AN4228
Application note

Migrating from STM32F1 Series to STM32F3 Series microcontrollers

Introduction

For the designers of STM32 microcontroller applications, being able to replace easily one microcontroller type by another one in the same product family is an important asset. Migrating an application to a different microcontroller is often needed, when the product requirements grow, putting extra demands on the memory size, or increasing the number of I/Os. The cost reduction objectives may also be an argument to switch to smaller components and to shrink the PCB area.

This application note analyzes the steps required to migrate from an existing STM32F1 Series device to an STM32F3 Series device. It gathers the most important information and lists the vital aspects that need to be addressed.

In order to migrate an application from the STM32F1 Series to the STM32F3 Series, four aspects need to be considered: the hardware migration, the peripheral migration, the firmware migration and the software migration.

To fully benefit from this application note, the user should be familiar with the STM32 microcontroller documentation available on *www.st.com*, with a particular focus on:

- STM32F1 Series reference manuals (RM0008 and RM0041), the STM32F1 datasheets, and the STM32F1 Flash programming manuals (PM0075, PM0063 and PM0068).
- STM32F3 Series reference manuals and the STM32F3 datasheets.

For an overview of the whole STM32 series and a comparison of the different features of each STM32 product series, refer to the application note *Migration and compatibility guidelines for STM32 microcontroller applications (AN3364)*.

Contents

1 Hardware migration 6

2 Boot mode compatibility 9

3 Peripheral migration 10

3.1 STM32 product cross-compatibility 10

3.2 System architecture 13

3.3 Memory mapping 15

3.4 Reset and clock controller (RCC) interface 17

3.4.1 System clock configuration 18

3.4.2 Peripheral access configuration 21

3.4.3 Peripheral clock configuration 22

3.5 DMA interface 23

3.6 Interrupt vectors 25

3.7 GPIO interface 26

3.7.1 Alternate function mode 27

3.8 EXTI source selection 28

3.9 FLASH interface 28

3.10 SAR ADC interface 29

3.11 PWR interface 31

3.12 Real-time clock (RTC) interface 32

3.13 SPI interface 32

3.14 I2C interface 33

3.15 USART interface 34

3.16 CEC interface 35

4 Firmware migration using the library 36

4.1 Migration steps 36

4.2 RCC driver 36

4.2.1 System clock configuration 36

4.2.2 Peripheral access configuration 37

4.2.3 Peripheral clock configuration 37

4.3 FLASH driver 38
4.4 CRC driver 40
4.5 GPIO configuration update 41
4.5.1 Output mode 41
4.5.2 Input mode 42
4.5.3 Analog mode 42
4.5.4 Alternate function mode 42
4.6 EXTI Line0 44
4.7 NVIC interrupt configuration 45
4.8 ADC configuration 46
4.9 DAC driver 59
4.10 PWR driver 60
4.11 Backup data registers 61
4.12 CEC application code 62
4.13 I2C driver 64
4.14 SPI driver 69
4.15 USART driver 71
4.16 IWDG driver 76
4.17 FMC driver 77
4.18 TIM driver 80
4.19 DBGMCU driver 87

5 Revision history 88

List of tables

Table 1. STM32F103 and STM32F3xx line available packages 6
Table 2. Main pinout differences between STM32F10x and STM32F30x lines 6
Table 3. LQFP144 pinout in STM32F103xx and STM32F30xxD/E 7
Table 4. Main pinout differences between STM32F373xx and STM32F103xx 7
Table 5. Boot modes. 9
Table 6. STM32 peripheral compatibility analysis STM32F1 Series versus STM32F3 Series. 11
Table 7. IP bus mapping differences between STM32F3 Series and STM32F1 Series. 15
Table 8. RCC differences between STM32F1 Series and STM32F3 Series 17
Table 9. Example of migrating system clock configuration code from STM32F100 Value line to STM32F3 Series. 20
Table 10. RCC registers used for peripheral access configuration. 21
Table 11. DMA request differences between STM32F3 Series and STM32F1 Series. 23
Table 12. Interrupt vector differences between STM32F3 Series and STM32F1 Series 25
Table 13. GPIO differences between STM32F1 Series and STM32F3 Series. 27
Table 14. FLASH differences between STM32F1 Series and STM32F3 Series 28
Table 15. ADC differences between STM32F1 Series and STM32F30x lines 29
Table 16. ADC channels mapping differences 30
Table 17. PWR differences between STM32F1 Series and STM32F3 Series 31
Table 18. STM32F10x and STM32F3xx clock source API correspondence 37
Table 19. STM32F10x and STM32F3xx FLASH driver API correspondence. 38
Table 20. STM32F10x and STM32F3xx CRC driver API correspondence. 40
Table 21. STM32F10x and STM32F3xx MISC driver API correspondence 45
Table 22. STM32F10x and STM32F3xx ADC driver API correspondence. 47
Table 23. STM32F10x and STM32F3xx DAC driver API correspondence. 59
Table 24. STM32F10x and STM32F3xx PWR driver API correspondence 60
Table 25. STM32F10x and STM32F37x CEC driver API correspondence. 63
Table 26. STM32F10x and STM32F3xx I2C driver API correspondence. 65
Table 27. STM32F10x and STM32F3xx SPI driver API correspondence. 69
Table 28. STM32F10x and STM32F3xx USART driver API correspondence 72
Table 29. STM32F10x and STM32F3xx IWDG driver API correspondence. 76
Table 30. STM32F10x and STM32F303xD/E FMC driver API correspondence 77
Table 31. STM32F10x and STM32F3xx TIM driver API correspondence 80
Table 32. STM32F10x and STM32F3xx DBGMCU driver API correspondence 87
Table 33. Document revision history 88

List of figures

Figure 1. System architecture for STM32F30x and STM32F3x8 lines 14
Figure 2. System architecture for STM32F37x lines 14

1 Hardware migration

The STM32F30x and STM32F10x lines are pin-to-pin compatible. All the peripherals share the same pins in the two lines, but there are some minor differences between the packages. The transition from the STM32F1 Series to the STM32F3 Series is simple as only a few pins are impacted (as detailed in *Table 2* and *Table 4*).

Table 1 lists the available packages in the STM32F103 and STM32F3xx line products.

Table 1. STM32F103 and STM32F3xx line available packages[(1)]

Package	STM32F103xx	STM32F30xxB/C	STM32F30xxD/E	STM32F37xxx	STM32F302/ F301x6/8	STM32F303 x6/8
LQFP32	-	-	-	-	-	X
LQFP48	-	X	-	X	X	X
LQFP64	X	X	X	X	X	X
LQFP100	X	X	X	X	-	-
LQFP144	X	-	X	-	-	
UFQFN32	-	-	-	-	X	-
UFBGA100	-	-	-	X	-	-
WLCSP49	-	-	-	-	X	-

1. X = available.

Table 2. Main pinout differences between STM32F10x and STM32F30x lines

LQFP48	LQFP64	LQFP100	STM32F10xxx	STM32F302xB/C STM32F303xB/C	STM32F302xD/E STM32F303xD/E	STM32F302x6/8 STM32F301x6/8 STM32F303x6/8	Comments
-	-	10	V_{SS}	PF9	PF9	-	One additional I/O
-	-	11	V_{DD}	PF10	PF10	-	One additional I/O
5	5	12	OSC_IN	PF0/OSC_IN	PF0/OSC_IN	PF0/OSC_IN	-
6	6	13	OSC_OUT	PF1/OSC_OUT	PF1/OSC_OUT	PF1/OSC_OUT	-
-	-	19	V_{SSA}	PF2	PF2	-	One additional I/O
8	12	20	V_{REF-}	V_{REF-}/V_{SSA}	V_{REF-}/V_{SSA}	V_{REF-}/V_{SSA}	-
-	18	27	V_{SS}	PF4	V_{SS}	V_{SS}	One additional I/O

Table 2. Main pinout differences between STM32F10x and STM32F30x lines (continued)

LQFP48	LQFP64	LQFP100	STM32F10xxx	STM32F302xB/C STM32F303xB/C	STM32F302xD/E STM32F303xD/E	STM32F302x6/8 STM32F301x6/8 STM32F303x6/8	Comments
20	28	37	PB2/BOOT1	PB2	PB2	-	One additional I/O. The bootloader strategy has changed in the STM32F30xx and BOOT1 is not necessary anymore. The boot memory (RAM, Flash or System memory) is selectable through an option bit.
-	-	73	Not Connected	PF6	PF6	-	One additional I/O

Considering the 48-/64-/100-pin packages, the migration from STM32F1 Series to STM32F30xxx products has no impact on the pinout, except that the user gains additional GPIOs for their application.

Table 3 shows the LQFP144 pinout compatibility between STM32F302/F303xD/E and STM32F103xx products.

Table 3. LQFP144 pinout in STM32F103xx and STM32F30xxD/E

LQFP144	STM32F103xx	STM32F302xD/E, STM32F303xD/E
10	PF0	PH0
11	PF1	PH1
23	OSC_IN	PF0/OSC_IN
24	OSC_OUT	PF1/OSC_OUT
106	Not connected	PH2

Table 4. Main pinout differences between STM32F373xx and STM32F103xx

Pin number for 48-pin package	Pin number for 64-pin package	Pin number for 100-pin package	STM32F373xx	STM32F103xx	Comments
-	-	10	PF9	V_{SS}	-
-	-	11	PF10	V_{DD}	-
-	-	19	PF2	V_{SSA}	-
-	-	21	V_{DDA}	V_{REF+}	Supply compatible
9	13	22	V_{REF+}/V_{DDA}	V_{DDA}	Supply compatible
-	17	-	V_{REF+}	PA3	-

Table 4. Main pinout differences between STM32F373xx and STM32F103xx (continued)

Pin number for 48-pin package	Pin number for 64-pin package	Pin number for 100-pin package	STM32F373xx	STM32F103xx	Comments
-	18	-	PA3	V_{SS}	Supply compatible
-	19	-	V_{DD}	V_{DD}	Supply compatible
17	-	-	V_{DD}	PA7	-
23	31	-	V_{REFSD-}/V_{SSSD}	V_{SS}	Supply compatible
24	32	-	V_{DDSD}	V_{DD}	Supply compatible
-	-	27	PF4	V_{SS}	-
-	-	48	V_{REFSD-}	PB11	-
-	-	49	V_{SSSD}	V_{SS}	Supply compatible
-	-	50	V_{DDSD12}	V_{DD}	Supply compatible
25	33	51	V_{REFSD+}/V_{DDSD3}	PB12	-
-	-	52	V_{REFSD+}	PB13	-
35	47	73	PF6	V_{SS}	Supply compatible
36	48	-	PF7	V_{DD}	Supply compatible

2 Boot mode compatibility

The way to select the boot mode on the STM32F3 Series differs from the STM32F1 Series. Instead of using two pins for this setting, the STM32F3 Series devices get the nBOOT1 value from an option bit located in the user option bytes at 0x1FFFF800 memory address. Together with the BOOT0 pin, it selects the boot mode to the main Flash memory, the SRAM or to the System memory. *Table 5* summarizes the different configurations available for selecting the Boot mode.

Table 5. Boot modes

STM32F3 Series /STM32F1 Series 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

Note: The BOOT1 value is the opposite of the nBOOT1 option bit.

3 Peripheral migration

As shown in *Table 6*, there are three categories of peripherals. The common peripherals are supported with the dedicated firmware library without any modification, except if the peripheral instance is no longer present. The user can change the instance and, of course, all the related features (clock configuration, pin configuration, interrupt/DMA request).

The modified peripherals such as ADC, RCC and RTC are different from the STM32F1 Series ones and should be updated to take advantage of the enhancements and the new features in the STM32F3 Series.

All these modified peripherals in the STM32F3 Series are enhanced to obtain smaller silicon print with features designed to offer advanced high-end capabilities in economical end products and to fix some limitations present in the STM32F1 Series.

3.1 STM32 product cross-compatibility

The STM32 Series embeds a set of peripherals which can be classed in three categories:

- The first category is for the peripherals which are, by definition, common to all products. Those peripherals are identical, so they have the same structure, registers and control bits. There is no need to perform any firmware change to keep the same functionality, at the application level, after migration. All the features and behavior remain the same.
- The second category is for the peripherals which are shared by all products but have only minor differences (in general to support new features). The migration from one product to another is very easy and does not need any significant new development effort.
- The third category is for peripherals which have been considerably changed from one product to another (new architecture, new features...). For this category of peripherals, the migration will require new development, at the application level.

Table 6 gives a general overview of this classification.

Table 6. STM32 peripheral compatibility analysis STM32F1 Series versus STM32F3 Series

Peripheral	STM32F1 Series	STM32F3 Series	Compatibility		
			Feature	Pinout(1)	FW driver
SPI	Yes	Yes++	Two FIFO available, 4-bit to 16-bit data size selection	Partial compatibility	Partial compatibility
WWDG	Yes	Yes	Same features	NA	Full compatibility
IWDG	Yes	Yes+	Added a Window mode	NA	Full compatibility
DBGMCU	Yes	Yes	Same features	Identical	Full 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	2 DMA controllers with 12 channels(2)	NA	Full compatibility
TIM	Yes	Yes+	Enhancement	Partial compatibility	Full compatibility
PWR	Yes	Yes+	V_{DDA} can be higher than V_{DD}, 1.8 V mode for core, independent V_{DD} for SDADCs	Identical for the same feature	Partial compatibility
RCC	Yes	Yes+	-	PD0 & PD1 => PF0 & PF1 for the oscillator	Partial compatibility
USART	Yes	Yes+	Choice for independent clock sources, timeout feature, wakeup from stop mode	Identical for the same feature	Full compatibility
I2C	Yes	Yes++	Communication events managed by HW, FM+, wakeup from stop mode, digital filter	Partial compatibility	New driver
DAC	Yes	Yes+	DMA underrun interrupt	Identical	Full compatibility
ADC	Yes	Yes++	Same ADC or new fast ADCs	Partial compatibility	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	New GPIOs	Partial compatibility

Table 6. STM32 peripheral compatibility analysis STM32F1 Series versus STM32F3 Series (continued)

Peripheral	STM32F1 Series	STM32F3 Series	Compatibility		
			Feature	Pinout[1]	FW driver
CAN	Yes	Yes	In STM32F303/302xB/C, 512 bytes SRAM is not shared with USB. In STM32F302x6/x8 and STM32F30xxD/E, 256 bytes shared SRAM with USB.	Identical	Full compatibility
USB FS Device	Yes	Yes	In STM32F303/302xB/C, 512 bytes SRAM is not shared with CAN. In STM32F302x6/x8 and STM32F30xxD/E, 726 bytes dedicated SRAM and 256 bytes shared SRAM with CAN i.e. 1Kbytes dedicated SRAM in case CAN is disabled.	Identical	Full compatibility for SM32F303/302xB/C. Partial compatibility for STM32F302x6/x8 and STM32F30xxD/E as the USB is modified compared to STM32F1/F30xxB/C, comes with LPM support and has 726 bytes dedicated SRAM and 256 bytes shared SRAM with CAN, or 1Kbytes dedicated SRAM in case CAN is disabled.
Ethernet	Yes	NA	NA	NA	NA
SDIO	Yes	NA	NA	NA	NA
FMC	Yes	Yes	Available only on the STM32F30xxD/E.	Identical	Full compatibility.
Touch Sensing	NA	Yes	NA	NA	NA
COMP	NA	Yes	NA	NA	NA
OPAMP[3]	NA	Yes	NA	NA	NA
SYSCFG	NA	Yes	NA	NA	NA
SDADC[4]	NA	Yes	NA	NA	NA

1. For more details about the pinout compatibility refer to the pinout table in the related data sheets.
2. In the In the STM32F303/302/301x6/x8, one DMA controller is available, with 7 channels
3. The OPAMP is available only on STM32F30xxx/F358xx.
4. The SDADC is available only on STM32F37xxx/F378xx.

Note: *Yes++ = New feature or new architecture*
Yes+ = Same feature, but specification change or enhancement
Yes = Feature available
NA = Feature not available

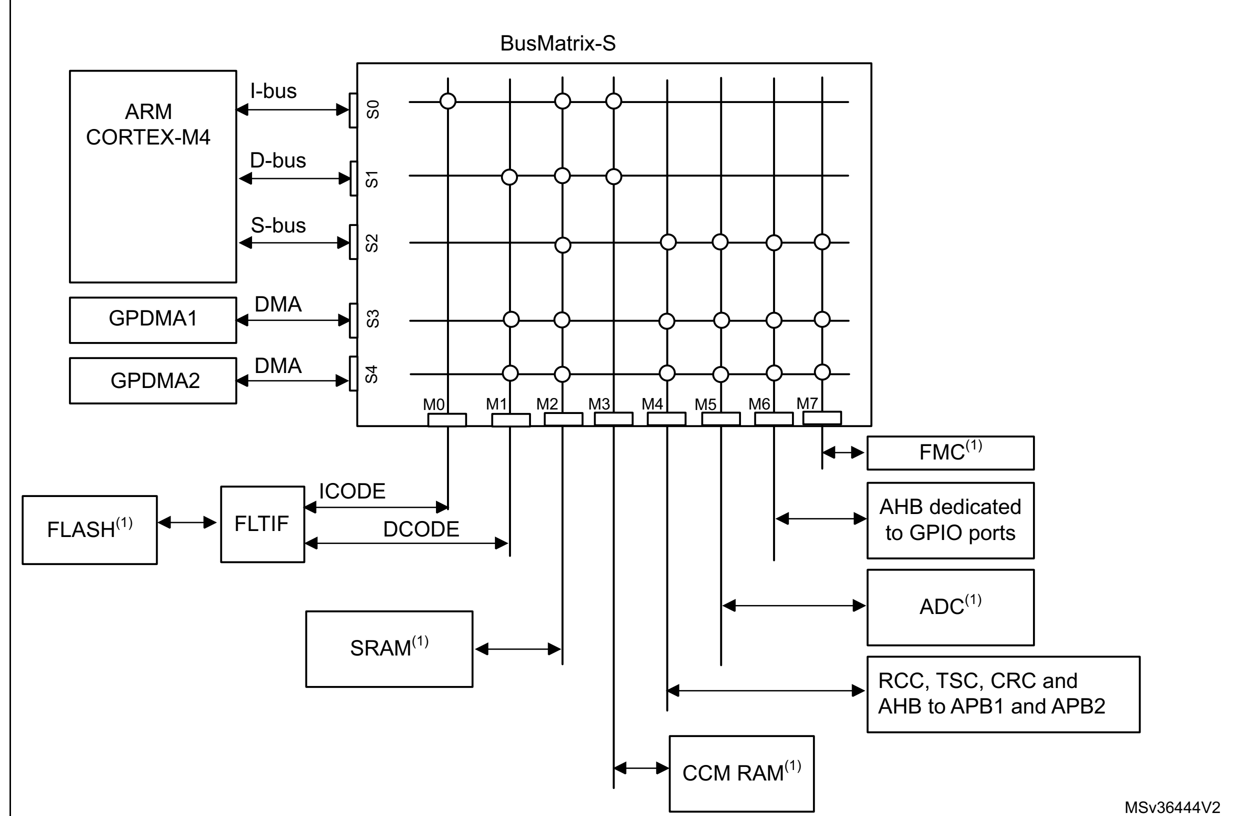
3.2 System architecture

The main system consists of:

- Five masters:
  - Cortex-M4 core I-bus
  - Cortex-M4 core D-bus
  - Cortex-M4 core S-bus
  - GP-DMA1 and GP-DMA2 (general-purpose DMAs)
- Seven or six slaves in STM32F30xxx microcontrollers depending on the CCM RAM and FMC availability in the product:
  - Internal Flash memory on the DCode and ICode
  - CCM SRAM on STM32F303xB/C/D/E and STM32F303x6/x8
  - Internal SRAM
  - AHB to APBx (APB1 or APB2), which connect all the APB peripherals
  - AHB dedicated to GPIO ports
  - ADCs
  - FMC
- Five slaves in STM32F37xxx microcontrollers:
  - Internal Flash memory on the DCode
  - Internal Flash memory on the ICode
  - 2 Kbyte internal SRAM memory
  - AHB to APBx (APB1 or APB2) connecting all the APB peripherals
  - AHB dedicated to GPIO ports.

These are interconnected using a multilayer AHB bus architecture as shown in *Figure 1* and *Figure 2*.

Figure 1. System architecture for STM32F30x and STM32F3x8 lines



1. Flash size, SRAM size, CCM and FMC availability and number of ADCs depend on the product. Refer to the STM32F30x line datasheets.

Figure 2. System architecture for STM32F37x lines

Bus matrix-S
ARM
Cortex-M4
I-bus
D-bus
S-bus
DMA1
DMA
DMA2
DMA
S0
S1
S2
S3
S4
M0
M1
M2
M4
M6
64-bit 256 KB
Flash memory
FLITF
ICODE
DCODE
32 KB SRAM
AHB dedicated to GPIO ports
RCC, TSC, CRC and
AHB to APB1 and APB2
MS19998V2

3.3 Memory mapping

The peripheral address mapping has been changed in the STM32F3 Series versus the STM32F1 Series. The main change concerns the GPIOs which have been moved from the APB bus to the AHB bus to allow them to operate at the maximum speed.

Table 7 provides the peripheral address mapping correspondence between STM32F3 and STM32F1 Series.

Table 7. IP bus mapping differences between STM32F3 Series and STM32F1 Series(1)

<table>
<tr><th rowspan="2">Peripheral</th><th colspan="2">STM32F30xxx</th><th colspan="2">STM32F37xxx</th><th colspan="2">STM32F1 Series</th></tr>
<tr><th>Bus</th><th>Base address</th><th>Bus</th><th>Base address</th><th>Bus</th><th>Base address</th></tr>
<tr><td>TSC</td><td>AHB1</td><td>0x4002 4000</td><td>AHB</td><td>0x4002 4000</td><td>NA</td><td>NA</td></tr>
<tr><td>GPIOH</td><td rowspan="8">AHB2</td><td>0x4800 1C00</td><td>NA</td><td>NA</td><td rowspan="8">APB2</td><td>0x4800 1FFF</td></tr>
<tr><td>GPIOG</td><td>0x4800 1800</td><td>NA</td><td>NA</td><td>0x4800 1BFF</td></tr>
<tr><td>GPIOF</td><td>0x4800 1400</td><td rowspan="6">AHB2</td><td>0x4800 1400</td><td>0x4001 1800</td></tr>
<tr><td>GPIOE</td><td>0x4800 1000</td><td>0x4800 1000</td><td>NA</td></tr>
<tr><td>GPIOD</td><td>0x4800 0C00</td><td>0x4800 0C00</td><td>0x4001 1400</td></tr>
<tr><td>GPIOC</td><td>0x4800 0800</td><td>0x4800 0800</td><td>0x4001 1000</td></tr>
<tr><td>GPIOB</td><td>0x4800 0400</td><td>0x4800 0400</td><td>0x4001 0C00</td></tr>
<tr><td>GPIOA</td><td>0x4800 0000</td><td>0x4800 0000</td><td>0x4001 0800</td></tr>
<tr><td>TIM1</td><td>APB2</td><td>0x4001 2C00</td><td>NA</td><td>NA</td><td>NA</td><td>0x4001 2C00</td></tr>
<tr><td>ADC/ADC1</td><td rowspan="4">AHB3</td><td rowspan="2">0x5000 0000</td><td>APB2</td><td>0x4001 2400</td><td>NA</td><td>0x4001 2400</td></tr>
<tr><td>ADC2</td><td>NA</td><td>NA</td><td>NA</td><td>0x40012 800</td></tr>
<tr><td>ADC3</td><td rowspan="2">0x5000 0400</td><td>NA</td><td>NA</td><td>NA</td><td>0x40013 C00</td></tr>
<tr><td>ADC4</td><td>NA</td><td>NA</td><td>NA</td><td>NA</td></tr>
<tr><td>SDADC3</td><td>NA</td><td>NA</td><td rowspan="4">APB2</td><td>0x4001 6800</td><td>NA</td><td>NA</td></tr>
<tr><td>SDADC2</td><td>NA</td><td>NA</td><td>0x4001 6400</td><td>NA</td><td>NA</td></tr>
<tr><td>SDADC1</td><td>NA</td><td>NA</td><td>0x4001 6000</td><td>NA</td><td>NA</td></tr>
<tr><td>TIM19</td><td>NA</td><td>NA</td><td>0x4001 5C00</td><td>NA</td><td>NA</td></tr>
<tr><td>SYSCFG</td><td rowspan="3">APB2 (through SYSCFG)</td><td rowspan="3">0x4001 0000</td><td rowspan="2">APB2 (through SYSCFG)</td><td rowspan="2">0x4001 0000</td><td rowspan="2">APB2</td><td>NA</td></tr>
<tr><td>COMP</td><td>NA</td></tr>
<tr><td>OPAMP</td><td>NA</td><td>NA</td><td>NA</td><td>NA</td></tr>
</table>

Table 7. IP bus mapping differences between STM32F3 Series and STM32F1 Series[1] (continued)

Peripheral	STM32F30xxx		STM32F37xxx		STM32F1 Series	
	Bus	Base address	Bus	Base address	Bus	Base address
CEC	APB1	0x4000 7800	APB1	0x4000 7800	APB1	0x4000 7800
DAC/DAC1		0x4000 7400		0x4000 7400		0x400 07400
DAC2		NA		0x4000 9800		NA
I2S2ext	APB1	0x4000 3400	APB1	NA	APB1	NA
TIM14	APB1	NA	APB1	0x4000 2000	NA	NA
TIM18		NA		0x4000 9C00	NA	NA
FSMC Registers	NA	NA	NA	NA	AHB	0xA000 0000
USB OTG FS	NA	NA	NA	NA		0x5000 0000
ETHERNET MAC	NA	NA	NA	NA		0x4002 8000
SDIO	NA	NA	NA	NA		0x4001 8000
TIM20	APB2	0x4001 5000	NA	NA	NA	NA
SPI4	APB2	0x4001 3C00	NA	NA	NA	NA
I2C3	APB1	0x4000 7800	NA	NA	NA	NA
TIM11	NA	NA	NA	NA	APB2	0x40015 400
TIM10	NA	NA	NA	NA		0x40015 000
TIM9	NA	NA	NA	NA		0x40014 C00
TIM8	APB2	0x4001 3400	NA	NA		0x40013 400

Table 7. IP bus mapping differences between STM32F3 Series and STM32F1 Series[1] (continued)

Peripheral	STM32F30xxx		STM32F37xxx		STM32F1 Series	
	Bus	Base address	Bus	Base address	Bus	Base address
CAN2	NA	NA	NA	NA	APB1	0x40006 800
CAN1	APB1	0x4000 6400	APB1	0x4000 6400		0x4000 6400
UART5		0x4000 5000	NA	NA		0x40005000
UART4		0x4000 4C00	NA	NA		0x40004C00
I2S2ext		0x4000 4000	NA	NA		NA
TIM13	NA	NA	APB1	0x4000 1C00		0x4000 1C00
TIM12	NA	NA		0x4000 1800		0x4000 1800
TIM5	NA	NA		0x4000 0C00		0x40000 C00
BKP registers	NA	NA	NA	NA		0x4000 6C00
AFIO	NA	NA	NA	NA	APB2	0x4001 0000

1. NA = not applicable

3.4 Reset and clock controller (RCC) interface

The main differences related to the RCC (Reset and clock controller) in STM32F3 Series versus STM32F1 Series are presented in *Table 8*.

Table 8. RCC differences between STM32F1 Series and STM32F3 Series

RCC	STM32F1 Series	STM32F3 Series
HSE	3 - 25 MHz depending on the product line used	4 - 32 MHz
PLL	– Connectivity line: main PLL + 2 PLLs for I2S, Ethernet and OTG FS clock – Other product lines: main PLL	Main PLL
RTC clock source	LSI, LSE or HSE/128	LSI, LSE or HSE clock divided by 32

Table 8. RCC differences between STM32F1 Series and STM32F3 Series (continued)

RCC	STM32F1 Series	STM32F3 Series
MCO clock source	MCO pin (PA8): – Connectivity line: HSI, HSE, PLLCLK/2, SYSCLK, PLL2, PLL3 or XT1 – Other product lines: HSI, HSE, PLLCLK/2 or SYSCLK	STM32F302/303xB/C and STM32F37xxx: SYSCLK, HSI, HSE, PLLCLK/2, LSE, LSI. STM32F302/301/303x6/x8: SYSCLK, HSI, HSE, PLLCLK/2 or PLLCLK, LSE, LSI; with the possibility to reduce MCO frequency by a configurable divider.
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 TIM16 timer (STM32F30/31xxx) and TIM14 (STM32F37/38xxx) with respect to HSI/HSE clock – HSE are indirectly measured through MCO using TIM16 timer (STM32F30/31xxx) and TIM14 (STM32F37/38xxx) channel 1 input capture with respect to HSI clock.

In addition to the differences described in the table above, the following additional adaptation steps may be needed for the migration:

- System clock configuration
- Peripheral access configuration
- Peripheral clock configuration

3.4.1 System clock configuration

When moving from STM32F1 Series to STM32F3 Series, only a few settings need to be updated in the system clock configuration code; mainly the Flash settings (configure the right wait states for the system frequency, prefetch enable/disable) or/and the PLL parameters configuration:

- If HSE or HSI is used directly as the system clock source, only the Flash parameters should be modified.
- If PLL (clocked by HSE or HSI) is used as the system clock source, the Flash parameters and PLL configuration need to be updated.

Table 9 below provides an example of how to port a system clock configuration from STM32F1 Series to STM32F3 Series:

- STM32F100 value line running at maximum performance: system clock at 24 MHz (PLL, clocked by the HSE (8 MHz), used as the system clock source), Flash with 0 wait states and Flash prefetch queue enabled.
- STM32F3 Series running at maximum performance: system clock at 72 MHz (PLL, clocked by the HSE (8 MHz), used as the system clock source), Flash memory with 2 wait states and Flash prefetch enabled.

Note: In the STM32F30xxD/E devices, the maximum system clock of 72 MHz can be reached when the PLL is the system clock source and the PLL clock source is the HSI. In the STM32F1 Series and other STM32F3 Series devices, when the PLL clock source is the HSI, the maximum system clock value is 64 MHz.

As shown in *Table 9*, only the Flash settings and PLL parameters (code in ***Bold Italic***) need to be rewritten to run on STM32F3 Series. However, HSE, AHB prescaler and the system clock source configuration are left unchanged, and the APB prescalers are adapted to the maximum APB frequency in the STM32F3 Series.

Note: The source code presented in Table 9 is intentionally simplified (timeout in wait loop removed) and is based on the assumption that the RCC and Flash registers are at their reset values.

For the STM32F3xxxx devices, the user can use the clock configuration tool, STM32F3xx_Clock_Configuration.xls, to generate a customized system_stm32F3xx.c file containing a system clock configuration routine, depending on the user application requirements.

Table 9. Example of migrating system clock configuration code from STM32F100 Value line to STM32F3 Series

<table>
<tr><th>STM32F100 Value Line running at 24 MHz
(PLL as clock source) with 0 wait states</th><th>STM32F3xx running at 48 MHz
(PLL as clock source) with 1 wait state</th></tr>
<tr><td>/* 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)
{
}</td><td>/* 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)
{
}</td></tr>
</table>

3.4.2 Peripheral access configuration

Since the address mapping of some peripherals has been changed in the STM32F3 Series versus the STM32F1 Series, the user needs to use different registers to [enable/disable] or [enter/exit] the peripheral [clock] or [from reset mode].

Table 10. 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

To configure the access to a given peripheral, the user has first to know to which bus this peripheral is connected; refer to *Table 7* then, depending on the action needed, program the right register as described in *Table 10* above. For example, if USART1 is connected to the APB2 bus, to enable the USART1 clock the user has to configure APB2ENR register as follows:

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

3.4.3 Peripheral clock configuration

Some peripherals have a dedicated clock source independent from the system clock, and used to generate the clock required for them to operate:

- ADC

  In the STM32F30xxx devices, the ADC features two possible clock sources:
  - The ADC clock can be derived from the PLL output. It can reach 72 MHz and can be divided by the following prescalers values: 1, 2, 4, 6, 8, 10, 12, 16, 32, 64, 128, or 256. It is asynchronous to the AHB clock
  - The ADC clock can also be derived from the AHB clock of the ADC bus interface, divided by a programmable factor (1, 2 or 4). When the programmable factor is 1, the AHB prescaler must be equal to '1'.
- SDADC

  The STM32F37xxx SDADC clock source is derived from the system clock divided by a wide range of prescalers, ranging from 2 to 48.
- RTC

  In the STM32F3 Series devices, the RTC features three possible clock sources:
  - The first one is based on the HSE Clock; a prescaler divides its frequency by 32 before going to the RTC
  - The second one is the LSE oscillator
  - The third clock source is the LSI RC with a value of 40 kHz.
- TIMx

  The timers clock frequencies are automatically defined by hardware. There are two cases:
  - If the APB prescaler equals 1, the timer clock frequencies are set to the same frequency as that of the APB domain
  - Otherwise, they are set to twice (×2) the frequency of the APB domain.

  PLL clock source

  A clock issued from the PLL (PLLCLKx2) can be selected for TIMx (x = 1, 8 on the STM32F303xB/C, x = 1 on the STM32F303x6/8, x = 1, 15, 16, 17 on the STM32F302/301x6/8), x = 1,2,3,4,8,15,16,17,20 in the STM32F303xD/E.

  This configuration allows to feed TIMx with a frequency up to 144 MHz when the system clock source is the PLL. In this configuration:
  - On the STM32F303/302xB/C, AHB and APB2 prescalers are set to 1 i.e. AHB and APB2 clocks are not divided with respect to the system clock
  - On the STM32F303/302/301x6/8 and STM32F302/303xD/E, AHB or APB2 subsystem clocks are not divided by more than 2 cumulatively with respect to the system clock.
- I2S

  In the STM32F30xxx devices, the I2S features 2 possible clock sources:
  - The first one is the System clock
  - The second one comes from the external clock provided on I2S_CKIN pin.

3.5 DMA interface

The STM32F1 Series and STM32F3 Series use the same fully compatible DMA controller.

Table 11 presents the correspondence between the DMA requests of the peripherals in STM32F1 Series and STM32F3 Series.

The STM32F303/302xB/C devices have two DMA controllers with 12 channels in total (as STM32F1), the STM32F302/301/303x6/8 devices have 1 DMA controller with 7 channels

Table 11. DMA request differences between STM32F3 Series and STM32F1 Series(1)

Peripheral	DMA request	STM32F1 Series	STM32F30xxx(2)	STM32F37xxx
ADC2	**ADC2**	NA	DMA2_Channel1 DMA2_Channel3	NA
ADC3	**ADC3**	DMA2_Channel5	DMA2_Channel5	NA
ADC4	**ADC4**	NA	DMA2_Channel2 DMA2_Channel4	NA
SDADC	**SDADC1** **SDADC2** **SDADC3**	NA	NA	DMA2_Channel3 DMA2_Channel4 DMA2_Channel5
SPI4(3)	SPI4_RX SPI4_TX	NA	DMA2_Channel4 DMA2_Channel5	NA
TIM6/DAC1_CH1 **TIM7/DAC1_CH2**	**DAC1_Channel1** **DAC1_Channel2**	DMA2_Channel3 DMA2_Channel4	DMA2_Channel3 DMA1_Channel3 DMA2_Channel4 DMA1_Channel4	DMA2_Channel3 DMA1_Channel3 DMA1_Channel4 DMA2_Channel4
TIM18/DAC2	**DAC2_Channel1** **DAC2_Channel2**	NA	NA	DMA2_Channel5 DMA1_Channel5
UART4	**UART4_Rx** **UART4_Tx**	DMA2_Channel3 DMA2_Channel5	DMA2_Channel3 DMA2_Channel5	NA
UART5	**UART5_Rx** **UART5_Tx**	DMA2_Channel4 DMA2_Channel1	NA	NA
I2C3(4)	**I2C3_Rx** **I2C3_Rx**	NA	DMA1_Channel2 DMA1_Channel1	NA
SDIO	**SDIO**	DMA2_Channel4	NA	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	DMA1_Channel5 DMA1_Channel2 DMA1_Channel3 DMA1_Channel6 DMA1_Channel4 DMA1_Channel4 DMA1_Channel4	NA

Table 11. DMA request differences between STM32F3 Series and STM32F1 Series[1] (continued)

Peripheral	DMA request	STM32F1 Series	STM32F30xxx[2]	STM32F37xxx
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	DMA2_Channel1 DMA2_Channel3 DMA2_Channel5 DMA2_Channel1 DMA2_Channel2 DMA2_Channel2 DMA2_Channel2	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	DMA2_Channel2 DMA2_Channel5 DMA2_Channel4 DMA2_Channel2 DMA2_Channel1 DMA2_Channel1
TIM16	**TIM16_UP** **TIM16_CH1**	DMA1_Channel6 DMA1_Channel6	DMA1_Channel3 DMA1_Channel3	DMA1_Channel3 DMA1_Channel6 DMA1_Channel3 DMA1_Channel6
TIM17	**TIM17_UP** **TIM17_CH1**	DMA1_Channel7 DMA1_Channel7	DMA1_Channel1 DMA1_Channel1	DMA1_Channel1 DMA1_Channel7 DMA1_Channel1 DMA1_Channel7
TIM19	**TIM19_UP** **TIM19_CH1** **TIM19_CH2** **TIM19_CH3** **TIM19_CH4**	NA	NA	DMA1_Channel4 DMA2_Channel5 DMA1_Channel2 DMA1_Channel1 DMA1_Channel1
TIM20[3]	**TIM20_UP** **TIM20_CH1** **TIM20_CH2** **TIM20_CH3** **TIM20_CH4** **TIM20_TRIG** **TIM20_COM**	NA	DMA2_Channel3 DMA2_Channel1 DMA2_Channel2 DMA2_Channel3 DMA2_Channel4 DMA2_Channel4 DMA2_Channel4	NA

1. NA = not applicable.
2. DMA2 channels available only on STM32F303/302xB/C/D/E.
3. Only on STM32F303/302xD/E.
4. Only on STM32F302/301x6/x8 and STM32F303/302xD/E.

3.6 Interrupt vectors

Table 12 presents the interrupt vectors in STM32F3 Series versus STM32F1 Series.

The switch from Cortex-M3 to Cortex-M4 induces an increase of the number of vector interrupts. This leads to many differences between the two devices.

Table 12. Interrupt vector differences between STM32F3 Series and STM32F1 Series

Position	STM32F1 Series	STM32F30xxx	STM32F37xxx
2	TAMPER	TAMP_STAMP	TAMP_STAMP
3	RTC	RTC_WKUP	RTC_WKUP
18	ADC1_2	ADC1_2	ADC1
19	CAN1_TX / USB_HP_CAN_TX	USB_HP/CAN_TX	CAN_TX
20	CAN1_RX0 / USB_LP_CAN_RX0	USB_LP/CAN_RX0	CAN_RXD
21	CAN1_RX1	CAN1_RX1	CAN_RXI
22	CAN1_SCE	CAN1_SCE	CAN_SCE
23	EXTI9_5	EXTI9_5	EXTI9_5
24	TIM1_BRK / TIM1_BRK _TIM9	TIM1_BRK / TIM15	TIM15
25	TIM1_UP / TIM1_UP_TIM10	TIM1_UP / TIM16	TIM16
26	TIM1_TRG_COM / TIM1_TRG_COM_TIM11	TIM1_TRG_COM / TIM17	TIM17
27	TIM1_CC	TIM1_CC	TIM18_DAC2
42	OTG_FS_WKUP / USBWakeUp	USB_WKUP	CEC
43	TIM8_BRK / TIM8_BRK_TIM12 (1)	TIM8_BRK	TIM12
44	TIM8_UP / TIM8_UP_TIM13 (1)	TIM8_UP	TIM13
45	TIM8_TRG_COM / TIM8_TRG_COM_TIM14 (1)	TIM8_TRG_COM	TIM14
46	TIM8_CC	TIM8_CC	Reserved
47	ADC3	ADC3	Reserved
48	FSMC	Reserved	Reserved
49	SDIO	Reserved	Reserved
50	TIM5	Reserved	TIM5
52	UART4	UART4	Reserved
53	UART5	UART5	Reserved
59	DMA2_Channel4 / DMA2_Channel4_5 (1)	DMA2_CH4	DMA2_CH4

Table 12. Interrupt vector differences between STM32F3 Series and STM32F1 Series (continued)

Position	STM32F1 Series	STM32F30xxx	STM32F37xxx
61	ETH	ADC4	SDADC1
62	ETH_WKUP	Reserved	SDADC2
63	CAN2_TX	Reserved	SDADC3
64	CAN2_RX01	COMP123	COMP1_2
65	CAN2_RX1	COMP456	Reserved
66	CAN2_SCE	COMP7	Reserved
67	OTG_FS	Reserved	Reserved
72	Reserved	I2C3_EV (only on STM32F302/301x6/x8) and STM32F303/302xD/E	Reserved
73	Reserved	I2C3_ER (only on STM32F302/301x6/x8 and STM32F302/F303xD/E)	Reserved
76	Reserved	USB_WKUP	USB_WKUP
77	Reserved	TIM20_BRK (only on STM32F303xD/E)	Reserved
78	Reserved	TIM20_UP (only on STM32F303xD/E)	TIM19 global interrupt
79	Reserved	TIM20_TRG_COM (only on STM32F303xD/E)	Reserved
80	Reserved	TIM20_CC (only on STM32F303xD/E)	Reserved
81	Reserved	FPU	FPU
84	Reserved	SPI4 (only on STM32F303xD/E)	Reserved

1. Depending on the product line used.

3.7 GPIO interface

The STM32F3 Series GPIO peripheral embeds new features compared to STM32F1 Series, below the main features:

- GPIO mapped on AHB bus for better performance
- I/O pin multiplexer and mapping: pins are connected to on-chip peripherals/modules through a multiplexer that allows only one peripheral alternate function (AF) connected to an I/O pin at a time. In this way, there can be no conflict between peripherals sharing the same I/O pin.
- More possibilities and features for I/O configuration

The STM32F3 Series GPIO peripheral is a new design and thus the architecture, the features and registers are different from the GPIO peripheral in the STM32F1 Series. Any

code written for the STM32F1 Series using the GPIO needs to be rewritten to run on STM32F3 Series.

For more information about STM32F3 Series GPIO programming and usage, refer to the “I/O pin multiplexer and mapping” section in the GPIO chapter of the STM32F3xx reference manuals (RM0313 and RM0316).

Table 13 presents the differences between GPIOs in the STM32F1 Series and STM32F3 Series.

Table 13. GPIO differences between STM32F1 Series and STM32F3 Series

GPIO	STM32F1 Series	STM32F3 Series
General purpose output	PP OD	PP PP + PU PP + PD OD OD + PU OD + PD
Alternate function output	PP OD	PP PP + PU PP + PD OD OD + PU OD + PD
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 frequency	18 MHz	36 MHz

3.7.1 Alternate function mode

STM32F1 Series

The configuration to use an I/O as an alternate function depends on the peripheral mode used. For example, the USART Tx pin should be configured as an alternate function push-pull, while the USART Rx pin should be configured as input floating or input pull-up.

To optimize the number of peripheral I/O functions for different device packages (especially those with a low pin count), it is possible to remap some alternate functions to other pins by software. For example, the USART2_RX pin can be mapped on PA3 (default remap) or PD6 (by software remap).

STM32F3 Series

Whatever the peripheral mode used, the I/O must be configured as an alternate function, then the system can use the I/O in the proper way (input or output).

The I/O pins are connected to on-chip peripherals/modules through a multiplexer that allows only one peripheral alternate function to be connected to an I/O pin at a time. In this way, there can be no conflict between peripherals sharing the same I/O pin. Each I/O pin has a multiplexer with 16 alternate function inputs (AF0 to AF15) that can be configured through the GPIOx_AFRL and GPIOx_AFRH registers: the peripheral alternate functions are mapped by configuring AF0 to AF15.

In addition to this flexible I/O multiplexing architecture, each peripheral has alternate functions mapped on different I/O pins to optimize the number of peripheral I/O functions for different device packages. For example, the USART2_RX pin can be mapped on PA3 or PA15 pin.

Note: Refer to the "Alternate function mapping" table in the STM32F3x datasheet for the detailed mapping of the system and the peripheral alternate function I/O pins.

1. Configuration procedure
   - Configure the desired I/O as an alternate function in the GPIOx_MODER register
   - Select the type, pull-up/pull-down and output speed via the GPIOx_OTYPER, GPIOx_PUPDR and GPIOx_OSPEEDER registers, respectively
   - Connect the I/O to the desired AFx in the GPIOx_AFRL or GPIOx_AFRH register

3.8 EXTI source selection

In the STM32F1 Series devices, the selection of the EXTI line source is performed through EXTIx bits in AFIO_EXTICRx registers, while in the STM32F3 Series this selection is done through EXTIx bits in SYSCFG_EXTICRx registers.

Only the mapping of the EXTICRx registers has been changed, without any changes to the meaning of the EXTIx bits. However, the maximum range of EXTIx bit values is 0b0101 as the last PORT is F (in STM32F1 Series, the maximum value is 0b0110).

3.9 FLASH interface

Table 14 presents the difference between the FLASH interface of STM32F1 Series and STM32F3 Series.

Table 14. FLASH differences between STM32F1 Series and STM32F3 Series

Feature		STM32F1 Series	STM32F3 Series
Main/Program memory granularity		Page size = 2 Kbytes except for Low and Medium density page size = 1 Kbyte	Page size = 2 Kbytes
System memory	**Start Address**	0x1FFF F000	0x1FFF D800
	End Address	0x1FFF F7FF	0x1FFF F7FF
Flash interface programming procedure		Same for all product lines	Same as STM32F1 Series for Flash program and erase operations. Different from STM32F1 Series for Option byte programming

Table 14. FLASH differences between STM32F1 Series and STM32F3 Series (continued)

Feature		STM32F1 Series	STM32F3 Series
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 a granularity of 2 pages
User Option bytes		STOP	STOP
		STANDBY	STANDBY
		WDG	WDG
		NA	RAM_PARITY_CHECK
		NA	VDDA_MONITOR
		NA	nBOOT1
		NA	SDADC12_VDD MONITOR (available only on STM32F37xxx)

3.10 SAR ADC interface

The STM32F37xxx ADC interface is identical to STM32F1 Series, while a different ADC interface is used in the STM32F30xxx devices.

Table 15 presents the differences between the ADC interface of STM32F1 Series and STM32F30x lines; these differences are the following:

- New digital interface
- New architecture and new features

Table 15. ADC differences between STM32F1 Series and STM32F30x lines

ADC	STM32F1 Series	STM32F30xxB/C	STM32F302x6/8/ STM32F301x6/8	STM32F303x6/8	STM32F30xxD/E
Instances	ADC1 / ADC2 / ADC3	ADC1 / ADC2 / ADC3 / ADC4	ADC1	ADC1 / ADC2	ADC1 / ADC2 / ADC3 / ADC4
Maximum sampling frequency	1 MSPS	5.1 MSPS	5.1 MSPS	5.1 MSPS	5.1 MSPS
Number of channels	Up to 21	Up to 39 + 7 internal	Up to 15 + 3 internal	Up to 21 + 4 internal	Up to 40 + 7 internal
Resolution	12 bits	Configurable to 12, 10, 8, and 6 bits	Configurable to 12, 10, 8, and 6 bits	Configurable to 12, 10, 8, and 6 bits	Configurable to 12, 10, 8, and 6 bits

Table 15. ADC differences between STM32F1 Series and STM32F30x lines (continued)

ADC	STM32F1 Series	STM32F30xxB/C	STM32F302x6/8/ STM32F301x6/8	STM32F303x6/8	STM32F30xxD/E
Conversion modes	Single / continuous / scan / discontinuous / dual mode	Single / continuous / scan / discontinuous / dual mode	Single / continuous / scan / discontinuous mode	Single / continuous / scan / discontinuous / dual mode	Single / continuous / scan / discontinuous / dual mode
Supply requirement	2.4 V to 3.6 V	2.0 to 3.6 V	2.0 to 3.6 V	2.0 to 3.6 V	2.0 to 3.6 V

Table 16. ADC channels mapping differences

	STM32F103xx	STM32F303xB/C/D/E	STM32F302/301x6/8	STM32F303x6/8
PA0	ADC123_IN0	ADC1_IN1	ADC1_IN1	ADC1_IN1
PA1	ADC123_IN1	ADC1_IN2	ADC1_IN2	ADC1_IN2
PA2	ADC123_IN2	ADC1_IN3	ADC1_IN3	ADC1_IN3
PA3	ADC123_IN3	ADC1_IN4	ADC1_IN4	ADC1_IN4
PA4	ADC12_IN4	ADC2_IN1	ADC1_IN5	ADC2_IN1
PA5	ADC12_IN5	ADC2_IN2	-	ADC2_IN2
PC0	ADC123_IN10	ADC12_IN6	ADC1_IN6	ADC12_IN6
PC1	ADC123_IN11	ADC12_IN7	ADC1_IN7	ADC12_IN7
PC2	ADC123_IN12	ADC12_IN8	ADC1_IN8	ADC12_IN8
PC3	ADC123_IN13	ADC12_IN9	ADC1_IN9	ADC12_IN9
PC4	ADC12_IN14	ADC2_IN5	-	ADC2_IN5
PC5	ADC12_IN15	ADC2_IN11	-	ADC2_IN11
PA6	ADC12_IN6	ADC2_IN3	ADC1_IN10	ADC2_IN3
PB0	ADC12_IN8	ADC3_IN12	ADC1_IN11	ADC1_IN11
PB1	ADC12_IN9	ADC3_IN14	ADC1_IN12	ADC1_IN12
PB2	-	ADC2_IN12	-	ADC2_IN12
PB11	-	ADC12_IN14(1)	ADC1_IN14	-
PB12	-	ADC2_IN13	-	ADC2_IN13
PB13	-	ADC3_IN5	ADC1_IN13	ADC1_IN13
PB14	-	ADC4_IN4	-	ADC2_IN14
PB15	-	ADC4_IN5	-	ADC2_IN15
PA7	ADC12_IN7	ADC2_IN4	ADC1_IN15	ADC2_IN4

1. Only on STM32F303/302xD/E.

Note: On STM32F37xxx, the ADC channels mapping is the same as in STM32F10xxx.

3.11 PWR interface

In STM32F3 Series the PWR controller presents some differences versus STM32F1 Series, these differences are summarized in *Table 17*. However, the programming interface is unchanged.

Table 17. PWR differences between STM32F1 Series and STM32F3 Series

PWR	STM32F1 Series	STM32F3 Series
Power supplies	– V_{DD} = 2.0 to 3.6 V: external power supply for I/Os and the internal regulator. Provided externally through VDD pins. – V_{SSA}, V_{DDA} = 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. – V_{BAT} = 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.	– V_{DD} = 2.0 to 3.6 V: external power supply for I/Os and the internal regulator. Provided externally through VDD pins. – V_{SSA}, V_{DDA} = 2.0 to 3.6 V: external analog power supplies for ADC, DAC, Reset blocks, RCs and PLL. V_{SSA} must be connected to V_{SS}. – V_{BAT} = 1.65 to 3.6 V: power supply for RTC, external clock 32 kHz oscillator and backup registers (through power switch) when VDD is not present. – V_{SSSD}, V_{DDSDx} = 2.0 to 3.6V: external analog power supplies for SDADCx peripherals and some GPIOs (STM32F37xxx only).
Battery backup domain	– Backup registers – RTC – LSE – PC13 to PC15 I/Os	– Backup registers – RTC – LSE – RCC Backup Domain Control Register
Wake-up sources	Standby mode – WKUP pin rising edge	Standby mode – WKUP1, WKUP2 or WKUP3 pin rising edge

3.12 Real-time clock (RTC) interface

The STM32F3 Series embeds a new RTC peripheral versus the STM32F1 Series. The architecture, features and programming interface are different.

As a consequence, the STM32F3 Series RTC programming procedures and registers are different from those of the STM32F1 Series, so any code written for the STM32F1 Series using the RTC needs to be rewritten to run on the STM32F3 Series.

The STM32F3 Series RTC provides best-in-class features:

- BCD timer/counter
- Time-of-day clock/calendar featuring subsecond precision with programmable daylight saving compensation
- Two programmable alarms
- Digital calibration circuit
- Time-stamp function for event saving
- Accurate synchronization with an external clock using the subsecond shift feature.
- STM32F30xxx/STM32F37xxx feature 16 and 32 backup registers, respectively (64 and 128 bytes) which are reset when a tamper detection event occurs

For more information about the STM32F3's RTC features, refer to RTC section of STM32F30/31xx and STM32F37/38xx Reference Manuals (RM0316 and RM0313).

For advanced information about the RTC programming, refer to the application note *Using the STM32 HW real-time clock (AN3371)*.

3.13 SPI interface

The STM32F3 Series embeds a new SPI peripheral versus the STM32F1 Series. The architecture, features and programming interface are modified to introduce new capabilities.

As a consequence, the STM32F3 Series SPI programming procedures and registers are similar to those of the STM32F1 Series but with new features. The code written for the STM32F1 Series using the SPI needs little rework to run on the STM32F3 Series, if it did not use new capabilities.

The STM32F3 Series SPI provides best-in-class added features:

- Enhanced NSS control - NSS pulse mode (NSSP) and TI mode
- Programmable data frame length from 4-bit to 16-bit
- Two 32-bit Tx/Rx FIFO buffers with DMA capability and data packing access for frames fitted into one byte (up to 8-bit)
- 8-bit or 16-bit CRC calculation length for 8-bit and 16-bit data.

Furthermore, the SPI peripheral, available in the STM32F3 Series, fixes the CRC limitation present in the STM32F1 Series. For more information about the STM32F3 SPI features, refer to SPI section of STM32F30/31xx and STM32F37/38xx Reference Manuals (RM0316 and RM0313).

3.14 I2C interface

The STM32F3 Series embeds a new I2C peripheral versus the STM32F1 Series. The architecture, features and programming interface are different.

As a consequence, the STM32F3 Series I2C programming procedures and registers are different from those of the STM32F1 Series, so any code written for the STM32F1 Series using the I2C needs to be rewritten to run on the STM32F3 Series.

The STM32F3 Series I2C provides best-in-class new features:

- Communication events managed by hardware.
- Programmable analog and digital noise filters.
- Independent clock source: HSI or SYSCLK.
- Wake-up from STOP mode.
- Fast mode + (up to 1MHz) with 20mA I/O output current drive.
- 7-bit and 10-bit addressing mode, multiple 7-bit slave address support with configurable masks.
- Address sequence automatic sending (both 7-bit and 10-bit) in master mode.
- Automatic end of communication management in master mode.
- Programmable Hold and Setup times.
- Command and Data Acknowledge control.

For more information about the STM32F3 I2C features, refer to I2C section of STM32F30/31xx and STM32F37/38xx Reference Manuals (RM0316 and RM0313).

3.15 USART interface

The STM32F3 Series embeds a new USART peripheral versus the STM32F1 Series. The architecture, features and programming interface are modified to introduce new capabilities.

As a consequence, the STM32F3 Series USART programming procedures and registers are modified from those of the STM32F1 Series, so any code written for the STM32F1 Series using the USART needs to be updated to run on the STM32F3 Series.

The STM32F3 Series USART provides best-in-class added features:

- A choice of independent clock sources allowing
  - UART functionality and wake-up from low power modes,
  - convenient baud-rate programming independently of the APB clock reprogramming.
- SmartCard emulation capability: T=0 with auto retry and T=1
- Swappable Tx/Rx pin configuration
- Binary data inversion
- Tx/Rx pin active level inversion
- Transmit/receive enable acknowledge flags
- New Interrupt sources with flags:
  - Address/character match
  - Block length detection and timeout detection
- Timeout feature
- Modbus communication
- Overrun flag disable
- DMA disable on reception error
- Wake-up from STOP mode
- Auto baud rate detection capability
- Driver Enable signal (DE) for RS485 mode

For more information about the STM32F3 USART features, refer to USART section in the corresponding Reference Manual.

3.16 CEC interface

The CEC interface is available only on the STM32F37xxx devices.

The STM32F3 Series embeds a new CEC peripheral versus the STM32F1 Series. The architecture, features and programming interface are modified to introduce new capabilities.

As a consequence, the STM32F3 Series CEC programming procedures and registers are different from those of the STM32F1 Series, so any code written for the STM32F1 Series using the CEC needs to be rewritten to run on the STM32F3 Series.

The STM32F3 Series CEC provides best-in-class added features:

- 32 kHz CEC kernel with dual clock
  - LSE
  - HSI/244
- Reception in listen mode
- Rx tolerance margin: standard or extended
- Arbitration (signal free time): standard (by H/W) or aggressive (by S/W)
- Arbitration lost detected flag/interrupt
- Automatic transmission retry supported in case of arbitration lost
- Multi-address configuration
- Wake-up from STOP mode
- Receive error detection
  - Bit rising error (with stop reception)
  - Short bit period error
  - Long bit period error
- Configurable error bit generation
  - on bit rising error detection
  - on long bit period error detection
- Transmission under run detection
- Reception overrun detection

The following features present in the STM32F1 Series are now handled by the new STM32F3 Series CEC features and thus are no more available.

- Bit timing error mode & bit period error mode, by the new error handler
- Configurable prescaler frequency divider, by the CEC fixed kernel clock

For more information about the STM32F3 CEC features, refer to CEC section of STM32F37/38xxx Reference Manual (RM0313).

4 Firmware migration using the library

This section describes how to migrate an application based on the STM32F1xx Standard Peripherals Library to the STM32F3xx Standard Peripherals Library.

The STM32F1xx and STM32F3xx libraries have the same architecture and are CMSIS compliant; they use the same driver naming and the same APIs for all compatible peripherals.

Only a few peripheral drivers need to be updated to migrate the application from an STM32F1 Series to an STM32F3 Series product.

Note: In the rest of this section (unless otherwise specified), the term "STM32F3xx library" is used to refer to the STM32F3xx Standard Peripherals Library, and the term "STM32F10x library" is used to refer to the STM32F10x Standard Peripherals Library.

The STM32F3xxxx devices refer to the STM32F30/31xxx devices and STM32F37/38xxx devices.

4.1 Migration steps

To update the user application code to run on the STM32F3xx library, follow the steps listed below:

1. Update the toolchain startup files:
   a) *Project files:* device connections and Flash memory loader. These files are provided with the latest version of the user toolchain that supports STM32F3xxxx devices. For more information, refer to the user toolchain documentation.
   b) *Linker configuration and vector table location files:* these files are developed following the CMSIS standard and are included in the STM32F3xx library install package under the following directory: *Libraries\CMSIS\Device\ST\STM32F3xx.*
2. Add STM32F3xx library source files to the application sources:
   a) Replace the *stm32f10x_conf.h* file of the user application by *stm32f3xx_conf.h* provided in STM32F3xx library.
   b) Replace the existing *stm32f10x_it.c/stm32f10x_it.h files* in the user application by *stm32f3xx_it.c/Stm32f3xx_it.h* provided in STM32F3xx library.
3. Update the part of the user application code that uses the PWR, GPIO, FLASH, ADC and RTC drivers. Further details are provided in the next section.

Note: The STM32F3xx library comes with a rich set of examples (around 76 in total) demonstrating how to use the different peripherals. They are located under Project\STM32F3xx_StdPeriph_Examples\.

4.2 RCC driver

4.2.1 System clock configuration

As presented in *Section 3.4: Reset and clock controller (RCC) interface*, the STM32F3 Series and STM32F1 Series have the same clock sources and configuration procedures. However, there are some differences related to the product voltage range, PLL configuration, maximum frequency and Flash wait state configuration. Thanks to the CMSIS

layer, these differences are hidden from the application code: only replace the *system_stm32f10x.c* file by *system_stm32f3xx.c* file. This file provides an implementation of *SystemInit()* function used to configure the microcontroller system at start-up and before branching to the main() program.

Note: For the STM32F3xx, the user can use the clock configuration tool, STM32F3xx_Clock_Configuration.xls, to generate a customized SystemInit() function *depending on the user application requirements. For more information, refer to AN4152 "Clock configuration tool for STM32F30xx microcontrollers" and to AN4132 Clock configuration tool for STM32F37xxx microcontrollers".*

4.2.2 Peripheral access configuration

As presented in *Section 3.4: Reset and clock controller (RCC) interface*, the user needs to call different functions to enable/disable the peripheral clock or enter/exit from reset mode.

As an example, GPIOA is mapped on AHB bus on STM32F3 Series (APB2 bus on STM32F1 Series). To enable its clock in the STM32F1 Series, use

```
_AHBPeriphClockCmd(_AHBPeriph_GPIOA, ENABLE)
```

instead of:

```
_APB2PeriphClockCmd(_APB2Periph_GPIOA, ENABLE);
```

Refer to *Table 7* for the peripheral bus mapping changes between the STM32F3 Series and the STM32F1 Series.

4.2.3 Peripheral clock configuration

Some STM32F3xx peripherals support dual clock features. *Table 18* summarizes the clock sources for these IPs in comparison with STM32F10x peripherals.

Table 18. STM32F10x and STM32F3xx clock source API correspondence

Peripherals	Source clock in STM32F10x	Source clock in STM32F3xx
CEC	APB1 clock with prescaler	– 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	I2C can be clocked with: – System clock – HSI
SPI/I2S	System clock	– APB clock – External clock for I2S clock source(1)
USART	– USART1 can be clocked by PCLK2 (72 MHz Max) – Other USARTs can be clocked by PCLK1 (36 MHz Max)	USART can be clocked with: – system clock – LSE clock – HSI clock – APB clock (PCLK)

1. It is applicable only for STM32F30xxx devices.

4.3 FLASH driver

Table 19 shows the FLASH driver API correspondence between STM32F10x and STM32F3xx Libraries. The user can easily update his application code by replacing STM32F10x functions by the corresponding function in the STM32F3xx library.

The API compatibility has been maintained as much as possible. However legacy define statements are available within the Flash driver in case API or parameter names change.

Table 19. STM32F10x and STM32F3xx FLASH driver API correspondence

	STM32F10x Flash driver API	STM32F3xx 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 NewState);
	void FLASH_HalfCycleAccessCmd(uint32_t FLASH_HalfCycleAccess);	void FLASH_HalfCycleAccessCmd(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_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 19. STM32F10x and STM32F3xx FLASH driver API correspondence (continued)

	STM32F10x Flash driver API	STM32F3xx Flash driver API
Option Byte Programming	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);
	FLASH_Status FLASH_EraseOptionBytes(void);	FLASH_Status FLASH_OB_Erase(void);

Table 19. STM32F10x and STM32F3xx FLASH driver API correspondence (continued)

	STM32F10x Flash driver API	STM32F3xx Flash driver API
Interrupts and flags 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);
	void FLASH_ITConfig(uint32_t FLASH_IT, FunctionalState NewState);	void FLASH_ITConfig(uint32_t FLASH_IT, FunctionalState NewState);
	FlagStatus FLASH_GetPrefetchBufferStatus(void);	NA

Color key:

= New function

= Same function, but API was changed

= Function not available (NA)

4.4 CRC driver

Table 20 shows the CRC driver API correspondence between STM32F10x and STM32F3xx Libraries.

Table 20. STM32F10x and STM32F3xx CRC driver API correspondence

	STM32F10x CRC driver API	STM32F3xx CRC driver API
Configuration	NA	void CRC_DeInit(void);
	void CRC_ResetDR(void);	void CRC_ResetDR(void);
	NA	void CRC_PolynomialSizeSelect(uint32_t CRC_PolSize);
	NA	void CRC_ReverseInputDataSelect(uint32_t CRC_ReverseInputData);
	NA	void CRC_ReverseOutputDataCmd(FunctionalState NewState);
	NA	void CRC_SetPolynomial(uint32_t CRC_Pol);
	NA	void CRC_SetInitRegister(uint32_t CRC_InitValue);

Table 20. STM32F10x and STM32F3xx CRC driver API correspondence (continued)

	STM32F10x CRC driver API	STM32F3xx CRC driver API
Computation	uint32_t CRC_CalcCRC(uint32_t CRC_Data);	uint32_t CRC_CalcCRC(uint32_t CRC_Data);
	NA	uint32_t CRC_CalcCRC16bits(uint16_t CRC_Data);
	NA	uint32_t CRC_CalcCRC8bits(uint8_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);
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);

Color key:

= New function

= Same function, but API was changed

= Function not available (NA)

4.5 GPIO configuration update

This section explains how to update the configuration of the various GPIO modes when porting the application code from STM32F1 Series to STM32F3 Series.

4.5.1 Output mode

The example below shows how to configure an I/O in output mode (for example to drive a LED) in STM32F1 Series:

```
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);
```

In STM32F3 Series, the user has to update this code as follows:

```
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);
```

4.5.2 Input mode

The example below shows how to configure an I/O in input mode (for example to be used as an EXTI line) in STM32F1 Series:

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

In STM32F3 Series, the user has to update this code as follows:

```
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);
```

4.5.3 Analog mode

The example below shows how to configure an I/O in analog mode (for example, an ADC or DAC channel) in STM32F1 Series:

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

In STM32F3 Series, the user has to update this code as follows:

```
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 Alternate function mode

STM32F1 Series

The configuration to use an I/O as an alternate function depends on the peripheral mode used; for example, the USART Tx pin should be configured as an alternate function push-pull while the USART Rx pin should be configured as an input floating or an input pull-up.

To optimize the number of peripheral I/O functions for different device packages, it is possible, by software, to remap some alternate functions to other pins. For example, the USART2_RX pin can be mapped on PA3 (default remap) or PD6 (by software remap).

STM32F3 Series

Whatever the peripheral mode used, the I/O must be configured as an alternate function, then the system can use the I/O in the proper way (input or output).

The I/O pins are connected to on-board peripherals/modules through a multiplexer that allows only one peripheral alternate function to be connected to an I/O pin at a time. In this way, there can be no conflict between peripherals sharing the same I/O pin. Each I/O pin

has a multiplexer with sixteen alternate function inputs (AF0 to AF15) that can be configured through the GPIO_PinAFConfig () function:

- After reset, all I/Os are connected to the system's alternate function 0 (AF0)
- The peripherals' alternate functions are mapped by configuring AF0 to AF15.

In addition to this flexible I/O multiplexing architecture, each peripheral has alternate functions mapped onto different I/O pins to optimize the number of peripheral I/O functions for different device packages; for example, the USART2_RX pin can be mapped on PA3 or PA15 pin.

The configuration procedure is the following:

1. Connect the pin to the desired peripherals' Alternate Function (AF) using GPIO_PinAFConfig() function
2. Use GPIO_Init() function to configure the I/O pin:
   a) Configure the desired pin in alternate function mode using GPIO_InitStructure->GPIO_Mode = GPIO_Mode_AF;
   b) Select the type, pull-up/pull-down and output speed via GPIO_PuPd, GPIO_OType and GPIO_Speed members

The example below shows how to remap USART2 Tx/Rx I/Os on PD5/PD6 pins in STM32F1 Series:

```
  /* Enable APB2 interface clock for GPIOD and AFIO (AFIO peripheral is used
     to configure the I/Os software remapping) */
  _APB2PeriphClockCmd(_APB2Periph_GPIOD | _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;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_Init(GPIOD, &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(GPIOD, &GPIO_InitStructure);
```

In STM32F3 Series, update this code as follows:

```
  /* Enable GPIOA's AHB interface clock */
  _AHBPeriphClockCmd(_AHBPeriph_GPIOA, ENABLE);
/* Select USART2 I/Os mapping on PA14/15 pins
[(USART2_TX,USART2_RX):(PA.14,PA.15)] */
  /* Connect PA14 to USART2_Tx */
  GPIO_PinAFConfig(GPIOA, GPIO_PinSource14, GPIO_AF_2);
  /* Connect PA15 to USART2_Rx*/
  GPIO_PinAFConfig(GPIOA, GPIO_PinSource15, GPIO_AF_2);
/* Configure USART2_Tx and USART2_Rx as alternate function */
  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 Line0

The example below shows how to configure the PA0 pin to be used as EXTI Line0 in STM32F1 Series:

```
/* Enable APB interface clock for GPIOA and AFIO */
_APB2PeriphClockCmd(_APB2Periph_GPIOA | _APB2Periph_AFIO, ENABLE);

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

/* Connect EXTI Line0 to PA0 pin */
GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_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);
```

In STM32F3 Series, the configuration of the EXTI line source pin is performed in the SYSCFG peripheral (instead of AFIO in STM32F1 Series). As a result, the source code should be updated as follows:

```
/* Enable GPIOA's AHB interface clock */
_AHBPeriphClockCmd(_AHBPeriph_GPIOA, ENABLE);
/* Enable SYSCFG's APB interface clock */
_APB2PeriphClockCmd(_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 NVIC interrupt configuration

This section explains how to configure the NVIC interrupts (IRQ).

STM32F1 Series

In the STM32F1 Series, the NVIC supports:

- Up to 68 interrupts (68 ex. Core IT
- A programmable priority level of 0-15 for each interrupt (4 bits of interrupt priority are used). A higher level corresponds to a lower priority; level 0 is the highest interrupt priority.
- Grouping of priority values into group priority and sub priority fields.
- Dynamic changing of priority levels.

The Cortex-M3 exceptions are managed by CMSIS functions:

- Enabling and configuration of the preemption priority and sub priority of the selected IRQ channels according to the Priority grouping configuration.

STM32F3 Series

In the STM32F3 Series, the NVIC supports:

- Up to 66 interrupts
- 16 programmable priority levels.
- The priority level of an interrupt should not be changed after it is enabled.

The Cortex-M4 exceptions are managed by CMSIS functions:

- Enabling and configuration of the priority of the selected IRQ channels. The priority ranges between 0 and 15. Lower priority values give a higher priority.

Table 21 gives the MISC driver API correspondence between STM32F10x and STM32F3xx Libraries.

Table 21. STM32F10x and STM32F3xx MISC driver API correspondence

STM32F10x MISC Driver API	STM32F3xx 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);

Table 21. STM32F10x and STM32F3xx MISC driver API correspondence (continued)

STM32F10x MISC Driver API	STM32F3xx MISC Driver API
NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup);	NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup);
void NVIC_SetVectorTable(uint32_t NVIC_VectTab, uint32_t Offset);	void NVIC_SetVectorTable(uint32_t NVIC_VectTab, uint32_t Offset);

4.8 ADC configuration

The STM32F10x and STM32F37x ADC peripherals are similar. The STM32F30xxx devices feature a new ADC.

Table 22 describes the difference between STM32F10x functions and STM32F3xx libraries.

Table 22. STM32F10x and STM32F3xx ADC driver API correspondence

	STM32F10x ADC driver API	STM32F37x ADC driver API	STM32F30x ADC driver API
Initialization and Configuration	void ADC_DeInit(ADC_TypeDef* ADCx);	void ADC_DeInit(ADC_TypeDef* ADCx);	void ADC_DeInit(ADC_TypeDef* ADCx);
	void ADC_Init(ADC_TypeDef* ADCx, ADC_InitTypeDef* ADC_InitStruct);	void ADC_Init(ADC_TypeDef* ADCx, ADC_InitTypeDef* ADC_InitStruct);	void ADC_Init(ADC_TypeDef* ADCx, ADC_InitTypeDef* ADC_InitStruct);
	void ADC_StructInit(ADC_InitTypeDef * ADC_InitStruct);	void ADC_StructInit(ADC_InitTypeDef* ADC_InitStruct);	void ADC_StructInit(ADC_InitTypeDef* ADC_InitStruct);
	NA	NA	void ADC_InjectedInit(ADC_TypeDef* ADCx, ADC_InjectedInitTypeDef* ADC_InjectedInitStruct);
	NA	NA	void ADC_InjectedStructInit(ADC_InjectedInitTypeDef* ADC_InjectedInitStruct);
	NA	NA	void ADC_CommonInit(ADC_TypeDef* ADCx, ADC_CommonInitTypeDef* ADC_CommonInitStruct);
	NA	NA	void ADC_CommonStructInit(ADC_CommonInitTypeDef* ADC_CommonInitStruct);
	void ADC_Cmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_Cmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_Cmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	void ADC_StartCalibration(ADC_TypeDef* ADCx);	void ADC_StartCalibration(ADC_TypeDef* ADCx);	void ADC_StartCalibration(ADC_TypeDef* ADCx);
	NA	NA	uint32_t ADC_GetCalibrationValue(ADC_TypeDef* ADCx);
	NA	NA	void ADC_SetCalibrationValue(ADC_TypeDef* ADCx, uint32_t ADC_Calibration);

Table 22. STM32F10x and STM32F3xx ADC driver API correspondence (continued)

	STM32F10x ADC driver API	STM32F37x ADC driver API	STM32F30x ADC driver API
Initialization and Configuration	NA	NA	void ADC_SelectCalibrationMode(ADC_TypeDef* ADCx, uint32_t ADC_CalibrationMode);
	FlagStatus ADC_GetCalibrationStatus(ADC_TypeDef* ADCx);	FlagStatus ADC_GetCalibrationStatus(ADC_TypeDef* ADCx);	FlagStatus ADC_GetCalibrationStatus(ADC_TypeDef* ADCx);
	NA	NA	void ADC_DisableCmd(ADC_TypeDef* ADCx);
	NA	NA	FlagStatus ADC_GetDisableCmdStatus(ADC_TypeDef* ADCx);
	NA	NA	void ADC_VoltageRegulatorCmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	NA	NA	void ADC_SelectDifferentialMode(ADC_TypeDef* ADCx, uint8_t ADC_Channel, FunctionalState NewState);
	NA	NA	void ADC_SelectQueueOfContextMode(ADC_TypeDef* ADCx, FunctionalState NewState);
	NA	NA	void ADC_AutoDelayCmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	void ADC_ResetCalibration(ADC_TypeDef* ADCx);	void ADC_ResetCalibration(ADC_TypeDef* ADCx);	NA

Table 22. STM32F10x and STM32F3xx ADC driver API correspondence (continued)

	STM32F10x ADC driver API	STM32F37x ADC driver API	STM32F30x ADC driver API
Analog Watchdog configuration	void ADC_AnalogWatchdogCmd(ADC_TypeDef* ADCx, uint32_t ADC_AnalogWatchdog);	void ADC_AnalogWatchdogCmd(ADC_TypeDef* ADCx, uint32_t ADC_AnalogWatchdog);	void ADC_AnalogWatchdogCmd(ADC_TypeDef* ADCx, uint32_t ADC_AnalogWatchdog);
	void ADC_AnalogWatchdogThresholdsConfig(ADC_TypeDef* ADCx, uint16_t HighThreshold, uint16_t LowThreshold);	void ADC_AnalogWatchdogThresholdsConfig(ADC_TypeDef* ADCx, uint16_t HighThreshold, uint16_t LowThreshold);	void ADC_AnalogWatchdog1ThresholdsConfig(ADC_TypeDef* ADCx, uint16_t HighThreshold, uint16_t LowThreshold);
			void ADC_AnalogWatchdog2ThresholdsConfig(ADC_TypeDef* ADCx, uint8_t HighThreshold, uint8_t LowThreshold);
			void ADC_AnalogWatchdog3ThresholdsConfig(ADC_TypeDef* ADCx, uint8_t HighThreshold, uint8_t LowThreshold);
	void ADC_AnalogWatchdogSingleChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel);	void ADC_AnalogWatchdogSingleChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel);	void ADC_AnalogWatchdog1SingleChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel);
			void ADC_AnalogWatchdog2SingleChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel);
			void ADC_AnalogWatchdog3SingleChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel);

Table 22. STM32F10x and STM32F3xx ADC driver API correspondence (continued)

	STM32F10x ADC driver API	STM32F37x ADC driver API	STM32F30x ADC driver API
Temperature Sensor, Vrefint and VBAT management	void ADC_TempSensorVrefintCmd(FunctionalState NewState);	void ADC_TempSensorVrefintCmd(FunctionalState NewState);	void ADC_TempSensorCmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	NA	NA	void ADC_VrefintCmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	NA	NA	void ADC_VbatCmd(ADC_TypeDef* ADCx, FunctionalState NewState);

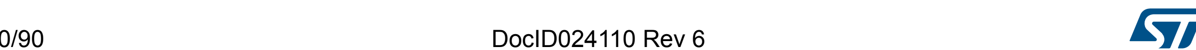


Table 22. STM32F10x and STM32F3xx ADC driver API correspondence (continued)

	STM32F10x ADC driver API	STM32F37x ADC driver API	STM32F30x ADC driver API
Regular Channels	void ADC_RegularChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime);	void ADC_RegularChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime);	void ADC_RegularChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime);
	NA	NA	void ADC_RegularChannelSequencerLengthConfig(ADC_TypeDef* ADCx, uint8_t SequencerLength);
	void ADC_ExternalTrigConvCmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_ExternalTrigConvCmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_ExternalTriggerConfig(ADC_TypeDef* ADCx, uint16_t ADC_ExternalTrigConvEvent, uint16_t ADC_ExternalTrigEventEdge);
	void ADC_SoftwareStartConv(ADC_TypeDef* ADCx);	void ADC_SoftwareStartConv(ADC_TypeDef* ADCx);	void ADC_StartConversion(ADC_TypeDef* ADCx);
	FlagStatus ADC_GetSoftwareStartConvStatus(ADC_TypeDef* ADCx);	FlagStatus ADC_GetSoftwareStartConvStatus(ADC_TypeDef* ADCx);	FlagStatus ADC_GetStartConversionStatus(ADC_TypeDef* ADCx);
	NA	NA	void ADC_StopConversion(ADC_TypeDef* ADCx);
	void ADC_DiscModeChannelCountConfig(ADC_TypeDef* ADCx, uint8_t Number);	void ADC_DiscModeChannelCountConfig(ADC_TypeDef* ADCx, uint8_t Number);	void ADC_DiscModeChannelCountConfig(ADC_TypeDef* ADCx, uint8_t Number);

Table 22. STM32F10x and STM32F3xx ADC driver API correspondence (continued)

	STM32F10x ADC driver API	STM32F37x ADC driver API	STM32F30x ADC driver API
Regular Channels	void ADC_DiscModeCmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_DiscModeCmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_DiscModeCmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	uint16_t ADC_GetConversionValue(ADC_TypeDef* ADCx);	uint16_t ADC_GetConversionValue(ADC_TypeDef* ADCx);	uint16_t ADC_GetConversionValue(ADC_TypeDef* ADCx);
	uint32_t ADC_GetDualModeConversionValue(void); |	Not supported by STM32F37xxx	uint32_t ADC_GetDualModeConversionValue(ADC_TypeDef* ADCx);
	NA	NA	void ADC_SetChannelOffset1(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint16_t Offset);
	NA	NA	void ADC_SetChannelOffset2(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint16_t Offset);
	NA	NA	void ADC_SetChannelOffset3(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint16_t Offset);
	NA	NA	void ADC_SetChannelOffset4(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint16_t Offset);
	NA	NA	void ADC_ChannelOffset1Cmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	NA	NA	void ADC_ChannelOffset2Cmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	NA	NA	void ADC_ChannelOffset3Cmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	NA	NA	void ADC_ChannelOffset4Cmd(ADC_TypeDef* ADCx, FunctionalState NewState);

Table 22. STM32F10x and STM32F3xx ADC driver API correspondence (continued)

	STM32F10x ADC driver API	STM32F37x ADC driver API	STM32F30x ADC driver API
Regular Channels DMA configuration	void ADC_DMACmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_DMACmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_DMACmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	NA	NA	void ADC_DMAConfig(ADC_TypeDef* ADCx, uint32_t ADC_DMAMode);

Table 22. STM32F10x and STM32F3xx ADC driver API correspondence (continued)

	STM32F10x ADC driver API	STM32F37x ADC driver API	STM32F30x ADC driver API
Injected channels Configuration	void ADC_InjectedChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime);	void ADC_InjectedChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime);	NA
	void ADC_InjectedSequencerLengthConfig(ADC_TypeDef* ADCx, uint8_t SequencerLength);	void ADC_InjectedSequencerLengthConfig(ADC_TypeDef* ADCx, uint8_t SequencerLength);	NA
	void ADC_ExternalTrigInjectedConvConfig(ADC_TypeDef* ADCx, uint32_t ADC_ExternalTrigInjecConv);	void ADC_ExternalTrigInjectedConvConfig(ADC_TypeDef* ADCx, uint32_t ADC_ExternalTrigInjecConv);	NA
	void ADC_SoftwareStartInjectedConvCmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_SoftwareStartInjectedConvCmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_StartInjectedConversion(ADC_TypeDef* ADCx);
	FlagStatus ADC_GetSoftwareStartInjectedConvCmdStatus(ADC_TypeDef* ADCx);	FlagStatus ADC_GetSoftwareStartInjectedConvCmdStatus(ADC_TypeDef* ADCx);	FlagStatus ADC_GetStartInjectedConversionStatus(ADC_TypeDef* ADCx);
	ADC_InjectedChannelSampleTimeConfig(ADC_TypeDef* ADCx, uint8_t ADC_InjectedChannel, uint8_t ADC_SampleTime);	NA	ADC_InjectedChannelSampleTimeConfig(ADC_TypeDef* ADCx, uint8_t ADC_InjectedChannel, uint8_t ADC_SampleTime);
	NA	NA	void ADC_StopInjectedConversion(ADC_TypeDef* ADCx);
	ADC_AutoInjectedConvCmd(ADC_TypeDef* ADCx, FunctionalState NewState);	ADC_AutoInjectedConvCmd(ADC_TypeDef* ADCx, FunctionalState NewState);	ADC_AutoInjectedConvCmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	void ADC_InjectedDiscModeCmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_InjectedDiscModeCmd(ADC_TypeDef* ADCx, FunctionalState NewState);	void ADC_InjectedDiscModeCmd(ADC_TypeDef* ADCx, FunctionalState NewState);
	uint16_t ADC_GetInjectedConversionValue(ADC_TypeDef* ADCx, uint8_t ADC_InjectedChannel);	uint16_t ADC_GetInjectedConversionValue(ADC_TypeDef* ADCx, uint8_t ADC_InjectedChannel);	uint16_t ADC_GetInjectedConversionValue(ADC_TypeDef* ADCx, uint8_t ADC_InjectedChannel);

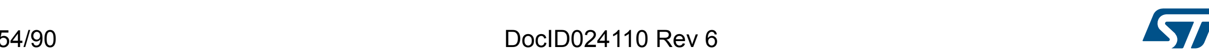


Table 22. STM32F10x and STM32F3xx ADC driver API correspondence (continued)

	STM32F10x ADC driver API	STM32F37x ADC driver API	STM32F30x ADC driver API
ADC Dual mode configuration	NA	NA	FlagStatus ADC_GetCommonFlagStatus(ADC_TypeDef* ADCx, uint32_t ADC_FLAG);
	NA	NA	void ADC_ClearCommonFlag(ADC_TypeDef* ADCx, uint32_t ADC_FLAG);
Interrupts and flags management	void ADC_ITConfig(ADC_TypeDef* ADCx, uint32_t ADC_IT, FunctionalState NewState);	void ADC_ITConfig(ADC_TypeDef* ADCx, uint32_t ADC_IT, FunctionalState NewState);	void ADC_ITConfig(ADC_TypeDef* ADCx, uint32_t ADC_IT, FunctionalState NewState);
	FlagStatus ADC_GetFlagStatus(ADC_TypeDef* ADCx, uint8_t ADC_FLAG);	FlagStatus ADC_GetFlagStatus(ADC_TypeDef* ADCx, uint8_t ADC_FLAG);	FlagStatus ADC_GetFlagStatus(ADC_TypeDef* ADCx, uint8_t ADC_FLAG);
	void ADC_ClearFlag(ADC_TypeDef* ADCx, uint8_t ADC_FLAG);	void ADC_ClearFlag(ADC_TypeDef* ADCx, uint8_t ADC_FLAG);	void ADC_ClearFlag(ADC_TypeDef* ADCx, uint8_t ADC_FLAG);
	ITStatus ADC_GetITStatus(ADC_TypeDef* ADCx, uint16_t ADC_IT);	ITStatus ADC_GetITStatus(ADC_TypeDef* ADCx, uint16_t ADC_IT);	ITStatus ADC_GetITStatus(ADC_TypeDef* ADCx, uint16_t ADC_IT);
	void ADC_ClearITPendingBit(ADC_TypeDef* ADCx, uint16_t ADC_IT);	void ADC_ClearITPendingBit(ADC_TypeDef* ADCx, uint16_t ADC_IT);	void ADC_ClearITPendingBit(ADC_TypeDef* ADCx, uint16_t ADC_IT);
	FlagStatus ADC_GetCalibrationStatus(ADC_TypeDef* ADCx);	FlagStatus ADC_GetCalibrationStatus(ADC_TypeDef* ADCx);	NA
	FlagStatus ADC_GetResetCalibrationStatus(ADC_TypeDef* ADCx);	FlagStatus ADC_GetResetCalibrationStatus(ADC_TypeDef* ADCx);	NA

Color key:

= New function

= Same function, but API was changed

= Function not available (NA)

This section gives an example of how to port existing code from STM32F1 Series to STM32F3 Series.

The example below shows how to configure the ADC1 to convert continuously channel 14 in STM32F1 Series:

```
  /* ADCCLK = PCLK2/4 */
  _ADCCLKConfig(_PCLK2_Div4);

  /* Enable ADC's APB interface clock */
  _APB2PeriphClockCmd(_APB2Periph_ADC1, ENABLE);

  /* Configure ADC1 to convert continously 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);
  ...
```

In F30 Series, the user has to update this code as follows:

```
...
/* Configure the ADC clock */
```

```
  RCC_ADCCLKConfig(RCC_ADC12PLLCLK_Div2);

  /* Enable ADC1 clock */
  RCC_AHBPeriphClockCmd(RCC_AHBPeriph_ADC12, ENABLE);
ADC_StructInit(&ADC_InitStructure);

  /* Calibration procedure */
  ADC_VoltageRegulatorCmd(ADC1, ENABLE);

  ADC_SelectCalibrationMode(ADC1, ADC_CalibrationMode_Single);
  ADC_StartCalibration(ADC1);

  while(ADC_GetCalibrationStatus(ADC1) != RESET );
  calibration_value = ADC_GetCalibrationValue(ADC1);

 ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent;
  ADC_CommonInitStructure.ADC_Clock = ADC_Clock_AsynClkMode;
  ADC_CommonInitStructure.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;
  ADC_CommonInitStructure.ADC_DMAMode = ADC_DMAMode_OneShot;
  ADC_CommonInitStructure.ADC_TwoSamplingDelay = 0;

  ADC_CommonInit(ADC1, &ADC_CommonInitStructure);

  ADC_InitStructure.ADC_ContinuousConvMode = ADC_ContinuousConvMode_Enable;
  ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;
  ADC_InitStructure.ADC_ExternalTrigConvEvent =
ADC_ExternalTrigConvEvent_0;
  ADC_InitStructure.ADC_ExternalTrigEventEdge =
ADC_ExternalTrigEventEdge_None;
  ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
  ADC_InitStructure.ADC_OverrunMode = ADC_OverrunMode_Disable;
  ADC_InitStructure.ADC_AutoInjMode = ADC_AutoInjec_Disable;
  ADC_InitStructure.ADC_NbrOfRegChannel = 1;
  ADC_Init(ADC1, &ADC_InitStructure);

  /* ADC1 regular channel14 configuration */
  ADC_RegularChannelConfig(ADC1, ADC_Channel_14, 1,
ADC_SampleTime_7Cycles5);

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

  /* wait for ADRDY */
  while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_RDY));

  /* Start ADC1 Software Conversion */
```

```
ADC_StartConversion(ADC1);

/* Infinite loop */
while (1)
{
  /* Test EOC flag */
  while(ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);

  /* Get ADC1 converted data */
  ADC1ConvertedValue =ADC_GetConversionValue(ADC1);
}
...
```

4.9 DAC driver

Table 23 describes the difference between STM32F10x functions and STM32F3xx libraries.

Table 23. STM32F10x and STM32F3xx DAC driver API correspondence

	STM32F10x DAC driver API	STM32F3xx DAC driver API
Configuration	void DAC_DeInit(void);	void DAC_DeInit(DAC_TypeDef* DACx);
	void DAC_Init(uint32_t DAC_Channel, DAC_InitTypeDef* DAC_InitStruct);	void DAC_Init(DAC_TypeDef* DACx, 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(DAC_TypeDef* DACx, uint32_t DAC_Channel, FunctionalState NewState);
	void DAC_SoftwareTriggerCmd(uint32_t DAC_Channel, FunctionalState NewState);	void DAC_SoftwareTriggerCmd(DAC_TypeDef* DACx, uint32_t DAC_Channel, FunctionalState NewState);
	DAC_DualSoftwareTriggerCmd(FunctionalState NewState);	void DAC_DualSoftwareTriggerCmd(DAC_TypeDef* DACx, FunctionalState NewState);
	void DAC_WaveGenerationCmd(uint32_t DAC_Channel, uint32_t DAC_Wave, FunctionalState NewState);	void DAC_WaveGenerationCmd(DAC_TypeDef* DACx, uint32_t DAC_Channel, uint32_t DAC_Wave, FunctionalState NewState);
	void DAC_SetChannel1Data(uint32_t DAC_Align, uint16_t Data);	void DAC_SetChannel1Data(DAC_TypeDef* DACx, uint32_t DAC_Align, uint16_t Data);
	void DAC_SetChannel2Data(uint32_t DAC_Align, uint16_t Data);	void DAC_SetChannel2Data(DAC_TypeDef* DACx, uint32_t DAC_Align, uint16_t Data);
	void DAC_SetDualChannelData(uint32_t DAC_Align, uint16_t Data2, uint16_t Data1);	void DAC_SetDualChannelData(DAC_TypeDef* DACx, uint32_t DAC_Align, uint16_t Data2, uint16_t Data1);
	uint16_t DAC_GetDataOutputValue(uint32_t DAC_Channel);	uint16_t DAC_GetDataOutputValue(DAC_TypeDef* DACx, uint32_t DAC_Channel);
	NA	void DAC_DualSoftwareTriggerCmd(DAC_TypeDef* DACx, FunctionalState NewState)
	NA	void DAC_WaveGenerationCmd(DAC_TypeDef* DACx, uint32_t DAC_Channel, uint32_t DAC_Wave, FunctionalState NewState);
DMA management	void DAC_DMACmd(uint32_t DAC_Channel, FunctionalState NewState);	void DAC_DMACmd(DAC_TypeDef* DACx, uint32_t DAC_Channel, FunctionalState NewState);

Table 23. STM32F10x and STM32F3xx DAC driver API correspondence (continued)

	STM32F10x DAC driver API	STM32F3xx DAC driver API
Interrupts and flags management	void DAC_ITConfig(uint32_t DAC_Channel, uint32_t DAC_IT, FunctionalState NewState);(1)	void DAC_ITConfig(DAC_TypeDef* DACx, uint32_t DAC_Channel, uint32_t DAC_IT, FunctionalState NewState);
	FlagStatus DAC_GetFlagStatus(uint32_t DAC_Channel, uint32_t DAC_FLAG);(1)	FlagStatus DAC_GetFlagStatus(DAC_TypeDef* DACx, uint32_t DAC_Channel, uint32_t DAC_FLAG);
	void DAC_ClearFlag(uint32_t DAC_Channel, uint32_t DAC_FLAG);(1)	void DAC_ClearFlag(DAC_TypeDef* DACx, uint32_t DAC_Channel, uint32_t DAC_FLAG);
	ITStatus DAC_GetITStatus(uint32_t DAC_Channel, uint32_t DAC_IT);(1)	ITStatus DAC_GetITStatus(DAC_TypeDef* DACx, uint32_t DAC_Channel, uint32_t DAC_IT);
	void DAC_ClearITPendingBit(uint32_t DAC_Channel, uint32_t DAC_IT);(1)	void DAC_ClearITPendingBit(DAC_TypeDef* DACx, uint32_t DAC_Channel, uint32_t DAC_IT);

Color key:

= New function

= Same function, but API was changed

= Function not available (NA)

1. These functions exist only on STM32F10X_LD_VL, STM32F10X_MD_VL, and STM32F10X_HD_VL devices.

4.10 PWR driver

Table 24 presents the PWR driver API correspondence between STM32F10x and STM32F3xx Libraries. The user can easily update the application code by replacing STM32F10x functions by the corresponding function in the STM32F3xx library.

Table 24. STM32F10x and STM32F3xx PWR driver API correspondence

	STM32F10x PWR driver API	STM32F3xx PWR driver API
Interface configuration	void PWR_DeInit(void);	void PWR_DeInit(void);
	void PWR_BackupAccessCmd(FunctionalState NewState);	void PWR_BackupAccessCmd(FunctionalState NewState);

Table 24. STM32F10x and STM32F3xx PWR driver API correspondence (continued)

	STM32F10x PWR driver API	STM32F3xx PWR driver API
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);(1)
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);
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);

Color key:
- = New function
- = Same function, but API was changed
- = Function not available (NA)

1. Additional Wake-up pins are available on STM32F3 Series.

4.11 Backup data registers

In STM32F1 Series, the Backup data registers are managed through the BKP peripheral, while in STM32F3 Series they are a part of the RTC peripheral (there is no BKP peripheral).

The example below shows how to write to/read from Backup data registers in STM32F1 Series:

```
uint16_t BKPdata = 0;
...
  /* Enable APB2 interface clock for PWR and BKP */
  _APB1PeriphClockCmd(_APB1Periph_PWR | _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);
```

In STM32F3 Series, the user has to update this code as follows:

```
uint16_t BKPdata = 0;

  ...
  /* PWR Clock Enable */
  _APB1PeriphClockCmd(_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);
```

The main changes in the source code in STM32F3 Series versus STM32F1 Series are described below:

- There is no BKP peripheral
- Write to/read from Backup data registers are performed through the RTC driver
- Backup data registers naming changed from BKP_DRx to RTC_BKP_DRx, and numbering starts from 0 instead of 1.

4.12 CEC application code

The user can easily update the user CEC application code by replacing STM32F10x functions by the corresponding function of the STM32F3xx library.

Table 25 presents the CEC driver API correspondence between STM32F10x and STM32F37x Libraries.

Table 25. STM32F10x and STM32F37x CEC driver API correspondence

	STM32F10x CEC driver API	STM32F37x 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);
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);

Color key:

= New function

= Same function, but API was changed

= Function not available (NA)

The main changes in the source code/procedure in STM32F3 Series versus STM32F1 Series are described below:

- Dual Clock Source (Refer to section for more details).
- No Prescaler feature configuration.
- It supports more than one address (multiple addressing).
- Each event flag has an associate enable control bit to generate the adequate interrupt.
- In the CEC structure definition, seven fields should be initialized.

  The example below shows how to configure the CEC STM32F1 Series:

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

In STM32F37xxx devices, the user have to update this code as follows:

```
/* 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_Off;
  CEC_InitStructure.CEC_BRDNoGen = CEC_BRDNoGen_Off;
  CEC_InitStructure.CEC_SFTOption = CEC_SFTOption_Off;
  CEC_Init(&CEC_InitStructure);
```

4.13 I2C driver

The STM32F3 Series devices incorporate new I2C features. *Table 26* presents the I2C driver API correspondence between STM32F10x and STM32F3xx Libraries. Update the user application code by replacing STM32F10x functions by the corresponding functions of the STM32F3xx library.

Table 26. STM32F10x and STM32F3xx I2C driver API correspondence

	STM32F10x I2C Driver API	STM32F3xx 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 26. STM32F10x and STM32F3xx I2C driver API correspondence (continued)

	STM32F10x I2C Driver API	STM32F3xx 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 I2C_Direction)	NA

Table 26. STM32F10x and STM32F3xx I2C driver API correspondence (continued)

	STM32F10x I2C Driver API	STM32F3xx 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_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

Table 26. STM32F10x and STM32F3xx I2C driver API correspondence (continued)

	STM32F10x I2C Driver API	STM32F3xx 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);

Color key:

= New function

= Same function, but API was changed

= Function not available (NA)

Though some API functions are identical in STM32F1 Series and STM32F3 Series devices, in most cases the application code needs to be rewritten when migrating from STM32F1 Series to STM32F3 Series. However, STMicroelectronics provides an “I2C Communication peripheral application library (CPAL)”, which allows to move seamlessly from STM32F1 Series to STM32F3 Series: the user needs to modify only few settings without any changes on the application code. For more details about STM32F1 I2C CPAL, refer to UM1029. For STM32F3 Series, the I2C CPAL is provided within the Standard Peripherals Library package.

4.14 SPI driver

The STM32F3xx SPI includes some new features as compared with STM32F10x SPI. *Table 27* presents the SPI driver API correspondence between STM32F10x and STM32F3xx Libraries.

Table 27. STM32F10x and STM32F3xx SPI driver API correspondence

	STM32F10x SPI driver API	STM32F3xx 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);
	NA	void I2S_FullDuplexConfig(SPI_TypeDef* I2Sxext, I2S_InitTypeDef* I2S_InitStruct);(1)
	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 27. STM32F10x and STM32F3xx SPI driver API correspondence (continued)

	STM32F10x SPI driver API	STM32F3xx 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_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);

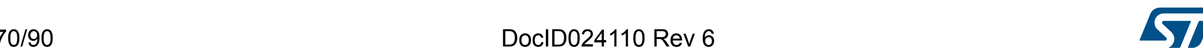


Table 27. STM32F10x and STM32F3xx SPI driver API correspondence (continued)

	STM32F10x SPI driver API	STM32F3xx SPI driver API
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);(2)
	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);(2)
	ITStatus SPI_I2S_GetITStatus(SPI_TypeDef* SPIx, uint8_t SPI_I2S_IT);	ITStatus SPI_I2S_GetITStatus(SPI_TypeDef* SPIx, uint8_t SPI_I2S_IT);(2)

Color key:

= New function

= Same function, but API was changed

= Function not available (NA)

1. It is applicable only for STM32F30xxx devices.
2. One more flag in STM32F3xx (TI frame format error) can generate an event in comparison with STM32F10x driver API.

4.15 USART driver

The STM32F3xx USART includes enhancements in comparison with STM32F10x USART. *Table 28* presents the USART driver API correspondence between STM32F10x and STM32F3xx Libraries.

Table 28. STM32F10x and STM32F3xx USART driver API correspondence

	STM32F10x USART driver API	STM32F3xx 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 28. STM32F10x and STM32F3xx USART driver API correspondence (continued)

	STM32F10x USART driver API	STM32F3xx 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);

Table 28. STM32F10x and STM32F3xx USART driver API correspondence (continued)

	STM32F10x USART driver API	STM32F3xx USART driver API
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);
Smart Card mode	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 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);

Table 28. STM32F10x and STM32F3xx USART driver API correspondence (continued)

	STM32F10x USART driver API	STM32F3xx USART driver API
DMA transfers	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);
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:

= New function

= Same function, but API was changed

= Function not available (NA)

4.16 IWDG driver

The existing IWDG available on the STM32F10x and STM32F3xx devices have the same specifications, with window capability additional feature in STM32F3 Series which detect over frequency on external oscillators. *Table 29* lists the IWDG driver APIs.

Table 29. STM32F10x and STM32F3xx IWDG driver API correspondence

	STM32F10x IWDG driver API	STM32F3xx 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);

Table 29. STM32F10x and STM32F3xx IWDG driver API correspondence (continued)

	STM32F10x IWDG driver API	STM32F3xx IWDG driver API
Flag management	FlagStatus IWDG_GetFlagStatus(uint16_t IWDG_FLAG);	FlagStatus IWDG_GetFlagStatus(uint16_t IWDG_FLAG);

Color key:
- = New function
- = Same function, but API was changed
- = Function not available (NA)

4.17 FMC driver

The STM32F303xD/E devices provide a Flexible Memory Controller (FMC) supporting asynchronous and synchronous Memories (SRAM, PSRAM, NOR and NAND). Existing FSMC available on STM32F10x and FMC available on the STM32F303xD/E devices have the same specifications.

Table 30 shows the FSMC/FMC driver API correspondence between STM32F10x and STMF3xx libraries.

Table 30. STM32F10x and STM32F303xD/E FMC driver API correspondence

	STM32F10x FSMC driver API	STM32F303xD/E FMC driver API
NOR/SRAM Controller functions	void FSMC_NORSRAMDeInit(uint32_t FSMC_Bank);	void FMC_NORSRAMDeInit(uint32_t FMC_Bank);
	void FSMC_NORSRAMInit(FSMC_NORSRAMInitTypeDef* FSMC_NORSRAMInitStruct);	void FMC_NORSRAMInit(FMC_NORSRAMInitTypeDef* FMC_NORSRAMInitStruct);
	void FSMC_NORSRAMStructInit(FSMC_NORSRAMInitTypeDef* FSMC_NORSRAMInitStruct);	void FMC_NORSRAMStructInit(FMC_NORSRAMInitTypeDef* FMC_NORSRAMInitStruct);
	void FSMC_NORSRAMCmd(uint32_t FSMC_Bank, FunctionalState NewState);	void FMC_NORSRAMCmd(uint32_t FMC_Bank, FunctionalState NewState);

Table 30. STM32F10x and STM32F303xD/E FMC driver API correspondence (continued)

	STM32F10x FSMC driver API	STM32F303xD/E FMC driver API
NAND Controller functions	void FSMC_NANDDeInit(uint32_t FSMC_Bank);	void FMC_NANDDeInit(uint32_t FMC_Bank);
	void FSMC_NANDInit(FSMC_NANDInitTypeDef* FSMC_NANDInitStruct);	void FMC_NANDInit(FMC_NANDInitTypeDef* FMC_NANDInitStruct);
	void FSMC_NANDStructInit(FSMC_NANDInitTypeDef* FSMC_NANDInitStruct);	void FMC_NANDStructInit(FMC_NANDInitTypeDef* FMC_NANDInitStruct);
	void FSMC_NANDCmd(uint32_t FSMC_Bank, FunctionalState NewState);	void FMC_NANDCmd(uint32_t FMC_Bank, FunctionalState NewState);
	void FSMC_NANDECCCmd(uint32_t FSMC_Bank, FunctionalState NewState);	void FMC_NANDECCCmd(uint32_t FMC_Bank, FunctionalState NewState);
	uint32_t FSMC_GetECC(uint32_t FSMC_Bank);	uint32_t FMC_GetECC(uint32_t FMC_Bank);
PCCARD Controller functions	void FSMC_PCCARDDeInit(void);	void FMC_PCCARDDeInit(void);
	void FSMC_PCCARDInit(FSMC_PCCARDInitTypeDef* FSMC_PCCARDInitStruct);	void FMC_PCCARDInit(FMC_PCCARDInitTypeDef* FMC_PCCARDInitStruct);
	void FSMC_PCCARDStructInit(FSMC_PCCARDInitTypeDef* FSMC_PCCARDInitStruct);	void FMC_PCCARDStructInit(FMC_PCCARDInitTypeDef* FMC_PCCARDInitStruct);
	void FSMC_PCCARDCmd(FunctionalState NewState);	void FMC_PCCARDCmd(FunctionalState NewState);

Table 30. STM32F10x and STM32F303xD/E FMC driver API correspondence (continued)

	STM32F10x FSMC driver API	STM32F303xD/E FMC driver API
Interrupts and flags management functions	void FSMC_ITConfig(uint32_t FMC_Bank,uint32_t FMC_IT,FunctionalState NewState);	void FMC_ITConfig(uint32_t FMC_Bank uint32_t FMC_IT, FunctionalState NewState);
	FlagStatus FSMC_GetFlagStatus(uint32_t FSMC_Bank,uint32_t FSMC_FLAG);	FlagStatus FMC_GetFlagStatus(uint32_t FMC_Bank,uint32_t FMC_FLAG);
	void FSMC_ClearFlag(uint32_t FSMC_Bank,uint32_t FSMC_FLAG);	void FMC_ClearFlag(uint32_t FMC_Bank,uint32_t FMC_FLAG);
	ITStatus FSMC_GetITStatus(uint32_t FSMC_Bank,uint32_t FSMC_IT);	ITStatus FMC_GetITStatus(uint32_t FMC_Bank uint32_t FMC_IT);
	void FSMC_ClearITPendingBit(uint32_t FSMC_Bank,uint32_t FSMC_IT);	void FMC_ClearITPendingBit(uint32_t FMC_Bank,uint32_t FMC_IT);

4.18 TIM driver

The existing TIM available on the STM32F10x and STM32F3xx devices have the same specifications, with additional features in the STM32F3 Series. *Table 31* lists the TIM driver APIs.

Table 31. STM32F10x and STM32F3xx TIM driver API correspondence

	STM32F10x TIM driver API	STM32F3xx TIM driver API
Time Base management	void TIM_DeInit(TIM_TypeDef* TIMx);	void TIM_DeInit(TIM_TypeDef* TIMx);
	void TIM_TimeBaseInit(TIM_TypeDef* TIMx, TIM_TimeBaseInitTypeDef* TIM_TimeBaseInitStruct);	void TIM_TimeBaseInit(TIM_TypeDef* TIMx, TIM_TimeBaseInitTypeDef* TIM_TimeBaseInitStruct);
	void TIM_TimeBaseStructInit(TIM_TimeBaseInitTypeDef* TIM_TimeBaseInitStruct);	void TIM_TimeBaseStructInit(TIM_TimeBaseInitTypeDef* TIM_TimeBaseInitStruct);
	void TIM_PrescalerConfig(TIM_TypeDef* TIMx, uint16_t Prescaler, uint16_t TIM_PSCReloadMode);	void TIM_PrescalerConfig(TIM_TypeDef* TIMx, uint16_t Prescaler, uint16_t TIM_PSCReloadMode);
	void TIM_CounterModeConfig(TIM_TypeDef* TIMx, uint16_t TIM_CounterMode);	void TIM_CounterModeConfig(TIM_TypeDef* TIMx, uint16_t TIM_CounterMode);
	void TIM_SetCounter(TIM_TypeDef* TIMx, uint16_t Counter);	void TIM_SetCounter(TIM_TypeDef* TIMx, uint16_t Counter);
	void TIM_SetAutoreload(TIM_TypeDef* TIMx, uint16_t Autoreload);	void TIM_SetAutoreload(TIM_TypeDef* TIMx, uint16_t Autoreload);
	uint16_t TIM_GetCounter(TIM_TypeDef* TIMx);	uint16_t TIM_GetCounter(TIM_TypeDef* TIMx);
	uint16_t TIM_GetPrescaler(TIM_TypeDef* TIMx);	uint16_t TIM_GetPrescaler(TIM_TypeDef* TIMx);
	void TIM_UpdateDisableConfig(TIM_TypeDef* TIMx, FunctionalState NewState);	void TIM_UpdateDisableConfig(TIM_TypeDef* TIMx, FunctionalState NewState);
	void TIM_UpdateRequestConfig(TIM_TypeDef* TIMx, uint16_t TIM_UpdateSource);	void TIM_UpdateRequestConfig(TIM_TypeDef* TIMx, uint16_t TIM_UpdateSource);
	NA	void TIM_UIFRemap(TIM_TypeDef* TIMx, FunctionalState NewState);(1)
	void TIM_ARRPreloadConfig(TIM_TypeDef* TIMx, FunctionalState NewState);	void TIM_ARRPreloadConfig(TIM_TypeDef* TIMx, FunctionalState NewState);
	void TIM_SelectOnePulseMode(TIM_TypeDef* TIMx, uint16_t TIM_OPMode);	void TIM_SelectOnePulseMode(TIM_TypeDef* TIMx, uint16_t TIM_OPMode);
	void TIM_SetClockDivision(TIM_TypeDef* TIMx, uint16_t TIM_CKD);	void TIM_SetClockDivision(TIM_TypeDef* TIMx, uint16_t TIM_CKD);
	void TIM_Cmd(TIM_TypeDef* TIMx, FunctionalState NewState);	void TIM_Cmd(TIM_TypeDef* TIMx, FunctionalState NewState);

Table 31. STM32F10x and STM32F3xx TIM driver API correspondence (continued)

	STM32F10x TIM driver API	STM32F3xx TIM driver API
Output Compare management	void TIM_OC1Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);	void TIM_OC1Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);
	void TIM_OC2Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);	void TIM_OC2Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);
	void TIM_OC3Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);	void TIM_OC3Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);
	void TIM_OC4Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);	void TIM_OC4Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);
	NA	void TIM_OC5Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);(1)
	NA	void TIM_OC6Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct);(1)
	NA	void TIM_SelectGC5C1(TIM_TypeDef* TIMx, FunctionalState NewState);(1)
	NA	void TIM_SelectGC5C2(TIM_TypeDef* TIMx, FunctionalState NewState);(1)
	NA	void TIM_SelectGC5C3(TIM_TypeDef* TIMx, FunctionalState NewState);(1)
	void TIM_OCStructInit(TIM_OCInitTypeDef* TIM_OCInitStruct);	void TIM_OCStructInit(TIM_OCInitTypeDef* TIM_OCInitStruct);
	NA	void TIM_SelectOCxM(TIM_TypeDef* TIMx, uint16_t TIM_Channel, uint32_t TIM_OCMode);
	void TIM_SetCompare1(TIM_TypeDef* TIMx, uint32_t Compare1);	void TIM_SetCompare1(TIM_TypeDef* TIMx, uint32_t Compare1);
	void TIM_SetCompare2(TIM_TypeDef* TIMx, uint32_t Compare2);	void TIM_SetCompare2(TIM_TypeDef* TIMx, uint32_t Compare2);
	void TIM_SetCompare3(TIM_TypeDef* TIMx, uint32_t Compare3);	void TIM_SetCompare3(TIM_TypeDef* TIMx, uint32_t Compare3);
	void TIM_SetCompare4(TIM_TypeDef* TIMx, uint32_t Compare4);	void TIM_SetCompare4(TIM_TypeDef* TIMx, uint32_t Compare4);
	NA	void TIM_SetCompare5(TIM_TypeDef* TIMx, uint32_t Compare5);(1)
	NA	void TIM_SetCompare6(TIM_TypeDef* TIMx, uint32_t Compare6);(1)
	void TIM_ForcedOC1Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);	void TIM_ForcedOC1Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);
	void TIM_ForcedOC2Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);	void TIM_ForcedOC2Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);
	void TIM_ForcedOC3Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);	void TIM_ForcedOC3Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);
	void TIM_ForcedOC4Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);	void TIM_ForcedOC4Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);

Table 31. STM32F10x and STM32F3xx TIM driver API correspondence (continued)

	STM32F10x TIM driver API	STM32F3xx TIM driver API
Output Compare management	NA	void TIM_ForcedOC5Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);(1)
	NA	void TIM_ForcedOC6Config(TIM_TypeDef* TIMx, uint16_t TIM_ForcedAction);(1)
	void TIM_OC1PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);	void TIM_OC1PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
	void TIM_OC2PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);	void TIM_OC2PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
	void TIM_OC3PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);	void TIM_OC3PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
	void TIM_OC4PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);	void TIM_OC4PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);
	NA	void TIM_OC5PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);(1)
	NA	void TIM_OC6PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload);(1)
	void TIM_OC1FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);	void TIM_OC1FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
	void TIM_OC2FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);	void TIM_OC2FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
	void TIM_OC3FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);	void TIM_OC3FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
	void TIM_OC4FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);	void TIM_OC4FastConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCFast);
	void TIM_ClearOC1Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);	void TIM_ClearOC1Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
	void TIM_ClearOC2Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);	void TIM_ClearOC2Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
	void TIM_ClearOC3Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);	void TIM_ClearOC3Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
	void TIM_ClearOC4Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);	void TIM_ClearOC4Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);
	NA	void TIM_ClearOC5Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);(1)
	NA	void TIM_ClearOC6Ref(TIM_TypeDef* TIMx, uint16_t TIM_OCClear);(1)
	NA	void TIM_SelectOCREFClear(TIM_TypeDef* TIMx, uint16_t TIM_OCReferenceClear);(1)
	void TIM_OC1PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);	void TIM_OC1PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);
	void TIM_OC1NPolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCNPolarity);	void TIM_OC1NPolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCNPolarity);

Table 31. STM32F10x and STM32F3xx TIM driver API correspondence (continued)

	STM32F10x TIM driver API	STM32F3xx TIM driver API
Output Compare management	void TIM_OC2PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);	void TIM_OC2PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);
	void TIM_OC2NPolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCNPolarity);	void TIM_OC2NPolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCNPolarity);
	void TIM_OC3PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);	void TIM_OC3PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);
	void TIM_OC3NPolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCNPolarity);	void TIM_OC3NPolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCNPolarity);
	void TIM_OC4PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);	void TIM_OC4PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);
	NA	void TIM_OC5PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);
	NA	void TIM_OC6PolarityConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPolarity);
	void TIM_CCxCmd(TIM_TypeDef* TIMx, uint16_t TIM_Channel, uint16_t TIM_CCx);	void TIM_CCxCmd(TIM_TypeDef* TIMx, uint16_t TIM_Channel, uint16_t TIM_CCx);
	void TIM_CCxNCmd(TIM_TypeDef* TIMx, uint16_t TIM_Channel, uint16_t TIM_CCxN);	void TIM_CCxNCmd(TIM_TypeDef* TIMx, uint16_t TIM_Channel, uint16_t TIM_CCxN);
Input Capture management	void TIM_ICInit(TIM_TypeDef* TIMx, TIM_ICInitTypeDef* TIM_ICInitStruct);	void TIM_ICInit(TIM_TypeDef* TIMx, TIM_ICInitTypeDef* TIM_ICInitStruct);
	void TIM_ICStructInit(TIM_ICInitTypeDef* TIM_ICInitStruct);	void TIM_ICStructInit(TIM_ICInitTypeDef* TIM_ICInitStruct);
	void TIM_PWMIConfig(TIM_TypeDef* TIMx, TIM_ICInitTypeDef* TIM_ICInitStruct);	void TIM_PWMIConfig(TIM_TypeDef* TIMx, TIM_ICInitTypeDef* TIM_ICInitStruct);
	uint32_t TIM_GetCapture1(TIM_TypeDef* TIMx);	uint32_t TIM_GetCapture1(TIM_TypeDef* TIMx);
	uint32_t TIM_GetCapture2(TIM_TypeDef* TIMx);	uint32_t TIM_GetCapture2(TIM_TypeDef* TIMx);
	uint32_t TIM_GetCapture3(TIM_TypeDef* TIMx);	uint32_t TIM_GetCapture3(TIM_TypeDef* TIMx);
	uint32_t TIM_GetCapture4(TIM_TypeDef* TIMx);	uint32_t TIM_GetCapture4(TIM_TypeDef* TIMx);
	void TIM_SetIC1Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);	void TIM_SetIC1Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);
	void TIM_SetIC2Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);	void TIM_SetIC2Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);
	void TIM_SetIC3Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);	void TIM_SetIC3Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);
	void TIM_SetIC4Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);	void TIM_SetIC4Prescaler(TIM_TypeDef* TIMx, uint16_t TIM_ICPSC);

Table 31. STM32F10x and STM32F3xx TIM driver API correspondence (continued)

	STM32F10x TIM driver API	STM32F3xx TIM driver API
Advanced-control timers (TIM1 and TIM8) specific features	void TIM_BDTRConfig(TIM_TypeDef* TIMx, TIM_BDTRInitTypeDef *TIM_BDTRInitStruct);	void TIM_BDTRConfig(TIM_TypeDef* TIMx, TIM_BDTRInitTypeDef *TIM_BDTRInitStruct);
	NA	void TIM_Break1Config(TIM_TypeDef* TIMx, uint32_t TIM_Break1Polarity, uint8_t TIM_Break1Filter);(1)
	NA	void TIM_Break2Config(TIM_TypeDef* TIMx, uint32_t TIM_Break2Polarity, uint8_t TIM_Break2Filter);(1)
	NA	void TIM_Break1Cmd(TIM_TypeDef* TIMx, FunctionalState NewState);(1)
	NA	void TIM_Break2Cmd(TIM_TypeDef* TIMx, FunctionalState NewState);(1)
	void TIM_BDTRStructInit(TIM_BDTRInitTypeDef* TIM_BDTRInitStruct);	void TIM_BDTRStructInit(TIM_BDTRInitTypeDef* TIM_BDTRInitStruct);
	void TIM_CtrlPWMOutputs(TIM_TypeDef* TIMx, FunctionalState NewState);	void TIM_CtrlPWMOutputs(TIM_TypeDef* TIMx, FunctionalState NewState);
	void TIM_SelectCOM(TIM_TypeDef* TIMx, FunctionalState NewState);	void TIM_SelectCOM(TIM_TypeDef* TIMx, FunctionalState NewState);
	void TIM_CCPreloadControl(TIM_TypeDef* TIMx, FunctionalState NewState);	void TIM_CCPreloadControl(TIM_TypeDef* TIMx, FunctionalState NewState);
Interrupts, DMA and flags management	void TIM_ITConfig(TIM_TypeDef* TIMx, uint16_t TIM_IT, FunctionalState NewState);	void TIM_ITConfig(TIM_TypeDef* TIMx, uint16_t TIM_IT, FunctionalState NewState);
	void TIM_GenerateEvent(TIM_TypeDef* TIMx, uint16_t TIM_EventSource);	void TIM_GenerateEvent(TIM_TypeDef* TIMx, uint16_t TIM_EventSource);
	FlagStatus TIM_GetFlagStatus(TIM_TypeDef* TIMx, uint32_t TIM_FLAG);	FlagStatus TIM_GetFlagStatus(TIM_TypeDef* TIMx, uint32_t TIM_FLAG);
	void TIM_ClearFlag(TIM_TypeDef* TIMx, uint16_t TIM_FLAG);	void TIM_ClearFlag(TIM_TypeDef* TIMx, uint16_t TIM_FLAG);
	ITStatus TIM_GetITStatus(TIM_TypeDef* TIMx, uint16_t TIM_IT);	ITStatus TIM_GetITStatus(TIM_TypeDef* TIMx, uint16_t TIM_IT);
	void TIM_ClearITPendingBit(TIM_TypeDef* TIMx, uint16_t TIM_IT);	void TIM_ClearITPendingBit(TIM_TypeDef* TIMx, uint16_t TIM_IT);
	void TIM_DMAConfig(TIM_TypeDef* TIMx, uint16_t TIM_DMABase, uint16_t TIM_DMABurstLength);	void TIM_DMAConfig(TIM_TypeDef* TIMx, uint16_t TIM_DMABase, uint16_t TIM_DMABurstLength);
	void TIM_DMACmd(TIM_TypeDef* TIMx, uint16_t TIM_DMASource, FunctionalState NewState);	void TIM_DMACmd(TIM_TypeDef* TIMx, uint16_t TIM_DMASource, FunctionalState NewState);
	void TIM_SelectCCDMA(TIM_TypeDef* TIMx, FunctionalState NewState);	void TIM_SelectCCDMA(TIM_TypeDef* TIMx, FunctionalState NewState);

Table 31. STM32F10x and STM32F3xx TIM driver API correspondence (continued)

	STM32F10x TIM driver API	STM32F3xx TIM driver API
Clocks management	void TIM_InternalClockConfig(TIM_TypeDef* TIMx);	void TIM_InternalClockConfig(TIM_TypeDef* TIMx);
	void TIM_ITRxExternalClockConfig(TIM_TypeDef* TIMx, uint16_t TIM_InputTriggerSource);	void TIM_ITRxExternalClockConfig(TIM_TypeDef* TIMx, uint16_t TIM_InputTriggerSource);
	void TIM_TIxExternalClockConfig(TIM_TypeDef* TIMx, uint16_t TIM_TIxExternalCLKSource, uint16_t TIM_ICPolarity, uint16_t ICFilter);	void TIM_TIxExternalClockConfig(TIM_TypeDef* TIMx, uint16_t TIM_TIxExternalCLKSource, uint16_t TIM_ICPolarity, uint16_t ICFilter);
	void TIM_ETRClockMode1Config(TIM_TypeDef* TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter);	void TIM_ETRClockMode1Config(TIM_TypeDef* TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter);
	void TIM_ETRClockMode2Config(TIM_TypeDef* TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter);	void TIM_ETRClockMode2Config(TIM_TypeDef* TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter);
Synchronization management	void TIM_SelectInputTrigger(TIM_TypeDef* TIMx, uint16_t TIM_InputTriggerSource);	void TIM_SelectInputTrigger(TIM_TypeDef* TIMx, uint16_t TIM_InputTriggerSource);
	void TIM_SelectOutputTrigger(TIM_TypeDef* TIMx, uint16_t TIM_TRGOSource);	void TIM_SelectOutputTrigger(TIM_TypeDef* TIMx, uint16_t TIM_TRGOSource);
	NA	void TIM_SelectOutputTrigger2(TIM_TypeDef* TIMx, uint32_t TIM_TRGO2Source);(1)
	void TIM_SelectSlaveMode(TIM_TypeDef* TIMx, uint32_t TIM_SlaveMode);	void TIM_SelectSlaveMode(TIM_TypeDef* TIMx, uint32_t TIM_SlaveMode);
	void TIM_SelectMasterSlaveMode(TIM_TypeDef* TIMx, uint16_t TIM_MasterSlaveMode);	void TIM_SelectMasterSlaveMode(TIM_TypeDef* TIMx, uint16_t TIM_MasterSlaveMode);
	void TIM_ETRConfig(TIM_TypeDef* TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter);	void TIM_ETRConfig(TIM_TypeDef* TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter);

Table 31. STM32F10x and STM32F3xx TIM driver API correspondence (continued)

	STM32F10x TIM driver API	STM32F3xx TIM driver API
Specific interface management	void TIM_EncoderInterfaceConfig(TIM_TypeDef* TIMx, uint16_t TIM_EncoderMode, uint16_t TIM_IC1Polarity, uint16_t TIM_IC2Polarity);	void TIM_EncoderInterfaceConfig(TIM_TypeDef* TIMx, uint16_t TIM_EncoderMode, uint16_t TIM_IC1Polarity, uint16_t TIM_IC2Polarity);
	void TIM_SelectHallSensor(TIM_TypeDef* TIMx, FunctionalState NewState);	void TIM_SelectHallSensor(TIM_TypeDef* TIMx, FunctionalState NewState);
Specific remapping management	NA	void TIM_RemapConfig(TIM_TypeDef* TIMx, uint16_t TIM_Remap);

Color key:

- = New function
- = Same function, but API was changed
- = Function not available (NA)

1. Those functions are applicable only for STM32F30xxx devices.

4.19 DBGMCU driver

Existing DBGMCU available on STM32F10x and STM32F3xx devices have the same specifications. Table 32 lists the DBGMCU driver APIs.

Table 32. STM32F10x and STM32F3xx DBGMCU driver API correspondence

	STM32F10x DBGMCU driver API	STM32F3xx DBGMCU driver API
Device and Revision ID management	uint32_t DBGMCU_GetREVID(void);	uint32_t DBGMCU_GetREVID(void);
	uint32_t DBGMCU_GetDEVID(void);	uint32_t DBGMCU_GetDEVID(void);
Peripherals configuration	void DBGMCU_Config(uint32_t DBGMCU_Periph, FunctionalState NewState);	void DBGMCU_Config(uint32_t DBGMCU_Periph, FunctionalState NewState);
	NA	void DBGMCU_APB1PeriphConfig(uint32_t DBGMCU_Periph, FunctionalState NewState);
	NA	void DBGMCU_APB2PeriphConfig(uint32_t DBGMCU_Periph, FunctionalState NewState);

Color key:

= New function

= Same function, but API was changed

= Function not available (NA)

5 Revision history

Table 33. Document revision history

Date	Revision	Changes
19-Feb-2013	1	Initial release
20-Mar-2014	2	Updated *Table 1*, *Table 2*, *Table 6*, *Table 7*, *Table 8*, *Table 11*, *Table 12*, *Table 13*, *Table 14*, *Table 15*, *Table 17* and *Table 23*. Added *Table 16: ADC channels mapping differences*. Updated *Section 3.2: System architecture*, *Section 3.4.3: Peripheral clock configuration*, *Section 3.15: USART interface* and *Section 3.16: CEC interface*. Updated *Figure 1*.
04-Apr-2014	3	Removed references to products with 16 Kbytes of Flash memory (STM32F301x4, STM32F302x4 and STM32F303x4). Updated *Table 2*
28-Jan-2015	4	Extended the document applicability to STM32F303xDxE devices. Updated: – *Table 1: STM32F103 and STM32F3xx line available packages* – The text after *Table 2: Main pinout differences between STM32F10x and STM32F30x lines* – *Table 6: STM32 peripheral compatibility analysis STM32F1 Series versus STM32F3 Series* – *Section 3.2: System architecture* – *Figure 1: System architecture for STM32F30x and STM32F3x8 lines* – *Table 7: IP bus mapping differences between STM32F3 Series and STM32F1 Series* – *Section 3.4.1: System clock configuration* – *Section 3.4.3: Peripheral clock configuration* – *Table 11: DMA request differences between STM32F3 Series and STM32F1 Series* – *Table 12: Interrupt vector differences between STM32F3 Series and STM32F1 Series* – *Section 3.7.1: Alternate function mode* – *Table 15: ADC differences between STM32F1 Series and STM32F30x lines* – *Table 16: ADC channels mapping differences* – *Table 19: STM32F10x and STM32F3xx FLASH driver API correspondence* – *Table 22: STM32F10x and STM32F3xx ADC driver API correspondence* – *Table 23: STM32F10x and STM32F3xx DAC driver API correspondence* Added – *Section 4.17: FMC driver*

Table 33. Document revision history (continued)

Date	Revision	Changes
05-Mar-2015	5	Updated: – the number of ADC channels for STM32F303x6/8 in *Table 15: ADC differences between STM32F1 Series and STM32F30x lines*, – PB1, PB2 and PB23 entries for STM32F303x6/8 in *Table 16: ADC channels mapping differences*.
10-Feb-2017	6	Updated *Table 6: STM32 peripheral compatibility analysis STM32F1 Series versus STM32F3 Series*.

IMPORTANT NOTICE – PLEASE READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgement.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of Purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2017 STMicroelectronics – All rights reserved