取队列。
- STM32F0 系列芯片运行在最高性能模式：系统时钟为 48Hz（锁相环用作系统时钟源），FLASH 设置在等待模式 1 及使能预取功能。

如表 7 所示，在 F0 系列运行时，只有 FLASH 参数和锁相环配置需要重新写入。但是，HSE，AHB 预分频器和系统时钟源配置都未改变，APB 总线的预分频器也适用 F0 系列 APB 总线的最高频率。

- 注意：
- 1 表 7 中所列的源码是有意简化的，并且是基于 RCC 和 FLASH 的寄存器都处于复位值这一假设。
 - 2 对于 F0 系列芯片，根据读者的应用程序要求，可以利用时钟配置工具——STM32F0xx_Clock_Configuration.xls, 来生成包含系统时钟配置程序的自定义系统文件。

Table 7. Example of migrating system clock configuration code from F1 to F0

STM32F100x Value Line running at 24 MHz (PLL as clock source) with 0 wait states	STM32F0xx running at 48 MHz (PLL as clock source) with 1 wait state
--	---

```

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

/* Wait till HSE is ready */
while((RCC->CR & RCC_CR_HSERDY) == 0)
{
}

/* Flash configuration -----*/
/* Prefetch ON, Flash 0 wait state */
FLASH->ACR |= FLASH_ACR_PRFTBE | FLASH_ACR_LATENCY_0;

/* AHB and APB prescaler configuration --*/
/* 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 configuration = (HSE / 2) * 6 = 24 MHz */
RCC->CFGR |= (uint32_t)(RCC_CFGR_PLLSRC_PREDIV1 |
RCC_CFGR_PLLXTPRE_PREDIV1_Div2 | RCC_CFGR_PLLMULL6);

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

/* Wait till PLL is ready */
while((RCC->CR & RCC_CR_PLLRDY) == 0)
{
}

/* Select PLL as system clock source ----*/
RCC->CFGR &= (uint32_t)((uint32_t)~(RCC_CFGR_SW));
RCC->CFGR |= (uint32_t)RCC_CFGR_SW_PLL;

/* Wait till PLL is used as system clock source */
while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS) !=
(uint32_t)0x08)
{
}

```

```

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

/* Wait till HSE is ready */
while((RCC->CR & RCC_CR_HSERDY) == 0)
{
}

/* Flash configuration -----*/
/* Prefetch ON, Flash 1 wait state */
FLASH->ACR |= FLASH_ACR_PRFTBE | FLASH_ACR_LATENCY;

/* AHB and APB prescaler configuration --*/
/* HCLK = SYSCLK */
RCC->CFGR |= (uint32_t)RCC_CFGR_HPRE_DIV1;

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

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

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

/* Wait till PLL is ready */
while((RCC->CR & RCC_CR_PLLRDY) == 0)
{
}

/* Select PLL as system clock source ----*/
RCC->CFGR |= (uint32_t)RCC_CFGR_SW_PLL;

/* Wait till PLL is used as system clock source */
while ((RCC->CFGR & (uint32_t)RCC_CFGR_SWS) !=
(uint32_t)RCC_CFGR_SWS_PLL)
{
}

```

2. 外设通道配置：由于 F1 到 F0 的一些外设地址映射发生了变化，读者需要使用不同的寄存器来使能/失能时钟以及进入/退出复位模式。

Table 8. RCC registers used for peripheral access configuration

Bus	Register	Comments
AHB	RCC_AHBRSTR	Used to [enter/exit] the AHB peripheral from reset
	RCC_AHBENR	Used to [enable/disable] the AHB peripheral clock
APB1	RCC_APB1RSTR	Used to [enter/exit] the APB1 peripheral from reset
	RCC_APB1ENR	Used to [enable/disable] the APB1 peripheral clock
APB2	RCC_APB2RSTR	Used to [enter/exit] the APB2 peripheral from reset
	RCC_APB2ENR	Used to [enable/disable] the APB2 peripheral clock

为了配置给定外设的通道，读者首先应知道该外设是挂靠在哪根总线上的。参考表 5，然后根据需要的动作，为表 8 中描述的合适寄存器安排计划。例如，如果 USART1 是挂靠在 APB2 总线上的，读者需要按照如下的指令配置 APB2ENR 寄存器以使能串口一时钟。

RCC->APB2ENR|=RCC_APB2ENR_USART1EN;

3. 外设时钟配置：一些外设拥有独立于系统时钟以外的专用的时钟源，用来产生其操作所必须的时钟信号。
 - a) ADC：在 STM32F0 系列中，ADC 的特点是拥有两种可能的时钟源。
 - 第一种是基于可编程时钟 PCLK，通过预分频器可使 ADC 时钟频率缩减到 1/2 或 1/4 倍后作为 ADC 的输入时钟。
 - 另一种方式是一种全新的特色，一个 14MHz 的专用振荡器被集成在芯片上作为 ADC 的输入时钟频率
 - b) RTC：在 STM32 系列中，RTC 模块的特点是拥有三种可能的时钟源：
 - 第一种是基于 HSE 时钟，预分频器将 HSE 时钟信号分为 1/32 作为 RTC 输入信号。
 - 第二种是 LSE 振荡器
 - 第三种时钟源是值为 40KHz 的集成 RC 振荡器

3.5 DMA 接口

F1 系列与 F0 系列的 DMA 控制器是完全兼容的。

F0 系列使用 5 通道 DMA 控制器而 F1 使用 2 通道控制器。每个通道处理来自一个或多个通道的内存访问请求。

下表列出了 F0 系列与 F1 系列外设对应的 DMA 请求。

Table 9. DMA request differences between STM32F1 series and STM32F0 series

Peripheral	DMA request	STM32F1 series	STM32F0 series
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	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_Channel6 DMA_Channel7
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_Channel7 DMA_Channel6
I2C2	I2C2_Rx I2C2_Tx	DMA1_Channel5 DMA1_Channel4	DMA_Channel5 DMA_Channel4
SDIO	SDIO	DMA2_Channel4	NA

TIM1	TIM1_UP	DMA1_Channel5	DMA_Channel5
	TIM1_CH1	DMA1_Channel2	DMA_Channel2
	TIM1_CH2	DMA1_Channel3	DMA_Channel3
	TIM1_CH3	DMA1_Channel6	DMA_Channel6
	TIM1_CH4	DMA1_Channel4	DMA_Channel4

Table 9.DMA request differences between STM32F1 series and STM32F0 series (continued)

Peripheral	DMA request	STM32F1 series	STM32F0 series
TIM8	TIM8_UP	DMA2_Channel1	NA
	TIM8_CH1	DMA2_Channel3	
	TIM8_CH2	DMA2_Channel5	
	TIM8_CH3	DMA2_Channel1	
	TIM8_CH4	DMA2_Channel2	
TIM2	TIM2_UP	DMA1_Channel2	DMA_Channel2
	TIM2_CH1	DMA1_Channel5	DMA_Channel5
	TIM2_CH2	DMA1_Channel7	DMA_Channel3
	TIM2_CH3	DMA1_Channel1	DMA_Channel1
TIM3	TIM3_UP	DMA1_Channel3	DMA_Channel3
	TIM3_CH1	DMA1_Channel6	DMA_Channel4
	TIM3_TRIG	DMA1_Channel6	DMA_Channel4
	TIM3_CH3	DMA1_Channel2	DMA_Channel2
TIM4	TIM4_UP	DMA1_Channel7	NA
	TIM4_CH1	DMA1_Channel1	
	TIM4_CH2	DMA1_Channel4	
TIM5	TIM5_UP	DMA2_Channel2	NA
	TIM5_CH1	DMA2_Channel5	
	TIM5_CH2	DMA2_Channel4	
	TIM5_CH3	DMA2_Channel2	
	TIM5_CH4	DMA2_Channel1	
TIM6	TIM6_UP	DMA2_Channel3 DMA1_Channel3 ⁽¹⁾	DMA_Channel3
TIM7	TIM7_UP	DMA2_Channel4 DMA1_Channel4 ⁽¹⁾	NA
TIM15	TIM15_UP	DMA1_Channel5	DMA_Channel5
	TIM15_CH1	DMA1_Channel5	DMA_Channel5
	TIM15_TRIG	DMA1_Channel5	DMA_Channel5
	TIM15_COM	DMA1_Channel5	DMA_Channel5
TIM16	TIM16_UP	DMA1_Channel6	DMA_Channel3/DMA_Channel4
	TIM16_CH1	DMA1_Channel6	DMA_Channel3/DMA_Channel4

TIM17	TIM17_UP	DMA1_Channel7	DMA_Channel1/DMA_Channel2
	TIM17_CH1	DMA1_Channel7	DMA_Channel1/DMA_Channel2

1. 对于高密度数据线器件，DAC 的 DMA 请求是分别映射到 DMA1 的通道 3 和通道 4 的。

3.6 中断向量

表 10 给出了 F1 系列与 F0 系列的中断向量对比。

从 Cortex-M3 到 M0 的改变导致了中断向量的减少。这种改变使两种器件有了许多不同。

Table 10. Interrupt vector differences between STM32F1 series and STM32F0 series

Position	STM32F1 series	STM32F0 series
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	Reserved
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_T	SPI2
27	TIM1_CC	USART1
28	TIM2	USART2
29	TIM3	Reserved

Table 10. Interrupt vector differences between STM32F1 series and STM32F0 series

Position	STM32F1 series	STM32F0 series
30	TIM4	CEC
31	I2C1_EV	Reserved
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
63	CAN2_TX	NA

Table 10. Interrupt vector differences between STM32F1 series and STM32F0 series

Position	STM32F1 series	STM32F0 series
64	CAN2_RX01	NA
65	CAN2_RX1	NA
66	CAN2_SCE	NA
67	OTG_FS	NA

1. 取决于所用的生产线

Cortex-M0 内核使用 2 位长度设置中断优先级，省略了副优先级。用户可以在嵌套向量中断控制器中设置 4 级优先级。F1 和 Cortex-M3 内核使用 4 位长度设置，因此可以达到 16 个优先级。

3.7 GPIO 接口

相比于 F1 系列，STM32F0 的 GPIO 外设嵌入了一些新特性，主要如下：

- GPIO 挂载在 AHB 总线上以获取更高的性能
- I/O 引脚多路复用和映射：引脚通过一个多路复用器连接到片上外设，同一时间一个引脚只允许使用一种复用功能。这种模式下，可以避免不同外设竞争 I/O 引脚引发冲突。
- I/O 的配置有了更多特性和发展可能。

F0 的 GPIO 是一种新的设计，因此与 F1 系列的 GPIO 外设的结构、特性、寄存器方面有所不同。F1 上任何与 GPIO 有关的代码在 F0 上都要重新编写。

更多关于 STM32F0 的 GPIO 模块的设计和使用的信息，请参考 STM32F0xx Reference Manual (RM0091)。

下表列出了 F1 系列与 F0 系列关于 GPIO 的不同点。

Table 11. GPIO differences between STM32F1 series and STM32F0 series

GPIO	STM32F1 series	STM32F0 series
Input mode	Floating PU	Floating PU
General purpose output	PP OD	PP PP + PU PP+PD OD OD + PU OD + PD

Table 11.GPIO differences between STM32F1 series and STM32F0 series (continued)

GPI	STM32F1 series	STM32F0 series
Alternate function output	PP OD	PP PP + PU PP+ PD OD OD+ PU OD + PD
Input / Output	Analog	Analog
Output speed	2 MHz 10 MHz 50 MHz	2 MHz 10 MHz 48 MHz
Alternate function selection	To optimize the number of peripheral I/O functions for different device packages, it is possible to remap some alternate functions to some other pins (software remap).	Highly flexible pin multiplexing allows no conflict between peripherals sharing the same I/O pin.
Max IO toggle	18 MHz	12 MHz

管脚复用模式:

STM32F1 系列:

1. I/O 引脚的复用配置取决于外设的模式选择。例如，串口的发送引脚应该配置为复用推挽输出，而接收引脚配置为上拉输入或浮空输入。
2. 为了合理地把外设 I/O 引脚分配给不同的设备（特别是对于那些引脚较少的芯片），可以对一些引脚的复用功能进行重映射。例如，USART2 的接收数据引脚可以用 PA2 引脚（默认映射），也可以用 PD6（软件重映射）。

STM32F0 系列:

1. 不管外设使用什么模式，I/O 引脚必须使用管脚复用模式，这样系统才可正确的使用 I/O 引脚（输入或输出）。
2. I/O 引脚多路复用和映射：引脚通过一个多路复用器连接到片上外设，同一时间一个引脚只允许使用一种复用功能。这种模式下，可以避免不同外设竞争 I/O 引脚引发冲突。每一个引脚对应一个八种复用输入（AF0 到 AF7）的多路复用器，可以通过 GPIOx_AFRL 与 GPIOx_AFRH 寄存器来配置。

——外设的管脚复用通过配置 AF0 到 AF7 来映射。

3. 除了这种灵活的引脚复用结构，每个外设拥有映射到不同引脚的复用功能，使不同设备分配的引脚数量最优化。例如，USART2 的接收引脚可以使用 PA3 或者 PA15。

注意：请参阅 STM32F0 的管脚复用映射表来获取更多关于系统、外设管脚复用的信息。

4. 配置过程

——在 GPIO 的模式设置寄存器中将目标引脚配置为管脚复用模式

——分别通过 GPIOx_OTYPER、GPIOx_PUPDR、GPIOx_OSPEEDER 来配置工作方式、上拉或下拉以及输出速度

——将配置好的 I/O 引脚通过 GPIOx_AFR1 与 GPIOx_AFR2 连接到目标复用 AF 引脚 (AFx)。

3.8 外部中断源选择

在 STM32F1 中，外部中断源的选择是通过寄存器 AFIO_EXTICRx 的 EXTIx 位来配置，而 F0 系列的中断源选择是通过 SYSCFG_EXTICRx 寄存器的 EXTIx 位来配置。

只有 EXTIx 寄存器的映射有所变化，EXTIx 位没有任何改变。但是，EXTIx 位的取值范围有所变化，F0 系列最大值为 0b0101，最后一个端口为 F（而 F1 的最大值为 0110）。

3.9 FLASH 接口

下表列出了 STM32F0 系列与 F1 系列 FLASH 接口的不同之处，可以分为以下几类：

- 新接口，新技术
- 新的结构
- 新的读保护机制，3 个读保护级别

因此，F0 系列与 F1 系列的 FLASH 程序设计流程以及寄存器有很大不同，F1 上任何有关 FLASH 接口的代码都要重新编写以便在 F0 上运行。

Table 12. FLASH differences between STM32F1 series and STM32F0 series

Feature		STM32F1 series	STM32F0 series
Main/Program memory	Start Address	0x0800 0000	0x0800 0000
	End Address	up to 0x080F FFFF	Up to 0x0805 FFFF
	Granularity	Page size = 2 Kbytes except for Low and Medium density page size = 1 Kbyte	64 pages of 1 Kbyte
EEPROM memory	Start Address	Available through SW emulation	Available through SW emulation
	End Address		
System memory	Start Address	0x1FFF F000	0x1FFF EC00
	End Address	0x1FFF F7FF	0x1FFF F7FF

Table 12. FLASH differences between STM32F1 series and STM32F0 series (continued)

Feature		STM32F1 series	STM32F0 series
Option Bytes	Start Address	0x1FFF F800	0x1FFF F800
	End Address	0x1FFF F80F	0x1FFF F80B
Flash interface	Start address	0x4002 2000	0x4002 2000
	Programming procedure	Same for all product lines	Same as F1 series for Flash program and erase operations. Different from F1 series for Option byte programming
Read Protection	Unprotection	Read protection disable RDP = 0xA55A	Level 0 no protection RDP = 0xAA
	Protection	Read protection enable RDP != 0xA55A	Level 1 memory protection RDP != (Level 2 & Level 0) Level 2: Lvl 1 + Debug disabled
Write protection		Protection by 4-Kbyte block	Protection by 4-Kbyte block
User Option bytes		STOP	STOP
		STANDBY	STANDBY
		WDG	WDG
		NA	RAM_PARITY_CHECK
		NA	VDDA_MONITOR
		NA	nBOOT1
Erase granularity		Page (1 or 2 Kbytes)	Page (1 Kbyte)
Program mode		Half word (16 bits)	Half word (16 bits)

3.10 ADC 接口

下表列出了 STM32F0 系列与 F1 系列 ADC 接口的不同之处，有以下几点：

- 新的数字接口
- 新的结构和新的特性

Table 13. ADC differences between STM32F1 series and STM32F0 series

ADC	STM32F1 series	STM32F0 series
ADC Type	SAR	SAR structure
Instances	ADC1 / ADC2 / ADC3	ADC

Table 13. ADC differences between STM32F1 series and STM32F0 series

ADC	STM32F1 series		STM32F0 series
Maximum sampling frequency	1 MSPS		1 MSPS
Number of channels	Up to 21 channels		Up to 16 channels + 3 internal
Resolution	12-bit		12-bit
Conversion modes	Single / continuous / scan / discontinuous /dual mode		Single / continuous / scan / discontinuous / dual mode / triple mode
DMA	Yes		Yes
External Trigger	Yes		Yes
	<u>External event for regular group</u> For ADC1 and ADC2: TIM1 CC1 TIM1 CC2 TIM1 CC3 TIM2 CC2 TIM3 TRGO TIM4 CC4 EXTI line 11 / TIM8_TRGO For ADC3: TIM3 CC1 TIM2 CC3 TIM1 CC3 TIM8 CC1 TIM8 TRGO TIM5 CC1	<u>External event for injected group</u> For ADC1 and ADC2: TIM1 TRGO TIM1 CC4 TIM2 TRGO TIM2 CC1 TIM3 CC4 TIM4 TRGO EXTI line15 / TIM8_CC4 For ADC3: TIM1 TRGO TIM1 CC4 TIM4 CC3 TIM8 CC2 TIM8 CC4 TIM5 TRGO	<u>External event</u> TIM1_TRGO TIM1_CC4 TIM2_TRGO TIM3_TRGO TIM15_TRGO
Supply requirement	2.4 V to 3.6 V		2.4 V to 3.6 V
Input range	$V_{REF-} \leq V_{IN} \leq V_{REF+}$		V_{DD} and $2.4 \leq V_{DDA} \leq 3.6$

3.11 电源接口

在 STM32F0 系列芯片中，电源控制器与 F1 相比有一些不同，概述在下表中。不过程序接口部分并未改变。

Table 14. PWR differences between STM32F1 series and STM32F0 series

PWR	STM32F1 series	STM32F0 series
Power supplies	<p>1- VDD = 2.0 to 3.6 V: external power supply for I/Os and the internal regulator. Provided externally through VDD pins.</p> <p>2- VSSA, VDDA = 2.0 to 3.6 V: external analog power supplies for ADC, Reset blocks, RCs and PLL. VDDA and VSSA must be connected to VDD and VSS, respectively.</p> <p>3- VBAT = 1.8 to 3.6 V: power supply for RTC, external clock 32 kHz oscillator and backup registers (through power switch) when VDD is not present.</p>	<p>1- VDD = 2.0 to 3.6 V: external power supply for I/Os and the internal regulator. Provided externally through VDD pins.</p> <p>2- VSSA, VDDA = 2.0 to 3.6 V: external analog power supplies for ADC, DAC, Reset blocks, RCs and PLL. VDDA and VSSA must be connected to VDD and VSS, respectively.</p> <p>3- VBAT = 1.8 to 3.6 V: power supply for RTC, external clock 32 kHz oscillator and backup registers (through power switch) when VDD is not present.</p>
Battery backup domain	<ul style="list-style-type: none"> – Backup registers – RTC – LSE 	<ul style="list-style-type: none"> – Backup registers – RTC – LSE
Power supply supervisor	<p>Integrated POR / PDR circuitry</p> <p>Programmable voltage detector (PVD)</p>	<p>Integrated POR / PDR circuitry</p> <p>Programmable voltage detector (PVD)</p>
Low-power modes	<p>Sleep mode</p> <p>Stop mode</p> <p>Standby mode (1.8V domain powered-off)</p>	<p>Sleep mode</p> <p>Stop mode</p> <p>Standby mode (1.8V domain powered-off)</p>
Wake-up sources	<p><u>Sleep mode</u></p> <ul style="list-style-type: none"> – Any peripheral interrupt/wakeup event <p><u>Stop mode</u></p> <ul style="list-style-type: none"> – Any EXTI line event/interrupt <p><u>Standby mode</u></p> <ul style="list-style-type: none"> – WKUP pin rising edge – RTC alarm 	<p><u>Sleep and sleep low power modes</u></p> <ul style="list-style-type: none"> – Any peripheral interrupt/wakeup event <p><u>Stop mode</u></p> <ul style="list-style-type: none"> – Any EXTI line event/interrupt <p><u>Standby mode</u></p> <ul style="list-style-type: none"> – WKUP0 or WKUP1 pin rising edge – RTC alarm

3.12 实时时钟（RTC）接口

与 F1 系列相比，F0 系列嵌入了一种新的 RTC 外设。在结构、特性、程序接口方面都有不同。因此，F0 的 RTC 程序接口设计与 F1 有所不同，F1 上任何有关 RTC 的代码都要重新编写 以便在 F0 上运行。

F0 的 RTC 拥有一流的特性：

- 可编程定时/计数器
- 日历时间精确到亚秒级，配备可编程日光补偿。
- 一个可编程警报器
- 数字校准电路
- 为记录时间设计的时间标记功能
- 与一个拥有亚秒转换特性的外部时钟源精确同步
- 5 个备份寄存器，当篡改检测事件发生时复位

请参阅 STM32F0 参考手册（RM0091）以获得关于 F0 系列 RTC 的更多特性。

要得到关于 RTC 程序设计的高级资料，请参阅应用注释 AN3371。

3.13 SPI 接口

与 F1 系列相比，F0 嵌入了一种新的 SPI 外设。在结构、特性、程序设计接口方面作了修改以便拓展新的功能。

因此，F0 的程序设计以及寄存器与 F1 类似，只是有了一些新特性。如果不使用新的功能，F1 上有关 SPI 的代码几乎不需要改动就可以在 F0 上运行。

F0 的 SPI 拥有一流的新增特性：

- 加强的 NSS 管脚控制——NSS 脉冲模式（NSSP）或 TI 模式
- 可编程数据帧的结构从 4 位到 16 位
- 两个 32 位的数据收发缓冲区，可使用 DMA 功能，拥有适用 8 位数据帧的通道
- 适合 8 位或 16 位的数据的循环冗余校验

另外，F0 中的 SPI 修复了 F1 中的 CRC 限制。请参阅 STM32F0 参考手册以得到更多关于 F0 的 SPI 的特性。

3.14 I2C 接口

与 F1 系列相比，F0 系列嵌入了一种新的 I2C 外设。在结构、特性、程序接口方面都有不同。因此，F0 的 I2C 程序接口设计与寄存器与 F1 有所不同，F1 上任何有关 I2C 的代码都要重新编写以便在 F0 上运行。

F0 的 I2C 拥有一流的新增特性：

- 通讯事件由硬件管理
- 可编程模拟/数字噪声滤波器
- 独立时钟源：HSI 或 SYSCLK
- 停止模式唤醒功能
- 20mA 的 I/O 输出电流驱动的快速模式（最高 1Mz）
- 7 位或 10 位地址模式，通过可配置遮掩功能支持多个 7 位从机地址。
- 主模式下地址连续自动发送（7 或 10 位）
- 主模式下自动结束通讯管理
- 可编程的保留和设置时间
- 命令和数据应答控制

请参阅 STM32 参考手册以获取更多关于 F0 的 I2C 的特性

3.15 USART（串口）接口

与 F1 系列相比，F0 嵌入了一种新的 USART 外设。在结构、特性、程序设计接口方面作了修改以便拓展新的功能。

因此，F0 的 USART 程序接口设计与寄存器与 F1 有所不同，F1 上任何有关 USART 的代码都要重新编写以便在 F0 上运行。

F0 的 USART 拥有一流的新增特性：

- 允许使用独立时钟源：
 - UART 功能以及从低功耗模式唤醒
 - 独立于 APB 时钟重编程之外的方便的波特率编程
- 智能卡仿真功能：T=0（自动重试）和 T=1
- 可交换的 Tx（发送）、Rx（接收）引脚配置
- 二进制数据反演
- 收发引脚有效电平转换
- 使能收发自动应答标志
- 新的带标记的中断源
 - 地址/字符匹配
 - 字块长度检测以及超时检测
- 超时特性
- 总线通讯
- 超限运转标志失能
- 接收错误失能 DMA
- 从停止模式唤醒
- 自动检测波特率功能
- RS485 模式下驱动器使能信号

请参阅 STM32F0 参考手册（RM0091）USART 章节以获取更多关于 USART 的信息。

3.16 CEC 接口

与 F1 系列相比，F0 嵌入了一种新的 CEC 外设。在结构、特性、程序设计接口方面作了修改以便拓展新的功能。

因此，F0 的 CEC 程序接口设计与寄存器与 F1 有所不同，F1 上任何有关 CEC 的代码都要重新编写以便在 F0 上运行。

F0 的 CEC 拥有一流的新增特性：

- 配备双时钟的 CEC 内核
 - LSE
 - HSI/244
- 监听模式下的接收功能
- 接收容许偏差边限：标准的或扩展的
- 仲裁（空闲信号）：标准的（通过 H/W）或扩展的（通过 S/W）
- 仲裁丢失检测标志/中断
- 仲裁丢失模式下支持自动重发
- 多地址配置
- 从停止模式唤醒
- 接收错误检测
 - 上升位错误
 - 短位周期错误
 - 长位周期错误
- 运行监测时发送
- 接收超限检测

以下 F1 中的特性在 F0 中通过新的 CEC 特性处理，因此不再可用：

- 位时间&位周期模式，通过新的错误处理
- 可配置的预分频器，通过 CEC 固定的内核时钟

请参阅 STM32F0 参考手册（RM0091）CEC 章节以获取更多关于 CEC 的信息。

4 固件库移植

这一部分描述了如何移植一段基于 STM32F1 固件库的程序以使它适应 STM32F0 的固件库。

STM32F1 的固件库与 F0 的固件库有相同的结构而且服从 CMSIS 协议，它们使用相同命名的驱动，所有的可移植外设使用相同的程序接口（API）

只有一小部分外设驱动需要更新以便从 F1 移植到 F0 上

注意：在本章的以下部分（除非特别说明），“STM32F0xx Library”代表“STM32F0xx Standard Peripherals Library”，“STM32F1xx Library”代表“STM32F1xx Standard Peripherals Library”。

4.1 移植步骤

要更新应用程序代码以适应 STM32F0 的固件库，用户应按照以下所列步骤进行操作：

1. 更新工具链启动文件

a) 工程文件：设备连接和闪存装载程序。这些文件配备有用户工具链的最新版本，支持 F0 的固件库。请参阅工具文件获取更多信息。

b) 连接器配置和向量表位置文件。这些文件由 CMSIS 标准发展而来，包含在一下目录下：Libraries/CMSIS/Device/ST/STM32F0xx。

2. 添加 STM32F0 固件库源文件到应用程序。

a) 用 STM32F0 固件库中提供的 STM32F0xx_conf.h 文件代替应用程序中的 STM32F0xx_conf.h 文件。

b) 用 STM32F0 固件库中提供的 STM32F0xx_it.c 及 STM32F0xx_it.h 文件代替应用程序中的 STM32F1xx_it.c 和 STM32F1xx_it.h 文件

3. 用 RCC, PWR, GPIO, FLASH, ADC 和 RTC 模块驱动更新应用程序代码，具体细节在后面一部分描述。

注意：STM32F0 的固件库提供了丰富的例程（共 67 例）来展示如何使用固件库（在 Project/STM32F0xx_StdPeriph_Examples 路径下）。

4.2 RCC 驱动

1. 系统时钟设置：如 3.4：《重置和时钟控制器接口》部分所述，STM32F1 系列与 F0 系列使用相同的时钟源，配置过程也一致。但是，产品的相关电压范围、锁相环配置、最大频率以及 FLASH 等待模式配置有所不同。感谢 CMSIS 层，这些不同之处与应用程序层隔离开了，用户只需用 system_stm32f00x.c 去代替 system_stm32f10x.c 文件即可。这个文件提供了 System_Init 函数的实现，用来实现微控制器系统的启动和进入到主程序之前的配置。

注意：对于 STM32F0，用户可以用始终配置工具 STM32F0xx_Clock_Configuration.xls 来根据程序要求自定义系统配置文件

2. 外设通道配置：如 3.4 节：《重置和时钟控制器接口》部分所述：用户需要调用不同的函数来使能/失能或进入/退出外设时钟或重置模式。例如，在 F0 系列中 GPIOA 挂载在 AHB 总线上（F1 系列中挂载在 APB 总线上），要使能它的时钟，需要用
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE) 语句来替代
RCC_APBPeriphClockCmd(RCC_APBPeriph_GPIOA, ENABLE) 语句来使能它。
参考表 5 中 F1 与 F0 系列的外设映射改变。

3. 外设时钟改变

一些 STM32 的外设支持双时钟特性。下表列出了这些外设的时钟源与 F1 系列的对比情况：

Table 15. STM32F10x and STM32F0xx source clock API correspondence

Peripherals	Source clock in STM32F10xx device	Source clock in STM32F0xx device
ADC	APB2 clock with prescaler	<ul style="list-style-type: none"> - HSI14: by default - APB2 clock/2 - APB2 clock/4
CEC	APB1 clock with prescaler	<ul style="list-style-type: none"> - HSI/244: by default - LSE - APB clock: Clock for the digital interface (used for register read/write access). This clock is equal to the APB2 clock.
I2C	APB1 clock	<ol style="list-style-type: none"> I2C1 can be clocked with: <ul style="list-style-type: none"> - System clock - HSI I2C2 can be only clocked with: <ul style="list-style-type: none"> - HSI
SPI/I2S	System clock	System clock
USART	<ol style="list-style-type: none"> USART1 can be clocked with: <ul style="list-style-type: none"> - PCLK2 (72 MHz Max) Other USARTs can be clocked with: <ul style="list-style-type: none"> - PCLK1 (36 MHz Max) 	<ol style="list-style-type: none"> USART1 can be clocked with: <ul style="list-style-type: none"> - system clock - LSE clock - HSI clock - APB clock (PCLK) USART2 can be only clocked with: <ul style="list-style-type: none"> - system clock

4.3 FLASH 驱动

下表列出了 F1 和 F0 系列的 FLASH 驱动程序接口对应情况。用户可以轻易地用 F0 固件库中的对应函数来代替 F1 系列的函数来更新应用程序代码。

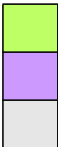
Table 16. STM32F10x and STM32F0xx FLASH driver API correspondence

	STM32F10x Flash driver API	STM32F0xx Flash driver API
Interface configuration	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
	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);
Memory Programming	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);

Table 16.STM32F10x and STM32F0xx FLASH driver API correspondence (continued)

	STM32F10x Flash driver API	STM32F0xx Flash driver API
	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_RDPCConfig(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);

Table 16. STM32F10x and STM32F0xx FLASH driver API correspondence (continued)

	STM32F10x Flash driver API	STM32F0xx Flash driver API
FLAG management	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);
<p>Color key: top = New function; middle=Same function, but API was changed; bottom=Function not available (NA)</p> 		

4.4 CRC 驱动


下表列出了 F1 和 F0 系列的 FLASH 驱动程序接口对应情况。用户可以轻易地用 F0 固件库中的对应函数来代替 F1 系列的函数来更新应用程序代码。

Table 17. STM32F10xx and STM32F0xx CRC driver API correspondence

	STM32F10xx CRC driver API	STM32F0xx CRC driver API
Configuration	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);

Computation	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);

Table 17. STM32F10xx and STM32F0xx CRC driver API correspondence (continued)

	STM32F10xx CRC driver API	STM32F0xx CRC driver API
IDR access	void CRC_SetIDRegister(uint8_t CRC_IDValue);	void CRC_SetIDRegister(uint8_t CRC_IDValue);
	uint8_t CRC_GetIDRegister(void);	uint8_t CRC_GetIDRegister(void);
<p>Color key: top= New function; middle= Same function, but API was changed; bottom= Function not available (NA)</p> 		

4.5 GPIO 配置更新

这一部分解释了从 F1 到 F0 移植的过程中如何更新各种各样的 GPIO 模式配置。

4.51 输出模式

以下这个例子展示了在 F1 系列中如何配置一个 GPIO 引脚为输出模式（例如驱动一个 LED）

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_xxMHz; /* 2, 10 or
50 MHz */
```

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_Init(GPIOy, &GPIO_InitStructure);
```

在F0系列中，用户需要按以下方式配置：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP; /* Push-pull or
open drain */
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP; /* None, Pull-up or
pull-down */
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_xxMHz; /* 10, 2 or 50MHz
*/
GPIO_Init(GPIOy, &GPIO_InitStructure);
```

4.52 输入模式

以下这个例子展示了在F1系列中如何配置一个GPIO引脚为输出模式（例如作为外部中断使用）。

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IN_FLO
ATNG;
```

```
GPIO_Init(GPIOy, &GPIO_InitStructure);
```

在F0系列中，用户需要按以下方式配置：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_x;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL; /* None, Pull-up or
pull-down */
GPIO_Init(GPIOy, &GPIO_InitStructure);
```

4.53 模拟模式

以下这个例子展示了在F1系列中如何配置一个GPIO引脚为模拟模式（例如作为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，用户需要按以下方式配置：

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

4.54 复用模式

F1系列中

1. I/O引脚的配置方式取决于外设的使用模式；例如，串口的发送引脚复用模式应设为推挽输出，而发送引脚应设为上拉输入或浮空输入。
2. 为了将外设引脚合理的分配给不同的设备，可以通过软件将一些引脚的复用功能进行重映射。例如，USART2的接收引脚可以映射到PA3（默认映射），也可以映射到PD6（软件重映射）

F0系列中

1. 不管外设使用何种模式，I/O引脚必须配置为复用模式，这样系统才可以正确的使用引脚（输入或输出）。
2. I/O引脚通过一个多路复用器连接到片上外设，同一时间，一个引脚只允许被用作一个

外设的复用功能。在这种模式下，可以避免外设共享I/O引脚引发冲突。每个I/O引脚拥有一个16个复用输入功能的多路复用器，可以通过GPIO_PinAFConfig()函数进行配置。

——复位之后，所有I/O的连接到系统复用功能0（AF0）

——外设的复用功能通过配置AF1到AF6来进行映射

3. 除了灵活的I/O多路映射结构，每个外设的复用功能映射到不同的引脚来为不同设备最优分配外设引脚，例如USART的接收引脚可以映射到PA3或PA15。

4. 配置过程

——通过GPIO_PinAFConfig()函数来将引脚连接到外设复用功能

——使用GPIO_Init()函数来配置I/O引脚：

-通过GPIO_InitStructure->GPIO_Mode=GPIO_Mode_AF来配置引脚为复用功能模式；

-通过GPIO_PuPd、GPIO_OType、GPIO_Speed来配置工作方式、上拉或下拉、输出速度3个量

下例展示了如何将F1中USART2的收发引脚（PD5/PD6）进行重映射：

```
/* Enable APB2 interface clock for GPIO and AFIO (AFIO peripheral is
used to configure the I/Os software remapping) */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIO | RCC_APB2Periph_AFIO,
ENABLE);

/* Enable USART2 I/Os software remapping [(USART2_Tx,USART2_Rx):(PD5,PD6)] */
GPIO_PinRemapConfig(GPIO_Remap_USART2, ENABLE);

/* Configure USART2 Tx as alternate function push-pull */
GPIO_InitStructure.GPIO_Pin=GPIO_Pin_5;
GPIO_InitStructure.GPIO_Mode=GPIO_Mode_AF_PP;
P;
GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50
MHz;
GPIO_Init(GPIO, &GPIO_InitStructure);
/* Configure USART2 Rx as input floating */
GPIO_InitStructure.GPIO_Pin=GPIO_Pin_6;
GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIO, &GPIO_InitStructure);
```

在F0系列中，用户需要按以下方式配置：

```
/* Enable GPIOA's AHB interface clock */
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
GPIO_PinAFConfig(GPIOA,GPIO_PinSource14,GPIO_AF_2);
GPIO_PinAFConfig(GPIOA,GPIO_PinSource15,GPIO_AF_2);
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_14 | GPIO_Pin_15;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50
MHz;
GPIO_InitStructure.GPIO_OType=GPIO_OType_PP;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
```

```
GPIO_Init(GPIOA, &GPIO_InitStructure);
```

4.6 外部中断线0

下例展示了如何在F1系列中将引脚PA0配置为外部中断0的引脚：

```
/* Enable APB interface clock for GPIOA and AFIO */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO,
ENABLE);

/* Configure PA0 pin in input mode */
GPIO_InitStructure.GPIO_Pin=GPIO_Pin_0;
GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IN_FLOA
TING;
GPIO_Init(GPIOA, &GPIO_InitStructure);

/* Connect EXTI Line0 to PA0 pin */
GPIO_EXTILineConfig(GPIO_PortSourceGPIOA,GPIO_PinSo
urce0);

/* Configure EXTI line0 */
EXTI_InitStructure.EXTI_Line =EXTI_Line0;
EXTI_InitStructure.EXTI_Mode=EXTI_Mode_Interrupt;
EXTI_InitStructure.EXTI_Trigger=EXTI_Trigger_Fal
ling;
EXTI_InitStructure.EXTI_LineCmd=ENABLE;
EXTI_Init(&EXTI_InitStructure);
```

在F0系列中，外部中断1的配置通过SYSCFG配置（而不是像F1中通过AFIO）。因此，源代码应该按照如下方式更新：

```
/* Enable GPIOA's AHB interface clock */
RCC_AHBPeriphClockCmd(RCC_AHBPeriph_GPIOA, ENABLE);
/* Enable SYSCFG's APB interface clock */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

/* Configure PA0 pin in input mode */ 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);

/* Connect EXTI Line0 to PA0 pin */
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA,
```

```
EXTI_PinSource0);

/* Configure EXTI line0 */
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 嵌套向量中断控制器配置

这一部分描述了嵌套向量中断控制器的中断配置（IRQ）

在F1系列中，NVIC支持：

- 最多81个中断
- 16个可编程中断优先级（使用4位）
- 将优先级分为优先级组和副优先级区域
- 优先级动态变化

Cortex-M3的异常处理由CMSIS功能进行管理

- 根据优先级组的配置来使能和配置所选IRQ中断通道的抢占优先级和副优先级。

下例展示了如何配置F1系列的CEC中断：

```
/* Configure two bits for preemption priority */
NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);

/* Enable the CEC global Interrupt (with higher priority) */
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：

- 最多32个中断
- 4个可编程优先级（使用2位作为中断优先级）
- 中断优先级使能之后就不应改变

Cortex-M0异常处理由CMSIS功能处理：

- 使能和配置已选通道的中断优先级。优先级范围从0到3。小的优先级值代表高优先级。

在F0系列中，CEC中断配置源代码应按如下方式进行更新：

```
/* Enable the CEC global Interrupt (with higher priority) */
NVIC_InitStructure.NVIC_IRQChannel = CEC_IRQn;

NVIC_InitStructure.NVIC_IRQChannelPriority = 0;

NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;

NVIC_Init(&NVIC_InitStructure);
```

下表列出了F1与F0系列固件库MISC驱动程序接口的对应关系：

Table 18. STM32F10x and STM32F0xx MISC driver API correspondence

STM32F10xx MISC Driver API	STM32F0xx MISC Driver 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

这一部分提供了如何将F1上的代码移植到F0上。

下例展示了在F1系列中如何配置ADC1的通道14以进行连续数据转换

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

/* Enable ADC's APB interface clock */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1, ENABLE);

/* Configure ADC1 to convert continuously channel14 */
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 regular channel14 configuration */
ADC-RegularChannelConfig(ADC1, ADC_Channel_14, 1,
ADC_SampleTime_55Cycles5);

/* Enable ADC1's DMA interface */
ADC_DMACmd(ADC1, ENABLE);

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

/* Enable ADC1 reset calibration register */
ADC_ResetCalibration(ADC1);
/* Check the end of ADC1 reset calibration register */
while(ADC_GetResetCalibrationStatus(ADC1));

/* Start ADC1 calibration */ ADC_StartCalibration(ADC1);
/* Check the end of ADC1 calibration */
while(ADC_GetCalibrationStatus(ADC1));

/* Start ADC1 Software Conversion */ ADC_SoftwareStartConvCmd(ADC1,
ENABLE);
...

```

在F0系列中，用户应按如下方式更新代码：

```

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

/* Enable ADC1 clock */ RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1,
ENABLE);

/* ADC1 configuration */ 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);

/* Convert the ADC1 Channel 1 with 55.5 Cycles as sampling time */
ADC_ChannelConfig(ADC1, ADC_Channel_11 ,
ADC_SampleTime_55_5Cycles);

/* ADC Calibration */ ADC_GetCalibrationFactor(ADC1);

```

```
/* ADC DMA request in circular mode */ ADC_DMAREquestModeConfig(ADC1,
ADC_DMAMode_Circular);

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

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

/* Wait the ADCEN flag */
while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_ADEN));

/* ADC1 regular Software Start Conv */ ADC_StartOfConversion(ADC1);
...
```


4.9 DAC驱动

下表描述了F1系列与F0系列的固件库的功能区别：

Table 19. STM32F10x and STM32F0xx DAC driver API correspondence

	STM32F10x DAC driver API	STM32F0xx DAC driver API
Configuration	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 management	void DAC_DMAMCmd(uint32_t DAC_Channel, FunctionalState NewState);	void DAC_DMAMCmd(uint32_t DAC_Channel, FunctionalState NewState);
Interrupts and flags management	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);

F1与F0系列源代码/程序方面的变化主要如下所述：

- DAC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，只有两个区域（外触发器和输出缓冲区）需要初始化。

下例展示了F1系列中，如何配置DAC的通道1：

```
/* DAC channel1 Configuration */
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);
```

F0系列中，用户应按如下方式更新：

```
/* DAC channel1 Configuration */
DAC_InitStructure.DAC_Trigger = DAC_Trigger_None;
DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
/* DAC Channel1 Init */
DAC_Init(DAC_Channel_1, &DAC_InitStructure);
```

4.10 电源驱动

下表列出了F1系列与F0系列固件库在电源驱动的应用程序接口的对应特性。你可以轻易用F0系列固件库中对应特性来替代F1中相关的应用程序代码。

Table 20. STM32F10x and STM32F0xx PWR driver API correspondence

	STM32F10x PWR driver API	STM32F0xx PWR driver API
--	--------------------------	--------------------------

Interface configuration	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);
WakeUp	void PWR_WakeUpPinCmd(FunctionalState NewState);	void PWR_WakeUpPinCmd(uint32_t PWR_WakeUpPin, FunctionalState NewState);(*)
Power Management	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);

Table 20. STM32F10x and STM32F0xx PWR driver API correspondence (continued)

	STM32F10x PWR driver API	STM32F0xx PWR driver API
FLAG management	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);
<p>Color key:top= New function; middle=Same function, but API was changed; bottom= Function not available (NA)</p> <div><div></div><div></div><div></div></div>		

4.11 后备数据寄存器

在F1系列中，后备数据寄存器通过BKP外设管理，在F0系列中则是RTC外设的一部分（该系列无BKP外设）

下例展示了在F1系列中如何读写后备数据寄存器：

```
uint16_t BKPdata = 0;

...

/* Enable APB2 interface clock for PWR and BKP */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);

/* Enable write access to Backup domain */
PWR_BackupAccessCmd(ENABLE);

/* Write data to Backup data register 1 */
BKP_WriteBackupRegister(BKP_DR1, 0x3210);

/* Read data from Backup data register 1 */
BKPdata =BKP_ReadBackupRegister(BKP_DR1);
```

在F0系列中，用户应按如下方式更新代码：

```
uint16_t BKPdata = 0;

...
```

```
/* PWR Clock Enable */ RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR,  
ENABLE);  
  
/* Enable write access to RTC domain */ PWR_RTCAccessCmd(ENABLE);  
  
/* Write data to Backup data register 1 */  
RTC_WriteBackupRegister(RTC_BKP_DR1, 0x3220);  
  
/* Read data from Backup data register 1 */  
BKPdata=RTC_ReadBackupRegister(RTC_BKP_DR1);
```

F0与F1系列源代码的主要区别描述如下：

1. 没有BKP外设
2. 通过RTC驱动来向后备数据寄存器读取/写入数据。
3. 后备数据寄存器的名字由BKP_DRx改为RTC_BKP_DRx, 从0开始编号而不是从一开始。

4.12 CEC程序代码

你可以轻易地通过用F0系列固件库中的相关功能替代F1中的功能来完成应用程序代码更新。下表列出了F1与F0系列固件库中CEC驱动的应用程序接口的对应情况。

Table 21. STM32F10xx and STM32F0xx CEC driver API correspondence

	STM32F10xx CEC driver API	STM32F0xx CEC driver API
Interface Configuration	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
DATA Transfers	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);

Table 21. STM32F10xx and STM32F0xx CEC driver API correspondence (continued)

	STM32F10xx CEC driver API	STM32F0xx CEC driver API
Interrupt and Flag management	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);
<p>Color key: top= New function; middle= Same function, but API was changed; bottom= Function not available (NA)</p> <div><div></div><div></div><div></div></div>		

与F1系列相比，F0的主要变化描述如下：

- 双重时钟源（细节请参考RCC部分）
- 无预分频特性配置
- 支持不知一个地址（多级地址）
- 每个事件标志都有相应的使能控制位来生成适当的中断
- 在CEC结构定义中，七个部分需要初始化

下例展示了在F1系列中如何配置CEC：

```
/* Configure the CEC peripheral */  
  
CEC_InitStructure.CEC_BitTimingMode = CEC_BitTimingStdMode;  
CEC_InitStructure.CEC_BitPeriodMode = CEC_BitPeriodStdMode;  
  
CEC_Init(&CEC_InitStructure);
```

在F0系列中，需要按如下方式更新代码：

```
/* Configure 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_O
ff;

CEC_InitStructure.CEC_BRDNoGen = CEC_BRDNoGen_Off;

CEC_InitStructure.CEC_SFTOption = CEC_SFTOption_Off;

CEC_Init(&CEC_InitStructure);
```


4.13 I2C驱动

STM32系列器件包含新的I2C特性。

你可以轻易地通过用F0系列固件库中的相关功能替代F1中的功能来完成应用程序代码更新。下表列出了F1与F0系列固件库中I2C驱动的应用程序接口的对应情况。

Table 22. STM32F10xx and STM32F0xx I2C driver API correspondence

	STM32F10xx I2C Driver API	STM32F0xx I2C Driver API
Initialization and Configuration	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

Table 22. STM32F10xx and STM32F0xx I2C driver API correspondence (continued)

	STM32F10xx I2C Driver API	STM32F0xx I2C Driver API
Communications handling	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	NA

Table 22. STM32F10xx and STM32F0xx I2C driver API correspondence (continued)

	STM32F10xx I2C Driver API	STM32F0xx I2C Driver API
SMBUS management	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);
Data transfers	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 management	void I2C_DMAMCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);	void I2C_DMAMCmd(I2C_TypeDef* I2Cx, uint32_t I2C_DMAMReq, FunctionalState NewState);
	Void I2C_DMALastTransferCmd(I2C_TypeDef* I2Cx, FunctionalState NewState);	NA

Table 22. STM32F10xx and STM32F0xx I2C driver API correspondence (continued)

	STM32F10xx I2C Driver API	STM32F0xx I2C Driver API
Interrupts and flags management	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);
<p>Color key: top=New function ; middle=Same function,but API was changed;bottom=Function not available(NA)</p> <div><div></div><div></div><div></div></div>		

尽管在 F1 系列与 F0 系列中某些 API 函数是相同的，但是大部分情况下，从 F1 到 F0 移植时，程序代码需要重新编写。STM 系列微控制器提供一种 “I2C 通讯外设应用程序库（CPAL）” 来实现 F1 到 F0 的无缝移植：用户只需要修改部分设置而不需要改变任何代码。请参阅 UN1029 以获取更多关于 CPAL 的信息。对于 F0 系列，CPAL 在标准外设库中提供。

4.14 SPI 驱动

与 F1 系列相比，F0 系列 SPI 包含一些新特性。表 23 列出了 F1 与 F0 系列固件库中 SPI 程序接口的对比。

Table 23. STM32F10xx and STM32F0xx SPI driver API correspondence

	STM32F10xx SPI driver API	STM32F0xx SPI driver API
Initialization and Configuration	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);

Data Transfers	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);

Table 23. STM32F10xx and STM32F0xx SPI driver API correspondence (continued)

	STM32F10xx SPI driver API	STM32F0xx SPI driver API
Hardware CRC Calculation functions	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 transfers	void SPI_I2S_DMAMCmd(SPI_TypeDef* SPIx, uint16_t SPI_I2S_DMAMReq, FunctionalState NewState);	void SPI_I2S_DMAMCmd(SPI_TypeDef* SPIx, uint16_t SPI_I2S_DMAMReq, FunctionalState NewState);
	NA	void SPI_LastDMATransferCmd(SPI_TypeDef* SPIx, uint16_t SPI_LastDMATransfer);
Interrupts and flags management	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);(*)

Color key: top=New function;middle=Same function, but API was changed;bottom= Function not available (NA)



(*) One more flag in STM32F0xx (TI frame format error) can generate an event in comparison with STM32F10xx driver API

4.15 串口驱动

与 F1 系列相比, F0 的串口拥有一些增强功能。表 9 列出了 F1 与 F0 系列固件库的串口程序接口的对应情况。

Table 24. STM32F10x and STM32F0xx USART driver API correspondence

	STM32F10xx USART driver API	STM32F0xx USART driver API
Initialization and Configuration	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);

Table 24. STM32F10x and STM32F0xx USART driver API correspondence (continued)

	STM32F10xx USART driver API	STM32F0xx USART driver API
STOP Mode	NA	void USART_STOPModeCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_StopModeWakeUpSourceConfig(USART_TypeDef* USARTx, uint32_t USART_WakeUpSource);
AutoBaudRate	NA	void USART_AutoBaudRateCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	NA	void USART_AutoBaudRateConfig(USART_TypeDef* USARTx, uint32_t USART_AutoBaudRate);
Data transfers	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);
Multi-Processor Communication	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 mode	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);

Half-duplex mode	void USART_HalfDuplexCmd(USART_TypeDef * USARTx, FunctionalState NewState);	Void USART_HalfDuplexCmd(USART_TypeDef* USARTx, FunctionalState NewState);
------------------	---	---

Table 24. STM32F10x and STM32F0xx USART driver API correspondence (continued)

	STM32F10xx USART driver API	STM32F0xx USART driver API
Smart Card mode	void USART_SmartCardCmd(USART_TypeDef * USARTx, FunctionalState NewState);	void USART_SmartCardCmd(USART_TypeDef* USARTx, FunctionalState NewState);
	void USART_SmartCardNACKCmd(USART_Typ eDef* 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 mode	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 mode	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 transfers	void USART_DMAMCmd(USART_TypeDef* USARTx, uint32_t USART_DMAReq, FunctionalState NewState);	void USART_DMAMCmd(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);

Table 24. STM32F10x and STM32F0xx USART driver API correspondence (continued)

	STM32F10xx USART driver API	STM32F0xx USART driver API
Interrupts and flags management	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);


Color key: top=New function;middle=Same function,but API was changed;bottom=Function not available



4.16 独立看门狗驱动

F1 系列与 F0 系列器件的独立看门狗拥有相同的技术参数，F0 系列附加了窗口功能以监测外部振荡器的过高频率。下表列出了独立看门狗驱动的程序接口。

Table 25. STM32F10xx and STM32Fxx IWDG driver API correspondence

	STM32F10xx IWDG driver API	STM32F0xx IWDG driver API
Prescaler and Counter configuration	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 activation	void IWDG_Enable(void);	void IWDG_Enable(void);
Flag management	FlagStatus IWDG_GetFlagStatus(uint16_t IWDG_FLAG);	FlagStatus IWDG_GetFlagStatus(uint16_t IWDG_FLAG);
<p>Color key:top= New function;middle= Same function, but API was changed;bottom= Function not available (NA)</p> 		

5 修订记录

Table 26. Document revision history

Date	Revision	Changes
10-Jul-2012	1	Initial release