Introduction

The STM32F2xx Standard Peripheral Library covers 3 abstraction levels, and includes:

- A complete register address mapping with all bits, bitfields and registers declared in C. This avoids a cumbersome task and more important, it brings the benefits of a bug free reference mapping file, speeding up the early project phase.
- A collection of routines and data structures covering all peripheral functions (drivers with common API). It can directly be used as a reference framework, since it also includes macros for supporting core-related intrinsic features, common constants, and definition of data types.
- A set of examples covering all available peripherals with template projects for the most common development tools. With the appropriate hardware evaluation board, this allows to get started with a brand-new micro within few hours.

Each driver consists of a set of functions covering all peripheral features. The development of each driver is driven by a common API (application programming interface) which standardizes the driver structure, the functions and the parameter names.

The driver source code is developed in ‘Strict ANSI-C’ (relaxed ANSI-C for projects and example files). It is fully documented and is MISRA-C 2004 compliant. Writing the whole library in ‘Strict ANSI-C’ makes it independent from the development tools. Only the start-up files depend on the development tools. Thanks to the Standard Peripheral Library, low-level implementation details are transparent so that reusing code on a different MCU requires only to reconfigure the compiler. As a result, developers can easily migrate designs across the STM32 series to quickly bring product line extensions to market without any redesign. In addition, the library is built around a modular architecture that makes it easy to tailor and run it on the same MCU using hardware platforms different from ST evaluation boards.

The Standard Peripheral Library implements run-time failure detection by checking the input values for all library functions. Such dynamic checking contributes towards enhancing the robustness of the software. Run-time detection is suitable for user application development and debugging. It adds an overhead which can be removed from the final application code to minimize code size and execution speed. For more details refer to Section 1.1.5: "Run-time checking".

Since the Standard Peripheral Library is generic and covers all peripheral features, the size and/or execution speed of the application code may not be optimized. For many applications, the library may be used as is. However, for applications having tough constraints in terms of code size and/or execution speed, the library drivers should be used as a reference on how to configure the peripheral and tailor them to specific application requirements.

The firmware library user manual is structured as follows:

- Document conventions, rules, architecture and overview of the Library package.
- How to use and customize the Library (step by step).
- Detailed description of each peripheral driver: configuration structure, functions and how to use the provided API to build your application.

The STM32F2xx Standard Peripheral Library will be referred to as Library throughout the document, unless otherwise specified.
Contents

1  STM32F2xx Standard Peripheral Library................................................. 18
   1.1  Coding rules and conventions...................................................... 18
       1.1.1  Acronyms.............................................................................. 18
       1.1.2  Naming conventions.............................................................. 19
       1.1.3  Coding rules ....................................................................... 20
       1.1.4  Bit-Banding............................................................................ 22
       1.1.5  Run-time checking................................................................. 23
       1.1.6  MISRA-C 2004 compliance...................................................... 25
   1.2  Architecture..................................................................................... 27
   1.3  Package description......................................................................... 28
       1.3.1  Library folder structure......................................................... 29
       1.3.2  Project folder ........................................................................ 32
       1.3.3  Utilities folder........................................................................ 33
   1.4  Supported devices and development tools ....................................... 35
       1.4.1  Supported devices.................................................................... 35
       1.4.2  Supported development tools and compilers............................. 36

2  How to use and customize the library ...................................................... 37
   2.1  Library configuration parameters.................................................... 37
   2.2  Library programming model ............................................................ 39
   2.3  Peripheral initialization and configuration........................................ 40
   2.4  How to run your first example......................................................... 41
       2.4.1  Prerequisites........................................................................... 41
       2.4.2  Run your first example............................................................ 41
       2.4.3  Run a peripheral example....................................................... 43
   2.5  STM32F2xx programming model using the library............................ 44
   2.6  How to develop your first application............................................... 46
       2.6.1  Starting point ......................................................................... 46
       2.6.2  Library configuration parameters............................................ 46
       2.6.3  system_stm32f2xx.c................................................................ 46
       2.6.4  main.c.................................................................................. 47
       2.6.5  stm32f2xx_it.c....................................................................... 49

3  Analog-to-digital converter (ADC).......................................................... 50
   3.1  ADC Firmware driver registers structures......................................... 50
       3.1.1  ADC_TypeDef........................................................................ 50
3.1.2 ADC_Common_TypeDef........................................... 51
3.1.3 ADC_InitTypeDef............................................... 52
3.1.4 ADC_CommonInitTypeDef.................................... 52

3.2 ADC Firmware driver API description........................................ 53
3.2.1 How to use this driver ........................................... 53
3.2.2 Initialization and configuration ................................... 54
3.2.3 Interrupt and flag management ................................... 56
3.2.4 Initialization and configuration functions............................ 57
3.2.5 Analog Watchdog configuration functions............................. 60
3.2.6 Temperature Sensor, Vrefint (Voltage Reference internal)............ 62
3.2.7 Regular Channels Configuration functions............................ 62
3.2.8 Regular Channels DMA Configuration functions.......................... 67
3.2.9 Injected channels Configuration functions.............................. 68
3.2.10 Interrupt and flag management functions............................. 74

3.3 ADC Firmware driver defines.............................................. 76
3.3.1 ADC Firmware driver defines..................................... 76

3.4 ADC Programming Example............................................. 88

4 Controller area network (CAN)............................................. 90
4.1 CAN Firmware driver registers structures................................. 90
4.1.1 CAN_TxMailBox_TypeDef....................................... 90
4.1.2 CAN_FIFOMailBox_TypeDef..................................... 90
4.1.3 CAN_FilterRegister_TypeDef................................... 91
4.1.4 CAN_TypeDef ..................................................... 91
4.1.5 CAN_InitTypeDef............................................... 92
4.1.6 CAN_FilterInitTypeDef......................................... 94
4.1.7 CanTxMsg........................................................ 95
4.1.8 CanRxMsg........................................................ 95

4.2 CAN Firmware driver API description..................................... 96
4.2.1 How to use this driver ........................................... 96
4.2.2 Initialization and configuration ................................... 97
4.2.3 Interrupt and flag management ................................... 98
4.2.4 Initialization and configuration functions............................ 100
4.2.5 CAN Frames Transmission functions ................................ 103
4.2.6 CAN Frames Reception functions ................................ 104
4.2.7 CAN Operation modes functions ................................ 105
4.2.8 CAN Bus Error management functions.............................. 106
4.2.9 Interrupt and flag management functions............................. 108
<table>
<thead>
<tr>
<th>Contents</th>
<th>UM1061</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3 CAN Firmware driver defines</td>
<td>111</td>
</tr>
<tr>
<td>4.3.1 CAN Firmware driver defines</td>
<td>111</td>
</tr>
<tr>
<td>4.4 CAN Programming Example</td>
<td>121</td>
</tr>
<tr>
<td>5 CRC calculation unit (CRC)</td>
<td>123</td>
</tr>
<tr>
<td>5.1 CRC Firmware driver registers structures</td>
<td>123</td>
</tr>
<tr>
<td>5.1.1 CRC_TypeDef</td>
<td>123</td>
</tr>
<tr>
<td>5.2 CRC Firmware driver API description</td>
<td>123</td>
</tr>
<tr>
<td>5.2.1 Functions</td>
<td>124</td>
</tr>
<tr>
<td>5.3 CRC Programming Example</td>
<td>126</td>
</tr>
<tr>
<td>6 Cryptographic processor (CRYP)</td>
<td>127</td>
</tr>
<tr>
<td>6.1 CRYP Firmware driver registers structures</td>
<td>127</td>
</tr>
<tr>
<td>6.1.1 CRYP_TypeDef</td>
<td>127</td>
</tr>
<tr>
<td>6.1.2 CRYP_InitTypeDef</td>
<td>128</td>
</tr>
<tr>
<td>6.1.3 CRYP_KeyInitTypeDef</td>
<td>129</td>
</tr>
<tr>
<td>6.1.4 CRYP_IVInitTypeDef</td>
<td>129</td>
</tr>
<tr>
<td>6.1.5 CRYP_Context</td>
<td>130</td>
</tr>
<tr>
<td>6.2 CRYP Firmware driver API description</td>
<td>131</td>
</tr>
<tr>
<td>6.2.1 How to use this driver</td>
<td>131</td>
</tr>
<tr>
<td>6.2.2 Initialization and configuration</td>
<td>132</td>
</tr>
<tr>
<td>6.2.3 Interrupt and flag management</td>
<td>133</td>
</tr>
<tr>
<td>6.2.4 High level functions</td>
<td>134</td>
</tr>
<tr>
<td>6.2.5 Initialization and configuration functions</td>
<td>135</td>
</tr>
<tr>
<td>6.2.6 CRYP Data processing functions</td>
<td>138</td>
</tr>
<tr>
<td>6.2.7 Context swapping functions</td>
<td>139</td>
</tr>
<tr>
<td>6.2.8 CRYPTO DMA interface Configuration function</td>
<td>140</td>
</tr>
<tr>
<td>6.2.9 Interrupt and flag management functions</td>
<td>140</td>
</tr>
<tr>
<td>6.2.10 High Level AES functions</td>
<td>141</td>
</tr>
<tr>
<td>6.2.11 High Level TDES functions</td>
<td>143</td>
</tr>
<tr>
<td>6.2.12 High Level DES functions</td>
<td>144</td>
</tr>
<tr>
<td>6.3 CRYP Firmware driver defines</td>
<td>145</td>
</tr>
<tr>
<td>6.3.1 CRYP Firmware driver defines</td>
<td>145</td>
</tr>
<tr>
<td>6.4 CRYP Programming Example</td>
<td>148</td>
</tr>
<tr>
<td>7 Digital-to-analog converter (DAC)</td>
<td>150</td>
</tr>
<tr>
<td>7.1 DAC Firmware driver registers structures</td>
<td>150</td>
</tr>
<tr>
<td>7.1.1 DAC_TypeDef</td>
<td>150</td>
</tr>
<tr>
<td>7.1.2 DAC_InitTypeDef</td>
<td>151</td>
</tr>
</tbody>
</table>
7.2 DAC Firmware driver API description..................................................151
  7.2.1 DAC peripheral features.................................................................151
  7.2.2 How to use this driver .................................................................153
  7.2.3 DAC channels configuration: trigger, output buffer, data format.... 153
  7.2.4 DMA management........................................................................153
  7.2.5 Interrupt and flag management ......................................................153
  7.2.6 DAC channels configuration..........................................................153
  7.2.7 DMA management function...........................................................158
  7.2.8 Interrupt and flag management functions......................................159

7.3 DAC Firmware driver defines .............................................................161
  7.3.1 DAC Firmware driver defines ......................................................161

7.4 DAC Programming Example ...............................................................165

8 Debug support (DBGMCU).....................................................................167
  8.1 DBGMCU Firmware driver registers structures..............................167
     8.1.1 DBGMCU_TypeDef.................................................................167
  8.2 DBGMCU Firmware driver API description ....................................167
     8.2.1 Functions..................................................................................167
  8.3 DBGMCU Firmware driver defines................................................170

9 Digital camera interface (DCMI) ..........................................................173
  9.1 DCMI Firmware driver registers structures.....................................173
     9.1.1 DCMI_TypeDef...........................................................................173
     9.1.2 DCMI_InitTypeDef......................................................................174
     9.1.3 DCMI_CROPInitTypeDef............................................................174
     9.1.4 DCMI_CodesInitTypeDef............................................................175
  9.2 DCMI Firmware driver API description ..........................................175
     9.2.1 How to use this driver..............................................................175
     9.2.2 Initialization and configuration.................................................176
     9.2.3 Image capture............................................................................176
     9.2.4 Interrupt and flag management ................................................176
     9.2.5 Initialization and configuration functions.................................177
     9.2.6 Image capture functions............................................................179
     9.2.7 Interrupt and flag management functions................................180
  9.3 DCMI Firmware driver defines.........................................................183
     9.3.1 DCMI Firmware driver defines.................................................183
  9.4 DCMI Programming Example............................................................186

10 DMA controller (DMA) .........................................................................187
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 DMA Firmware driver registers structures</td>
<td>187</td>
</tr>
<tr>
<td>10.1.1 DMA_TypeDef</td>
<td>187</td>
</tr>
<tr>
<td>10.1.2 DMA_Stream_TypeDef</td>
<td>187</td>
</tr>
<tr>
<td>10.1.3 DMA_InitTypeDef</td>
<td>188</td>
</tr>
<tr>
<td>10.2 DMA Firmware driver API description</td>
<td>189</td>
</tr>
<tr>
<td>10.2.1 How to use this driver</td>
<td>189</td>
</tr>
<tr>
<td>10.2.2 Initialization and configuration</td>
<td>191</td>
</tr>
<tr>
<td>10.2.3 Interrupt and flag management</td>
<td>193</td>
</tr>
<tr>
<td>10.2.4 Initialization and configuration functions</td>
<td>194</td>
</tr>
<tr>
<td>10.2.5 Data Counter functions</td>
<td>197</td>
</tr>
<tr>
<td>10.2.6 Double Buffer mode functions</td>
<td>198</td>
</tr>
<tr>
<td>10.2.7 Interrupt and flag management functions</td>
<td>200</td>
</tr>
<tr>
<td>10.3 DMA Firmware driver defines</td>
<td>203</td>
</tr>
<tr>
<td>10.3.1 DMA Firmware driver defines</td>
<td>203</td>
</tr>
<tr>
<td>10.4 DMA Programming Example</td>
<td>215</td>
</tr>
<tr>
<td>11 External interrupt/event controller (EXTI)</td>
<td>217</td>
</tr>
<tr>
<td>11.1 EXTI Firmware driver registers structures</td>
<td>217</td>
</tr>
<tr>
<td>11.1.1 EXTI_TypeDef</td>
<td>217</td>
</tr>
<tr>
<td>11.1.2 EXTI_InitTypeDef</td>
<td>217</td>
</tr>
<tr>
<td>11.2 EXTI Firmware driver API description</td>
<td>218</td>
</tr>
<tr>
<td>11.2.1 EXTI features</td>
<td>218</td>
</tr>
<tr>
<td>11.2.2 How to use this driver</td>
<td>218</td>
</tr>
<tr>
<td>11.2.3 Initialization and configuration</td>
<td>218</td>
</tr>
<tr>
<td>11.2.4 Interrupt and flag management</td>
<td>219</td>
</tr>
<tr>
<td>11.2.5 Initialization and configuration functions</td>
<td>219</td>
</tr>
<tr>
<td>11.2.6 Interrupt and flag management functions</td>
<td>220</td>
</tr>
<tr>
<td>11.3 EXTI Firmware driver defines</td>
<td>222</td>
</tr>
<tr>
<td>11.3.1 EXTI Firmware driver defines</td>
<td>222</td>
</tr>
<tr>
<td>11.4 EXTI Programming Example</td>
<td>224</td>
</tr>
<tr>
<td>12 FLASH Memory (FLASH)</td>
<td>225</td>
</tr>
<tr>
<td>12.1 FLASH Firmware driver registers structures</td>
<td>225</td>
</tr>
<tr>
<td>12.1.1 FLASH_TypeDef</td>
<td>225</td>
</tr>
<tr>
<td>12.2 FLASH Firmware driver API description</td>
<td>225</td>
</tr>
<tr>
<td>12.2.1 How to use this driver</td>
<td>225</td>
</tr>
<tr>
<td>12.2.2 FLASH interface configuration</td>
<td>226</td>
</tr>
<tr>
<td>12.2.3 FLASH memory programming</td>
<td>227</td>
</tr>
<tr>
<td>12.2.4 Option bytes programming</td>
<td>228</td>
</tr>
</tbody>
</table>
12.2.5 Interrupt and flag management ........................................ 229
12.2.6 FLASH interface configuration functions .......................... 229
12.2.7 FLASH memory programming functions .......................... 231
12.2.8 Option bytes programming functions .............................. 235
12.2.9 Interrupt and flag management functions ....................... 239

12.3 FLASH Firmware driver defines ........................................ 242
12.3.1 FLASH Firmware driver defines .................................... 242

12.4 FLASH Programming Example ........................................... 247

13 Flexible static memory controller (FSMC) ............................. 250

13.1 FSMC Firmware driver registers structures .......................... 250
  13.1.1 FSMC_Bank1_TypeDef .................................................. 250
  13.1.2 FSMC_Bank1E_TypeDef ............................................... 250
  13.1.3 FSMC_Bank2_TypeDef .................................................. 250
  13.1.4 FSMC_Bank3_TypeDef .................................................. 251
  13.1.5 FSMC_Bank4_TypeDef .................................................. 252
  13.1.6 FSMC_NORSRAMTimingInitTypeDef ................................. 252
  13.1.7 FSMC_NORSRAMInitTypeDef ......................................... 253
  13.1.8 FSMC_NAND_PCCARDTimingInitTypeDef ............................ 255
  13.1.9 FSMC_NANDInitTypeDef .............................................. 255
  13.1.10 FSMC_PCCARDInitTypeDef ......................................... 256

13.2 FSMC Firmware driver API description ............................... 257
  13.2.1 NOR/SRAM controller .................................................. 257
  13.2.2 NAND controller ........................................................ 258
  13.2.3 PCCARD controller ..................................................... 258
  13.2.4 Interrupt and flag management .................................... 259
  13.2.5 NOR_SRAM controller functions .................................... 259
  13.2.6 NAND controller functions .......................................... 261
  13.2.7 PCCARD controller functions ....................................... 263
  13.2.8 Interrupt and flag management functions ....................... 265

13.3 FSMC Firmware driver defines ........................................... 267
  13.3.1 FSMC Firmware driver defines ...................................... 267

13.4 FSMC Programming Example .............................................. 273

14 General-purpose I/Os (GPIO) ............................................. 274

14.1 GPIO Firmware driver registers structures ........................... 274
  14.1.1 GPIO_TypeDef .......................................................... 274
  14.1.2 GPIO_InitTypeDef .................................................... 274

14.2 GPIO Firmware driver API description ................................ 275
## 14.2 How to use this driver

14.2.1 How to use this driver ................................................. 275  
14.2.2 Initialization and configuration ........................................ 276  
14.2.3 GPIO Read and Write .................................................. 276  
14.2.4 GPIO Alternate functions configuration ............................... 276  
14.2.5 Initialization and configuration functions ............................ 276  
14.2.6 GPIO Read and Write functions ........................................ 278  
14.2.7 GPIO Alternate functions configuration function ..................... 281  

## 14.3 GPIO Firmware driver defines

14.3.1 GPIO Firmware driver defines ........................................ 283  

## 14.4 GPIO Programming Example

14.4 GPIO Programming Example ............................................ 289  

## 15 Hash processor (HASH)

15.1 HASH Firmware driver registers structures ............................... 292  
15.1.1 HASH_TypeDef ......................................................... 292  
15.1.2 HASH_InitTypeDef .................................................... 292  
15.1.3 HASH_MsgDigest ....................................................... 293  
15.1.4 HASH_Context ........................................................ 293  

15.2 HASH Firmware driver API description .................................. 294  
15.2.1 How to use this driver ................................................ 294  
15.2.2 Initialization and configuration ....................................... 295  
15.2.3 Message Digest generation .......................................... 295  
15.2.4 Context swapping ...................................................... 296  
15.2.5 Initialization and configuration ....................................... 296  
15.2.6 Interrupt and flag management ...................................... 296  
15.2.7 High level functions .................................................. 297  
15.2.8 Initialization and configuration functions ............................ 297  
15.2.9 Message Digest generation functions ................................ 299  
15.2.10 Context swapping functions ......................................... 301  
15.2.11 HASH DMA interface Configuration function ...................... 301  
15.2.12 Interrupt and flag management functions .......................... 302  
15.2.13 High Level SHA1 functions ......................................... 304  
15.2.14 High Level MD5 functions .......................................... 305  

## 15.3 HASH Firmware driver defines ......................................... 306  

## 15.4 HASH Programming Example ........................................... 307  

## 16 Inter-integrated circuit interface (I2C)

16.1 I2C Firmware driver registers structures ............................... 309  
16.1.1 I2C_TypeDef ........................................................... 309
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18.2.9  Flags management functions</td>
<td>351</td>
</tr>
<tr>
<td>18.3    PWR Firmware driver defines</td>
<td>352</td>
</tr>
<tr>
<td>18.3.1  PWR Firmware driver defines</td>
<td>352</td>
</tr>
<tr>
<td>18.4    PWR Programming Example</td>
<td>353</td>
</tr>
<tr>
<td><strong>19</strong>  Reset and clock control (RCC)</td>
<td>355</td>
</tr>
<tr>
<td>19.1    RCC Firmware driver registers structures</td>
<td>355</td>
</tr>
<tr>
<td>19.1.1  RCC_TypeDef</td>
<td>355</td>
</tr>
<tr>
<td>19.1.2  RCC_ClocksTypeDef</td>
<td>357</td>
</tr>
<tr>
<td>19.2    RCC Firmware driver API description</td>
<td>357</td>
</tr>
<tr>
<td>19.2.1  RCC specific features</td>
<td>357</td>
</tr>
<tr>
<td>19.2.2  Internal and external clocks, PLL, CSS and MCO configuration</td>
<td>358</td>
</tr>
<tr>
<td>19.2.3  System AHB and APB busses clocks configuration</td>
<td>359</td>
</tr>
<tr>
<td>19.2.4  Peripheral clocks configuration</td>
<td>360</td>
</tr>
<tr>
<td>19.2.5  Interrupt and flag management functions</td>
<td>361</td>
</tr>
<tr>
<td>19.2.6  Internal and external clocks, PLL, CSS and MCO configuration</td>
<td>361</td>
</tr>
<tr>
<td>19.2.7  System AHB and APB busses clocks configuration</td>
<td>368</td>
</tr>
<tr>
<td>19.2.8  Peripheral clocks configuration</td>
<td>372</td>
</tr>
<tr>
<td>19.2.9  Interrupt and flag management functions</td>
<td>384</td>
</tr>
<tr>
<td>19.3    RCC Firmware driver defines</td>
<td>386</td>
</tr>
<tr>
<td>19.3.1  RCC Firmware driver defines</td>
<td>386</td>
</tr>
<tr>
<td>19.4    RCC Programming Example</td>
<td>400</td>
</tr>
<tr>
<td><strong>20</strong>  Random number generator (RNG)</td>
<td>403</td>
</tr>
<tr>
<td>20.1    RNG Firmware driver registers structures</td>
<td>403</td>
</tr>
<tr>
<td>20.1.1  RNG_TypeDef</td>
<td>403</td>
</tr>
<tr>
<td>20.2    RNG Firmware driver API description</td>
<td>403</td>
</tr>
<tr>
<td>20.2.1  How to use this driver</td>
<td>403</td>
</tr>
<tr>
<td>20.2.2  Initialization and configuration</td>
<td>403</td>
</tr>
<tr>
<td>20.2.3  Getting 32-bit Random number</td>
<td>404</td>
</tr>
<tr>
<td>20.2.4  Interrupt and flag management</td>
<td>404</td>
</tr>
<tr>
<td>20.2.5  Initialization and configuration functions</td>
<td>405</td>
</tr>
<tr>
<td>20.2.6  Get 32 bit Random number function</td>
<td>406</td>
</tr>
<tr>
<td>20.2.7  Interrupt and flag management functions</td>
<td>406</td>
</tr>
<tr>
<td>20.3    RNG Firmware driver defines</td>
<td>408</td>
</tr>
<tr>
<td>20.3.1  RNG Firmware driver defines</td>
<td>408</td>
</tr>
<tr>
<td>20.4    RNG Programming Example</td>
<td>409</td>
</tr>
<tr>
<td><strong>21</strong>  Real-time clock (RTC)</td>
<td>410</td>
</tr>
</tbody>
</table>
21.1 RTC Firmware driver registers structures ........................................... 410
  21.1.1 RTC_TypeDef.............................................................. 410
  21.1.2 RTC_InitTypeDef......................................................... 412
  21.1.3 RTC_TimeTypeDef......................................................... 413
  21.1.4 RTC_DateTypeDef......................................................... 413
  21.1.5 RTC_AlarmTypeDef......................................................... 414

21.2 RTC Firmware driver API description .................................................. 414
  21.2.1 Backup Domain operating conditions............................................. 414
  21.2.2 How to use the RTC driver.................................................... 415
  21.2.3 RTC configuration........................................................... 416
  21.2.4 Backup Data registers configuration.......................................... 418
  21.2.5 RTC and low power modes.................................................... 419
  21.2.6 RTC Tamper and TimeStamp pin selection and Output Type Config
       configuration................................................................. 419
  21.2.7 Interrupt and flag management................................................. 419
  21.2.8 Initialization and configuration functions................................... 421
  21.2.9 Backup Data registers configuration functions.............................. 424
  21.2.10 RTC Tamper and TimeStamp pin selection and Output Type Config
          configuration functions.................................................. 425
  21.2.11 Interrupt and flag management functions.................................. 426
  21.2.12 Time and Date configuration functions.................................... 429
  21.2.13 Alarm configuration functions............................................ 431
  21.2.14 WakeUp Timer configuration functions.................................... 433
  21.2.15 Daylight Saving configuration functions.................................... 435
  21.2.16 Output pin Configuration function........................................ 436
  21.2.17 Coarse Calibration configuration functions................................ 436
  21.2.18 TimeStamp configuration functions........................................ 438
  21.2.19 Tampers configuration functions.......................................... 439

21.3 RTC Firmware driver defines.......................................................... 440
  21.3.1 RTC Firmware driver defines................................................ 440

21.4 RTC Programming Example.................................................................... 449

22 Secure digital input/output interface (SDIO).......................................... 451
  22.1 SDIO Firmware driver registers structures........................................ 451
   22.1.1 SDIO_TypeDef.............................................................. 451
   22.1.2 SDIO_InitTypeDef......................................................... 452
   22.1.3 SDIO_CmdInitTypeDef..................................................... 453
   22.1.4 SDIO_DataInitTypeDef.................................................... 453

  22.2 SDIO Firmware driver API description............................................ 454
**22.2.1** How to use this driver ................................................................. 454
**22.2.2** Initialization and configuration .................................................... 456
**22.2.3** Command path state machine (CPSM) management ...................... 456
**22.2.4** Interrupt and flag management .................................................... 457
**22.2.5** Initialization and configuration functions .................................... 457
**22.2.6** Command path state machine (CPSM) management functions .... 459
**22.2.7** Data path state machine (DPSM) management functions .............. 461
**22.2.8** SDIO IO Cards mode management functions .................................. 463
**22.2.9** CE-ATA mode management functions ........................................... 464
**22.2.10** DMA transfers management function ......................................... 465
**22.2.11** Interrupt and flag management functions .................................. 466

**22.3** SDIO Firmware driver defines .......................................................... 471
  **22.3.1** SDIO Firmware driver defines .................................................... 471

**22.4** SDIO Programming Example ............................................................ 479

**23** Serial peripheral interface (SPI) ......................................................... 481

**23.1** SPI Firmware driver registers structures ........................................ 481
  **23.1.1** SPI_TypeDef .............................................................................. 481
  **23.1.2** SPI_InitTypeDef ....................................................................... 482
  **23.1.3** I2S_InitTypeDef ....................................................................... 483

**23.2** SPI Firmware driver API description ............................................... 484
  **23.2.1** How to use this driver ............................................................... 484
  **23.2.2** Initialization and configuration .................................................. 485
  **23.2.3** Data transfers ........................................................................... 485
  **23.2.4** Hardware CRC calculation ......................................................... 485
  **23.2.5** DMA transfers management ....................................................... 487
  **23.2.6** Interrupt and flag management .................................................. 487
  **23.2.7** Initialization and configuration functions .................................... 488
  **23.2.8** Data transfers functions ............................................................. 493
  **23.2.9** Hardware CRC calculation functions .......................................... 494
  **23.2.10** DMA transfers management function ........................................ 495
  **23.2.11** Interrupt and flag management functions .................................. 496

**23.3** SPI Firmware driver defines ............................................................ 499
  **23.3.1** SPI Firmware driver defines ....................................................... 499

**23.4** SPI Programming Example ............................................................... 507
  **23.4.1** I2S Programming Example .......................................................... 509

**24** System configuration controller (SYSCFG) ........................................... 510

**24.1** SYSCFG Firmware driver registers structures .................................... 510
## 24 SYSCFG Firmware driver defines

**24.1 SYSCFG_TypeDef**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>510</td>
<td>SYSCFG_TypeDef</td>
</tr>
</tbody>
</table>

**24.2 SYSCFG Firmware driver API description**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>511</td>
<td>Functions</td>
</tr>
</tbody>
</table>

**24.3 SYSCFG Firmware driver defines**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>513</td>
<td>SYSCFG_TypeDef</td>
</tr>
</tbody>
</table>

## 25 General-purpose timers (TIM)

**25.1 TIM Firmware driver registers structures**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>517</td>
<td>TIM_TypeDef</td>
</tr>
</tbody>
</table>

**25.2 TIM Firmware driver API description**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>522</td>
<td>How to use this driver</td>
</tr>
</tbody>
</table>

**25.3 TIM Firmware driver defines**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>568</td>
<td>TIM_TypeDef</td>
</tr>
</tbody>
</table>

### 25.4 TIM Programming Example

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>586</td>
<td>TIM_TypeDef</td>
</tr>
</tbody>
</table>

## 26 Universal synchronous asynchronous receiver transmitter (USART)

**26.1 USART Firmware driver registers structures**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>589</td>
<td>USART_TypeDef</td>
</tr>
</tbody>
</table>

**26.2 USART Firmware driver API description**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>589</td>
<td>USART_TypeDef</td>
</tr>
</tbody>
</table>

**26.3 USART Firmware driver defines**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>590</td>
<td>USART_TypeDef</td>
</tr>
</tbody>
</table>

**26.4 USART Programming Example**

<table>
<thead>
<tr>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>590</td>
<td>USART_TypeDef</td>
</tr>
</tbody>
</table>

---

Stable. The text is clear and readable.
<table>
<thead>
<tr>
<th>Contents</th>
<th>UM1061</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.1.3 USART_ClockInitTypeDef</td>
<td>591</td>
</tr>
<tr>
<td>26.2 USART Firmware driver API description</td>
<td>591</td>
</tr>
<tr>
<td>26.2.1 How to use this driver</td>
<td>591</td>
</tr>
<tr>
<td>26.2.2 Initialization and configuration</td>
<td>592</td>
</tr>
<tr>
<td>26.2.3 Data transfers</td>
<td>593</td>
</tr>
<tr>
<td>26.2.4 Multiprocessor communication</td>
<td>593</td>
</tr>
<tr>
<td>26.2.5 LIN mode</td>
<td>593</td>
</tr>
<tr>
<td>26.2.6 Halfduplex mode</td>
<td>594</td>
</tr>
<tr>
<td>26.2.7 Smartcard mode</td>
<td>595</td>
</tr>
<tr>
<td>26.2.8 IrDA mode</td>
<td>596</td>
</tr>
<tr>
<td>26.2.9 DMA transfers management</td>
<td>596</td>
</tr>
<tr>
<td>26.2.10 Interrupt and flag management</td>
<td>596</td>
</tr>
<tr>
<td>26.2.11 Initialization and configuration functions</td>
<td>598</td>
</tr>
<tr>
<td>26.2.12 Data transfers functions</td>
<td>601</td>
</tr>
<tr>
<td>26.2.13 MultiProcessor communication functions</td>
<td>602</td>
</tr>
<tr>
<td>26.2.14 LIN mode functions</td>
<td>603</td>
</tr>
<tr>
<td>26.2.15 Halfduplex mode function</td>
<td>605</td>
</tr>
<tr>
<td>26.2.16 Smartcard mode functions</td>
<td>605</td>
</tr>
<tr>
<td>26.2.17 IrDA mode functions</td>
<td>606</td>
</tr>
<tr>
<td>26.2.18 DMA transfers management functions</td>
<td>607</td>
</tr>
<tr>
<td>26.2.19 Interrupt and flag management functions</td>
<td>608</td>
</tr>
<tr>
<td>26.3 USART Firmware driver defines</td>
<td>611</td>
</tr>
<tr>
<td>26.3.1 USART Firmware driver defines</td>
<td>611</td>
</tr>
<tr>
<td>26.4 USART Programming Example</td>
<td>616</td>
</tr>
<tr>
<td>27 Window watchdog (WWDG)</td>
<td>619</td>
</tr>
<tr>
<td>27.1 WWDG Firmware driver registers structures</td>
<td>619</td>
</tr>
<tr>
<td>27.1.1 WWDG_TypeDef</td>
<td>619</td>
</tr>
<tr>
<td>27.2 WWDG Firmware driver API description</td>
<td>619</td>
</tr>
<tr>
<td>27.2.1 Prescaler, Refresh window and Counter configuration functions</td>
<td>620</td>
</tr>
<tr>
<td>27.2.2 WWDG activation function</td>
<td>622</td>
</tr>
<tr>
<td>27.2.3 Interrupt and flag management functions</td>
<td>622</td>
</tr>
<tr>
<td>27.3 WWDG Firmware driver defines</td>
<td>623</td>
</tr>
<tr>
<td>27.4 WWDG Programming Example</td>
<td>624</td>
</tr>
<tr>
<td>28 Miscellaneous add-on to CMSIS functions(misc)</td>
<td>625</td>
</tr>
<tr>
<td>28.1 MISC Firmware driver registers structures</td>
<td>625</td>
</tr>
<tr>
<td>28.1.1 NVIC_InitTypeDef</td>
<td>625</td>
</tr>
<tr>
<td>28.2 MISC Firmware driver API description</td>
<td>625</td>
</tr>
</tbody>
</table>
28.2.1 How to configure Interrupts using driver ................................................................. 625
28.2.2 Functions ................................................................................................................. 626

28.3 MISC Firmware driver defines ................................................................................... 628
  28.3.1 MISC Firmware driver defines ............................................................................... 628

28.4 Interrupt Programming Example .................................................................................. 629

29 Revision history .............................................................................................................. 633

30 Disclaimer ..................................................................................................................... 634
List of tables

Table 1: List of abbreviations ................................................................. 18
Table 2: MSIRA-C 2004 compliance matrix ............................................ 25
Table 3: Description of CMSIS files ....................................................... 30
Table 4: STM32F2xx_StdPeriph_Driver files description ......................... 31
Table 5: STM32F2xx_StdPeriph_Template files description .................... 32
Table 6: Utilities/STM32_EVAL files description .................................. 34
Table 7: Library configuration parameters ......................................... 37
Table 8: Default clock configuration in system_stm32F2xxx.c .................. 47
Table 9: Number of wait states according to CPU clock (HCLK) frequency . 226
Table 10: Program/erase parallelism .................................................... 227
Table 11: Number of wait states according to CPU clock (HCLK) frequency 359
Table 12: Selection of RTC_AF1 alternate functions ............................... 417
Table 13: Selection of RTC_AF2 alternate functions ............................... 418
Table 14: Revision history ................................................................... 633
# List of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Library architecture</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>Library package structure</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>Library folder structure</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Project folder structure</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>Utilities folder structure</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>Message displayed on the LCD when running the template example</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>How to run a peripheral example</td>
<td>43</td>
</tr>
<tr>
<td>8</td>
<td>STM32F2xx programming model using the library</td>
<td>44</td>
</tr>
</tbody>
</table>
1 STM32F2xx Standard Peripheral Library

1.1 Coding rules and conventions
The conventions used in the present user manual and in the library are described in the sections below.

1.1.1 Acronyms
Table 1: "List of abbreviations" describes the acronyms used in this document.

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Peripheral / unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog-to-digital</td>
</tr>
<tr>
<td>BKPSRAM</td>
<td>Backup SRAM memory</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller area network</td>
</tr>
<tr>
<td>CRC</td>
<td>CRC calculation unit</td>
</tr>
<tr>
<td>CRYP</td>
<td>Cryptographic processor</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital to analog converter</td>
</tr>
<tr>
<td>DBGMCU</td>
<td>Debug MCU</td>
</tr>
<tr>
<td>DCMI</td>
<td>Digital camera interface</td>
</tr>
<tr>
<td>DMA</td>
<td>DMA controller</td>
</tr>
<tr>
<td>EXTI</td>
<td>External interrupt/event controller</td>
</tr>
<tr>
<td>FSMC</td>
<td>Flexible static memory controller</td>
</tr>
<tr>
<td>FLASH</td>
<td>Flash memory</td>
</tr>
<tr>
<td>GPIO</td>
<td>General purpose I/O</td>
</tr>
<tr>
<td>HASH</td>
<td>Hash processor</td>
</tr>
<tr>
<td>I²C</td>
<td>Inter-integrated circuit</td>
</tr>
<tr>
<td>I²S</td>
<td>Inter-integrated sound</td>
</tr>
<tr>
<td>IWDG</td>
<td>Independent watchdog</td>
</tr>
<tr>
<td>NVIC</td>
<td>Nested vectored interrupt controller</td>
</tr>
<tr>
<td>PWR</td>
<td>Power control</td>
</tr>
<tr>
<td>RCC</td>
<td>Reset and clock controller</td>
</tr>
<tr>
<td>RNG</td>
<td>Random number generator</td>
</tr>
<tr>
<td>RTC</td>
<td>Real-time clock</td>
</tr>
<tr>
<td>SDIO</td>
<td>SDIO interface</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial peripheral interface</td>
</tr>
<tr>
<td>SysTick</td>
<td>System tick timer</td>
</tr>
<tr>
<td>TIM</td>
<td>Advanced-control, general-purpose or basic timer</td>
</tr>
<tr>
<td>USART</td>
<td>Universal synchronous asynchronous receiver transmitter</td>
</tr>
</tbody>
</table>
1.1.2 Naming conventions

The following naming conventions are used in the library:

- **PPP** refers to any peripheral acronym, for example **ADC**. See Section 1.1: “Coding rules and conventions” for more information.
- System and source/header file names are preceded by ‘stm32f2xx_’, for example stm32f2xx_conf.h.
- Constants used in one file are defined within this file. A constant used in more than one file is defined in a header file. All constants are written in upper case, except for peripheral driver function parameters.
- typedef variable names should be suffixed with _TypeDef.
- Registers are considered as constants. In most cases, their name is in upper case and uses the same acronyms as in the STM32F2xx reference manual document.
- Peripheral registers are declared in the **PPP_TypeDef** structure (e.g. **ADC_TypeDef**) in stm32f2xx.h file.
- Almost all peripheral function names are preceded by the corresponding peripheral acronym in upper case followed by an underscore. The first letter in each word is in upper case, for example **USART_SendData**. Only one underscore is allowed in a function name to separate the peripheral acronym from the rest of the function name.
- The structure containing the initialization parameters for the PPP peripheral are named **PPP_InitTypeDef** (e.g. **ADC_InitTypeDef**).
- The functions used to initialize the PPP peripheral according to parameters specified in **PPP_InitTypeDef** are named **PPP_Init**, e.g. **TIM_Init**.
- The functions used to reset the PPP peripheral registers to their default values are named **PPP_DeInit**, e.g. **TIM_DeInit**.
- The functions used to fill the **PPP_InitTypeDef** structure with the reset values of each member are named **PPP_StructInit**, e.g. **USART_StructInit**.
- The functions used to enable or disable the specified PPP peripheral are named **PPP_Cmd**, for example **USART_Cmd**.
- The functions used to enable or disable an interrupt source for the specified PPP peripheral are named **PPP_ITConfig**, e.g. **RCC_ITConfig**.
- The functions used to enable or disable the DMA interface for the specified PPP peripheral are named **PPP_DMAConfig**, e.g. **TIM_DMAConfig**.
- The functions used to configure a peripheral function always end with the string ‘Config’, for example **GPIO_PinAFConfig**.
- The functions used to check whether the specified PPP flag is set or reset are named **PPP_GetFlagStatus**, e.g. **I2C_GetFlagStatus**.
- The functions used to clear a PPP flag are named **PPP_ClearFlag**, for example **I2C_ClearFlag**.
- The functions used to check whether the specified PPP interrupt has occurred or not are named **PPP_GetITStatus**, e.g. **I2C_GetITStatus**.
- The functions used to clear a PPP interrupt pending bit are named **PPP_ClearITPendingBit**, e.g. **I2C_ClearITPendingBit**.
1.1.3 Coding rules

This section describes the coding rules used in the library.

General

- All codes should comply with ANSI C standard and should compile without warning under at least its main compiler. Any warnings that cannot be eliminated should be commented in the code.
- The library uses ANSI standard data types defined in the ANSI C header file `<stdint.h>`.
- The library has no blocking code and all required waiting loops (polling loops) are controlled by an expiry programmed timeout.

Variable types

Specific variable types are already defined with a fixed type and size. These types are defined in the file `stm32f2xx.h`

```
typedef enum {
    RESET = 0,
    SET = !RESET
} FlagStatus, ITStatus;

typedef enum {
    DISABLE = 0,
    ENABLE = !DISABLE
} FunctionalState;

typedef enum {
    ERROR = 0,
    SUCCESS = !ERROR
} ErrorStatus;
```

Peripherals

Pointers to peripherals are used to access the peripheral control registers. They point to data structures that represent the mapping of the peripheral control registers.

Peripheral registers structure

`stm32f2xx.h` contains the definition of all peripheral register structures. The example below illustrates the SPI register structure declaration:

```
/*--------------------- Serial Peripheral Interface ---------------------*/
typedef struct
{
    __IO uint16_t CR1; /*!< SPI control register 1 (not used in I2S mode),
    Address offset: 0x00 */
    uint16_t RESERVED0;/*< Reserved, 0x02 */
    __IO uint16_t CR2; /*!< SPI control register 2, Address offset: 0x04 */
    uint16_t RESERVED1;/*< Reserved, 0x06 */
    __IO uint16_t SR;  /*!< SPI status register, Address offset: 0x08 */
    uint16_t RESERVED2;/*< Reserved, 0x0A */
    __IO uint16_t DR;  /*!< SPI data register, Address offset: 0x0C */
    uint16_t RESERVED3;/*< Reserved, 0x0E */
    __IO uint16_t CRCPR; /*!< SPI CRC polynomial register (not used in I2S mode),
    Address offset: 0x10 */
    uint16_t RESERVED4;/*< Reserved, 0x12 */
```
The register names are the register acronyms written in upper case for each peripheral. RESERVEDi (I being an integer that indexes the reserved field) indicates a reserved field.

Each peripheral has several dedicated registers which contain different flags. Registers are defined within a dedicated structure for each peripheral. Flags are defined as acronyms written in upper case and preceded by 'PPP_FLAG_'. The flag definition is adapted to each peripheral case and defined in stm32f2xx_ppp.h.

Peripheral declaration

All peripherals are declared in stm32f2xx.h. The following example shows the declaration of the SPI peripheral:

```c
#define SPI2_BASE             (APB1PERIPH_BASE + 0x3800)
#define SPI3_BASE             (APB1PERIPH_BASE + 0x3C00)
...
#define SPI1_BASE             (APB2PERIPH_BASE + 0x3000)
...
#define SPI2               ((SPI_TypeDef *) SPI2_BASE)
#define SPI3               ((SPI_TypeDef *) SPI3_BASE)
...
#define SPI1               ((SPI_TypeDef *) SPI1_BASE)
```

SPIx_BASE is the base address of a specific SPI and SPIx is a pointer to a register structure that refers to a specific SPI.

The peripheral registers are accessed as follows:

```c
SPI1->CR1 = 0x0001;
```

Peripheral registers bits

All the peripheral registers bits are defined as constants in the stm32f2xx.h file. They are defined as acronyms written in upper-case into the form:

`PPP_<register_name>_<bit_name>`
1.1.4 Bit-Banding

The Cortex-M3 memory map includes two bit-band memory regions. These regions map each word in an alias region of memory to a bit in a bit-band region of memory. Writing to a word in the alias region has the same effect as a read/modify/write operation on the targeted bit in the bit-band region.

All the STM32F2xx peripheral registers are mapped in a bit-band region. This feature is consequently intensively used in functions which perform single bit set/reset in order to reduce and optimize code size.

The sections below describe how the bit-band access is used in the Library.

Mapping formula

The mapping formula shows how to link each word in the alias region to a corresponding target bit in the bit-band region. The mapping formula is given below:

\[
\text{bit\_word\_offset} = (\text{byte\_offset} \times 32) + (\text{bit\_number} \times 4)
\]
\[
\text{bit\_word\_addr} = \text{bit\_band\_base} + \text{bit\_word\_offset}
\]

where:

- \text{bit\_word\_offset} is the position of the target bit in the bit-band memory region
- \text{bit\_word\_addr} is the address of the word in the alias memory region that maps to the targeted bit.
- \text{bit\_band\_base} is the starting address of the alias region
- \text{byte\_offset} is the number of the byte in the bit-band region that contains the targeted bit
- \text{bit\_number} is the bit position (0-7) of the targeted bit.

Example of implementation

The following example shows how to map the PLLON[24] bit of RCC_CR register in the alias region:

```c
/*!< Peripheral base address in the alias region */
#define PERIPH_BASE   ((uint32_t)0x40000000)

/*!< Peripheral base address in the bit-band region */
#define PERIPH_BB_BASE    ((uint32_t)0x42000000)

/* Alias word address of PLLON bit */
#define RCC_OFFSET                (RCC_BASE - PERIPH_BASE)
```

Example:

```c
#define SPI_CR1_CPHA  ((uint16_t)0x0001) /*!< Clock Phase */
#define SPI_CR1_CPOL  ((uint16_t)0x0002) /*!< Clock Polarity */
#define SPI_CR1_MSTR  ((uint16_t)0x0004) /*!< Master Selection */
#define SPI_CR1_BR    ((uint16_t)0x0038) /*!< BR[2:0] bits (Baud Rate Control) */
#define SPI_CR1_BR_0  ((uint16_t)0x0008) /*!< Bit 0 */
#define SPI_CR1_BR_1  ((uint16_t)0x0010) /*!< Bit 1 */
#define SPI_CR1_BR_2  ((uint16_t)0x0020) /*!< Bit 2 */
```
#define CR_OFFSET       (RCC_OFFSET + 0x00)
#define PLLON_BitNumber 0x18
#define CR_PLLON_BB      (PERIPH_BB_BASE + (CR_OFFSET * 32) +
                      (PLLON_BitNumber * 4))

To code a function which enables/disables the PLL, the usual method is the following:

...  
void RCC_PLLCmd(FunctionalState NewState)
{
  if (NewState != DISABLE)
  { /* Enable PLL */
    RCC->CR |= RCC_CR_PLLON;
  }  
  else
  { /* Disable PLL */
    RCC->CR &= ~RCC_CR_PLLON;
  }
}

Using bit-band access this function will be coded as follows:

void RCC_PLLCmd(FunctionalState NewState)
{
  *(__IO uint32_t *) CR_PLLON_BB = (uint32_t)NewState;
}

1.1.5 Run-time checking

The library implements run-time failure detection by checking the input values of all library functions. The run-time checking is achieved by using an assert_param macro. This macro is used in all the library functions which have an input parameter. It allows checking that the input value lies within the parameter allowed values.

To enable the run-time checking, use the assert_param macro, and leave the define USE_FULL_ASSERT uncommented in stm32f2xx_conf.h file.

Example:PWR_ClearFlag function

stm32f2xx_pwr.c:

void PWR_ClearFlag(uint32_t PWR_FLAG)
{
  /* Check the parameters */
  assert_param(IS_PWR_CLEAR_FLAG(PWR_FLAG));
  PWR->CR |= PWR_FLAG << 2;
}

stm32f2xx_pwr.h:

/* PWR Flag */
#define PWR_FLAG_WU       ((uint32_t)0x00000001)
#define PWR_FLAG_SB       ((uint32_t)0x00000002)
#define PWR_FLAG_PVDO     ((uint32_t)0x00000004)
#define PWR_FLAG_BRR      ((uint32_t)0x00000008)
...
#define IS_PWR_CLEAR_FLAG(FLAG) (((FLAG) == PWR_FLAG_WU) || ((FLAG) == PWR_FLAG_SB))
If the expression passed to the `assert_param` macro is false, the `assert_failed` function is called and returns the name of the source file and the source line number of the call that failed. If the expression is true, no value is returned.

The `assert_param` macro is implemented in `stm32f2xx_conf.h`:

```c
/* Exported macro -----------------------------------------------------------*/
#ifdef USE_FULL_ASSERT
/**
 * @brief The assert_param macro is used for function's parameters check.
 * @param expr: If expr is false, it calls assert_failed function
 * which reports the name of the source file and the source line number of the call that failed.
 * If expr is true, it returns no value.
 * @retval None
 */
#define assert_param(expr) ((expr) ? (void)0 : assert_failed((uint8_t *)__FILE__, __LINE__))
/* Exported functions ------------------------------------------------------*/
void assert_failed(uint8_t* file, uint32_t line);
#else
#define assert_param(expr) ((void)0)
#endif /* USE_FULL_ASSERT */
```

The `assert_failed` function is implemented in the main.c file or in any other user C file:

```c
#elifdef USE_FULL_ASSERT
/**
 * @brief Reports the name of the source file and the source line number
 * where the assert_param error has occurred.
 * @param file: pointer to the source file name
 * @param line: assert_param error line source number
 * @retval None
 */
void assert_failed(uint8_t* file, uint32_t line)
{
    /* User can add his own implementation to report the file name and line number */
    printf("Wrong parameter value detected on\r\n");
    printf(" file %s\r\n", file);
    printf(" line %d\r\n", line);
    /* Infinite loop */
    while (1)
    {
        
    }
#endif /* USE_FULL_ASSERT */
```

Because of the overhead it introduces, it is recommended to use run-time checking during application code development and debugging, and to remove it from the final application to improve code size and speed.

However if you want to keep this functionality in your final application, reuse the `assert_param` macro defined within the library to test the parameter values before calling the library functions.
1.1.6 MISRA-C 2004 compliance

The C programming language is growing in importance for embedded systems. However, when it comes to developing code for safety-critical applications, this language has many drawbacks. There are several unspecified, implementation-defined, and undefined aspects of the C language that make it unsuited for developing safety-critical systems.

The Motor Industry Software Reliability Association’s Guidelines for the use of the C language in critical systems (MISRA-C 2004 [1]) describe a subset of the C language well suited for developing safety-critical systems.

The STM32F2xx standard peripheral drivers (STM32F2xx_StdPeriph_Driver) have been developed to be MISRA-C 2004 compliant.

The following section describes how the StdPeriph_Driver complies with MISRA-C 2004 (as described in section 4.4 Claiming compliance of the standard [1]):

- A compliance matrix has been completed which shows how compliance has been enforced.
- The whole STM32F2xx_StdPeriph_Driver C code is compliant with MISRA-C 2004 rules. Deviations are documented.
- A list of all instances of rules not being followed is being maintained, and for each instance there is an appropriately signed-off deviation.
- All the issues listed in section 4.2 "The programming language and coding context of the standard" [1], that need to be checked during the firmware development phase, have been addressed during the development of the STM32F2xx standard peripherals driver and appropriate measures have been taken.

Compliance matrix

The compliance of the STM32F2xx standard peripherals driver (STM32F2xx_StdPeriph_Driver) with MISRA-C 2004 has been checked using the IAR C/C++ Compiler for ARM. MISRA compliance applies only to STM32F2xx standard peripherals driver source file. Examples and project files are not MISRA compliant.

Two options are available for checking MISRA compliance:

- The compiler: IAR C/C++ Compiler for ARM V6.20
- Manual checking (code review)

The following table lists the MISRA-C 2004 rules that are frequently violated in the code.

Table 2: MISRA-C 2004 compliance matrix

<table>
<thead>
<tr>
<th>MISRA-C 2004 rule number</th>
<th>Required/Advisory</th>
<th>Summary</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Required</td>
<td>Compiler is configured to allow extensions - all code shall conform to ISO 9899 standard C, with no extensions permitted</td>
<td>IAR compiler extensions are enabled. This was allowed to support new CMSIS types.</td>
</tr>
<tr>
<td>5.1</td>
<td>Required</td>
<td>Identifiers (internal and external) shall not rely on significance of more than 31 characters</td>
<td>Some long parameters names are defined for code readability.</td>
</tr>
<tr>
<td>8.1</td>
<td>Required</td>
<td>No prototype seen - functions shall always have prototype declarations and the prototype shall be visible at both the function definition</td>
<td>This rule is violated as there is no function prototype for __WFI and __WFE macros in the CMSIS layer.</td>
</tr>
<tr>
<td>10.1</td>
<td>Required</td>
<td>The value of an expression of integer type shall not be implicitly converted to a</td>
<td>Complexity</td>
</tr>
<tr>
<td>MISRA-C 2004 rule number</td>
<td>Required/Advisory</td>
<td>Summary</td>
<td>Reason</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------------</td>
<td>---------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>different underlying type.</td>
<td></td>
</tr>
<tr>
<td>10.6</td>
<td>Required</td>
<td>A 'U' suffix shall be applied to all constants of &quot;unsigned&quot; type</td>
<td>The &quot;stdint.h&quot; defined types are used to be CMSIS compliant.</td>
</tr>
<tr>
<td>11.2</td>
<td>Required</td>
<td>Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.</td>
<td>Needed when addressing memory mapped registers</td>
</tr>
<tr>
<td>11.3</td>
<td>Advisory</td>
<td>A cast should not be performed between a pointer type and an integral type.</td>
<td>Needed when addressing memory mapped registers</td>
</tr>
<tr>
<td>16.7</td>
<td>Advisory</td>
<td>A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.</td>
<td></td>
</tr>
<tr>
<td>19.1</td>
<td>Advisory</td>
<td>#include statements in a file shall only be preceded by other preprocessor directives or comments</td>
<td>This rule was violated to be in line with the CMSIS architecture.</td>
</tr>
</tbody>
</table>

How to check that your code is MISRA-C 2004 compliant

The default IAR project template provided with the STM32F2xx Standard Peripheral Library is already pre-configured for MISRA-C 2004 compliance. Then, the user has to enable the MISRA-C 2004 checker if needed.

To enable the IAR MISRA-C 2004 checker, go to Project->Options (ALT+F7) and then in "General Options" Category select the "MISRA-C:2004" tab and check the "Enable MISRA-C" box. With the default EWARM template project, all violated rules described above are unchecked.

To use the IAR MISRA-C Checker to verify that your code is MISRA-C 2004 compliant, please follow the following steps:

1. Enable the IAR MISRA-C 2004 Checker
2. Inside the core_cm3.h file add the following directive "#pragma system_include" to prevent the MISRA-C checker to check this file.
3. Uncomment the "USE_FULL_ASSERT" inside the STM32f2xx_conf.h file

Only the STM32F2xx standard peripherals driver are MISRA-C 2004 Compliant.

1.2 Architecture

The library is built around a modular programming model ensuring the independencies between the several components building the main application and allowing an easy porting on a large product range, evaluation boards and even the use of some integrated firmware components for other application with the minimum changes on the code of the common parts.

The following figure provides a global view of the STM32F2xx Standard Peripheral Library usage and interaction with other firmware components.

**Figure 1: Library architecture**

**HAL**

HAL is a Hardware Abstraction Layer (HAL) that allows controlling the different STM32F2xx device’s registers and features.

- CMSIS layer
  - Core Peripheral Access Layer: contains name definitions, address definitions and helper functions to access core registers and peripherals. It defines also a device independent interface for RTOS Kernels that includes debug channel definitions.
  - STM32F2xx Device Peripheral Access Layer: provides definitions for all the peripheral register’s definitions, bits definitions and memory mapping for STM32F2xx devices.

- STM32F2xx standard peripheral driver that provides drivers and header files for all the peripherals. It uses CMSIS layer to access STM32F2xx registers.

**BSP**

BSP is a board specific package (BSP) that implements an abstraction layer to interact with the Human Interface resources: buttons, LEDs, LCD and COM ports (USARTs) available on STMicroelectronics evaluation boards. A common API is provided to manage these different resources, and can be easily tailored to support any other development board, by just adapting the initialization routine.
Application layer

The application layer consists of a set of examples covering all available peripherals with template projects for the most common development Tools. With the appropriate hardware evaluation board, this allows to get started with a brand new micro within few hours.

1.3 Package description

The Library is supplied in one single zip file. The extraction of the zip file generates one folder, STM32F2xx_StdPeriph_Lib_VX.Y.Z, which contains the following subfolders:

Figure 2: Library package structure

![Library package structure diagram]

1. VX.Y.Z refer to the library version, ex. V1.0.0

The library package consists of three main folders, described in Section 1.3.1: "Library folder structure"
1.3.1 Library folder structure

This folder contains all CMSIS files and STM32F2xx Standard Peripheral Drivers.

The library folder structure is shown in the figure below:

Figure 3: Library folder structure

CMSIS subfolder

This subfolder contains the STM32F2xx and Cortex-M3 CMSIS files:

- Cortex-M CMSIS files containing name definitions, address definitions and helper functions to access Cortex-M3 core registers and peripherals. It defines also a device independent interface for RTOS kernels that includes debug channel definitions.
- STM32F2xx CMSIS files consist of:
  - stm32f2xx.h: this file contains the definitions of all peripheral registers, bits, and memory mapping for STM32F2xx devices. It is the unique include file used in the application programmer C source code, usually in the main.c.
  - system_stm32f2xx.c/.h: this file contains the system clock configuration for STM32F2xx devices. It exports SystemInit() function which sets up the system clock source, PLL multiplier and divider factors, AHB/APBx prescalers and Flash settings. This function is called at startup just after reset and before connecting to the main program. The call is made inside the startup_stm32f2xx.s file.
  - startup_stm32f4xx.s: this file contains the Cortex-M3 startup code and interrupt vectors for all STM32F2xx device interrupt handlers.

STM32F2xx_StdPeriph_Driver subfolder

This subfolder contains all the subdirectories and files that make up the core of the library. They do not need to be modified by the user:

- inc subfolder contains the peripheral drivers header files.
- src subfolder contains the peripheral drivers source files.

Each peripheral has a source code file, stm32f2xx_ppp.c, and a header file, stm32f2xx_ppp.h. The stm32f2xx_ppp.c file contains all the firmware functions required to use the PPP peripheral.
The library files are listed and described in details in the following tables.

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>core_cm3.c</td>
<td>Defines several helper functions that access Cortex-M3 core registers.</td>
</tr>
<tr>
<td>core_cm3.h</td>
<td>Describes the data structures for the Cortex-M3 core peripherals and</td>
</tr>
<tr>
<td></td>
<td>performs the address mapping of these structures. It also provides basic</td>
</tr>
<tr>
<td></td>
<td>access to the Cortex-M3 core registers and core peripherals using efficient</td>
</tr>
<tr>
<td></td>
<td>functions defined as static inline.</td>
</tr>
<tr>
<td>stm32f2xx.h</td>
<td>CMSIS Cortex-M3 STM32F2xx peripheral access layer header file. This file</td>
</tr>
<tr>
<td></td>
<td>contains the definitions of all peripheral registers, bits, and memory</td>
</tr>
<tr>
<td></td>
<td>mapping for STM32F2xx devices. The file is the unique include file used in</td>
</tr>
<tr>
<td></td>
<td>the application programmer C source code, usually in the main.c. This file</td>
</tr>
<tr>
<td></td>
<td>contains:</td>
</tr>
<tr>
<td></td>
<td>• Configuration section allowing:</td>
</tr>
<tr>
<td></td>
<td>− To select the device used in the target application</td>
</tr>
<tr>
<td></td>
<td>− To use or not the peripheral drivers in your application code</td>
</tr>
<tr>
<td></td>
<td>(meaning that the code is based on direct access to peripheral</td>
</tr>
<tr>
<td></td>
<td>registers rather than drivers API). This option is controlled by #define</td>
</tr>
<tr>
<td></td>
<td>USE_STDPERIPH_DRIVER</td>
</tr>
<tr>
<td></td>
<td>− To change few application-specific parameters such as the HSE crystal</td>
</tr>
<tr>
<td></td>
<td>frequency</td>
</tr>
<tr>
<td></td>
<td>• Data structures and address mapping for all peripherals</td>
</tr>
<tr>
<td></td>
<td>• Peripheral registers declarations and bits definition</td>
</tr>
<tr>
<td></td>
<td>• Macros to access peripheral registers hardware</td>
</tr>
<tr>
<td></td>
<td>This file also contains the library release number defined by the define</td>
</tr>
<tr>
<td></td>
<td>statement __STM32F2XX_STDPERIPH_VERSION</td>
</tr>
<tr>
<td>system_stm32f2xx.c</td>
<td>This file contains the system clock configuration for STM32F2xx devices.</td>
</tr>
<tr>
<td></td>
<td>This file includes two functions and one global variable to be called from</td>
</tr>
<tr>
<td></td>
<td>the user application:</td>
</tr>
<tr>
<td></td>
<td>• SystemInit(): this function setups the system clock source, PLL</td>
</tr>
<tr>
<td></td>
<td>multiplier and divider factors, AHB/APBx prescalers and Flash</td>
</tr>
<tr>
<td></td>
<td>settings. This function is called at startup just after reset and</td>
</tr>
<tr>
<td></td>
<td>before branch to the main program. The call is made inside the</td>
</tr>
<tr>
<td></td>
<td>startup_stm32f2xx.s file.</td>
</tr>
<tr>
<td></td>
<td>• SystemCoreClock: this variable contains the core clock (HCLK). It can be</td>
</tr>
<tr>
<td></td>
<td>used by the application code to set up the SysTick timer or configure</td>
</tr>
<tr>
<td></td>
<td>other parameters.</td>
</tr>
<tr>
<td></td>
<td>• SystemCoreClockUpdate(): this function updates the variable SystemCore</td>
</tr>
<tr>
<td></td>
<td>Clock and must be called whenever the core clock is changed during</td>
</tr>
<tr>
<td></td>
<td>program execution.</td>
</tr>
<tr>
<td></td>
<td>This file is automatically generated by the clock configuration</td>
</tr>
<tr>
<td></td>
<td>tool &quot;STM32f2xx_Clock_Configuration.xls&quot;. Using this tool, you can generate</td>
</tr>
<tr>
<td></td>
<td>a configuration file customized for your application requirements. For more</td>
</tr>
<tr>
<td></td>
<td>information, please refer to AN3362 available from ST web site.</td>
</tr>
<tr>
<td>system_stm32f2xx.h</td>
<td>Header file for system_stm32f2xx.c</td>
</tr>
<tr>
<td>startup_stm32f2xx.s</td>
<td>Provides the Cortex-M3 startup code and interrupt vectors for all STM32F2xx</td>
</tr>
<tr>
<td></td>
<td>device interrupt handlers. This module performs the following functions:</td>
</tr>
</tbody>
</table>
|                 | • It sets the initial SP
It sets the initial PC == Reset_Handler
- It sets the vector table entries with the exceptions ISR address
- It branches to __main in the C library (which eventually calls main()). A file is provided for each compiler.

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stm32f2xx_ppp.c</td>
<td>Driver source code file of PPP peripheral coded in Strict ANSI-C, and independent from the development Tools.</td>
</tr>
<tr>
<td>stm32f2xx_ppp.h</td>
<td>Provides functions prototypes and variable definitions used within for stm32f2xx_ppp.c file</td>
</tr>
<tr>
<td>misc.c</td>
<td>Provides all the miscellaneous firmware functions (add-on to CMSIS functions)</td>
</tr>
<tr>
<td>misc.h</td>
<td>Header for misc.c file</td>
</tr>
</tbody>
</table>
1.3.2 Project folder

This folder contains template projects and peripheral examples. Its structure is shown in the figure below.

**Figure 4: Project folder structure**

STM32F2xx_StdPeriph_Template subfolder

This subfolder contains standard template projects for the supported development tools that compile the needed STM32F2xx standard peripheral drivers plus all the user-modifiable files that are necessary to create a new project.

The files are listed and described in details in the following table.

**Table 5: STM32F2xx_StdPeriph_Template files description**

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.c</td>
<td>Template source file allowing starting a development from scratch using the library drivers.</td>
</tr>
<tr>
<td>main.h</td>
<td>header file for main.c</td>
</tr>
<tr>
<td>stm32f2xx_conf.h</td>
<td>Header file allowing to enable/disable the peripheral drivers header files inclusion. This file can also be used to enable or disable the library run-time failure detection before compiling the firmware library drivers, through the</td>
</tr>
</tbody>
</table>
This file contains the system clock configuration for STM32F2xx devices. This file provides two functions and one global variable to be called from user application:

- **SystemInit():** this function sets up the system clock source, PLL multiplier and divider factors, AHB/APBx prescalers and Flash settings. This function is called at startup just after reset and before branch to main program. This call is made inside the "startup_stm32f2xx.s" file.
- **SystemCoreClock:** this variable contains the core clock (HCLK). It can be used by the user application to set up the SysTick timer or configure other parameters.
- **SystemCoreClockUpdate():** this function updates the variable SystemCoreClock and must be called whenever the core clock is changed during program execution.

This file is automatically generated by the clock configuration tool “STM32f2xx_Clock_Configuration.xls”. Using this tool, you can generate a configuration file customized for your application requirements. For more information, please refer to AN3362 available from ST web site.

Template source file containing the interrupt service routine (ISR) for Cortex-M3 exceptions. You can add additional ISR(s) for the used peripheral(s) (for the available peripheral interrupt handler name, please refer to the startup file startup_stm32f2xx.s).

Header file for stm32f2xx_it.c

### STM32F2xx_StdPeriph_Examples sub folder

This subfolder contains, for each peripheral, the minimum set of files needed to run a typical example on this peripheral. In addition to the user files described in the section above, each subfolder contains a readme.txt file describing the example and how to make it work.

For more details about the available examples within the library please refer to Library_Examples.html file located in the root of this folder.

### 1.3.3 Utilities folder

This folder contains the abstraction layer allowing interacting with the human interface resources (buttons, LEDs, LCD and COM ports (USARTs)) available on STMicroelectronics evaluation boards. A common API is provided to manage these different resources. It and can be easily tailored to support any other development board, by adapting the initialization routine.

Additional drivers are provided to manage the different memories and storage media available on these boards.

For each hardware module (e.g. LCD, I2C, EEPROM, external SRAM memory...) the API is fully compatible across all STMicroelectronics evaluation board drivers.
The Utilities folder structure is shown below.

![Utilities folder structure](image_url)

It contains common files and folder, plus a folder for STM322xG_EVAL board files.

**Table 6: Utilities/STM32_EVAL files description**

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stm32xg_eval.c</td>
<td>This file provides:</td>
</tr>
<tr>
<td></td>
<td>• A set of firmware functions to manage LEDs, pushbuttons, and COM ports</td>
</tr>
<tr>
<td></td>
<td>• Low level initialization functions for SDcard (on SDIO) and serial EEPROM (sEE) available on STM322xG-EVAL board.</td>
</tr>
<tr>
<td>stm32xg_eval.h</td>
<td>Header file for stm32xg_eval.c</td>
</tr>
<tr>
<td>stm32xg_eval_audio_codec.c</td>
<td>This file includes the low layer driver for CS43L22 Audio Codec available on STM322xG-EVAL board.</td>
</tr>
<tr>
<td>stm32xg_eval_audio_codec.h</td>
<td>Header file for stm32xg_eval_audio_codec.c</td>
</tr>
<tr>
<td>stm32xg_eval_fsmc_onenand.c</td>
<td>This file provides a set of functions needed to drive the KFG1216U2A/B-DIB6</td>
</tr>
</tbody>
</table>
### 1.4 Supported devices and development tools

#### 1.4.1 Supported devices

The library supports all STM32F205xx, STM32F207xx, STM32F215xx and STM32F217xx microcontroller memory and peripherals. By using this library moving the application firmware from one STM32F2xx device to another becomes straightforward.

The device part number is defined as follows in `stm32f2xx.h` file:

```c
#if !defined (STM32F2XX)
#define STM32F2XX
#endif
```

This define statement can be used at application level to configure the application firmware for STM32F2xx devices.
1.4.2 Supported development tools and compilers

STM32F2xx devices are supported by a full range of development solutions from lead suppliers that deliver start-to-finish control of application development from a single integrated development environment.

The library is supported by all major tool providers.

A template project is available for each development tool:

- **IAR Embedded Workbench for ARM (EWARM) development tool**
  - Compiler: IAR’s C/C++
- **RealView Microcontroller Development Kit (MDK-ARM) development tool**
  - Compiler: ARM C/C++ compiler
- **TASKING VX-toolset for ARM Cortex-M3 development tool**
  - Compiler: Tasking VX C/C++
- **Raisonance IDE RIDE7 (RIDE) development tool**
  - Compiler: GNU C/C++
- **Atollic TrueSTUDIO STM32 (TrueSTUDIO) development tool**
  - Compiler: GNU C/C++

Refer to the library release notes to know about the supported development tool version.
2 How to use and customize the library

The following sections explain all the steps required to configure, customize, run your first example, and develop your application based on the library.

2.1 Library configuration parameters

The configuration interface allows customizing the library for your application. It is not mandatory to modify this configuration and you can use the default configuration without any modification.

To configure these parameters, you should enable, disable or modify some options by uncommenting, commenting or modifying the values of the related define statements as described in the table below.

Table 7: Library configuration parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STM32F2XX</td>
<td>stm32f2xx.h</td>
<td>Default status: enabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defines the root number of STM32F2xx devices. This define statement can be used at application level to configure the application firmware for STM32F2xx.</td>
</tr>
<tr>
<td>USE_STDPERIPH_DRIVER(1)</td>
<td>stm32f2xx.h</td>
<td>Default status: disabled</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When disabled, the peripheral drivers are not included and the application code is based on direct access to peripherals registers.</td>
</tr>
<tr>
<td>HSE_VALUE</td>
<td>stm32f2xx.h</td>
<td>Default value: 25 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defines the value of the external oscillator (HSE) expressed in Hz. The user must adjust this define statement when using a different crystal value.</td>
</tr>
<tr>
<td>HSE_STARTUP_TIMEOUT</td>
<td>stm32f2xx.h</td>
<td>Default value: 0x0500</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defines the maximum external oscillator (HSE) startup timeout value. The user must adjust this define statement when using a different statement startup time.</td>
</tr>
<tr>
<td>HSI_VALUE</td>
<td>stm32f2xx.h</td>
<td>Default value: 16 MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Defines the value of the internal oscillator (HSI) expressed in Hz.</td>
</tr>
<tr>
<td>__MPU_PRESENT</td>
<td>stm32f2xx.h</td>
<td>These define statements are used by Cortex-M3 CMSIS layer to inform about the options supported by STM32F2xx devices:</td>
</tr>
<tr>
<td>__NVIC_PRIO_BITS</td>
<td></td>
<td>/*!&lt; STM32F2XX provide an MPU */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#define __MPU_PRESENT 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/*!&lt; STM32F2XX uses 4 Bits for the Priority Levels */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#define __NVIC_PRIO_BITS 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/*!&lt; CMSIS SysTick Config is used */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#define __Vendor_SysTickConfig 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>They should not be modified by the user.</td>
</tr>
<tr>
<td>USE_FULL_ASSERT</td>
<td>stm32f2xx_conf.h</td>
<td>Default status: disabled</td>
</tr>
</tbody>
</table>

**How to use and customize the library**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>This define statement is used to enable or disable the library run-time failure detection before compiling the firmware library drivers. When enabled, the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&quot;assert_param&quot; macro is expanded in the library drivers code. Run-time detection can be used for user application development and debugging. It adds an overhead which can be removed from the final application code to minimize code size and maximize execution speed.</td>
</tr>
<tr>
<td>I2S_EXTERNAL_CLOCK_VAL</td>
<td>stm32f2xx_conf.h</td>
<td>Default value: 12288000 Hz Default status: disabled If an external clock source is used to drive the I2S clock, this value must be set to the value of the external clock source, otherwise keep this define statement commented.</td>
</tr>
<tr>
<td>Peripheral header file inclusion</td>
<td>stm32f2xx_conf.h</td>
<td>This file allows to enable/disable the inclusion of the peripheral driver header files. By default all header files are included.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#include &quot;stm32f2xx_adc.h&quot; #include &quot;stm32f2xx_can.h&quot; #include &quot;stm32f2xx_crc.h&quot; #include &quot;stm32f2xx_crypt.h&quot; #include &quot;stm32f2xx_dac.h&quot; #include &quot;stm32f2xx_dbgmcu.h&quot; #include &quot;stm32f2xx_dcmi.h&quot; #include &quot;stm32f2xx_dma.h&quot; #include &quot;stm32f2xx_exti.h&quot; #include &quot;stm32f2xx_flash.h&quot; #include &quot;stm32f2xx_fsmc.h&quot; #include &quot;stm32f2xx_hash.h&quot; #include &quot;stm32f2xx_gpio.h&quot; #include &quot;stm32f2xx_i2c.h&quot; #include &quot;stm32f2xx_i2c.h&quot; #include &quot;stm32f2xx_iwdg.h&quot; #include &quot;stm32f2xx_pwr.h&quot; #include &quot;stm32f2xx_rcc.h&quot; #include &quot;stm32f2xx_rng.h&quot; #include &quot;stm32f2xx_rtc.h&quot; #include &quot;stm32f2xx_sdio.h&quot; #include &quot;stm32f2xx_spi.h&quot; #include &quot;stm32f2xx_syscfg.h&quot; #include &quot;stm32f2xx_tim.h&quot; #include &quot;stm32f2xx_usart.h&quot; #include &quot;stm32f2xx_wwdg.h&quot; #include &quot;misc.h&quot;</td>
</tr>
<tr>
<td>USE_STM322xG_EVAL (1)</td>
<td>stm322xg_eval.h</td>
<td>Default status: disabled This define statement is used to include the driver for STM322xG_EVAL board, when used.</td>
</tr>
<tr>
<td>DATA_IN_EXTSRAM</td>
<td>system_stm32f2xx.c</td>
<td>Default status: disabled This define statement enables the use of the external SRAM mounted on STM322xG_EVAL board as data memory</td>
</tr>
</tbody>
</table>
### Parameter

<table>
<thead>
<tr>
<th>Parameter</th>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VECT_TAB_SRAM</td>
<td></td>
<td>Default status: disabled. When enabled, this define statement relocate the vector table in the Internal SRAM.</td>
</tr>
<tr>
<td>VECT_TAB_OFFSET</td>
<td></td>
<td>Default value: 0x00. Defines the vector table base offset. It must be a multiple of 0x200. Use this define statement to build an application that will be loaded at an address different from the Flash memory base address (for example, when building an application to be loaded through in-application programming (IAP) program).</td>
</tr>
</tbody>
</table>

### Notes:

1. These define statements are declared in the compiler preprocessor section of the template projects provided within the library. As a consequence, you do not need to enable them in the corresponding header file.

### 2.2 Library programming model

#### Direct register Access

This model is based on direct register access using the CMSIS layer. This layer provides the definition of all STM32F2xx peripheral registers and bits, as well as memory mapping.

The advantage of this approach if that the code produced is compact and efficient. The drawback is that the developer should know in details the peripheral operation, registers and bits meaning, and the configuration procedure. This task is time consuming, and might lead to programming errors, which may slow down the project development phase.

To use this model, proceed as follows:

1. Comment the line `#define USE_STDPERIPH_DRIVER` in `stm32f2xx.h` file. Make sure that this define statement is not defined in the compiler preprocessor section.
2. Use peripheral registers structure and bits definition available within `stm32f2xx.h` to build the application.

#### Peripheral driver access

In this model the application code uses the peripheral driver API to control the peripheral configuration and operation. It allows any device to be used in the user application without the need for in-depth study of each peripheral specification. As a result, using the peripheral drivers saves significant time that would otherwise be spent in coding, while reducing the application development and integration cost.

However, since the drivers are generic and cover all peripherals functionalities, the size and/or execution speed of the application code may not be optimized.

To use this model, proceed as follows:

1. Add the line `#define USE_STDPERIPH_DRIVER` in the compiler preprocessor section or uncomment the line `#define USE_STDPERIPH_DRIVER` in `stm32f2xx.h`.
2. In `stm32f2xx_conf.h` file, select the peripherals to include their header file (by default all header files are included in the template file).
3. Use the peripheral drivers API provided by `stm32f2xx_ppp.h/.c` files under Libraries/STM32F2xx_StdPeriph_Driver to build your application. For more information, refer to the detailed description of each peripheral driver.
4. In addition to the peripheral drivers, you can reuse/adapt the rich set of examples available within the library. This reduces your application development time and allows you to start within few hours.

For many applications, the peripheral drivers can be used as is. However, for applications having tough constraints in terms of code size and/or execution speed, these drivers should be used as reference on how to configure the peripherals and tailor them to specific application requirements, in combination with peripheral direct register access.

The application code performance in terms of size and/or speed depends also on the C compiler optimization settings. To help you make the application code smaller, faster or balanced between size and speed, fine tune the optimizations according to your application needs. For more information please refer to your C compiler documentation.

2.3 Peripheral initialization and configuration

This section describes step by step how to initialize and configure a peripheral. The peripheral is referred to as PPP.

Before configuring a peripheral, its clock must be enabled by calling one of the following functions:

```c
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_PPPx, ENABLE);
RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_PPPx, ENABLE);
RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_PPPx, ENABLE);
RCC_APB2PeriphClockCmd(RCC_APB2Periph_PPPx, ENABLE);
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PPPx, ENABLE);
```

1. In the main application file, declare a `PPP_InitTypeDef` structure, for example:

```c
PPP_InitTypeDef  PPP_InitStructure;
```

The `PPP_InitTypeDef` is a working variable located in data memory area. It allows to initialize one or more PPP instances.

2. Fill the `PPP_InitTypeDef` variable with the allowed values of the structure member. Two solutions are possible:
   a. Configure the whole structure by following the procedure described below:

   ```c
   PPP_InitTypeDef  PPP_InitStructure;
   PPP_InitStructure.member1 = val1;
   PPP_InitStructure.member2 = val2;
   PPP_InitStructure.memberN = valN;
   /* where N is the number of the structure members */
   ```

   The previous initialization step can be merged in one single line to optimize the code size:

   ```c
   PPP_InitTypeDef  PPP_InitStructure = { val1, val2,.., valN}
   ```

   b. Configure only a few members of the structure: in this case modify the `PPP_InitTypeDef` variable that has been already filled by a call to the `PPP_StructInit(...)` function. This ensures that the other members of the `PPP_InitTypeDef` variable are initialized to the appropriate values (in most cases their default values).

   ```c
   PPP_StructInit(&PPP_InitStructure);
   PPP_InitTypeDef.memberX = valX;
   PPP_InitTypeDef.memberY = valY;
   /*where X and Y are the members the user wants to configure*/
   ```

3. Initialize the PPP peripheral by calling the `PPP_Init(..)` function.

```c
PPP_Init(PPP, &PPP_InitStructure);
```
4. At this stage the PPP peripheral is initialized and can be enabled by making a call to `PPP_Cmd(..)` function.

   ```c
   PPP_Cmd(PPP, ENABLE);
   ```

   The PPP peripheral can then be used through a set of dedicated functions. These functions are specific to the peripheral. For more details refer to the peripheral driver chapter.

   `PPP_DeInit(..)` function can be used to set all PPP peripheral registers to their default values (only for debug purpose):

   ```c
   PPP_DeInit(PPP);
   ```

   To modify the peripheral settings after configuring it, you have to proceed as follows:

   ```c
   PPP_InitStucture.memberX = valX;
   PPP_InitStucture.memberY = valY;
   PPP_Init(PPP, &PPP_InitStucture);
   /* where X and Y are the only members that user wants to modify*/
   ```

2.4 How to run your first example

The library provides a rich set of examples covering the main features of each peripheral. All the examples are independent from the development tools. These examples run on STMicroelectronics STM322xG-EVAL evaluation board and can be easily tailored to any other supported device and development board. Only source files are provided for each example and user can tailor the provided project template to run the selected example with his preferred development Tool.

2.4.1 Prerequisites

1. Latest release of documents and library. You can download the latest version of STM32F2xx related documents and library from STMicroelectronics web site:
   www.st.com/stm32
2. Hardware: to run the examples, you need an STM322xG-EVAL evaluation board from STMicroelectronics or any other compatible hardware.
3. To use your own hardware, simply adapt the example hardware configuration to your platform.
4. Development tools Use your preferred development tool, MDK-ARM (Keil), EWARM (IAR), RIDE (Raisonance), TASKING or TrueSTUDIO (Atollic). Just check that the version you are using supports STM32F2xx devices (see section Section 1.4.2: "Supported development tools and compilers"

2.4.2 Run your first example

This section describes how to load and execute the template example provided within the Library. This example configures the system clock to 120 MHz, initializes the EVAL board LEDs and LCD, then displays a welcome message on the LCD, and finally toggles two LEDs in an infinite loop.

To achieve this goal you have to proceed as described below:

1. Download and unzip the STM32F2xx_StdPeriph_Lib_VX.Y.Z.zip in the folder of your choice
2. Power-up the STM322xG-EVAL board
3. Connect your JTAG probe to the JTAG connector (CN14) of the EVAL board and to the USB port of your PC. The STM322xG-EVAL features a build-in ST-Link/V2 debugger and programmer which makes the external hardware debuggers useless to load and debug your program. Simply select ST-Link/V2 as your debugger in your
Development Tool configuration menu and connect the CN21 to your host PC through an USB cable. Refer to your development tool documentation to know if it supports the ST-Link/V2 debugger.

4. Run the template example: go to STM32F2xx_StdPeriph_Lib_VX.Y.Z\Project\STM32F2xx_StdPeriph_Template folder, and proceed as follows depending on the development tool you are using:
   a. EWARM
      a. Open the EWARM\Project.eww workspace
      b. Rebuild all files: Project->Rebuild all
      c. Load project image: Project->Debug
      d. Run program: Debug->Go(F5)
   b. MDK-ARM
      a. Open the MDK-ARM\Project.uvproj project
      b. Rebuild all files: Project->Rebuild all target files
      c. Load project image: Debug->Start/Stop Debug Session
      d. Run program: Debug->Run (F5)
   c. TrueSTUDIO
      a. Open the TrueSTUDIO development tool.
      b. Click File->Switch Workspace->Other and browse to TrueSTUDIO workspace directory.
      c. Click File->Import, select General->Existing Projects into Workspace and then click Next.
      d. Browse to the TrueSTUDIO workspace directory and select the STM322xG-EVAL project
      e. Rebuild all project files: Select the project in the "Project explorer" window then click on Project->build project menu.
      f. Run program: Select the project in the "Project explorer" window then click Run->Debug (F11)
   d. RIDE
      a. Open the Project.rprj project
      b. Rebuild all files: Project->build project
      c. Load project image: Debug->start(ctrl+D)
      d. Run program: Debug->Run(ctrl+F9)
   e. TASKING
      a. Open the TASKING toolchain.
      b. Click on File->Import, select General->'Existing Projects into Workspace’ and click Next
      c. Browse to TASKING workspace directory and select the STM322xG-EVAL project to configure the project for STM32F2xx devices
      d. Rebuild all project files by selecting the project in the "Project explorer" window and clicking on Project->build project menu
      e. Run the program by selecting the project in the "Project explorer" window and clicking Run->Debug (F11).

If the above sequence has worked correctly, LED1 and LED3 should be ON, LED2 and LED4 should be blinking and the following message is displayed on the LCD screen.
2.4.3 Run a peripheral example

Only the source files of the library peripheral examples are provided. You can tailor the project template provided to run the selected example with your development tool.

As an example, the following sequence is required to run the ADC3_DMA example:

1. Copy all source files from Project\STM32F2xx_StdPeriph_Examples\ADC\ADC3_DMA to the template folder under Project\STM32F2xx_StdPeriph_Template, see Figure 7: "How to run a peripheral example."

2. Open your preferred development tool, and proceed as described in section Section 2.4.2: "Run your first example"

3. If the example use additional source files which are not included in the template project, add manually the files to the project source list. Refer to the readme.txt file of your example for more details.

Figure 6: Message displayed on the LCD when running the template example

Figure 7: How to run a peripheral example
STM32F2xx programming model using the library

This chapter contains useful general information for using the library to develop application based on STM32F2xx devices. It describes in details the sequence to use a peripheral, from the configuration of the system to the configuration of the peripheral registers.

After reset the device is running from Internal High Speed oscillator (HSI 16MHz) with 0 Flash wait state, Flash prefecct buffer, D-Cache and I-Cache disabled, and all peripherals off except internal SRAM, Flash and JTAG:

- There is no prescaler on High speed (AHB) and Low speed (APB) buses. All the peripherals mapped on these buses are running at HSI speed.
- The clock for all peripherals is switched off, except for SRAM and FLASH.
- All GPIOs are in input floating state, except for JTAG pins which are assigned to debug. Once the device started from reset, the user application has to configure the system clock and all peripheral hardware resources (GPIO, Interrupt, DMA…).

**Figure 8: STM32F2xx programming model using the library**

1. **System clock configuration:** the STM32F2xx devices can run at frequency up to 120 MHz and feature several prescalers to configure the AHB, APB1 and APB2 frequencies. The maximum frequency of the AHB domain is 120 MHz. The maximum allowed frequency of the high-speed APB2 domain is 60 MHz, while the maximum allowed frequency of the low speed APB1 domain is 30 MHz. If the application requires higher frequency/performance, follow the sequence below to configure the system clock:
a. Configure the Flash wait state through FLASH_ACR register. For more details refer to Section 12: "FLASH Memory (FLASH)"

b. Select the clock source to be used. Internal (HSI 16MHz) or external (HSE up to 26 MHz).

c. Configure the PLL (optional), system input clock and AHB, APB1 and APB2 prescaler. For more details, refer to Section 19: "Reset and clock control (RCC)". You can use the clock configuration tool (STM32F2xx_Clock_Configuration.xls) to generate a customized system stm32f2xx.c file depending on your application requirements.

2. **Enable the clock for the peripheral(s) to be used**: Before starting to use a peripheral, enable the corresponding interface clock, as well as the clock for the associated GPIOs. This is done by using one of the following functions:

   - RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_PPPx, ENABLE);
   - RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_PPPx, ENABLE);
   - RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_PPPx, ENABLE);
   - RCC_APB2PeriphClockCmd(RCC_APB2Periph_PPPx, ENABLE);
   - RCC_APB1PeriphClockCmd(RCC_APB1Periph_PPPx, ENABLE);

   For example, the following function should be used to enable USART1 interface clock:

   - RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE); For more details, refer to Section 19: "Reset and clock control (RCC)"

3. Configure the clock source(s) for peripherals which clocks are not derived from the System clock:

   a. I2S: STM32F2xx I2S clock can be derived either from a specific PLL (PLLI2S) or from an external clock mapped on the I2S_CKIN pin. For more details, refer to Section 23: "Serial peripheral interface (SPI)"

   b. RTC: STM32F2xx RTC clock can be derived either from a LSI, LSE or HSE clock divided by 2 to 31. For more details, refer to Section 21: "Real-time clock (RTC)"

   c. USB OTG FS and SDIO: in STM32F2xx devices, the USB OTG FS requires a frequency equal to 48 MHz to work correctly, while the SDIO requires a frequency equal or lower than to 48 MHz. For more details, refer to Section 22: "Secure digital input/output interface (SDIO)"

   d. ADC: STM32F2xx ADC features two clock schemes:

      - Clock for the analog circuitry (ADCCLOCK). This clock is common to all ADCs. It is generated from the APB2 clock divided by a programmable prescaler that allows the ADC to work at PCLK2/2, /4, /6 or /8. ADCCLOCK maximum value is 30 MHz when the APB2 clock is 60 MHz. ADCCLOCK is configure through the ADC registers.

      - Clock for the digital interface (used for registers read/write access). This clock is equal to the APB2 clock. The digital interface clock can be enabled/disabled individually for each ADC through the RCC APB2 peripheral clock enable register (RCC_APB2ENR). However, there is only one bit to reset the three ADCs at the same time.

      - For more details, refer to Section 3: "Analog-to-digital converter (ADC)"

4. **Configure the peripheral GPIOs**: Whatever the peripheral mode, the I/Os should be configured as alternate function, before being used as input or output. To configure the I/Os, follow the steps below:

   a. Connect the pin to the desired peripheral alternate function (AF) using GPIO_PinAFConfig() function

   b. Use GPIO_Init() function to configure the I/O pin

      - Configure the desired pin in alternate function mode using GPIO_InitStructure->GPIO_Mode = GPIO_Mode_AF;
      - Select the type, pull-up/pull-down and output speed via GPIO_PuPd, GPIO_OType and GPIO_Speed members. For more details, refer to Section 14: "General-purpose I/Os (GPIO)"
5. **Configure the peripheral in the desired mode:** refer to the peripheral firmware driver section for details on the initialization procedure and how to use the available API. Other modules need to be configured when using interrupt and DMA:
   a. Using the interrupts: after enabling the interrupt source(s) in the peripheral registers, enable the peripheral interrupt line and configure its priority in the NVIC. For more details, refer to Section 28: “Miscellaneous add-on to CMSIS (misc)”
   b. Using the DMA: after enabling the DMA source(s) in the peripheral registers, configure and enable the peripheral DMA channel in the DMA controller. For more details, refer to Section 10: “DMA controller (DMA)”

### 2.6 How to develop your first application

This section describes all steps required for using and customizing the library to build an application from scratch. It gives a real example based on the requirements described below:

- STM322xG-EVAL board used as reference hardware
- System clock configured to 120 MHz, with 3 Flash wait state, Flash prefetch, Instruction cache and Data cache enabled.
- PA0 pin used as EXTI Line0. This pin is connected externally to a pushbutton.
- PG6 and PG8 pins used in output mode to drive LED1 and LED2, respectively.
- LED1 toggles continuously, while LED2 toggles each time the pushbutton is pressed.

#### 2.6.1 Starting point

The typical starting point is the template project provided within the library package (Project/STM32F2xx_StdPeriph_Template). This folder contains all the required template files as well as the project files for different development tools.

Reuse the template files as follow:

- main.c: first move the template main.c file to another location (to backup the template for future use), then create a new empty C file and rename it to main.c. This file will be used to implement the program code as described in the section below.
- stm32f2xx_it.c: use this template file to add the code required to manage the EXTI Line0 interrupt.
- stm32f2xx_it.h: use this template file to add the EXTI Line0 interrupt prototype.
- stm32f2xx_conf.h: use this template file without any change
- system_stm32f2xx.c: use the template file without any change

Follow the steps described in Section 2.5: “STM32F2xx programming model using the library” to develop your application.

#### 2.6.2 Library configuration parameters

To configure the library for your application, use the library default parameters as defined in Table 7: “Library configuration parameters”

#### 2.6.3 system_stm32f2xx.c

This file contains the SystemInit() function that configures the system clock, system clock source, PLL Multiplier and Divider factors, AHB/APBx prescalers and Flash settings. This function is called at startup just after reset and before branch to main program. This call is made inside the “startup_stm32f2xx.s” file.

The clock configuration tool "STM32f2xx_Clock_Configuration.xls" is used to generate system_stm32f2xx.c file that configures the device as follow. The table below shows the default configuration of system_stm32f2xx.c provided within the library:
### Table 8: Default clock configuration in `system_stm32F2xxx.c`

<table>
<thead>
<tr>
<th>System Clock source</th>
<th>PLL (HSESystem Clock source)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSLCK</td>
<td>120000000 Hz</td>
</tr>
<tr>
<td>HCLK</td>
<td>120000000 Hz</td>
</tr>
<tr>
<td>AHB Prescaler</td>
<td>1</td>
</tr>
<tr>
<td>APB1 Prescaler</td>
<td>4</td>
</tr>
<tr>
<td>APB2 Prescaler</td>
<td>2</td>
</tr>
<tr>
<td>HSE Frequency</td>
<td>25000000 Hz</td>
</tr>
<tr>
<td>PLL_M</td>
<td>25</td>
</tr>
<tr>
<td>PLL_N</td>
<td>240</td>
</tr>
<tr>
<td>PLL_P</td>
<td>2</td>
</tr>
<tr>
<td>PLL_Q</td>
<td>5</td>
</tr>
<tr>
<td>VDD</td>
<td>3.3 V</td>
</tr>
<tr>
<td>Flash Latency</td>
<td>3 WS</td>
</tr>
<tr>
<td>Prefetch Buffer</td>
<td>ON</td>
</tr>
<tr>
<td>Prefetch Buffer</td>
<td>ON</td>
</tr>
<tr>
<td>Prefetch Buffer</td>
<td>ON</td>
</tr>
</tbody>
</table>

48 MHz required for USB OTG FS, SDIO and RNG clock

---

**2.6.4 main.c**

The `main.c` file calls the library driver functions to configure the EXTI, GPIO and NIVC peripherals.

Include the library and STM322xG-EVAL board resources:

```c
/* Includes -----------------------------------------------------*/
#include "stm32f2xx.h"  /* The Library entry point */
#include "stm322xg_eval" /* Needed when using STM322xG-EVAL board*/
```

Declare three structure variables, used to initialize the EXTI, GPIO and NIVC peripherals:

```c
/* Private typedef -----------------------------------------------*/
EXTI_InitTypeDef   EXTI_InitStructure;
GPIO_InitTypeDef   GPIO_InitStructure;
NVIC_InitTypeDef   NVIC_InitStructure;
```

Declare prototype for a local function:

```c
/* Private function prototypes ----------------------------------*/
void Delay(__IO uint32_t nCount);
```

The main program will be structured as follow:

```c
/**
 * @brief  Main program.
 * @param  None
 * @retval None
 * /
int main(void)
{
```
1. System Clock configuration:

    /*! At this stage the microcontroller clock setting is already configured, this is done through SystemInit() function which is called from startup file (startup_stm32f2xx.s) before to branch to application main.
    To reconfigure the default setting of SystemInit() function, refer to system_stm32f2xx.c file */

2. Enable the clock for the peripheral(s) to be used (EXTI interface clock is always enabled):

    /* Enable GPIOA's AHB interface clock */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
    /* Enable SYSCFG's APB interface clock */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

3. Configure the peripheral's GPIOs:

    /* Connect EXTI Line0 to PA0 pin */
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);
    /* 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);

4. Configure the peripheral in the desired mode: /* 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);
    /* Enable and set EXTI line0 Interrupt to the lowest priority */
    NVIC_InitStructure NVIC_IRQChannel = EXTI0_IRQn;
    NVIC_InitStructure NVIC_IRQChannelPreemptionPriority = 0x0F;
    NVIC_InitStructure NVIC_IRQChannelSubPriority = 0x0F;
    NVIC_InitStructure NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

5. Insert the code below to use the evaluation board HAL to drive the LEDs:

    /* Initialize LED1 and LED2 mounted on STM322xG-EVAL board */
    STM_EVAL_LEDInit(LED1);
    STM_EVAL_LEDInit(LED2);
    while (1)
    {
        /* Toggle LD1 */
        STM_EVAL_LEDToggle(LED1);
        /* Insert some delay */
        Delay(0xFFFFF);
    }

    /*! @brief Inserts a delay time.  
    * @param nCount: specifies the delay time length.  
    * @retval None  
    */
    void Delay(__IO uint32_t nCount)
for(; nCount != 0; nCount--);

### 2.6.5 stm32f2xx_it.c

The `stm32f2xx_it.c` file can be used to implement the EXTI Line0 interrupt service routine (ISR) in which LED2 toggles each time the ISR is executed.

1. In “STM32F2xx Peripherals Interrupt Handlers” section, add the following code:

```c
/** 
 * @brief This function handles External line 0 interrupt request. 
 * @param None 
 * @retval None 
 */
void EXTI0_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line0) != RESET)
    {
        /* Toggle LED2 */
        STM_EVAL_LEDToggle(LED2);
        /* Clear the EXTI line 0 pending bit */
        EXTI_ClearITPendingBit(EXTI_Line0);
    }
}
```

2. In `stm32f2xx_it.h` file add the EXTI Line0 ISR prototype as follows (just after the line `void SysTick_Handler(void);`)

```c
void EXTI0_IRQHandler(void);
```
3 Analog-to-digital converter (ADC)

3.1 ADC Firmware driver registers structures

3.1.1 ADC_TypeDef

`ADC_TypeDef` is defined in the `stm32f2xx.h` file and contains the ADC peripheral registers definition.

**Data Fields**

- `__IO uint32_t SR`
- `__IO uint32_t CR1`
- `__IO uint32_t CR2`
- `__IO uint32_t SMPR1`
- `__IO uint32_t SMPR2`
- `__IO uint32_t JOFR1`
- `__IO uint32_t JOFR2`
- `__IO uint32_t JOFR3`
- `__IO uint32_t JOFR4`
- `__IO uint32_t HTR`
- `__IO uint32_t LTR`
- `__IO uint32_t SQR1`
- `__IO uint32_t SQR2`
- `__IO uint32_t SQR3`
- `__IO uint32_t JSQR`
- `__IO uint32_t JDR1`
- `__IO uint32_t JDR2`
- `__IO uint32_t JDR3`
- `__IO uint32_t JDR4`
- `__IO uint32_t DR`

**Field Documentation**

- `__IO uint32_t ADC_TypeDef::SR`
  - ADC status register, Address offset: 0x00
- `__IO uint32_t ADC_TypeDef::CR1`
  - ADC control register 1, Address offset: 0x04
- `__IO uint32_t ADC_TypeDef::CR2`
  - ADC control register 2, Address offset: 0x08
- `__IO uint32_t ADC_TypeDef::SMPR1`
  - ADC sample time register 1, Address offset: 0x0C
- `__IO uint32_t ADC_TypeDef::SMPR2`
  - ADC sample time register 2, Address offset: 0x10
- `__IO uint32_t ADC_TypeDef::JOFR1`
  - ADC injected channel data offset register 1, Address offset: 0x14
- `__IO uint32_t ADC_TypeDef::JOFR2`
  - ADC injected channel data offset register 2, Address offset: 0x18
3.1.2 ADC_Common_TypeDef

`ADC_Common_TypeDef` is defined in the `stm32f2xx.h` file and contains the ADC common registers definition.

Data Fields

- __IO uint32_t CSR
- __IO uint32_t CCR
- __IO uint32_t CDR

Field Documentation

- __IO uint32_t ADC_Common_TypeDef::CSR
  - ADC common status register, Address offset: ADC1 base address + 0x300
- __IO uint32_t ADC_Common_TypeDef::CCR
  - ADC common control register, Address offset: ADC1 base address + 0x304
- __IO uint32_t ADC_Common_TypeDef::CDR
  - ADC common regular data register for dual AND triple modes, Address offset: ADC1 base address + 0x308
3.1.3 ADC_InitTypeDef

`ADC_InitTypeDef` is defined in the `stm32f2xx_adc.h` file and contains the ADC initialization parameters. This structure is passed as parameter to `ADC_Init()` function.

Data Fields

- `uint32_t ADC_Resolution`
- `FunctionalState ADC_ScanConvMode`
- `FunctionalState ADC_ContinuousConvMode`
- `uint32_t ADC_ExternalTrigConvEdge`
- `uint32_t ADC_ExternalTrigConv`
- `uint32_t ADC_DataAlign`
- `uint8_t ADC_NbrOfConversion`

Field Documentation

- `uint32_t ADC_InitTypeDef::ADC_Resolution`
  - Configures the ADC resolution dual mode. This parameter can be a value of `ADC_resolution`
- `FunctionalState ADC_InitTypeDef::ADC_ScanConvMode`
  - Specifies whether the conversion is performed in Scan (multichannels) or Single (one channel) mode. This parameter can be set to ENABLE or DISABLE
- `FunctionalState ADC_InitTypeDef::ADC_ContinuousConvMode`
  - Specifies whether the conversion is performed in Continuous or Single mode. This parameter can be set to ENABLE or DISABLE.
- `uint32_t ADC_InitTypeDef::ADC_ExternalTrigConvEdge`
  - Select the external trigger edge and enable the trigger of a regular group. This parameter can be a value of `ADC_external_trigger_edge_for_regular_channels_conversion`
- `uint32_t ADC_InitTypeDef::ADC_ExternalTrigConv`
  - Select the external event used to trigger the start of conversion of a regular group. This parameter can be a value of `ADC_external_trigger_sources_for_regular_channels_conversion`
- `uint32_t ADC_InitTypeDef::ADC_DataAlign`
  - Specifies whether the ADC data alignment is left or right. This parameter can be a value of `ADC_data_align`
- `uint8_t ADC_InitTypeDef::ADC_NbrOfConversion`
  - Specifies the number of ADC conversions that will be done using the sequencer for regular channel group. This parameter must range from 1 to 16.

3.1.4 ADC_CommonInitTypeDef

`ADC_CommonInitTypeDef` is defined in the `stm32f2xx_adc.h` file and contains the common ADC initialization parameters. This structure is passed as parameter to `ADC_CommonInit()` function.

Data Fields

- `uint32_t ADC_Mode`
• `uint32_t ADC_Prescaler`
• `uint32_t ADC_DMAAccessMode`
• `uint32_t ADC_TwoSamplingDelay`

Field Documentation

• `uint32_t ADC_CommonInitTypeDef::ADC_Mode`
  – Configures the ADC to operate in independent or multi mode. This parameter can be a value of `ADC_Common_mode`
• `uint32_t ADC_CommonInitTypeDef::ADC_Prescaler`
  – Select the frequency of the clock to the ADC. The clock is common for all the ADCs. This parameter can be a value of `ADC_Prescaler`
• `uint32_t ADC_CommonInitTypeDef::ADC_DMAAccessMode`
  – Configures the Direct memory access mode for multi ADC mode. This parameter can be a value of `ADC_Direct_memory_access_mode_for_multi_mode`
• `uint32_t ADC_CommonInitTypeDef::ADC_TwoSamplingDelay`
  – Configures the Delay between 2 sampling phases. This parameter can be a value of `ADC_delay_between_2_sampling_phases`

3.2 ADC Firmware driver API description

The following section lists the various functions of the ADC library.

3.2.1 How to use this driver

This section provides informations to use the driver.

1. Enable the ADC interface clock using
   `RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADCx, ENABLE);`
2. ADC pins configuration
   a. Enable the clock for the ADC GPIOs using the following function:
      `RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOx, ENABLE);`
   b. Configure these ADC pins in analog mode using `GPIO_Init()`;
3. Configure the ADC Prescaler, conversion resolution and data alignment using the `ADC_Init()` function.
4. Activate the ADC peripheral using `ADC_Cmd()` function.

   • Regular channels group configuration
     – To configure the ADC regular channels group features, use `ADC_Init()` and `ADC-RegularChannelConfig()` functions.
     – To activate the continuous mode, use the `ADC_continuousModeCmd()` function.
     – To configure and activate the Discontinuous mode, use the `ADC_DiscModeChannelCountConfig()` and `ADC_DiscModeCmd()` functions.
     – To read the ADC converted values, use the `ADC_GetConversionValue()` function.
   • Multi mode ADCs Regular channels configuration
     – Refer to "Regular channels group configuration" description to configure the ADC1, ADC2 and ADC3 regular channels.
     – Select the Multi mode ADC regular channels features (dual or triple mode) using `ADC_CommonInitDef()` function and configure the DMA mode using `ADC_MultiModeDMARequestAfterLastTransferCmd()` functions.
− Read the ADCs converted values using the ADC_GetMultiModeConversionValue() function.
− DMA for Regular channels group features configuration
  − To enable the DMA mode for regular channels group, use the ADC_DMACmd() function.
  − To enable the generation of DMA requests continuously at the end of the last DMA transfer, use the ADC_DMARequestAfterLastTransferCmd() function.
− Injected channels group configuration
  − To configure the ADC Injected channels group features, use ADCInjectedChannelConfig() and ADCInjectedSequencerLengthConfig() functions.
  − To activate the continuous mode, use the ADC_continuousModeCmd() function.
  − To activate the Injected Discontinuous mode, use the ADCInjectedDiscModeCmd() function.
  − To activate the AutoInjected mode, use the ADC_AutoInjectedConvCmd() function.
  − To read the ADC converted values, use the ADC_GetInjectedConversionValue() function.

3.2.2 Initialization and configuration

This section provides functions allowing to:
− Initialize and configure the ADC Prescaler
− ADC Conversion Resolution (12bit..6bit)
− Scan Conversion Mode (multichannels or one channel) for regular group
− ADC Continuous Conversion Mode (Continuous or Single conversion) for regular group
− External trigger Edge and source of regular group,
− Converted data alignment (left or right)
− The number of ADC conversions that will be done using the sequencer for regular channel group
− Multi ADC mode selection
− Direct memory access mode selection for multi ADC mode
− Delay between 2 sampling phases (used in dual or triple interleaved modes)
− Enable or disable the ADC peripheral
− Section 3.2.4.1: "ADC_DeInit"
− Section 3.2.4.2: "ADC_Init"
− Section 3.2.4.3: "ADC_StructInit"
− Section 3.2.4.4: "ADC_CommonInit"
− Section 3.2.4.6: "ADC_Cmd"

Analog Watchdog configuration functions

This section provides functions allowing to configure the Analog Watchdog (AWD) feature in the ADC.

A typical configuration Analog Watchdog is done following these steps :
1. the ADC guarded channel(s) is (are) selected using the ADC_AnalogWatchdogSingleChannelConfig() function
2. The Analog watchdog lower and higher threshold are configured using the ADC_AnalogWatchdogThresholdsConfig() function.
3. The Analog watchdog is enabled and configured to enable the check, on one or more channels, using the ADC_AnalogWatchdogCmd() function.

A typical configuration Analog Watchdog is done following these steps:

1. the ADC guarded channel(s) is (are) selected using the ADC_AnalogWatchdogSingleChannelConfig() function.
2. The Analog watchdog lower and higher threshold are configured using the ADC_AnalogWatchdogThresholdsConfig() function.
3. The Analog watchdog is enabled and configured to enable the check, on one or more channels, using the ADC_AnalogWatchdogCmd() function.

- ADC_AnalogWatchdogCmd()
- ADC_AnalogWatchdogThresholdsConfig()
- ADC_AnalogWatchdogSingleChannelConfig()

Temperature Sensor, Vrefint (Voltage Reference internal)

- ADC_TempSensorVrefintCmd()
- ADC_VBATCmd()

Regular Channels Configuration functions

This section provides functions allowing to manage the ADC's regular channels, it is composed of 2 sub sections:

1. Configuration and management functions for regular channels: This subsection provides functions allowing to configure the ADC regular channels: Please Note that the following features for regular channels are configured using the ADC_Init() function: scan mode activation continuous mode activation External trigger source External trigger edge number of conversion in the regular channels group sequencer. () and () are performing the same configuration
   - Configure the rank in the regular group sequencer for each channel
   - Configure the sampling time for each channel
   - Select the conversion Trigger for regular channels
   - Select the desired EOC event behavior configuration
   - Activate the continuous Mode
   - Activate the Discontinuous Mode

2. Get the conversion data: This subsection provides an important function in the ADC peripheral since it returns the converted data of the current regular channel. When the Conversion value is read, the EOC Flag is automatically cleared. For multi ADC mode, the last ADC1, ADC2 and ADC3 regular conversions results data (in the selected multi mode) can be returned in the same time using ADC_GetMultiModeConversionValue() function.

- ADC-RegularChannelConfig()
- ADC_SoftwareStartConv()
- ADC_GetSoftwareStartConvStatus()
- ADC_EOCOnEachRegularChannelCmd()
- ADC_ContinuousModeCmd()
- ADC_DiscModeChannelCountConfig()
- ADC_DiscModeCmd()
- ADC_GetConversionValue()
- ADC_GetMultiModeConversionValue()

Regular Channels DMA Configuration functions

This section provides functions allowing to configure the DMA for ADC regular channels.
Since converted regular channel values are stored into a unique data register, it is useful to use DMA for conversion of more than one regular channel. This avoids the loss of the data already stored in the ADC Data register.

When the DMA mode is enabled (using the ADC_DMACmd() function), after each conversion of a regular channel, a DMA request is generated.

Depending on the "DMA disable selection for Independent ADC mode" configuration (using the ADC_DMARequestAfterLastTransferCmd() function), at the end of the last DMA transfer, two possibilities are allowed:

- No new DMA request is issued to the DMA controller (feature DISABLED)
- Requests can continue to be generated (feature ENABLED).

Depending on the "DMA disable selection for multi ADC mode" configuration (using the void ADC_MultiModeDMARequestAfterLastTransferCmd() function), at the end of the last DMA transfer, two possibilities are allowed:

- No new DMA request is issued to the DMA controller (feature DISABLED)
- Requests can continue to be generated (feature ENABLED).

### Injected channels Configuration functions

This section provides functions allowing to configure the ADC Injected channels, it is composed of 2 sub sections :

1. Configuration functions for Injected channels: This subsection provides functions allowing to configure the ADC injected channels:
   - Configure the rank in the injected group sequencer for each channel
   - Configure the sampling time for each channel
   - Activate the Auto injected Mode
   - Activate the Discontinuous Mode
   - Scan mode activation
   - External/software trigger source
   - External trigger edge
   - Injected channels sequencer.

2. Get the Specified Injected channel conversion data: This subsection provides an important function in the ADC peripheral since it returns the converted data of the specific injected channel.
   - ADCInjectedChannelConfig()
   - ADCInjectedSequencerLengthConfig()
   - ADCSetInjectedOffset()
   - ADCExternalTrigInjectedConvConfig()
   - ADCExternalTrigInjectedConvEdgeConfig()
   - ADCSoftwareStartInjectedConv()
   - ADCGetSoftwareStartInjectedConvCmdStatus()
   - ADCAutoInjectedConvCmd()
   - ADCInjectedDiscModeCmd()
   - ADCGetInjectedConversionValue()

### 3.2.3 Interrupt and flag management

This section provides functions allowing to configure the ADC Interrupts and to get the status and clear flags and Interrupts pending bits.
Each ADC provides 4 Interrupts sources and 6 Flags which can be divided into 3 groups:

1. Flags and Interrupts for ADC regular channels Flags:
   Interrupts:
   a. ADC_FLAG_OVR: Overrun detection when regular converted data are lost
   b. ADC_FLAG_EOC: Regular channel end of conversion ==> to indicate (depending on EOCS bit, managed by ADC_EOCOnEachRegularChannelCmd()) the end of: ==> a regular CHANNEL conversion ==> sequence of regular GROUP conversions
   c. ADC_FLAG_STRT: Regular channel start ==> to indicate when regular CHANNEL conversion starts.
   d. ADC_IT_OVR: specifies the interrupt source for Overrun detection event.
   e. ADC_IT_EOC: specifies the interrupt source for Regular channel end of conversion event.

2. Flags and Interrupts for ADC Injected channels Flags:
   Interrupts:
   a. ADC_FLAG_JEOC: Injected channel end of conversion ==> to indicate at the end of injected GROUP conversion
   b. ADC_FLAG_JSTRT: Injected channel start ==> to indicate when injected GROUP conversion starts.
   c. ADC_IT_JEOC: specifies the interrupt source for Injected channel end of conversion event.

3. General Flags and Interrupts for the ADC Flags:
   Interrupts:
   a. ADC_FLAG_AWD: Analog watchdog ==> to indicate if the converted voltage crosses the programmed thresholds values.
   b. ADC_IT_AWD: specifies the interrupt source for Analog watchdog event.

The user should identify which mode will be used in his application to manage the ADC controller events: Polling mode or Interrupt mode.

In the Polling Mode it is advised to use the following functions:
- ADC_GetFlagStatus(): to check if flags events occur.
- ADC_ClearFlag(): to clear the flags events. In the Interrupt Mode it is advised to use the following functions:
- ADC_ITConfig(): to enable or disable the interrupt source.
- ADC_GetITStatus(): to check if Interrupt occurs.
- ADC_ClearITPendingBit(): to clear the Interrupt pending Bit (corresponding Flag).
- ADC_ITConfig()
- ADC_GetFlagStatus()
- ADC_ClearFlag()
- ADC_GetITStatus()
- ADC_ClearITPendingBit()

### 3.2.4 Initialization and configuration functions

#### 3.2.4.1 ADC_DeInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void ADC_DeInit ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Deinitializes all ADCs peripherals registers to their default reset values.</td>
</tr>
<tr>
<td>Parameters</td>
<td>None.</td>
</tr>
</tbody>
</table>
3.2.4.2  ADC_Init

Function Name  void ADC_Init ( ADC_TypeDef * ADCx, ADC_InitTypeDef * ADC_InitStruct)

Function Description  Initializes the ADCx peripheral according to the specified parameters in the ADC_InitStruct.

Parameters
- ADCx : where x can be 1, 2 or 3 to select the ADC peripheral.
- ADC_InitStruct : pointer to an ADC_InitTypeDef structure that contains the configuration information for the specified ADC peripheral.

Return values  None.

Notes
- This function is used to configure the global features of the ADC (Resolution and Data Alignment), however, the rest of the configuration parameters are specific to the regular channels group (scan mode activation, continuous mode activation, External trigger source and edge, number of conversion in the regular channels group sequencer).

3.2.4.3  ADC_StructInit

Function Name  void ADC_StructInit ( ADC_InitTypeDef * ADC_InitStruct)

Function Description  Fills each ADC_InitStruct member with its default value.

Parameters
- ADC_InitStruct : pointer to an ADC_InitTypeDef structure which will be initialized.

Return values  None.

Notes
- This function is used to initialize the global features of the ADC (Resolution and Data Alignment), however, the rest of the configuration parameters are specific to the regular channels group (scan mode activation, continuous mode activation, External trigger source and edge, number of conversion in the regular channels group sequencer).
3.2.4.4 ADC_CommonInit

Function Name: void ADC_CommonInit (ADC_CommonInitTypeDef *ADC_CommonInitStruct)

Function Description: Initializes the ADCs peripherals according to the specified parameters in the ADC_CommonInitStruct.

Parameters:
- ADC_CommonInitStruct: pointer to an ADC_CommonInitTypeDef structure that contains the configuration information for all ADCs peripherals.

Return values:
- None.

Notes:
- None.

3.2.4.5 ADC_CommonStructInit

Function Name: void ADC_CommonStructInit (ADC_CommonInitTypeDef *ADC_CommonInitStruct)

Function Description: Fills each ADC_CommonInitStruct member with its default value.

Parameters:
- ADC_CommonInitStruct: pointer to an ADC_CommonInitTypeDef structure which will be initialized.

Return values:
- None.

Notes:
- None.

3.2.4.6 ADC_Cmd

Function Name: void ADC_Cmd (ADC_TypeDef *ADCx, FunctionalState NewState)

Function Description: Enables or disables the specified ADC peripheral.

Parameters:
- ADCx: where x can be 1, 2 or 3 to select the ADC peripheral.
- NewState: new state of the ADCx peripheral. This
3.2.5 Analog Watchdog configuration functions

3.2.5.1 ADC_AnalogWatchdogCmd

Function Name
void ADC_AnalogWatchdogCmd ( ADC_TypeDef * ADCx, uint32_t ADC_AnalogWatchdog)

Function Description
Enables or disables the analog watchdog on single/all regular or injected channels.

Parameters
- ADCx : where x can be 1, 2 or 3 to select the ADC peripheral.
- ADC_AnalogWatchdog : the ADC analog watchdog configuration. This parameter can be one of the following values:
  - ADC_AnalogWatchdog_SingleRegEnable : Analog watchdog on a single regular channel
  - ADC_AnalogWatchdog_SingleInjecEnable : Analog watchdog on a single injected channel
  - ADC_AnalogWatchdog_SingleRegOrInjecEnable : Analog watchdog on a single regular or injected channel
  - ADC_AnalogWatchdog_AllRegEnable : Analog watchdog on all regular channel
  - ADC_AnalogWatchdog_AllInjecEnable : Analog watchdog on all injected channel
  - ADC_AnalogWatchdog_AllRegAllInjecEnable : Analog watchdog on all regular and injected channels
  - ADC_AnalogWatchdog_None : No channel guarded by the analog watchdog

Return values
- None.

Notes
- None.

3.2.5.2 ADC_AnalogWatchdogThresholdsConfig

Function Name
void ADC_AnalogWatchdogThresholdsConfig ( ADC_TypeDef
* ADCx, uint16_t HighThreshold, uint16_t LowThreshold*

**Function Description**
Configures the high and low thresholds of the analog watchdog.

**Parameters**
- **ADCx** : where x can be 1, 2 or 3 to select the ADC peripheral.
- **HighThreshold** : the ADC analog watchdog High threshold value. This parameter must be a 12-bit value.
- **LowThreshold** : the ADC analog watchdog Low threshold value. This parameter must be a 12-bit value.

**Return values**
- None.

**Notes**
- None.

### 3.2.5.3 ADC_AnalogWatchdogSingleChannelConfig

**Function Name**
void ADC_AnalogWatchdogSingleChannelConfig (ADC_TypeDef * ADCx, uint8_t ADC_Channel)

**Function Description**
Configures the analog watchdog guarded single channel.

**Parameters**
- **ADCx** : where x can be 1, 2 or 3 to select the ADC peripheral.
- **ADC_Channel** : the ADC channel to configure for the analog watchdog. This parameter can be one of the following values:
  - **ADC_Channel_0** : ADC Channel0 selected
  - **ADC_Channel_1** : ADC Channel1 selected
  - **ADC_Channel_2** : ADC Channel2 selected
  - **ADC_Channel_3** : ADC Channel3 selected
  - **ADC_Channel_4** : ADC Channel4 selected
  - **ADC_Channel_5** : ADC Channel5 selected
  - **ADC_Channel_6** : ADC Channel6 selected
  - **ADC_Channel_7** : ADC Channel7 selected
  - **ADC_Channel_8** : ADC Channel8 selected
  - **ADC_Channel_9** : ADC Channel9 selected
  - **ADC_Channel_10** : ADC Channel10 selected
  - **ADC_Channel_11** : ADC Channel11 selected
  - **ADC_Channel_12** : ADC Channel12 selected
  - **ADC_Channel_13** : ADC Channel13 selected
  - **ADC_Channel_14** : ADC Channel14 selected
  - **ADC_Channel_15** : ADC Channel15 selected
  - **ADC_Channel_16** : ADC Channel16 selected
  - **ADC_Channel_17** : ADC Channel17 selected
  - **ADC_Channel_18** : ADC Channel18 selected

**Return values**
- None.

**Notes**
- None.
3.2.6 Temperature Sensor, Vrefint (Voltage Reference internal)

3.2.6.1 ADC_TempSensorVrefintCmd

Function Name: void ADC_TempSensorVrefintCmd (FunctionalState NewState)
Function Description: Enables or disables the temperature sensor and Vrefint channels.
Parameters:
- NewState: new state of the temperature sensor and Vrefint channels. This parameter can be: ENABLE or DISABLE.
Return values:
- None.
Notes:
- None.

3.2.6.2 ADC_VBATCmd

Function Name: void ADC_VBATCmd (FunctionalState NewState)
Function Description: Enables or disables the VBAT (Voltage Battery) channel.
Parameters:
- NewState: new state of the VBAT channel. This parameter can be: ENABLE or DISABLE.
Return values:
- None.
Notes:
- None.

3.2.7 Regular Channels Configuration functions

3.2.7.1 ADC-RegularChannelConfig

Function Name: void ADC-RegularChannelConfig (ADC_TypeDef *ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime)
Function Description: Configures for the selected ADC regular channel its corresponding rank in the sequencer and its sample time.
### Parameters

- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **ADC_Channel**: the ADC channel to configure. This parameter can be one of the following values:
  - `ADC_Channel_0`: ADC Channel0 selected
  - `ADC_Channel_1`: ADC Channel1 selected
  - `ADC_Channel_2`: ADC Channel2 selected
  - `ADC_Channel_3`: ADC Channel3 selected
  - `ADC_Channel_4`: ADC Channel4 selected
  - `ADC_Channel_5`: ADC Channel5 selected
  - `ADC_Channel_6`: ADC Channel6 selected
  - `ADC_Channel_7`: ADC Channel7 selected
  - `ADC_Channel_8`: ADC Channel8 selected
  - `ADC_Channel_9`: ADC Channel9 selected
  - `ADC_Channel_10`: ADC Channel10 selected
  - `ADC_Channel_11`: ADC Channel11 selected
  - `ADC_Channel_12`: ADC Channel12 selected
  - `ADC_Channel_13`: ADC Channel13 selected
  - `ADC_Channel_14`: ADC Channel14 selected
  - `ADC_Channel_15`: ADC Channel15 selected
  - `ADC_Channel_16`: ADC Channel16 selected
  - `ADC_Channel_17`: ADC Channel17 selected
  - `ADC_Channel_18`: ADC Channel18 selected

- **Rank**: The rank in the regular group sequencer. This parameter must be between 1 and 16.
- **ADC_SampleTime**: The sample time value to be set for the selected channel. This parameter can be one of the following values:
  - `ADC_SampleTime_3Cycles`: Sample time equal to 3 cycles
  - `ADC_SampleTime_15Cycles`: Sample time equal to 15 cycles
  - `ADC_SampleTime_28Cycles`: Sample time equal to 28 cycles
  - `ADC_SampleTime_56Cycles`: Sample time equal to 56 cycles
  - `ADC_SampleTime_84Cycles`: Sample time equal to 84 cycles
  - `ADC_SampleTime_112Cycles`: Sample time equal to 112 cycles
  - `ADC_SampleTime_144Cycles`: Sample time equal to 144 cycles
  - `ADC_SampleTime_480Cycles`: Sample time equal to 480 cycles

### Return values
- None.

### Notes
- None.
3.2.7.2  ADC_SoftwareStartConv

Function Name  void ADC_SoftwareStartConv (ADC_TypeDef * ADCx)
Function Description  Enables the selected ADC software start conversion of the regular channels.
Parameters  • ADCx : where x can be 1, 2 or 3 to select the ADC peripheral.
Return values  • None.
Notes  • None.

3.2.7.3  ADC_GetSoftwareStartConvStatus

Function Name  FlagStatus ADC_GetSoftwareStartConvStatus (ADC_TypeDef * ADCx)
Function Description  Gets the selected ADC Software start regular conversion Status.
Parameters  • ADCx : where x can be 1, 2 or 3 to select the ADC peripheral.
Return values  • The new state of ADC software start conversion (SET or RESET).
Notes  • None.

3.2.7.4  ADC_EOCOnEachRegularChannelCmd

Function Name  void ADC_EOCOnEachRegularChannelCmd (ADC_TypeDef * ADCx, FunctionalState NewState)
Function Description  Enables or disables the EOC on each regular channel conversion.
Parameters  • ADCx : where x can be 1, 2 or 3 to select the ADC peripheral.
• NewState : new state of the selected ADC EOC flag rising
  This parameter can be: ENABLE or DISABLE.
Return values  • None.
3.2.7.5 ADC_ContinuousModeCmd

Function Name  void ADC_ContinuousModeCmd (ADC_TypeDef * ADCx, FunctionalState NewState)
Function Description Enables or disables the ADC continuous conversion mode.
Parameters  
- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **NewState**: new state of the selected ADC continuous conversion mode. This parameter can be: ENABLE or DISABLE.

Return values  • None.
Notes  • None.

3.2.7.6 ADC_DiscModeChannelCountConfig

Function Name  void ADC_DiscModeChannelCountConfig (ADC_TypeDef * ADCx, uint8_t Number)
Function Description Configures the discontinuous mode for the selected ADC regular group channel.
Parameters  
- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **Number**: specifies the discontinuous mode regular channel count value. This number must be between 1 and 8.

Return values  • None.
Notes  • None.

3.2.7.7 ADC_DiscModeCmd
Analog-to-digital converter (ADC)

### 3.2.7.8 ADC_GetConversionValue

**Function Name**
```c
uint16_t ADC_GetConversionValue (ADC_TypeDef *ADCx)
```

**Function Description**
Returns the last ADCx conversion result data for regular channel.

**Parameters**
- ADCx : where x can be 1, 2 or 3 to select the ADC peripheral.

**Return values**
- The Data conversion value.

**Notes**
- None.

### 3.2.7.9 ADC_GetMultiModeConversionValue

**Function Name**
```c
uint32_t ADC_GetMultiModeConversionValue (void)
```

**Function Description**
Returns the last ADC1, ADC2 and ADC3 regular conversions results data in the selected multi mode.

**Parameters**
- None.

**Return values**
- The Data conversion value.

**Notes**
- In dual mode, the value returned by this function is as following Data[15:0]: these bits contain the regular data of ADC1. Data[31:16]: these bits contain the regular data of ADC2.
- In triple mode, the value returned by this function is as following Data[15:0]: these bits contain alternatively the
3.2.8 Regular Channels DMA Configuration functions

3.2.8.1 ADC_DMACmd

Function Name: void ADC_DMACmd (ADC_TypeDef *ADCx, FunctionalState NewState)

Function Description: Enables or disables the specified ADC DMA request.

Parameters:
- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **NewState**: new state of the selected ADC DMA transfer. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.

3.2.8.2 ADC_DMARequestAfterLastTransferCmd

Function Name: void ADC_DMARequestAfterLastTransferCmd (ADC_TypeDef *ADCx, FunctionalState NewState)

Function Description: Enables or disables the ADC DMA request after last transfer (Single-ADC mode)

Parameters:
- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **NewState**: new state of the selected ADC DMA request after last transfer. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.
3.2.8.3 ADC_MultiModeDMARequestAfterLastTransferCmd

Function Name: void ADC_MultiModeDMARequestAfterLastTransferCmd (FunctionalState NewState)

Function Description: Enables or disables the ADC DMA request after last transfer in multi ADC mode.

Parameters:
- **NewState**: new state of the selected ADC DMA request after last transfer. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- if Enabled, DMA requests are issued as long as data are converted and DMA mode for multi ADC mode (selected using ADC_CommonInit() function by ADC_CommonInitStruct.ADC_DMAAccessMode structure member) is ADC_DMAAccessMode_1, ADC_DMAAccessMode_2 or ADC_DMAAccessMode_3.

3.2.9 Injected channels Configuration functions

3.2.9.1 ADCInjectedChannelConfig

Function Name: void ADCInjectedChannelConfig (ADC_TypeDef * ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime)

Function Description: Configures for the selected ADC injected channel its corresponding rank in the sequencer and its sample time.

Parameters:
- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **ADC_Channel**: the ADC channel to configure. This parameter can be one of the following values:
  - **ADC_Channel_0**: ADC Channel0 selected
  - **ADC_Channel_1**: ADC Channel1 selected
  - **ADC_Channel_2**: ADC Channel2 selected
  - **ADC_Channel_3**: ADC Channel3 selected
  - **ADC_Channel_4**: ADC Channel4 selected
  - **ADC_Channel_5**: ADC Channel5 selected
  - **ADC_Channel_6**: ADC Channel6 selected
  - **ADC_Channel_7**: ADC Channel7 selected
  - **ADC_Channel_8**: ADC Channel8 selected
  - **ADC_Channel_9**: ADC Channel9 selected
  - **ADC_Channel_10**: ADC Channel10 selected
  - **ADC_Channel_11**: ADC Channel11 selected
- **ADC_Channel_12**: ADC Channel12 selected
- **ADC_Channel_13**: ADC Channel13 selected
- **ADC_Channel_14**: ADC Channel14 selected
- **ADC_Channel_15**: ADC Channel15 selected
- **ADC_Channel_16**: ADC Channel16 selected
- **ADC_Channel_17**: ADC Channel17 selected
- **ADC_Channel_18**: ADC Channel18 selected

- **Rank**: The rank in the injected group sequencer. This parameter must be between 1 and 4.
- **ADC_SampleTime**: The sample time value to be set for the selected channel. This parameter can be one of the following values:
  - **ADC_SampleTime_3Cycles**: Sample time equal to 3 cycles
  - **ADC_SampleTime_15Cycles**: Sample time equal to 15 cycles
  - **ADC_SampleTime_28Cycles**: Sample time equal to 28 cycles
  - **ADC_SampleTime_56Cycles**: Sample time equal to 56 cycles
  - **ADC_SampleTime_84Cycles**: Sample time equal to 84 cycles
  - **ADC_SampleTime_112Cycles**: Sample time equal to 112 cycles
  - **ADC_SampleTime_144Cycles**: Sample time equal to 144 cycles
  - **ADC_SampleTime_480Cycles**: Sample time equal to 480 cycles

**Return values**
- None.

**Notes**
- None.

### 3.2.9.2 ADCInjectedSequencerLengthConfig

**Function Name**

```c
void ADC_InjectedSequencerLengthConfig (ADC_TypeDef *ADCx, uint8_t Length)
```

**Function Description**

Configures the sequencer length for injected channels.

**Parameters**

- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **Length**: The sequencer length. This parameter must be a number between 1 and 4.

**Return values**
- None.

**Notes**
- None.
3.2.9.3 ADC_SetInjectedOffset

Function Name
void ADC_SetInjectedOffset (ADC_TypeDef *ADCx, uint8_t ADCInjectedChannel, uint16_t Offset)

Function Description
Set the injected channels conversion value offset.

Parameters
- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **ADCInjectedChannel**: the ADC injected channel to set its offset. This parameter can be one of the following values:
  - **ADCInjectedChannel_1**: Injected Channel1 selected
  - **ADCInjectedChannel_2**: Injected Channel2 selected
  - **ADCInjectedChannel_3**: Injected Channel3 selected
  - **ADCInjectedChannel_4**: Injected Channel4 selected
- **Offset**: the offset value for the selected ADC injected channel. This parameter must be a 12bit value.

Return values
- None.

Notes
- None.

3.2.9.4 ADC_ExternalTrigInjectedConvConfig

Function Name
void ADC_ExternalTrigInjectedConvConfig (ADC_TypeDef *ADCx, uint32_t ADC_ExternalTrigInjecConv)

Function Description
Configures the ADCx external trigger for injected channels conversion.

Parameters
- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **ADC_ExternalTrigInjecConv**: specifies the ADC trigger to start injected conversion. This parameter can be one of the following values:
  - **ADC_ExternalTrigInjecConv_T1_CC4**: Timer1 capture compare4 selected
  - **ADC_ExternalTrigInjecConv_T1_TRGO**: Timer1 TRGO event selected
  - **ADC_ExternalTrigInjecConv_T2_CC1**: Timer2 capture compare1 selected
  - **ADC_ExternalTrigInjecConv_T2_TRGO**: Timer2 TRGO event selected
  - **ADC_ExternalTrigInjecConv_T3_CC2**: Timer3
capture compare2 selected

- `ADC_EveningInjectedConv_T3_CC4`: Timer3
capture compare4 selected

- `ADC_EveningInjectedConv_T4_CC1`: Timer4
capture compare1 selected

- `ADC_EveningInjectedConv_T4_CC2`: Timer4
capture compare2 selected

- `ADC_EveningInjectedConv_T4_CC3`: Timer4
capture compare3 selected

- `ADC_EveningInjectedConv_T4_TRGO`: Timer4
TRGO event selected

- `ADC_EveningInjectedConv_T5_CC4`: Timer5
capture compare4 selected

- `ADC_EveningInjectedConv_T5_TRGO`: Timer5
TRGO event selected

- `ADC_EveningInjectedConv_T8_CC2`: Timer8
capture compare2 selected

- `ADC_EveningInjectedConv_T8_CC3`: Timer8
capture compare3 selected

- `ADC_EveningInjectedConv_T8_CC4`: Timer8
capture compare4 selected

- `ADC_EveningInjectedConv_Ext_IT15`: External interrupt line 15 event selected

Return values

- None.

Notes

- None.

3.2.9.5 `ADC_EveningInjectedConvEdgeConfig`

Function Name

`void ADC_EveningInjectedConvEdgeConfig (ADC_TypeDef * ADCx, uint32_t ADC_EveningInjectedConvEdge)`

Function Description

Configures the ADCx external trigger edge for injected channels conversion.

Parameters

- `ADCx`: where x can be 1, 2 or 3 to select the ADC peripheral.
- `ADC_EveningInjectedConvEdge`: specifies the ADC external trigger edge to start injected conversion. This parameter can be one of the following values:
  - `ADC_EveningInjectedConvEdge_None`: external trigger disabled for injected conversion
  - `ADC_EveningInjectedConvEdge_Rising`: detection on rising edge
  - `ADC_EveningInjectedConvEdge_Falling`: detection on falling edge
  - `ADC_EveningInjectedConvEdge_RisingFalling`:
Analog-to-digital converter (ADC)

3.2.9.6 ADC_SoftwareStartInjectedConv

- **Function Name**: void ADC_SoftwareStartInjectedConv (ADC_TypeDef *ADCx)
- **Function Description**: Enables the selected ADC software start conversion of the injected channels.
- **Parameters**
  - ADCx : where x can be 1, 2 or 3 to select the ADC peripheral.
- **Return values**
  - None.
- **Notes**
  - None.

3.2.9.7 ADC_GetSoftwareStartInjectedConvCmdStatus

- **Function Name**: FlagStatus ADC_GetSoftwareStartInjectedConvCmdStatus (ADC_TypeDef *ADCx)
- **Function Description**: Gets the selected ADC Software start injected conversion Status.
- **Parameters**
  - ADCx : where x can be 1, 2 or 3 to select the ADC peripheral.
- **Return values**
  - The new state of ADC software start injected conversion (SET or RESET).
- **Notes**
  - None.

3.2.9.8 ADC_AutoInjectedConvCmd
Function Name: void ADC_AutoInjectedConvCmd (ADC_TypeDef * ADCx, FunctionalState NewState)

Function Description: Enables or disables the selected ADC automatic injected group conversion after regular one.

Parameters:
- ADCx: where x can be 1, 2 or 3 to select the ADC peripheral.
- NewState: new state of the selected ADC auto injected conversion. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

3.2.9.9 ADC_InjectedDiscModeCmd

Function Name: void ADC_InjectedDiscModeCmd (ADC_TypeDef * ADCx, FunctionalState NewState)

Function Description: Enables or disables the discontinuous mode for injected group channel for the specified ADC.

Parameters:
- ADCx: where x can be 1, 2 or 3 to select the ADC peripheral.
- NewState: new state of the selected ADC discontinuous mode on injected group channel. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

3.2.9.10 ADC_GetInjectedConversionValue

Function Name: uint16_t ADC_GetInjectedConversionValue (ADC_TypeDef * ADCx, uint8_t ADC_InjectedChannel)

Function Description: Returns the ADC injected channel conversion result.

Parameters:
- ADCx: where x can be 1, 2 or 3 to select the ADC peripheral.
- ADC_InjectedChannel: the converted ADC injected channel. This parameter can be one of the following values:
  - ADC_InjectedChannel_1: Injected Channel1 selected
Analog-to-digital converter (ADC)

- **ADCInjectedChannel_2**: Injected Channel 2 selected
- **ADCInjectedChannel_3**: Injected Channel 3 selected
- **ADCInjectedChannel_4**: Injected Channel 4 selected

**Return values**
- The Data conversion value.

**Notes**
- None.

### 3.2.10 Interrupt and flag management functions

#### 3.2.10.1 ADC_ITConfig

**Function Name**
void ADC_ITConfig (ADC_TypeDef *ADCx, uint16_t ADC_IT, FunctionalState NewState)

**Function Description**
Enables or disables the specified ADC interrupts.

**Parameters**
- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **ADC_IT**: specifies the ADC interrupt sources to be enabled or disabled. This parameter can be one of the following values:
  - **ADC_IT_EOC**: End of conversion interrupt mask
  - **ADC_IT_AWD**: Analog watchdog interrupt mask
  - **ADC_IT_JEOC**: End of injected conversion interrupt mask
  - **ADC_IT_OVR**: Overrun interrupt enable
- **NewState**: new state of the specified ADC interrupts. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.

**Notes**
- None.

#### 3.2.10.2 ADC_GetFlagStatus

**Function Name**
FlagStatus ADC_GetFlagStatus (ADC_TypeDef *ADCx, uint8_t ADC_FLAG)

**Function Description**
Checks whether the specified ADC flag is set or not.

**Parameters**
- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **ADC_FLAG**: specifies the flag to check. This parameter can
3.2.10.3  ADC_ClearFlag

Function Name  void ADC_ClearFlag ( ADC_TypeDef * ADCx, uint8_t ADC_FLAG)

Function Description  Clears the ADCx’s pending flags.

Parameters  
- ADCx : where x can be 1, 2 or 3 to select the ADC peripheral.
- ADC_FLAG : specifies the flag to clear. This parameter can be any combination of the following values:
  - ADC_FLAG_AWD : Analog watchdog flag
  - ADC_FLAG_EOC : End of conversion flag
  - ADC_FLAG_JEOC : End of injected group conversion flag
  - ADC_FLAG_JSTRT : Start of injected group conversion flag
  - ADC_FLAG_STRT : Start of regular group conversion flag
  - ADC_FLAG_OVR : Overrun flag

Return values  • None.

Notes  • None.
**Function Description**
Checks whether the specified ADC interrupt has occurred or not.

**Parameters**
- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **ADC_IT**: specifies the ADC interrupt source to check. This parameter can be one of the following values:
  - **ADC_IT_EOC**: End of conversion interrupt mask
  - **ADC_IT_AWD**: Analog watchdog interrupt mask
  - **ADC_IT_JEOC**: End of injected conversion interrupt mask
  - **ADC_IT_OVR**: Overrun interrupt mask

**Return values**
- The new state of ADC_IT (SET or RESET).

**Notes**
- None.

---

### 3.2.10.5 ADC_ClearITPendingBit

**Function Name**
```c
void ADC_ClearITPendingBit (ADC_TypeDef *ADCx, uint16_t ADC_IT);
```

**Function Description**
Clears the ADCx's interrupt pending bits.

**Parameters**
- **ADCx**: where x can be 1, 2 or 3 to select the ADC peripheral.
- **ADC_IT**: specifies the ADC interrupt pending bit to clear. This parameter can be one of the following values:
  - **ADC_IT_EOC**: End of conversion interrupt mask
  - **ADC_IT_AWD**: Analog watchdog interrupt mask
  - **ADC_IT_JEOC**: End of injected conversion interrupt mask
  - **ADC_IT_OVR**: Overrun interrupt mask

**Return values**
- None.

**Notes**
- None.

---

### 3.3 ADC Firmware driver defines

#### 3.3.1 ADC Firmware driver defines

**ADC_analog_watchdog_selection**

- define: `ADC_AnalogWatchdog_SingleRegEnable((uint32_t)0x00800200)`
- #define: `ADC_AnalogWatchdog_SingleInjecEnable((uint32_t)0x00400200)`

- #define: `ADC_AnalogWatchdog_SingleRegOrInjecEnable((uint32_t)0x00C00200)`

- #define: `ADC_AnalogWatchdog_AllRegEnable((uint32_t)0x00800000)`

- #define: `ADC_AnalogWatchdog_AllInjecEnable((uint32_t)0x00400000)`

- #define: `ADC_AnalogWatchdog_AllRegAllInjecEnable((uint32_t)0x00C00000)`

- #define: `ADC_AnalogWatchdog_None((uint32_t)0x00000000)`

**ADC_channels**

- #define: `ADC_Channel_0((uint8_t)0x00)`

- #define: `ADC_Channel_1((uint8_t)0x01)`

- #define: `ADC_Channel_2((uint8_t)0x02)`

- #define: `ADC_Channel_3((uint8_t)0x03)`

- #define: `ADC_Channel_4((uint8_t)0x04)`
• #define: `ADC_Channel_5((uint8_t)0x05)`

• #define: `ADC_Channel_6((uint8_t)0x06)`

• #define: `ADC_Channel_7((uint8_t)0x07)`

• #define: `ADC_Channel_8((uint8_t)0x08)`

• #define: `ADC_Channel_9((uint8_t)0x09)`

• #define: `ADC_Channel_10((uint8_t)0x0A)`

• #define: `ADC_Channel_11((uint8_t)0x0B)`

• #define: `ADC_Channel_12((uint8_t)0x0C)`

• #define: `ADC_Channel_13((uint8_t)0x0D)`

• #define: `ADC_Channel_14((uint8_t)0x0E)`

• #define: `ADC_Channel_15((uint8_t)0x0F)`

• #define: `ADC_Channel_16((uint8_t)0x10)`
• #define: \texttt{ADC\_Channel\_17((uint8\_t)0x11)}

• #define: \texttt{ADC\_Channel\_18((uint8\_t)0x12)}

• #define: \texttt{ADC\_Channel\_TempSensor((uint8\_t)ADC\_Channel\_16)}

• #define: \texttt{ADC\_Channel\_Vrefint((uint8\_t)ADC\_Channel\_17)}

• #define: \texttt{ADC\_Channel\_Vbat((uint8\_t)ADC\_Channel\_18)}

\texttt{ADC\_Common\_mode}

• #define: \texttt{ADC\_Mode\_Independent((uint32\_t)0x00000000)}

• #define: \texttt{ADC\_DualMode\_RegSimult\_InjecSimult((uint32\_t)0x00000001)}

• #define: \texttt{ADC\_DualMode\_RegSimult\_AlterTrig((uint32\_t)0x00000002)}

• #define: \texttt{ADC\_DualMode\_InjecSimult((uint32\_t)0x00000005)}

• #define: \texttt{ADC\_DualMode\_RegSimult((uint32\_t)0x00000006)}

• #define: \texttt{ADC\_DualMode\_Interl((uint32\_t)0x00000007)}

• #define: \texttt{ADC\_DualMode\_AlterTrig((uint32\_t)0x00000009)}
• #define: `ADC_TripleMode_RegSimult_InjecSimult((uint32_t)0x00000011)`

• #define: `ADC_TripleMode_RegSimult_AlterTrig((uint32_t)0x00000012)`

• #define: `ADC_TripleMode_InjecSimult((uint32_t)0x00000015)`

• #define: `ADC_TripleMode_RegSimult((uint32_t)0x00000016)`

• #define: `ADC_TripleMode_Interl((uint32_t)0x00000017)`

• #define: `ADC_TripleMode_AlterTrig((uint32_t)0x00000019)`

`ADC_data_align`
• #define: `ADC_DataAlign_Right((uint32_t)0x00000000)`

• #define: `ADC_DataAlign_Left((uint32_t)0x00000800)`

`ADC_delay_between_2_sampling_phases`
• #define: `ADC_TwoSamplingDelay_5Cycles((uint32_t)0x00000000)`

• #define: `ADC_TwoSamplingDelay_6Cycles((uint32_t)0x00000100)`

• #define: `ADC_TwoSamplingDelay_7Cycles((uint32_t)0x00000200)`
- #define: `ADC_TwoSamplingDelay_8Cycles((uint32_t)0x00000300)`

- #define: `ADC_TwoSamplingDelay_9Cycles((uint32_t)0x00000400)`

- #define: `ADC_TwoSamplingDelay_10Cycles((uint32_t)0x00000500)`

- #define: `ADC_TwoSamplingDelay_11Cycles((uint32_t)0x00000600)`

- #define: `ADC_TwoSamplingDelay_12Cycles((uint32_t)0x00000700)`

- #define: `ADC_TwoSamplingDelay_13Cycles((uint32_t)0x00000800)`

- #define: `ADC_TwoSamplingDelay_14Cycles((uint32_t)0x00000900)`

- #define: `ADC_TwoSamplingDelay_15Cycles((uint32_t)0x00000A00)`

- #define: `ADC_TwoSamplingDelay_16Cycles((uint32_t)0x00000B00)`

- #define: `ADC_TwoSamplingDelay_17Cycles((uint32_t)0x00000C00)`

- #define: `ADC_TwoSamplingDelay_18Cycles((uint32_t)0x00000D00)`

- #define: `ADC_TwoSamplingDelay_19Cycles((uint32_t)0x00000E00)`
- #define: `ADC_TwoSamplingDelay_20Cycles((uint32_t)0x00000F00)`

**ADC_Direct_memory_access_mode_for_multi_mode**
- #define: `ADC_DMAAccessMode_Disabled((uint32_t)0x00000000)`

- #define: `ADC_DMAAccessMode_1((uint32_t)0x00004000)`

- #define: `ADC_DMAAccessMode_2((uint32_t)0x00008000)`

- #define: `ADC_DMAAccessMode_3((uint32_t)0x0000C000)`

**ADC_external_trigger_edge_for_injected_channels_conversion**
- #define: `ADC_ExternalTrigInjecConvEdge_None((uint32_t)0x00000000)`

- #define: `ADC_ExternalTrigInjecConvEdge_Rising((uint32_t)0x00100000)`

- #define: `ADC_ExternalTrigInjecConvEdge_Falling((uint32_t)0x00200000)`

- #define: `ADC_ExternalTrigInjecConvEdge_RisingFalling((uint32_t)0x00300000)`

**ADC_external_trigger_edge_for_regular_channels_conversion**
- #define: `ADC_ExternalTrigConvEdge_None((uint32_t)0x00000000)`

- #define: `ADC_ExternalTrigConvEdge_Rising((uint32_t)0x10000000)`
• #define: `ADC_ExternalTrigConvEdge_Falling((uint32_t)0x20000000)`

• #define: `ADC_ExternalTrigConvEdge_RisingFalling((uint32_t)0x30000000)`

`ADC_extrenal_trigger_sources_for_injected_channels_conversion`
• #define: `ADC_ExternalTrigInjecConv_T1_CC4((uint32_t)0x00000000)`

• #define: `ADC_ExternalTrigInjecConv_T1_TRGO((uint32_t)0x00010000)`

• #define: `ADC_ExternalTrigInjecConv_T2_CC1((uint32_t)0x00020000)`

• #define: `ADC_ExternalTrigInjecConv_T2_TRGO((uint32_t)0x00030000)`

• #define: `ADC_ExternalTrigInjecConv_T3_CC2((uint32_t)0x00040000)`

• #define: `ADC_ExternalTrigInjecConv_T3_CC4((uint32_t)0x00050000)`

• #define: `ADC_ExternalTrigInjecConv_T4_CC1((uint32_t)0x00060000)`

• #define: `ADC_ExternalTrigInjecConv_T4_CC2((uint32_t)0x00070000)`

• #define: `ADC_ExternalTrigInjecConv_T4_CC3((uint32_t)0x00080000)`

• #define: `ADC_ExternalTrigInjecConv_T4_TRGO((uint32_t)0x00090000)`
• #define: `ADC_ExternalTrigInjecConv_T5_CC4((uint32_t)0x000A0000)`

• #define: `ADC_ExternalTrigInjecConv_T5_TRGO((uint32_t)0x000B0000)`

• #define: `ADC_ExternalTrigInjecConv_T8_CC2((uint32_t)0x000C0000)`

• #define: `ADC_ExternalTrigInjecConv_T8_CC3((uint32_t)0x000D0000)`

• #define: `ADC_ExternalTrigInjecConv_T8_CC4((uint32_t)0x000E0000)`

• #define: `ADC_ExternalTrigInjecConv_Ext_IT15((uint32_t)0x000F0000)`

`ADC_extrenal_trigger_sources_for_regular_channels_conversion`

• #define: `ADC_ExternalTrigConv_T1_CC1((uint32_t)0x00000000)`

• #define: `ADC_ExternalTrigConv_T1_CC2((uint32_t)0x01000000)`

• #define: `ADC_ExternalTrigConv_T1_CC3((uint32_t)0x02000000)`

• #define: `ADC_ExternalTrigConv_T2_CC2((uint32_t)0x03000000)`

• #define: `ADC_ExternalTrigConv_T2_CC3((uint32_t)0x04000000)`

• #define: `ADC_ExternalTrigConv_T2_CC4((uint32_t)0x05000000)`

• #define: `ADC_ExternalTrigConv_T2_CC4((uint32_t)0x05000000)`
- #define: `ADC_ExternalTrigConv_T2_TRGO((uint32_t)0x06000000)`

- #define: `ADC_ExternalTrigConv_T3_CC1((uint32_t)0x07000000)`

- #define: `ADC_ExternalTrigConv_T3_TRGO((uint32_t)0x08000000)`

- #define: `ADC_ExternalTrigConv_T4_CC4((uint32_t)0x09000000)`

- #define: `ADC_ExternalTrigConv_T5_CC1((uint32_t)0x0A000000)`

- #define: `ADC_ExternalTrigConv_T5_CC2((uint32_t)0x0B000000)`

- #define: `ADC_ExternalTrigConv_T5_CC3((uint32_t)0x0C000000)`

- #define: `ADC_ExternalTrigConv_T8_CC1((uint32_t)0x0D000000)`

- #define: `ADC_ExternalTrigConv_T8_TRGO((uint32_t)0x0E000000)`

- #define: `ADC_ExternalTrigConv_Ext_IT11((uint32_t)0x0F000000)`

`ADC_flags_definition`

- #define: `ADC_FLAG_AWD((uint8_t)0x01)`
Analog-to-digital converter (ADC)

- `#define: ADC_FLAG_EOC((uint8_t)0x02)`
- `#define: ADC_FLAG_JEOC((uint8_t)0x04)`
- `#define: ADC_FLAG_JSTRT((uint8_t)0x08)`
- `#define: ADC_FLAG_STRT((uint8_t)0x10)`
- `#define: ADC_FLAG_OVR((uint8_t)0x20)`

**ADC_injected_channel_selection**

- `#define: ADC_InjectedChannel_1((uint8_t)0x14)`
- `#define: ADC_InjectedChannel_2((uint8_t)0x18)`
- `#define: ADC_InjectedChannel_3((uint8_t)0x1C)`
- `#define: ADC_InjectedChannel_4((uint8_t)0x20)`

**ADC_interrupts_definition**

- `#define: ADC_IT_EOC((uint16_t)0x0205)`
- `#define: ADC_IT_AWD((uint16_t)0x0106)`
- `#define: ADC_IT_JEOC((uint16_t)0x0407)`
- #define: `ADC_IT_OVR((uint16_t)0x201A)`

**ADC_Prescaler**
- #define: `ADC_Prescaler_Div2((uint32_t)0x00000000)`
- #define: `ADC_Prescaler_Div4((uint32_t)0x00010000)`
- #define: `ADC_Prescaler_Div6((uint32_t)0x00020000)`
- #define: `ADC_Prescaler_Div8((uint32_t)0x00030000)`

**ADC_resolution**
- #define: `ADC_Resolution_12b((uint32_t)0x00000000)`
- #define: `ADC_Resolution_10b((uint32_t)0x01000000)`
- #define: `ADC_Resolution_8b((uint32_t)0x02000000)`
- #define: `ADC_Resolution_6b((uint32_t)0x03000000)`

**ADC_sampling_times**
- #define: `ADC_SampleTime_3Cycles((uint8_t)0x00)`
- #define: `ADC_SampleTime_15Cycles((uint8_t)0x01)`
3.4 ADC Programming Example

The example below explains how to configure the ADC1 to convert continuously channel 14 (this example assumes that ADC channel 14 pin and DMA are already configured). For more examples about ADC configuration and usage, please refer to the ADC examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\ADC\.

```c
ADC_InitTypeDef       ADC_InitStruct;
ADC_CommonInitStruct  ADC_CommonInitStruct;

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

/* Common configuration (applicable for the three ADCs) */
ADC_CommonInitStruct.ADC_Mode = ADC_Mode_Independent;
ADC_CommonInitStruct.ADC_Prescaler = ADC_Prescaler_Div2;
ADC_CommonInitStruct.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;

/* Delay between 2 sampling phases */
```

- **#define:** `ADC_SampleTime_28Cycles((uint8_t)0x02)`
- **#define:** `ADC_SampleTime_56Cycles((uint8_t)0x03)`
- **#define:** `ADC_SampleTime_84Cycles((uint8_t)0x04)`
- **#define:** `ADC_SampleTime_112Cycles((uint8_t)0x05)`
- **#define:** `ADC_SampleTime_144Cycles((uint8_t)0x06)`
- **#define:** `ADC_SampleTime_480Cycles((uint8_t)0x07)`
ADC_CommonInitStruct.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_5Cycles;
ADC_CommonInitStruct();

/* Configure ADC1 to convert continuously channel14 ***********/
ADC_InitStruct.ADC_Resolution = ADC_Resolution_12b;
ADC_InitStruct.ADC_ScanConvMode = DISABLE;
ADC_InitStruct.ADC_ContinuousConvMode = ENABLE;
ADC_InitStruct.ADC_ExternalTrigConvEdge =
ADC_ExternalTrigConvEdge_None;
ADC_InitStruct.ADC_DataAlign = ADC_DataAlign_Right;
ADC_InitStruct.ADC_NbrOfConversion = 1;
ADC_Init(ADC1, &ADC_InitStruct);

/* ADC1 regular channel14 configuration */
ADC_RegularChannelConfig(ADC1, ADC_Channel_14, 1,
ADC_SampleTime_3Cycles);

/* Enable DMA request after last transfer (Single-ADC mode) */
ADC_DMARequestAfterLastTransferCmd(ADC1, ENABLE);

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

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

/* Start ADC1 Software Conversion */
ADC_SoftwareStartConv(ADC1);
4  Controller area network (CAN)

4.1  CAN Firmware driver registers structures

4.1.1  CAN_TxMailBox_TypeDef

*CAN_TxMailBox_TypeDef* is defined in the stm32f2xx.h

Data Fields

- __IO uint32_t TIR
- __IO uint32_t TDTR
- __IO uint32_t TDLR
- __IO uint32_t TDHR

Field Documentation

- __IO uint32_t CAN_TxMailBox_TypeDef::TIR
  - CAN TX mailbox identifier register
- __IO uint32_t CAN_TxMailBox_TypeDef::TDTR
  - CAN mailbox data length control and time stamp register
- __IO uint32_t CAN_TxMailBox_TypeDef::TDLR
  - CAN mailbox data low register
- __IO uint32_t CAN_TxMailBox_TypeDef::TDHR
  - CAN mailbox data high register

4.1.2  CAN_FIFOMailBox_TypeDef

*CAN_FIFOMailBox_TypeDef* is defined in the stm32f2xx.h

Data Fields

- __IO uint32_t RIR
- __IO uint32_t RDTR
- __IO uint32_t RDLR
- __IO uint32_t RDHR

Field Documentation

- __IO uint32_t CAN_FIFOMailBox_TypeDef::RIR
  - CAN receive FIFO mailbox identifier register
- __IO uint32_t CAN_FIFOMailBox_TypeDef::RDTR
  - CAN receive FIFO mailbox data length control and time stamp register
- __IO uint32_t CAN_FIFOMailBox_TypeDef::RDLR
  - CAN receive FIFO mailbox data low register
- __IO uint32_t CAN_FIFOMailBox_TypeDef::RDHR
4.1.3 CAN_FilterRegister_TypeDef

_CAN_FilterRegister_TypeDef_ is defined in the stm32f2xx.h

Data Fields

- __IO uint32_t FR1
- __IO uint32_t FR2

Field Documentation

- __IO uint32_t CAN_FilterRegister_TypeDef::FR1
  - CAN Filter bank register 1
- __IO uint32_t CAN_FilterRegister_TypeDef::FR2
  - CAN Filter bank register 1

4.1.4 CAN_TypeDef

_CAN_TypeDef_ is defined in the stm32f2xx.h

Data Fields

- __IO uint32_t MCR
- __IO uint32_t MSR
- __IO uint32_t TSR
- __IO uint32_t RF0R
- __IO uint32_t RF1R
- __IO uint32_t IER
- __IO uint32_t ESR
- __IO uint32_t BTR
- uint32_t RESERVED0
- CAN_TxMailBox_TypeDef sTxMailBox
- CAN_FIFOMailBox_TypeDef sFIFOMailBox
- uint32_t RESERVED1
- __IO uint32_t FMR
- __IO uint32_t FM1R
- uint32_t RESERVED2
- __IO uint32_t FS1R
- uint32_t RESERVED3
- __IO uint32_t FFA1R
- uint32_t RESERVED4
- __IO uint32_t FA1R
- uint32_t RESERVED5
- CAN_FilterRegister_TypeDef sFilterRegister
Field Documentation

- **__IO uint32_t CAN_TypeDef::MCR**
  - CAN master control register, Address offset: 0x00
- **__IO uint32_t CAN_TypeDef::MSR**
  - CAN master status register, Address offset: 0x04
- **__IO uint32_t CAN_TypeDef::TSR**
  - CAN transmit status register, Address offset: 0x08
- **__IO uint32_t CAN_TypeDef::RF0R**
  - CAN receive FIFO 0 register, Address offset: 0x0C
- **__IO uint32_t CAN_TypeDef::RF1R**
  - CAN receive FIFO 1 register, Address offset: 0x10
- **__IO uint32_t CAN_TypeDef::IER**
  - CAN interrupt enable register, Address offset: 0x14
- **__IO uint32_t CAN_TypeDef::ESR**
  - CAN error status register, Address offset: 0x18
- **__IO uint32_t CAN_TypeDef::BTR**
  - CAN bit timing register, Address offset: 0x1C

- **uint32_t CAN_TypeDef::RESERVED0[88]**
  - Reserved, 0x020 - 0x17F
- **CAN_TxMailBox_TypeDef CAN_TypeDef::sTxMailBox[3]**
  - CAN Tx MailBox, Address offset: 0x180 - 0x1AC
- **CAN_FIFOMailBox_TypeDef CAN_TypeDef::sFIFOMailBox[2]**
  - CAN FIFO MailBox, Address offset: 0x1B0 - 0x1CC

- **uint32_t CAN_TypeDef::RESERVED1[12]**
  - Reserved, 0x1D0 - 0x1FF
- **__IO uint32_t CAN_TypeDef::FMR**
  - CAN filter master register, Address offset: 0x200
- **__IO uint32_t CAN_TypeDef::FM1R**
  - CAN filter mode register, Address offset: 0x204

- **uint32_t CAN_TypeDef::RESERVED2**
  - Reserved, 0x208
- **__IO uint32_t CAN_TypeDef::FS1R**
  - CAN filter scale register, Address offset: 0x20C

- **uint32_t CAN_TypeDef::RESERVED3**
  - Reserved, 0x210
- **__IO uint32_t CAN_TypeDef::FFA1R**
  - CAN filter FIFO assignment register, Address offset: 0x214

- **uint32_t CAN_TypeDef::RESERVED4**
  - Reserved, 0x218
- **__IO uint32_t CAN_TypeDef::FA1R**
  - CAN filter activation register, Address offset: 0x21C

- **uint32_t CAN_TypeDef::RESERVED5[8]**
  - Reserved, 0x220-0x23F

- **CAN_FilterRegister_TypeDef CAN_TypeDef::sFilterRegister[28]**
  - CAN Filter Register, Address offset: 0x240-0x31C

4.1.5 **CAN_InitTypeDef**

*CAN_InitTypeDef* is defined in the *stm32f2xx_can.h*
Data Fields

- `uint16_t CAN_Prescaler`
  - Specifies the length of a time quantum. It ranges from 1 to 1024.
- `uint8_t CAN_Mode`
  - Specifies the CAN operating mode. This parameter can be a value of `CAN_operating_mode`
- `uint8_t CAN_SJW`
  - Specifies the maximum number of time quanta the CAN hardware is allowed to lengthen or shorten a bit to perform resynchronization. This parameter can be a value of `CAN_synchronisation_jump_width`
- `uint8_t CAN_BS1`
  - Specifies the number of time quanta in Bit Segment 1. This parameter can be a value of `CAN_time_quantum_in_bit_segment_1`
- `uint8_t CAN_BS2`
  - Specifies the number of time quanta in Bit Segment 2. This parameter can be a value of `CAN_time_quantum_in_bit_segment_2`
- `FunctionalState CAN_TTCM`
  - Enable or disable the time triggered communication mode. This parameter can be set either to ENABLE or DISABLE.
- `FunctionalState CAN_ABOM`
  - Enable or disable the automatic bus-off management. This parameter can be set either to ENABLE or DISABLE.
- `FunctionalState CAN_AWUM`
  - Enable or disable the automatic wake-up mode. This parameter can be set either to ENABLE or DISABLE.
- `FunctionalState CAN_NART`
  - Enable or disable the non-automatic retransmission mode. This parameter can be set either to ENABLE or DISABLE.
- `FunctionalState CAN_RFLM`
  - Enable or disable the Receive FIFO Locked mode. This parameter can be set either to ENABLE or DISABLE.
- `FunctionalState CAN_TXFP`
  - Enable or disable the transmit FIFO priority. This parameter can be set either to ENABLE or DISABLE.

Field Documentation

- `uint16_t CAN_InitTypeDef::CAN_Prescaler`
  - Specifies the length of a time quantum. It ranges from 1 to 1024.
- `uint8_t CAN_InitTypeDef::CAN_Mode`
  - Specifies the CAN operating mode. This parameter can be a value of `CAN_operating_mode`
- `uint8_t CAN_InitTypeDef::CAN_SJW`
  - Specifies the maximum number of time quanta the CAN hardware is allowed to lengthen or shorten a bit to perform resynchronization. This parameter can be a value of `CAN_synchronisation_jump_width`
- `uint8_t CAN_InitTypeDef::CAN_BS1`
  - Specifies the number of time quanta in Bit Segment 1. This parameter can be a value of `CAN_time_quantum_in_bit_segment_1`
- `uint8_t CAN_InitTypeDef::CAN_BS2`
  - Specifies the number of time quanta in Bit Segment 2. This parameter can be a value of `CAN_time_quantum_in_bit_segment_2`
- `FunctionalState CAN_InitTypeDef::CAN_TTCM`
  - Enable or disable the time triggered communication mode. This parameter can be set either to ENABLE or DISABLE.
- `FunctionalState CAN_InitTypeDef::CAN_ABOM`
  - Enable or disable the automatic bus-off management. This parameter can be set either to ENABLE or DISABLE.
- `FunctionalState CAN_InitTypeDef::CAN_AWUM`
  - Enable or disable the automatic wake-up mode. This parameter can be set either to ENABLE or DISABLE.
- `FunctionalState CAN_InitTypeDef::CAN_NART`
  - Enable or disable the non-automatic retransmission mode. This parameter can be set either to ENABLE or DISABLE.
- `FunctionalState CAN_InitTypeDef::CAN_RFLM`
  - Enable or disable the Receive FIFO Locked mode. This parameter can be set either to ENABLE or DISABLE.
- `FunctionalState CAN_InitTypeDef::CAN_TXFP`
  - Enable or disable the transmit FIFO priority. This parameter can be set either to ENABLE or DISABLE.
**4.1.6 CAN_FilterInitTypeDef**

*CAN_FilterInitTypeDef* is defined in the *stm32f2xx_can.h*

**Data Fields**

- `uint16_t CAN_FilterIdHigh`
- `uint16_t CAN_FilterIdLow`
- `uint16_t CAN_FilterMaskIdHigh`
- `uint16_t CAN_FilterMaskIdLow`
- `uint16_t CAN_FilterFIFOAssignment`
- `uint8_t CAN_FilterNumber`
- `uint8_t CAN_FilterMode`
- `uint8_t CAN_FilterScale`
- `FunctionalState CAN_FilterActivation`

**Field Documentation**

- `uint16_t CAN_FilterInitTypeDef::CAN_FilterIdHigh`
  - Specifies the filter identification number (MSBs for a 32-bit configuration, first one for a 16-bit configuration). This parameter can be a value between 0x0000 and 0xFFFF
- `uint16_t CAN_FilterInitTypeDef::CAN_FilterIdLow`
  - Specifies the filter identification number (LSBs for a 32-bit configuration, second one for a 16-bit configuration). This parameter can be a value between 0x0000 and 0xFFFF
- `uint16_t CAN_FilterInitTypeDef::CAN_FilterMaskIdHigh`
  - Specifies the filter mask number or identification number, according to the mode (MSBs for a 32-bit configuration, first one for a 16-bit configuration). This parameter can be a value between 0x0000 and 0xFFFF
- `uint16_t CAN_FilterInitTypeDef::CAN_FilterMaskIdLow`
  - Specifies the filter mask number or identification number, according to the mode (LSBs for a 32-bit configuration, second one for a 16-bit configuration). This parameter can be a value between 0x0000 and 0xFFFF
- `uint16_t CAN_FilterInitTypeDef::CAN_FilterFIFOAssignment`
  - Specifies the FIFO (0 or 1) which will be assigned to the filter. This parameter can be a value of *CAN_filter_FIFO*
- `uint8_t CAN_FilterInitTypeDef::CAN_FilterNumber`
  - Specifies the filter which will be initialized. It ranges from 0 to 13.
- `uint8_t CAN_FilterInitTypeDef::CAN_FilterMode`
  - Specifies the filter mode to be initialized. This parameter can be a value of *CAN_filter_mode*
- `uint8_t CAN_FilterInitTypeDef::CAN_FilterScale`
  - Specifies the filter scale. This parameter can be a value of *CAN_filter_scale*
- `FunctionalState CAN_FilterInitTypeDef::CAN_FilterActivation`
  - Enable or disable the filter. This parameter can be set either to *ENABLE* or *DISABLE*.
4.1.7 CanTxMsg

*CanTxMsg* is defined in the *stm32f2xx_can.h*

Data Fields

- `uint32_t StdId`
- `uint32_t ExtId`
- `uint8_t IDE`
- `uint8_t RTR`
- `uint8_t DLC`
- `uint8_t Data`

Field Documentation

- `uint32_t CanTxMsg::StdId`
  - Specifies the standard identifier. This parameter can be a value between 0 and 0x7FF.
- `uint32_t CanTxMsg::ExtId`
  - Specifies the extended identifier. This parameter can be a value between 0 and 0x1FFFFFFF.
- `uint8_t CanTxMsg::IDE`
  - Specifies the type of identifier for the message that will be transmitted. This parameter can be a value of `CAN_identifier_type`
- `uint8_t CanTxMsg::RTR`
  - Specifies the type of frame for the message that will be transmitted. This parameter can be a value of `CAN_remote_transmission_request`
- `uint8_t CanTxMsg::DLC`
  - Specifies the length of the frame that will be transmitted. This parameter can be a value between 0 and 8
- `uint8_t CanTxMsg::Data[8]`
  - Contains the data to be transmitted. It ranges from 0 to 0xFF.

4.1.8 CanRxMsg

*CanRxMsg* is defined in the *stm32f2xx_can.h*

Data Fields

- `uint32_t StdId`
- `uint32_t ExtId`
- `uint8_t IDE`
- `uint8_t RTR`
- `uint8_t DLC`
- `uint8_t Data`
- `uint8_t FMI`

Field Documentation
- `uint32_t CanRxMsg::StdId`  
  Specifies the standard identifier. This parameter can be a value between 0 and 0x7FF.
- `uint32_t CanRxMsg::ExtId`  
  Specifies the extended identifier. This parameter can be a value between 0 and 0xFFFFFFFF.
- `uint8_t CanRxMsg::IDE`  
  Specifies the type of identifier for the message that will be received. This parameter can be a value of `CAN_identifier_type`.
- `uint8_t CanRxMsg::RTR`  
  Specifies the type of frame for the received message. This parameter can be a value of `CAN_remote_transmission_request`.
- `uint8_t CanRxMsg::DLC`  
  Specifies the length of the frame that will be received. This parameter can be a value between 0 and 8.
- `uint8_t CanRxMsg::Data[8]`  
  Contains the data to be received. It ranges from 0 to 0xFF.
- `uint8_t CanRxMsg::FMI`  
  Specifies the index of the filter the message stored in the mailbox passes through. This parameter can be a value between 0 and 0xFF.

### 4.2 CAN Firmware driver API description

The following section lists the various functions of the CAN library.

### 4.2.1 How to use this driver

The following section lists the various functions of the CAN library.

1. Enable the CAN controller interface clock using:  
   `RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);` for CAN1 and  
   `RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN2, ENABLE);` for CAN2. In case you are using CAN2 only, you have to enable the CAN1 clock.
2. CAN pins configuration - Enable the clock for the CAN GPIOs using the following function:  
   `RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOx, ENABLE);`  
   Connect the involved CAN pins to AF9 using the following function  
   `GPIO_PinAFConfig(GPIOx, GPIO_PinSourcex, GPIO_AF_CANx);`  
   Configure these CAN pins in alternate function mode by calling the function  
   `GPIO_Init();`  
3. Initialize and configure the CAN using `CAN_Init()` and `CAN_FilterInit()` functions.
4. Transmit the desired CAN frame using `CAN_Transmit()` function.
5. Check the transmission of a CAN frame using `CAN_TransmitStatus()` function.
6. Cancel the transmission of a CAN frame using `CAN_CancelTransmit()` function.
8. Release the receive FIFOs using `CAN_FIFORelease()` function.
9. Return the number of pending received frames using `CAN_MessagePending()` function.
10. To control CAN events you can use one of the following two methods:  
    - Check on CAN flags using the `CAN_GetFlagStatus()` function.  
    - Use CAN interrupts through the function `CAN_ITConfig()` at initialization phase and `CAN_GetITStatus()` function into interrupt routines to check if the event has occurred or not. After checking on a flag you should clear it using...
4.2.2 Initialization and configuration

This section provides functions allowing to:

- Initialize the CAN peripherals: Prescaler, operating mode, the maximum number of time quanta to perform resynchronization, the number of time quanta in Bit Segments 1 and 2 and many other modes. Refer to @ref CAN_InitTypeDef for more details.
- Configures the CAN reception filter.
- Select the start bank filter for slave CAN.
- Enables or disables the Debug Freeze mode for CAN.
- Enables or disables the CAN Time Trigger Operation communication mode.

Below is the list of functions that can be used to initialize and configure the CAN:

- `CAN_DeInit()`
- `CAN_Init()`
- `CAN_FilterInit()`
- `CAN_StructInit()`
- `CAN_SlaveStartBank()`
- `CAN_DBGFreeze()`
- `CAN_TTComModeCmd()`

**CAN Frames Transmission functions**

This section provides functions allowing to:

- Initiate and transmit a CAN frame message (if there is an empty mailbox)
- Check the transmission status of a CAN Frame
- Cancel a transmit request

Below is the list of CAN Frames Transmission functions:

- `CAN_Transmit()`
- `CAN_TransmitStatus()`
- `CAN_CancelTransmit()`

**CAN Frames Reception functions**

This section provides functions allowing to:

- Receive a correct CAN frame
- Release a specified receive FIFO (2 FIFOs are available)
- Return the number of the pending received CAN frames

Below is the list of CAN Frames Reception functions:

- `CAN_Receive()`
- `CAN_FIFORelease()`
- `CAN_MessagePending()`

**CAN Operation modes functions**

This section provides functions allowing to select the CAN Operation modes:

- Sleep mode
- Normal mode
Initialization mode

Below is the list of CAN Operating modes functions:

- `CAN_OperatingModeRequest()`
- `CAN_Sleep()`
- `CAN_WakeUp()`

**CAN Bus Error management functions**

This section provides functions allowing to:

- Return the CANx's last error code (LEC)
- Return the CANx Receive Error Counter (REC)
- Return the LSB of the 9-bit CANx Transmit Error Counter (TEC).

If TEC is greater than 255, the CAN is in bus-off state.

If REC or TEC are greater than 96, an Error warning flag occurs.

If REC or TEC are greater than 127, an Error Passive Flag occurs.

Below is the list of CAN Bus Error management functions:

- `Section 4.2.8.1: "CAN_GetLastErrorCode"`
- `Section 4.2.8.2: "CAN_GetReceiveErrorCounter"`
- `Section 4.2.8.3: "CAN_GetLSBS TransmitErrorCounter"`

### 4.2.3 Interrupt and flag management

This section provides functions allowing to configure the CAN Interrupts and to get the status and clear flags and Interrupts pending bits.

The CAN provides 15 Flags and 14 Interrupts sources:

**Flags**

The 15 flags can be divided into 4 groups:

- **Transmit Flags** `CAN_FLAG_RQCP0`, `CAN_FLAG_RQCP1`, `CAN_FLAG_RQCP2`:
  Request completed MailBoxes 0, 1 and 2 Flags Set when the last request (transmit or abort) has been performed.

- **Receive Flags**
  - `CAN_FLAG_FMP0`, `CAN_FLAG_FMP1`:
    Message Pending Flags set to signal that messages are pending in the receive FIFO. These Flags are cleared only by hardware.
  - `CAN_FLAG_FF0`, `CAN_FLAG_FF1`:
    Full Flags set when three messages are stored in the selected FIFO.
- **CAN_FLAG_FOV0** CAN_FLAG_FOV1 : FIFO 0 and 1 Overrun Flags set when a new message has been received and passed the filter while the FIFO was full.

- **Operating modes Flags**
  - CAN_FLAG_WKU : Wake up Flag set to signal that a SOF bit has been detected while the CAN hardware was in Sleep mode.
  - CAN_FLAG_SLAK : Sleep acknowledge Flag Set to signal that the CAN has entered Sleep Mode.

- **Error Flags**
  - CAN_FLAG_EWG : Error Warning Flag Set when the warning limit has been reached (Receive Error Counter or Transmit Error Counter greater than 96). This Flag is cleared only by hardware.
  - CAN_FLAG_EPV : Error Passive Flag Set when the Error Passive limit has been reached (Receive Error Counter or Transmit Error Counter greater than 127). This Flag is cleared only by hardware.
  - CAN_FLAG_BOF : Bus-Off Flag set when CAN enters the bus-off state. The bus-off state is entered on TEC overflow, greater than 255. This Flag is cleared only by hardware.
  - CAN_FLAG_LEC : Last error code Flag set If a message has been transferred (reception or transmission) with error, and the error code is hold.

**Interrupts**

The 14 interrupts can be divided on 4 groups:

- **Transmit interrupt** CAN_IT_TME : Transmit mailbox empty Interrupt if enabled, this interrupt source is pending when no transmit request are pending for Tx mailboxes.

- **Receive Interrupts**
  - CAN_IT_FMP0, CAN_IT_FMP1 : FIFO 0 and FIFO1 message pending Interrupts if enabled, these interrupt sources are pending when messages are pending in the receive FIFO. The corresponding interrupt pending bits are cleared only by hardware.
  - CAN_IT_FF0, CAN_IT_FF1 : FIFO 0 and FIFO1 full Interrupts if enabled, these interrupt sources are pending when three messages are stored in the selected FIFO.
  - CAN_IT_FOV0, CAN_IT_FOV1 : FIFO 0 and FIFO1 overrun Interrupts if enabled, these interrupt sources are pending when a new message has been received and passed the filter while the FIFO was full.

- **Operating Mode Interrupts**
  - CAN_IT_WKU : Wake-up Interrupt if enabled, this interrupt source is pending when a SOF bit has been detected while the CAN hardware was in Sleep mode.
  - CAN_IT_SLK : Sleep acknowledge Interrupt if enabled, this interrupt source is pending when the CAN has entered Sleep Mode.

- **Error Interrupts**
  - CAN_IT_EWG : Error warning Interrupt if enabled, this interrupt source is pending when the warning limit has been reached (Receive Error Counter or Transmit Error Counter=96).
  - CAN_IT_EPV : Error passive Interrupt if enabled, this interrupt source is pending when the Error Passive limit has been reached (Receive Error Counter or Transmit Error Counter>127).
  - CAN_IT_BOF : Bus-off Interrupt if enabled, this interrupt source is pending when CAN enters the bus-off state. The bus-off state is entered on TEC overflow, greater than 255. This Flag is cleared only by hardware.
  - CAN_IT_LEC : Last error code Interrupt if enabled, this interrupt source is pending when a message has been transferred (reception or transmission) with error, and the error code is hold.
Managing the CAN controller events

The user should identify which mode will be used in his application to manage the CAN controller events: Polling mode or Interrupt mode.

- In the Polling Mode it is advised to use the following functions:
  - CAN_GetFlagStatus() : to check if flags events occur.
  - CAN_ClearFlag() : to clear the flags events.
- In the Interrupt Mode it is advised to use the following functions: The functions used to manage the CAN controller events are the following:
  CAN_ITConfig() CAN_GetFlagStatus() CAN_ClearFlag() CAN_GetITStatus() CAN_ClearITPendingBit()
  This function has no impact on CAN_IT_FMP0 and CAN_IT_FMP1 Interrupts pending bits since there are cleared only by hardware.
  - CAN_ITConfig() : to enable or disable the interrupt source.
  - CAN_GetITStatus() : to check if Interrupt occurs.
  - CAN_ClearITPendingBit() : to clear the Interrupt pending Bit (corresponding Flag). This function has no impact on CAN_IT_FMP0 and CAN_IT_FMP1 Interrupts pending bits since there are cleared only by hardware.

4.2.4 Initialization and configuration functions

4.2.4.1 CAN_DeInit

Function Name: void CAN_DeInit ( CAN_TypeDef * CANx)
Function Description: Deinitializes the CAN peripheral registers to their default reset values.
Parameters:
- CANx : where x can be 1 or 2 to select the CAN peripheral.
Return values:
- None.
Notes:
- None.

4.2.4.2 CAN_Init

Function Name: uint8_t CAN_Init ( CAN_TypeDef * CANx, CAN_InitTypeDef * CAN_InitStruct)
Function Description: Initializes the CAN peripheral according to the specified
parameters in the CAN_InitStruct.

Parameters
- **CANx**: where x can be 1 or 2 to select the CAN peripheral.
- **CAN_InitStruct**: pointer to a CAN_InitTypeDef structure that contains the configuration information for the CAN peripheral.

Return values
- **Constant indicates initialization succeed which will be CAN_InitStatus_Failed or CAN_InitStatus_Success.**

Notes
- **None.**

### 4.2.4.3 CAN_FilterInit

**Function Name**
`void CAN_FilterInit (CAN_FilterInitStruct)`

**Function Description**
Configures the CAN reception filter according to the specified parameters in the CAN_FilterInitStruct.

**Parameters**
- **CAN_FilterInitStruct**: pointer to a CAN_FilterInitStruct type that contains configuration information.

**Return values**
- **None.**

**Notes**
- **None.**

### 4.2.4.4 CAN_StructInit

**Function Name**
`void CAN_StructInit (CAN_InitTypeDef * CAN_InitStruct)`

**Function Description**
Fills each CAN_InitStruct member with its default value.

**Parameters**
- **CAN_InitStruct**: pointer to a CAN_InitTypeDef structure which will be initialized.

**Return values**
- **None.**

**Notes**
- **None.**

### 4.2.4.5 CAN_SlaveStartBank
Function Name: `void CAN_SlaveStartBank ( uint8_t CAN_BankNumber)`
Function Description: Select the start bank filter for slave CAN.
Parameters:
- `CAN_BankNumber`: Select the start slave bank filter from 1..27.

Return values: None.
Notes: None.

### 4.2.4.6 CAN_DBGFreeze

Function Name: `void CAN_DBGFreeze ( CAN_TypeDef * CANx, FunctionalState NewState)`
Function Description: Enables or disables the DBG Freeze for CAN.
Parameters:
- `CANx`: where x can be 1 or 2 to to select the CAN peripheral.
- `NewState`: new state of the CAN peripheral. This parameter can be: ENABLE (CAN reception/transmission is frozen during debug. Reception FIFOs can still be accessed/controlled normally) or DISABLE (CAN is working during debug).

Return values: None.
Notes: None.

### 4.2.4.7 CAN_TTComModeCmd

Function Name: `void CAN_TTComModeCmd ( CAN_TypeDef * CANx, FunctionalState NewState)`
Function Description: Enables or disables the CAN Time TriggerOperation communication mode.
Parameters:
- `CANx`: where x can be 1 or 2 to to select the CAN peripheral.
- `NewState`: Mode new state. This parameter can be: ENABLE or DISABLE. When enabled, Time stamp (TIME[15:0]) value is sent in the last two data bytes of the 8-byte message: TIME[7:0] in data byte 6 and TIME[15:8] in...
data byte 7.

Return values

- None.

Notes

- DLC must be programmed as 8 in order Time Stamp (2 bytes) to be sent over the CAN bus.

4.2.5 CAN Frames Transmission functions

4.2.5.1 CAN_Transmit

Function Name

uint8_t CAN_Transmit (CAN_TypeDef * CANx, CanTxMsg * TxMessage)

Function Description

Initiates and transmits a CAN frame message.

Parameters

- **CANx:** where x can be 1 or 2 to select the CAN peripheral.
- **TxMessage:** pointer to a structure which contains CAN Id, CAN DLC and CAN data.

Return values

- The number of the mailbox that is used for transmission or CAN_TxStatus_NoMailBox if there is no empty mailbox.

Notes

- None.

4.2.5.2 CAN_TransmitStatus

Function Name

uint8_t CAN_TransmitStatus (CAN_TypeDef * CANx, uint8_t TransmitMailbox)

Function Description

Checks the transmission status of a CAN Frame.

Parameters

- **CANx:** where x can be 1 or 2 to select the CAN peripheral.
- **TransmitMailbox:** the number of the mailbox that is used for transmission.

Return values

- CAN_TxStatus_Ok if the CAN driver transmits the message, CAN_TxStatus_Failed in an other case.

Notes

- None.
### 4.2.5.3 CAN_CancelTransmit

**Function Name**  
void CAN_CancelTransmit (CAN_TypeDef * CANx, uint8_t Mailbox)

**Function Description**  
Cancels a transmit request.

**Parameters**
- **CANx**: where x can be 1 or 2 to select the CAN peripheral.
- **Mailbox**: Mailbox number.

**Return values**  
- None.

**Notes**  
- None.

### 4.2.6 CAN Frames Reception functions

#### 4.2.6.1 CAN_Receive

**Function Name**  
void CAN_Receive (CAN_TypeDef * CANx, uint8_t FIFONumber, CanRxMsg * RxMessage)

**Function Description**  
Receives a correct CAN frame.

**Parameters**
- **CANx**: where x can be 1 or 2 to select the CAN peripheral.
- **FIFONumber**: Receive FIFO number, CAN_FIFO0 or CAN_FIFO1.
- **RxMessage**: pointer to a structure receive frame which contains CAN Id, CAN DLC, CAN data and FMI number.

**Return values**  
- None.

**Notes**  
- None.

#### 4.2.6.2 CAN_FIFORelease

**Function Name**  
void CAN_FIFORelease (CAN_TypeDef * CANx, uint8_t FIFONumber)

**Function Description**  
Releases the specified receive FIFO.
Parameters
- **CANx**: where x can be 1 or 2 to select the CAN peripheral.
- **FIFONumber**: FIFO to release, CAN_FIFO0 or CAN_FIFO1.

Return values
- None.

Notes
- None.

### 4.2.6.3 CAN_MessagePending

**Function Name**
`uint8_t CAN_MessagePending (CAN_TypeDef * CANx, uint8_t FIFONumber)`

**Function Description**
Returns the number of pending received messages.

**Parameters**
- **CANx**: where x can be 1 or 2 to select the CAN peripheral.
- **FIFONumber**: Receive FIFO number, CAN_FIFO0 or CAN_FIFO1.

**Return values**
- **NbMessage**: which is the number of pending message.

**Notes**
- None.

### 4.2.7 CAN Operation modes functions

#### 4.2.7.1 CAN_OperatingModeRequest

**Function Name**
`uint8_t CAN_OperatingModeRequest (CAN_TypeDef * CANx, uint8_t CAN_OperatingMode)`

**Function Description**
Selects the CAN Operation mode.

**Parameters**
- **CAN_OperatingMode**: CAN Operating Mode. This parameter can be one of CAN_OperatingMode_TypeDef enumeration.

**Return values**
- **status of the requested mode which can be**
  - **CAN_ModeStatus_Failed**: CAN failed entering the specific mode
  - **CAN_ModeStatus_Success**: CAN Succeed entering the specific mode

**Notes**
- None.
4.2.7.2  CAN_Sleep

Function Name  
uint8_t CAN_Sleep ( CAN_TypeDef * CANx)

Function Description  Enters the Sleep (low power) mode.

Parameters  
•  CANx : where x can be 1 or 2 to select the CAN peripheral.

Return values  
•  CAN_Sleep_Ok if sleep entered, CAN_Sleep_Failed otherwise.

Notes  
•  None.

4.2.7.3  CAN_WakeUp

Function Name  
uint8_t CAN_WakeUp ( CAN_TypeDef * CANx)

Function Description  Wakes up the CAN peripheral from sleep mode.

Parameters  
•  CANx : where x can be 1 or 2 to select the CAN peripheral.

Return values  
•  CAN_WakeUp_Ok if sleep mode left, CAN_WakeUp_Failed otherwise.

Notes  
•  None.

4.2.8  CAN Bus Error management functions

4.2.8.1  CAN_GetLastErrorCode

Function Name  
uint8_t CAN_GetLastErrorCode ( CAN_TypeDef * CANx)

Function Description  Returns the CANx's last error code (LEC).

Parameters  
•  CANx: where x can be 1 or 2 to select the CAN peripheral.

Return values  
•  Error code:
  - CAN_ERRORCODE_NoErr: No Error
  - CAN_ERRORCODE_StuffErr: Stuff Error
  - CAN_ERRORCODE_FormErr: Form Error
  - CAN_ERRORCODE_ACKErr : Acknowledgment Error
- CAN_ERRORCODE_BitRecessiveErr: Bit Recessive Error
- CAN_ERRORCODE_BitDominantErr: Bit Dominant Error
- CAN_ERRORCODE_CRCErr: CRC Error
- CAN_ERRORCODE_SoftwareSetErr: Software Set Error

Notes
- None.

4.2.8.2 CAN_GetReceiveErrorCounter

Function Name          uint8_t CAN_GetReceiveErrorCounter (CAN_TypeDef * CANx)
Function Description   Returns the CANx Receive Error Counter (REC).
Parameters            CANx : where x can be 1 or 2 to select the CAN peripheral.
Return values          CAN Receive Error Counter.
Notes                  In case of an error during reception, this counter is incremented by 1 or by 8 depending on the error condition as defined by the CAN standard. After every successful reception, the counter is decremented by 1 or reset to 120 if its value was higher than 128. When the counter value exceeds 127, the CAN controller enters the error passive state.

4.2.8.3 CAN_GetLSBTransmitErrorCounter

Function Name          uint8_t CAN_GetLSBTransmitErrorCounter (CAN_TypeDef * CANx)
Function Description   Returns the LSB of the 9-bit CANx Transmit Error Counter(TEC).
Parameters            CANx : where x can be 1 or 2 to select the CAN peripheral.
Return values          LSB of the 9-bit CAN Transmit Error Counter.
Notes                  None.
4.2.9 Interrupt and flag management functions

4.2.9.1 CAN_ITConfig

Function Name
void CAN_ITConfig (CAN_TypeDef * CANx, uint32_t CAN_IT, FunctionalState NewState)

Function Description
Enables or disables the specified CANx interrupts.

Parameters
- **CANx**: where x can be 1 or 2 to select the CAN peripheral.
- **CAN_IT**: specifies the CAN interrupt sources to be enabled or disabled. This parameter can be:
  - **CAN_IT_TME**: Transmit mailbox empty Interrupt
  - **CAN_IT_FMP0**: FIFO 0 message pending Interrupt
  - **CAN_IT_FF0**: FIFO 0 full Interrupt
  - **CAN_IT_FOV0**: FIFO 0 overrun Interrupt
  - **CAN_IT_FMP1**: FIFO 1 message pending Interrupt
  - **CAN_IT_FF1**: FIFO 1 full Interrupt
  - **CAN_IT_FOV1**: FIFO 1 overrun Interrupt
  - **CAN_IT_WKU**: Wake-up Interrupt
  - **CAN_IT_SLK**: Sleep acknowledge Interrupt
  - **CAN_IT_EWG**: Error warning Interrupt
  - **CAN_IT_EPV**: Error passive Interrupt
  - **CAN_IT_BOF**: Bus-off Interrupt
  - **CAN_IT_LEC**: Last error code Interrupt
  - **CAN_IT_ERR**: Error Interrupt
- **NewState**: new state of the CAN interrupts. This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- None.

4.2.9.2 CAN_GetFlagStatus

Function Name
FlagStatus CAN_GetFlagStatus (CAN_TypeDef * CANx, uint32_t CAN_FLAG)

Function Description
Checks whether the specified CAN flag is set or not.

Parameters
- **CANx**: where x can be 1 or 2 to select the CAN peripheral.
- **CAN_FLAG**: specifies the flag to check. This parameter can be one of the following values:
4.2.9.3 CAN_ClearFlag

Function Name: void CAN_ClearFlag (CAN_TypeDef * CANx, uint32_t CAN_FLAG)

Function Description: Clears the CAN's pending flags.

Parameters:
- CANx: where x can be 1 or 2 to select the CAN peripheral.
- CAN_FLAG: specifies the flag to clear. This parameter can be one of the following values:
  - CAN_FLAG_RQCP0: Request MailBox0 Flag
  - CAN_FLAG_RQCP1: Request MailBox1 Flag
  - CAN_FLAG_RQCP2: Request MailBox2 Flag
  - CAN_FLAG_FMP0: FIFO 0 Message Pending Flag
  - CAN_FLAG_FF0: FIFO 0 Full Flag
  - CAN_FLAG_FOV0: FIFO 0 Overrun Flag
  - CAN_FLAG_FMP1: FIFO 1 Message Pending Flag
  - CAN_FLAG_FF1: FIFO 1 Full Flag
  - CAN_FLAG_FOV1: FIFO 1 Overrun Flag
  - CAN_FLAG_WKU: Wake up Flag
  - CAN_FLAG_SLAK: Sleep acknowledge Flag
  - CAN_FLAG_EWG: Error Warning Flag
  - CAN_FLAG_EPV: Error Passive Flag
  - CAN_FLAG_BOF: Bus-Off Flag
  - CAN_FLAG_LEC: Last error code Flag

Return values: The new state of CAN_FLAG (SET or RESET).

Notes: None.
4.2.9.4  CAN_GetITStatus

**Function Name**  
ITStatus CAN_GetITStatus ( CAN_TypeDef * CANx, uint32_t CAN_IT)

**Function Description**  
Checks whether the specified CANx interrupt has occurred or not.

**Parameters**
- **CANx**: where x can be 1 or 2 to select the CAN peripheral.
- **CAN_IT**: specifies the CAN interrupt source to check. This parameter can be one of the following values:
  - CAN_IT_TME: Transmit mailbox empty Interrupt
  - CAN_IT_FMP0: FIFO 0 message pending Interrupt
  - CAN_IT_FF0: FIFO 0 full Interrupt
  - CAN_IT_FOV0: FIFO 0 overrun Interrupt
  - CAN_IT_FMP1: FIFO 1 message pending Interrupt
  - CAN_IT_FF1: FIFO 1 full Interrupt
  - CAN_IT_FOV1: FIFO 1 overrun Interrupt
  - CAN_IT_WKU: Wake-up Interrupt
  - CAN_IT_SLK: Sleep acknowledge Interrupt
  - CAN_IT_EWG: Error warning Interrupt
  - CAN_IT_EPV: Error passive Interrupt
  - CAN_IT_BOF: Bus-off Interrupt
  - CAN_IT_LEC: Last error code Interrupt
  - CAN_IT_ERR: Error Interrupt

**Return values**
- The current state of CAN_IT (SET or RESET).

**Notes**
- None.

4.2.9.5  CAN_ClearITPendingBit

**Function Name**  
void CAN_ClearITPendingBit ( CAN_TypeDef * CANx, uint32_t CAN_IT)

**Function Description**  
Clears the CANx's interrupt pending bits.

**Parameters**
- **CANx**: where x can be 1 or 2 to select the CAN peripheral.
- **CAN_IT**: specifies the interrupt pending bit to clear. This parameter can be one of the following values:
  - CAN_IT_TME: Transmit mailbox empty Interrupt
  - CAN_IT_FMP0: FIFO 0 message pending Interrupt
  - CAN_IT_FF0: FIFO 0 full Interrupt
  - CAN_IT_FOV0: FIFO 0 overrun Interrupt
  - CAN_IT_FOV1: FIFO 1 overrun Interrupt
  - CAN_IT_WKU: Wake-up Interrupt
4.3 CAN Firmware driver defines

4.3.1 CAN Firmware driver defines

**CAN_Error_Code_constants**

- #define: `CAN_ErrorCode_NoErr((uint8_t)0x00)`
  No Error

- #define: `CAN_ErrorCode_StuffErr((uint8_t)0x10)`
  Stuff Error

- #define: `CAN_ErrorCode_FormErr((uint8_t)0x20)`
  Form Error

- #define: `CAN_ErrorCode_ACKErr((uint8_t)0x30)`
  Acknowledgment Error

- #define: `CAN_ErrorCode_BitRecessiveErr((uint8_t)0x40)`
  Bit Recessive Error

- #define: `CAN_ErrorCode_BitDominantErr((uint8_t)0x50)`
  Bit Dominant Error

- #define: `CAN_ErrorCode_CRCErr((uint8_t)0x60)`
  CRC Error

- #define: `CAN_ErrorCode_SoftwareSetErr((uint8_t)0x70)`
  Software Set Error
Software Set Error

**CAN_filter_FIFO**
- #define: `CAN_Filter_FIFO0((uint8_t)0x00)`
  Filter FIFO 0 assignment for filter x

- #define: `CAN_Filter_FIFO1((uint8_t)0x01)`
  Filter FIFO 1 assignment for filter x

- #define: `CAN_FilterFIFO0CAN_Filter_FIFO0`

- #define: `CAN_FilterFIFO1CAN_Filter_FIFO1`

**CAN_filter_mode**
- #define: `CAN_FilterMode_IdMask((uint8_t)0x00)`
  identifier/mask mode

- #define: `CAN_FilterMode_IdList((uint8_t)0x01)`
  identifier list mode

**CAN_filter_scale**
- #define: `CAN_FilterScale_16bit((uint8_t)0x00)`
  Two 16-bit filters

- #define: `CAN_FilterScale_32bit((uint8_t)0x01)`
  One 32-bit filter

**CAN_flags**
- #define: `CAN_FLAG_RQCP0((uint32_t)0x38000001)`
  Request MailBox0 Flag

- #define: `CAN_FLAG_RQCP1((uint32_t)0x38000100)`
  Request MailBox1 Flag
• #define:  **CAN_FLAG_RQCP2**((uint32_t)0x38010000)
  Request MailBox2 Flag

• #define:  **CAN_FLAG_FMP0**((uint32_t)0x12000003)
  FIFO 0 Message Pending Flag

• #define:  **CAN_FLAG_FF0**((uint32_t)0x32000008)
  FIFO 0 Full Flag

• #define:  **CAN_FLAG_FOV0**((uint32_t)0x32000010)
  FIFO 0 Overrun Flag

• #define:  **CAN_FLAG_FMP1**((uint32_t)0x14000003)
  FIFO 1 Message Pending Flag

• #define:  **CAN_FLAG_FF1**((uint32_t)0x34000008)
  FIFO 1 Full Flag

• #define:  **CAN_FLAG_FOV1**((uint32_t)0x34000010)
  FIFO 1 Overrun Flag

• #define:  **CAN_FLAG_WKU**((uint32_t)0x31000008)
  Wake up Flag

• #define:  **CAN_FLAG_SLAK**((uint32_t)0x31000012)
  Sleep acknowledge Flag

• #define:  **CAN_FLAG_EWG**((uint32_t)0xF000001)
  Error Warning Flag

• #define:  **CAN_FLAG_EPV**((uint32_t)0xF000002)
  Error Passive Flag

• #define:  **CAN_FLAG_BOF**((uint32_t)0xF000004)
  Bus-Off Flag
#define: CAN_FLAG_LEC((uint32_t)0x30F00070)
Last error code Flag

**CAN_identifier_type**

- #define: CAN_Id_Standard((uint32_t)0x00000000)
  Standard Id

- #define: CAN_Id_Extended((uint32_t)0x00000004)
  Extended Id

- #define: CAN_ID_STDCAN_Id_Standard

- #define: CAN_ID_EXTCAN_Id_Extended

**CAN_InitStatus**

- #define: CAN_InitStatus_Failed((uint8_t)0x00)
  CAN initialization failed

- #define: CAN_InitStatus_Success((uint8_t)0x01)
  CAN initialization OK

- #define: CANINITFAILEDCAN_InitStatus_Failed

- #define: CANINITOKCAN_InitStatus_Success

**CAN_interrufts**

- #define: CAN_IT_TME((uint32_t)0x00000001)
  Transmit mailbox empty Interrupt

- #define: CAN_IT_FMP0((uint32_t)0x00000002)
  FIFO 0 message pending Interrupt
- #define: `CAN_IT_FF0((uint32_t)0x00000004)`
  FIFO 0 full Interrupt

- #define: `CAN_IT_FOV0((uint32_t)0x00000008)`
  FIFO 0 overrun Interrupt

- #define: `CAN_IT_FMP1((uint32_t)0x00000010)`
  FIFO 1 message pending Interrupt

- #define: `CAN_IT_FF1((uint32_t)0x00000020)`
  FIFO 1 full Interrupt

- #define: `CAN_IT_FOV1((uint32_t)0x00000040)`
  FIFO 1 overrun Interrupt

- #define: `CAN_IT_WKU((uint32_t)0x00010000)`
  Wake-up Interrupt

- #define: `CAN_IT_SLK((uint32_t)0x00020000)`
  Sleep acknowledge Interrupt

- #define: `CAN_IT_EWG((uint32_t)0x00000100)`
  Error warning Interrupt

- #define: `CAN_IT_EPV((uint32_t)0x00000200)`
  Error passive Interrupt

- #define: `CAN_IT_BOF((uint32_t)0x00000400)`
  Bus-off Interrupt

- #define: `CAN_IT_LEC((uint32_t)0x00000800)`
  Last error code Interrupt

- #define: `CAN_IT_ERR((uint32_t)0x00008000)`
  Error Interrupt
• #define: `CAN_IT_RQCP0_CAN_IT_TME`

• #define: `CAN_IT_RQCP1_CAN_IT_TME`

• #define: `CAN_IT_RQCP2_CAN_IT_TME`

`CAN_operating_mode`

• #define: `CAN_Mode_Normal((uint8_t)0x00)`
  normal mode

• #define: `CAN_Mode_LoopBack((uint8_t)0x01)`
  loopback mode

• #define: `CAN_Mode_Silent((uint8_t)0x02)`
  silent mode

• #define: `CAN_Mode_Silent_LoopBack((uint8_t)0x03)`
  loopback combined with silent mode

• #define: `CAN_OperatingMode_Initialization((uint8_t)0x00)`
  Initialization mode

• #define: `CAN_OperatingMode_Normal((uint8_t)0x01)`
  Normal mode

• #define: `CAN_OperatingMode_Sleep((uint8_t)0x02)`
  sleep mode

`CAN_operating_mode_status`

• #define: `CAN_ModeStatus_Failed((uint8_t)0x00)`
  CAN entering the specific mode failed

• #define: `CAN_ModeStatus_Success((uint8_t)!CAN_ModeStatus_Failed)}`
CAN entering the specific mode Succeed

**CAN_receive_FIFO_number_constants**
- #define: `CAN_FIFO0((uint8_t)0x00)`
  CAN FIFO 0 used to receive
- #define: `CAN_FIFO1((uint8_t)0x01)`
  CAN FIFO 1 used to receive

**CAN_remote_transmission_request**
- #define: `CAN_RTR_Data((uint32_t)0x00000000)`
  Data frame
- #define: `CAN_RTR_Remote((uint32_t)0x00000002)`
  Remote frame
- #define: `CAN_RTR_DATA_CAN_RTR_Data`
- #define: `CAN_RTR_REMOTE_CAN_RTR_Remote`

**CAN_sleep_constants**
- #define: `CAN_Sleep_Failed((uint8_t)0x00)`
  CAN did not enter the sleep mode
- #define: `CAN_Sleep_OK((uint8_t)0x01)`
  CAN entered the sleep mode
- #define: `CANSLEEPFAILED_CAN_Sleep_Failed`
- #define: `CANSLEEPOK_CAN_Sleep_OK`
• #define: **CAN_SJW_1tq**((uint8_t)0x00)
  1 time quantum

• #define: **CAN_SJW_2tq**((uint8_t)0x01)
  2 time quantum

• #define: **CAN_SJW_3tq**((uint8_t)0x02)
  3 time quantum

• #define: **CAN_SJW_4tq**((uint8_t)0x03)
  4 time quantum

**CAN_time_quantum_in_bit_segment_1**

• #define: **CAN_BS1_1tq**((uint8_t)0x00)
  1 time quantum

• #define: **CAN_BS1_2tq**((uint8_t)0x01)
  2 time quantum

• #define: **CAN_BS1_3tq**((uint8_t)0x02)
  3 time quantum

• #define: **CAN_BS1_4tq**((uint8_t)0x03)
  4 time quantum

• #define: **CAN_BS1_5tq**((uint8_t)0x04)
  5 time quantum

• #define: **CAN_BS1_6tq**((uint8_t)0x05)
  6 time quantum

• #define: **CAN_BS1_7tq**((uint8_t)0x06)
  7 time quantum

• #define: **CAN_BS1_8tq**((uint8_t)0x07)
  8 time quantum
• #define: `CAN_BS1_9tq((uint8_t)0x08)`
  9 time quantum

• #define: `CAN_BS1_10tq((uint8_t)0x09)`
  10 time quantum

• #define: `CAN_BS1_11tq((uint8_t)0x0A)`
  11 time quantum

• #define: `CAN_BS1_12tq((uint8_t)0x0B)`
  12 time quantum

• #define: `CAN_BS1_13tq((uint8_t)0x0C)`
  13 time quantum

• #define: `CAN_BS1_14tq((uint8_t)0x0D)`
  14 time quantum

• #define: `CAN_BS1_15tq((uint8_t)0x0E)`
  15 time quantum

• #define: `CAN_BS1_16tq((uint8_t)0x0F)`
  16 time quantum

`CAN_time_quantum_in_bit_segment_2`

• #define: `CAN_BS2_1tq((uint8_t)0x00)`
  1 time quantum

• #define: `CAN_BS2_2tq((uint8_t)0x01)`
  2 time quantum

• #define: `CAN_BS2_3tq((uint8_t)0x02)`
  3 time quantum

• #define: `CAN_BS2_4tq((uint8_t)0x03)`
Controller area network (CAN)

4 time quantum

- \#define: \texttt{CAN\_BS2\_5tq((uint8\_t)0x04)}

5 time quantum

- \#define: \texttt{CAN\_BS2\_6tq((uint8\_t)0x05)}

6 time quantum

- \#define: \texttt{CAN\_BS2\_7tq((uint8\_t)0x06)}

7 time quantum

- \#define: \texttt{CAN\_BS2\_8tq((uint8\_t)0x07)}

8 time quantum

\textbf{CAN\_transmit\_constants}

- \#define: \texttt{CAN\_TxStatus\_Failed((uint8\_t)0x00)}
  CAN transmission failed

- \#define: \texttt{CAN\_TxStatus\_Ok((uint8\_t)0x01)}
  CAN transmission succeeded

- \#define: \texttt{CAN\_TxStatus\_Pending((uint8\_t)0x02)}
  CAN transmission pending

- \#define: \texttt{CAN\_TxStatus\_NoMailBox((uint8\_t)0x04)}
  CAN cell did not provide an empty mailbox

- \#define: \texttt{CANTXFAILED\_CAN\_TxStatus\_Failed}

- \#define: \texttt{CANTXOK\_CAN\_TxStatus\_Ok}

- \#define: \texttt{CANTXPENDING\_CAN\_TxStatus\_Pending}
### CAN_wake_up_constants

- **#define**: `CAN_WakeUp_Failed((uint8_t)0x00)`
  
  CAN did not leave the sleep mode

- **#define**: `CAN_WakeUp_Ok((uint8_t)0x01)`
  
  CAN leaved the sleep mode

- **#define**: `CANWAKEUPFAILED CAN_WakeUp_Failed`

- **#define**: `CANWAKEUPOK CAN_WakeUp_Ok`

### 4.4 CAN Programming Example

The example below provides a typical configuration of the CAN peripheral. For more examples about CAN configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\CAN\\

```c
GPIO_InitTypeDef GPIO_InitStructure;
CAN_InitTypeDef CAN_InitStructure;
CAN_FilterInitTypeDef CAN_FilterInitStructure;

/* CAN GPIOs configuration ***********************************/
/* Enable GPIOD clock */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);

/* Connect PD1 to CAN1_Tx pin */
GPIO_PinAFConfig(GPIOD, GPIO_PinSource0, GPIO_AF_CAN1);
/* Connect PD0 to CAN1_Rx pin */
GPIO_PinAFConfig(GPIOD, GPIO_PinSource1, GPIO_AF_CAN1);

/* Configure CAN1_Rx(PD0) and CAN1_Tx(PD1) pins */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1;
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(GPIOIOD, &GPIO_InitStructure);

/* CAN configuration *******************************************/
```
/* Enable CAN1 clock */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);

/* CAN cell init */
CAN_InitStructure.CAN_TTCM = DISABLE;
CAN_InitStructure.CAN_ABOM = DISABLE;
CAN_InitStructure.CAN_AWUM = DISABLE;
CAN_InitStructure.CAN_NART = DISABLE;
CAN_InitStructure.CAN_RFLM = DISABLE;
CAN_InitStructure.CAN_TXFP = DISABLE;
CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;
CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;
/* CAN Baudrate = 1MBps (CAN clocked at 30 MHz) */
CAN_InitStructure.CAN_BS1 = CAN_BS1_6tq;
CAN_InitStructure.CAN_BS2 = CAN_BS2_8tq;
CAN_InitStructure.CAN_Prescaler = 2;
CAN_Init(CAN1, &CAN_InitStructure);

/* CAN filter init */
CAN_FilterInitStructure.CAN_FilterNumber = 0;
CAN_FilterInitStructure.CAN_FilterMode = CAN_FilterMode_IdMask;
CAN_FilterInitStructure.CAN_FilterScale = CAN_FilterScale_32bit;
CAN_FilterInitStructure.CAN_FilterIdHigh = 0x0000;
CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
CAN_FilterInitStructure.CAN_FilterMaskIdHigh = 0x0000;
CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0x0000;
CAN_FilterInitStructure.CAN_FilterFIFOAssignment = 0;
CAN_FilterInitStructure.CAN_FilterActivation = ENABLE;
CAN_FilterInit(&CAN_FilterInitStructure);
5  **CRC calculation unit (CRC)**

5.1  **CRC Firmware driver registers structures**

5.1.1  **CRC_TypeDef**

`CRC_TypeDef` is defined in the `stm32f2xx.h` file and contains the CRC registers definition.

**Data Fields**

- `__IO uint32_t DR`
- `__IO uint8_t IDR`
- `uint8_t RESERVED0`
- `uint16_t RESERVED1`
- `__IO uint32_t CR`

**Field Documentation**

- `__IO uint32_t CRC_TypeDef::DR`
  - CRC Data register, Address offset: 0x00
- `__IO uint8_t CRC_TypeDef::IDR`
  - CRC Independent data register, Address offset: 0x04
- `uint8_t CRC_TypeDef::RESERVED0`
  - Reserved, 0x05
- `uint16_t CRC_TypeDef::RESERVED1`
  - Reserved, 0x06
- `__IO uint32_t CRC_TypeDef::CR`
  - CRC Control register, Address offset: 0x08

5.2  **CRC Firmware driver API description**

The following section lists the various functions of the CRC library.

**Functions**

- `CRC_ResetDR()`
- `CRC_CalcCRC()`  
- `CRC_CalcBlockCRC()`  
- `CRC_GetCRC()`  
- `CRC_SetIDRegister()`  
- `CRC_GetIDRegister()`
5.2.1 Functions

5.2.1.1 CRC_ResetDR

Function Name: void CRC_ResetDR ( void )
Function Description: Resets the CRC Data register (DR).
Parameters:
- None.
Return values:
- None.
Notes:
- None.

5.2.1.2 CRC_CalcCRC

Function Name: uint32_t CRC_CalcCRC ( uint32_t Data)
Function Description: Computes the 32-bit CRC of a given data word (32-bit).
Parameters:
- Data: data word (32-bit) to compute its CRC
Return values:
- 32-bit CRC
Notes:
- None.

5.2.1.3 CRC_CalcBlockCRC

Function Name: uint32_t CRC_CalcBlockCRC ( uint32_t pBuffer, uint32_t BufferLength)
Function Description: Computes the 32-bit CRC of a given buffer of data word (32-bit).
Parameters:
- pBuffer: pointer to the buffer containing the data to be computed
- BufferLength: length of the buffer to be computed
Return values:
- 32-bit CRC
Notes:
- None.
5.2.1.4 CRC_GetCRC

<table>
<thead>
<tr>
<th>Function Name</th>
<th>uint32_t CRC_GetCRC ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Returns the current CRC value.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• None.</td>
</tr>
<tr>
<td>Return values</td>
<td>• 32-bit CRC</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>

5.2.1.5 CRC_SetIDRegister

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void CRC_SetIDRegister ( uint8_t IDValue)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Stores a 8-bit data in the Independent Data(ID) register.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• IDValue : 8-bit value to be stored in the ID register</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>

5.2.1.6 CRC_GetIDRegister

<table>
<thead>
<tr>
<th>Function Name</th>
<th>uint8_t CRC_GetIDRegister ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Returns the 8-bit data stored in the Independent Data(ID) register.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• None.</td>
</tr>
<tr>
<td>Return values</td>
<td>• 8-bit value of the ID register</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>
5.3 **CRC Programming Example**

The example below explains how to compute the 32-bit CRC of a given data word (32-bit).
For more examples about CRC configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\CRC\.

```c
__IO uint32_t CRCValue = 0;

/* Enable CRC clock */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_CRC, ENABLE);

/* Compute the CRC of data 0x5634d94c */
CRCValue = CRC_CalcCRC(0x5634d94c);
```
6 Cryptographic processor (CRYP)

6.1 CRYP Firmware driver registers structures

6.1.1 CRYP_TypeDef

`CRYP_TypeDef` is defined in the `stm32f2xx.h` file and contains the CRYP registers definition.

Data Fields

- `__IO uint32_t CR`
- `__IO uint32_t SR`
- `__IO uint32_t DR`
- `__IO uint32_t DOUT`
- `__IO uint32_t DMACR`
- `__IO uint32_t IMSCR`
- `__IO uint32_t RISR`
- `__IO uint32_t MISR`
- `__IO uint32_t K0LR`
- `__IO uint32_t K0RR`
- `__IO uint32_t K1LR`
- `__IO uint32_t K1RR`
- `__IO uint32_t K2LR`
- `__IO uint32_t K2RR`
- `__IO uint32_t K3LR`
- `__IO uint32_t K3RR`
- `__IO uint32_t IV0LR`
- `__IO uint32_t IV0RR`
- `__IO uint32_t IV1LR`
- `__IO uint32_t IV1RR`

Field Documentation

- `__IO uint32_t CRYP_TypeDef::CR`
  - CRYP control register, Address offset: 0x00
- `__IO uint32_t CRYP_TypeDef::SR`
  - CRYP status register, Address offset: 0x04
- `__IO uint32_t CRYP_TypeDef::DR`
  - CRYP data input register, Address offset: 0x08
- `__IO uint32_t CRYP_TypeDef::DOUT`
  - CRYP data output register, Address offset: 0x0C
- `__IO uint32_t CRYP_TypeDef::DMACR`
  - CRYP DMA control register, Address offset: 0x10
- `__IO uint32_t CRYP_TypeDef::IMSCR`
  - CRYP interrupt mask set/clear register, Address offset: 0x14
- `__IO uint32_t CRYP_TypeDef::RISR`
  - CRYP raw interrupt status register, Address offset: 0x18
- `__IO uint32_t CRYP_TypeDef::MISR`
Cryptographic processor (CRYP)

- CRYP masked interrupt status register, Address offset: 0x1C
- __IO uint32_t CRYP_TypeDef::K0LR
  - CRYP key left register 0, Address offset: 0x20
- __IO uint32_t CRYP_TypeDef::K0RR
  - CRYP key right register 0, Address offset: 0x24
- __IO uint32_t CRYP_TypeDef::K1LR
  - CRYP key left register 1, Address offset: 0x28
- __IO uint32_t CRYP_TypeDef::K1RR
  - CRYP key right register 1, Address offset: 0x2C
- __IO uint32_t CRYP_TypeDef::K2LR
  - CRYP key left register 2, Address offset: 0x30
- __IO uint32_t CRYP_TypeDef::K2RR
  - CRYP key right register 2, Address offset: 0x34
- __IO uint32_t CRYP_TypeDef::K3LR
  - CRYP key left register 3, Address offset: 0x38
- __IO uint32_t CRYP_TypeDef::K3RR
  - CRYP key right register 3, Address offset: 0x3C
- __IO uint32_t CRYP_TypeDef::IV0LR
  - CRYP initialization vector left-word register 0, Address offset: 0x40
- __IO uint32_t CRYP_TypeDef::IV0RR
  - CRYP initialization vector right-word register 0, Address offset: 0x44
- __IO uint32_t CRYP_TypeDef::IV1LR
  - CRYP initialization vector left-word register 1, Address offset: 0x48
- __IO uint32_t CRYP_TypeDef::IV1RR
  - CRYP initialization vector right-word register 1, Address offset: 0x4C

6.1.2 CRYP_InitTypeDef

`CRYP_InitTypeDef` is defined in the `stm32f2xx_cryp.h` file and contains the CRYP initialization parameters.

Data Fields

- uint16_t CRYP_AlgoDir
- uint16_t CRYP_AlgoMode
- uint16_t CRYP_DataType
- uint16_t CRYP_KeySize

Field Documentation

- `uint16_t CRYP_InitTypeDef::CRYP_AlgoDir`
  - Encrypt or Decrypt. This parameter can be a value of `CRYP_Algorithm_Direction`
- `uint16_t CRYP_InitTypeDef::CRYP_AlgoMode`
  - TDES-ECB, TDES-CBC, DES-ECB, DES-CBC, AES-ECB, AES-CBC, AES-CTR, AES-Key. This parameter can be a value of `CRYP_Algorithm_Mode`
- `uint16_t CRYP_InitTypeDef::CRYP_DataType`
  - 32-bit data, 16-bit data, bit data or bit-string. This parameter can be a value of `CRYP_Data_Type`
- `uint16_t CRYP_InitTypeDef::CRYP_KeySize`
6.1.3 CRYP_KeyInitStructDef

`CRYP_KeyInitStructDef` is defined in the `stm32f2xx_cryp.h` file and contains the CRYP keys initialization parameters.

**Data Fields**

- `uint32_t CRYP_Key0Left`
- `uint32_t CRYP_Key0Right`
- `uint32_t CRYP_Key1Left`
- `uint32_t CRYP_Key1Right`
- `uint32_t CRYP_Key2Left`
- `uint32_t CRYP_Key2Right`
- `uint32_t CRYP_Key3Left`
- `uint32_t CRYP_Key3Right`

**Field Documentation**

- `uint32_t CRYP_KeyInitStructDef::CRYP_Key0Left`
  - Key 0 Left
- `uint32_t CRYP_KeyInitStructDef::CRYP_Key0Right`
  - Key 0 Right
- `uint32_t CRYP_KeyInitStructDef::CRYP_Key1Left`
  - Key 1 left
- `uint32_t CRYP_KeyInitStructDef::CRYP_Key1Right`
  - Key 1 Right
- `uint32_t CRYP_KeyInitStructDef::CRYP_Key2Left`
  - Key 2 left
- `uint32_t CRYP_KeyInitStructDef::CRYP_Key2Right`
  - Key 2 Right
- `uint32_t CRYP_KeyInitStructDef::CRYP_Key3Left`
  - Key 3 left
- `uint32_t CRYP_KeyInitStructDef::CRYP_Key3Right`
  - Key 3 Right

6.1.4 CRYP_IVInitStructDef

`CRYP_IVInitStructDef` is defined in the `stm32f2xx_cryp.h` file and contains the CRYP initialization Vectors(IV).

**Data Fields**

- `uint32_t CRYP_IV0Left`
- `uint32_t CRYP_IV0Right`
- `uint32_t CRYP_IV1Left`
• `uint32_t CRYP_IV1Right`

Field Documentation

• `uint32_t CRYP_IVInitTypeDef::CRYP_IV0Left`
  – Init Vector 0 Left
• `uint32_t CRYP_IVInitTypeDef::CRYP_IV0Right`
  – Init Vector 0 Right
• `uint32_t CRYP_IVInitTypeDef::CRYP_IV1Left`
  – Init Vector 1 left
• `uint32_t CRYP_IVInitTypeDef::CRYP_IV1Right`
  – Init Vector 1 Right

6.1.5 CRYP_Context

`CRYP_Context` is defined in the `stm32f2xx_cryp.h`

Data Fields

• `uint32_t CR_bits9to2`
• `uint32_t CRYP_IV0LR`
• `uint32_t CRYP_IV0RR`
• `uint32_t CRYP_IV1LR`
• `uint32_t CRYP_IV1RR`
• `uint32_t CRYP_K0LR`
• `uint32_t CRYP_K0RR`
• `uint32_t CRYP_K1LR`
• `uint32_t CRYP_K1RR`
• `uint32_t CRYP_K2LR`
• `uint32_t CRYP_K2RR`
• `uint32_t CRYP_K3LR`
• `uint32_t CRYP_K3RR`

Field Documentation

• `uint32_t CRYP_Context::CR_bits9to2`
  – < Configuration KEY
• `uint32_t CRYP_Context::CRYP_IV0LR`
• `uint32_t CRYP_Context::CRYP_IV0RR`
• `uint32_t CRYP_Context::CRYP_IV1LR`
• `uint32_t CRYP_Context::CRYP_IV1RR`
  – IV
• `uint32_t CRYP_Context::CRYP_K0LR`
• `uint32_t CRYP_Context::CRYP_K0RR`
• `uint32_t CRYP_Context::CRYP_K1LR`
• `uint32_t CRYP_Context::CRYP_K1RR`
• `uint32_t CRYP_Context::CRYP_K2LR`
• `uint32_t CRYP_Context::CRYP_K2RR`
6.2 CRYP Firmware driver API description

The following section lists the various functions of the CRYP library.

6.2.1 How to use this driver

1. Enable the CRYP controller clock using RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_CRYP, ENABLE); function.
2. Initialize the CRYP using CRYP_Init(), CRYP_KeyInit() and if needed CRYP_IVInit().
3. Flush the IN and OUT FIFOs by using CRYP_FIFOFlush() function.
4. Enable the CRYP controller using the CRYP_Cmd() function.
5. If using DMA for Data input and output transfer, Activate the needed DMA Requests using CRYP_DMA_Cmd() function.
6. If DMA is not used for data transfer, use CRYP_DataIn() and CRYP_DataOut() functions to enter data to IN FIFO and get result from OUT FIFO.
7. To control CRYP events you can use one of the following two methods:
   a. Check on CRYP flags using the CRYP_GetFlagStatus() function.
   b. Use CRYP interrupts through the function CRYP_ITConfig() at initialization phase and CRYP_GetITStatus() function into interrupt routines in processing phase.
8. Save and restore Cryptographic processor context using CRYP_SaveContext() and CRYP_RestoreContext() functions.

Procedure to perform an encryption or a decryption

- Initialization
  a. Initialize the peripheral using CRYP_Init(), CRYP_KeyInit() and CRYP_IVInit functions:
     a. Configure the key size (128-, 192- or 256-bit, in the AES only)
     b. Enter the symmetric key
     c. Configure the data type
     d. In case of decryption in AES-ECB or AES-CBC, you must prepare the key: configure the key preparation mode. Then Enable the CRYP peripheral using CRYP_Cmd() function: the BUSY flag is set. Wait until BUSY flag is reset : the key is prepared for decryption
     e. Configure the algorithm and chaining (the DES/TDES in ECB/CBC, the AES in ECB/CBC/CTR)
     f. Configure the direction (encryption/decryption). - Write the initialization vectors (in CBC or CTR modes only)
  b. Flush the IN and OUT FIFOs using the CRYP_FIFOFlush() function

- Basic Processing mode (polling mode)
  a. Enable the cryptographic processor using CRYP_Cmd() function.
  b. Write the first blocks in the input FIFO (2 to 8 words) using CRYP_DataIn() function.
  c. Repeat the following sequence until the complete message has been processed:
     a. Wait for flag CRYP_FLAG_OFNE occurs (using CRYP_GetFlagStatus() function), then read the OUT-FIFO using CRYP_DataOut() function (1 block or until the FIFO is empty)
Cryptographic processor (CRYP)

b. Wait for flag CRYP_FLAG_IFNF occurs, (using CRYP_GetFlagStatus() function then write the IN FIFO using CRYP_DataIn() function (1 block or until the FIFO is full)

d. At the end of the processing, CRYP_FLAG_BUSY flag will be reset and both FIFOs are empty (CRYP_FLAG_IFEM is set and CRYP_FLAG_OFNE is reset). You can disable the peripheral using CRYP_Cmd() function.

- Intermittent Processing mode In this mode, Processing is done when the data are transferred by the CPU during interrupts. Enable the interrupts CRYP_IT_INI and CRYP_IT_OUTI using CRYP_ITConfig() function. Enable the cryptographic processor using CRYP_Cmd() function. In the CRYP_IT_INI interrupt handler : load the input message into the IN FIFO using CRYP_DataIn() function. You can load 2 or 4 words at a time, or load data until the IN FIFO is full. When the last word of the message has been entered into the IN FIFO, disable the CRYP_IT_INI interrupt (using CRYP_ITConfig() function). In the CRYP_IT_OUTI interrupt handler : read the output message from the OUT FIFO using CRYP_DataOut() function. You can read 1 block (2 or 4 words) at a time or read data until the FIFO is empty. When the last word has been read, INIM=0, BUSY=0 and both FIFOs are empty (CRYP_FLAG_IFEM is set and CRYP_FLAG_OFNE is reset). You can disable the CRYP_IT_OUTI interrupt (using CRYP_ITConfig() function) and you can disable the peripheral using CRYP_Cmd() function.

- DMA Processing mode In this mode, Processing is done when the DMA is used to transfer the data from/to the memory.

a. Configure the DMA controller to transfer the input data from the memory using DMA_Init() function. The transfer length is the length of the message. As message padding is not managed by the peripheral, the message length must be an entire number of blocks. The data are transferred in burst mode. The burst length is 4 words in the AES and 2 or 4 words in the DES/TDES. The DMA should be configured to set an interrupt on transfer completion of the output data to indicate that the processing is finished. Refer to DMA peripheral driver for more details.

b. Enable the cryptographic processor using CRYP_Cmd() function. Enable the DMA requests CRYP_DMAReq_DataIN and CRYP_DMAReq_DataOUT using CRYP_DMACmd() function.

c. All the transfers and processing are managed by the DMA and the cryptographic processor. The DMA transfer complete interrupt indicates that the processing is complete. Both FIFOs are normally empty and CRYP_FLAG_BUSY flag is reset.

### 6.2.2 Initialization and configuration

This section provides functions allowing to:

- Initialize the cryptographic Processor using CRYP_Init() function
- Encrypt or Decrypt
- Mode : TDES-ECB, TDES-CBC, DES-ECB, DES-CBC, AES-ECB, AES-CBC, AES-CTR, AES-Key
- DataType : 32-bit data, 16-bit data, bit data or bit-string
- Key Size (only in AES modes)
- Configure the Encrypt or Decrypt Key using CRYP_KeyInit() function
- Configure the Initialization Vectors(IV) for CBC and CTR modes using CRYP_IVInit() function.
- Flushes the IN and OUT FIFOs : using CRYP_FIFOFlush() function.
- Enable or disable the CRYP Processor using CRYP_Cmd() function.

Below is the list of functions used to initialize and configure the cryptographic processor:
CRYP Data processing functions

This section provides functions allowing the encryption and decryption operations:

- Enter data to be treated in the IN FIFO: using CRYP_DataIn() function.
- Get the data result from the OUT FIFO: using CRYP_DataOut() function.

The data processing functions are the following:

- CRYP_DataIn()
- CRYP_DataOut()

Context swapping functions

This section provides functions allowing to save and store CRYP Context. It is possible to interrupt an encryption/decryption/key generation process to perform another processing with a higher priority, and to complete the interrupted process later on, when the higher-priority task is complete. To do so, the context of the interrupted task must be saved from the CRYP registers to memory, and then be restored from memory to the CRYP registers.

1. To save the current context, use CRYP_SaveContext() function
2. To restore the saved context, use CRYP_RestoreContext() function

- CRYP_SaveContext()
- CRYP_RestoreContext()

CRYP DMA interface Configuration function

This section provides functions allowing to configure the DMA interface for CRYP data input and output transfer. When the DMA mode is enabled (using the CRYP_DMACmd() function), data can be transferred:

- From memory to the CRYP IN FIFO using the DMA peripheral by enabling the CRYP_DMAReq_DataIN request.
- From the CRYP OUT FIFO to the memory using the DMA peripheral by enabling the CRYP_DMAReq_DataOUT request.

The function that can be used for CRYP DMA interface Configuration is:

- CRYP_DMACmd()

6.2.3 Interrupt and flag management

This section provides functions allowing to configure the CRYP Interrupts and to get the status and Interrupts pending bits.

The CRYP provides 7 Flags and 2 Interrupt sources:

Flags

- CRYP_FLAG_IFEM: Set when Input FIFO is empty. This Flag is cleared only by hardware.
Cryptographic processor (CRYP)

- **CRYP_FLAG_IFNF**: Set when Input FIFO is not full. This Flag is cleared only by hardware.
- **CRYP_FLAG_INRIS**: Set when Input FIFO Raw interrupt is pending it gives the raw interrupt state prior to masking of the input FIFO service interrupt. This Flag is cleared only by hardware.
- **CRYP_FLAG_OFNE**: Set when Output FIFO not empty. This Flag is cleared only by hardware.
- **CRYP_FLAG_OFFU**: Set when Output FIFO is full. This Flag is cleared only by hardware.
- **CRYP_FLAG_OUTRIS**: Set when Output FIFO Raw interrupt is pending it gives the raw interrupt state prior to masking of the output FIFO service interrupt. This Flag is cleared only by hardware.
- **CRYP_FLAG_BUSY**: Set when the CRYP core is currently processing a block of data or a key preparation (for AES decryption). This Flag is cleared only by hardware. To clear it, the CRYP core must be disabled and the last processing has completed.

**Interrupts**

- **CRYP_IT_INI**: The input FIFO service interrupt is asserted when there are less than 4 words in the input FIFO. This interrupt is associated to CRYP_FLAG_INRIS flag. This interrupt is cleared by performing write operations to the input FIFO until it holds 4 or more words. The input FIFO service interrupt INMIS is enabled with the CRYP enable bit. Consequently, when CRYP is disabled, the INMIS signal is low even if the input FIFO is empty.
- **CRYP_IT_OUTI**: The output FIFO service interrupt is asserted when there is one or more (32-bit word) data items in the output FIFO. This interrupt is associated to CRYP_FLAG_OUTRIS flag. This interrupt is cleared by reading data from the output FIFO until there is no valid (32-bit) word left (that is, the interrupt follows the state of the OFNE (output FIFO not empty) flag).

**Managing the CRYP controller events**

The user should identify which mode will be used in his application to manage the CRYP controller events: Polling mode or Interrupt mode.

- In the Polling Mode it is advised to use the CRYP_GetFlagStatus() function to check if flags events occurred. The CRYPT flags do not need to be cleared since they are cleared as soon as the associated event are reset.
- In the Interrupt Mode it is advised to use the following functions: The CRYPT interrupts have no pending bits, the interrupt is cleared as soon as the associated event is reset.
  - CRYP_ITConfig(): to enable or disable the interrupt source.
  - CRYP_GetITStatus(): to check if Interrupt occurs.

The functions used to manage the CRYP controller event are the following:

- **CRYP_ITConfig()**
- **CRYP_GetITStatus()**
- **CRYP_GetFlagStatus()**

**6.2.4 High level functions**

The library also includes high level function:

**High Level AES functions**

- **CRYP_AES_ECB()**
- **CRYP_AES_CBC()**
6.2.5 Initialization and configuration functions

6.2.5.1 CRYP_DeInit

Function Name: void CRYP_DeInit ( void )
Function Description: Deinitializes the CRYP peripheral registers to their default reset values.
Parameters:
  • None.
Return values:
  • None.
Notes:
  • None.

6.2.5.2 CRYP_Init

Function Name: void CRYP_Init ( CRYP_InitTypeDef * CRYP_InitStruct )
Function Description: Initializes the CRYP peripheral according to the specified parameters in the CRYP_InitStruct.
Parameters:
  • CRYP_InitStruct: pointer to a CRYP_InitTypeDef structure that contains the configuration information for the CRYP peripheral.
Return values:
  • None.
Notes:
  • None.

6.2.5.3 CRYP_StructInit
6.2.5.4 CRYP_KeyInit

Function Name  void CRYP_KeyInit ( CRYP_KeyInitTypeDef * CRYP_KeyInitStruct)
Function Description  Initializes the CRYP Keys according to the specified parameters in the CRYP_KeyInitStruct.
Parameters  •  CRYP_KeyInitStruct : pointer to a CRYP_KeyInitTypeDef structure that contains the configuration information for the CRYP Keys.
Return values  •  None.
Notes  •  None.

6.2.5.5 CRYP_KeyStructInit

Function Name  void CRYP_KeyStructInit ( CRYP_KeyInitTypeDef * CRYP_KeyInitStruct)
Function Description  Fills each CRYP_KeyInitStruct member with its default value.
Parameters  •  CRYP_KeyInitStruct : pointer to a CRYP_KeyInitTypeDef structure which will be initialized.
Return values  •  None.
Notes  •  None.
6.2.5.6 CRYP_IVInit

Function Name: void CRYP_IVInit (CRYP_IVInitTypeDef * CRYP_IVInitStruct)

Function Description: Initializes the CRYP Initialization Vectors(IV) according to the specified parameters in the CRYP_IVInitStruct.

Parameters:
- CRYP_IVInitStruct: pointer to a CRYP_IVInitTypeDef structure that contains the configuration information for the CRYP Initialization Vectors(IV).

Return values:
- None.

Notes:
- None.

6.2.5.7 CRYP_IVStructInit

Function Name: void CRYP_IVStructInit (CRYP_IVInitTypeDef * CRYP_IVInitStruct)

Function Description: Fills each CRYP_IVInitStruct member with its default value.

Parameters:
- CRYP_IVInitStruct: pointer to a CRYP_IVInitTypeDef Initialization Vectors(IV) structure which will be initialized.

Return values:
- None.

Notes:
- None.

6.2.5.8 CRYP_FIFOFlush

Function Name: void CRYP_FIFOFlush (void)

Function Description: Flushes the IN and OUT FIFOs (that is read and write pointers of the FIFOs are reset)

Parameters:
- None.

Return values:
- None.

Notes:
- The FIFOs must be flushed only when BUSY flag is reset.
6.2.5.9 CRYP_Cmd

Function Name: void CRYP_Cmd ( FunctionalState NewState)
Function Description: Enables or disables the CRYP peripheral.
Parameters:
- NewState: new state of the CRYP peripheral. This parameter can be: ENABLE or DISABLE.
Return values:
- None.
Notes:
- None.

6.2.6 CRYP Data processing functions

6.2.6.1 CRYP_DataIn

Function Name: void CRYP_DataIn ( uint32_t Data)
Function Description: Writes data in the Data Input register (DIN).
Parameters:
- Data: data to write in Data Input register
Return values:
- None.
Notes:
- After the DIN register has been read once or several times, the FIFO must be flushed (using CRYP_FIFOFlush() function).

6.2.6.2 CRYP_DataOut

Function Name: uint32_t CRYP_DataOut ( void )
Function Description: Returns the last data entered into the output FIFO.
Parameters:
- None.
Return values:
- Last data entered into the output FIFO.
Notes:
- None.
6.2.7 Context swapping functions

6.2.7.1 CRYP_SaveContext

<table>
<thead>
<tr>
<th>Function Name</th>
<th>ErrorStatus CRYP_SaveContext (CRYP_Context * CRYP_ContextSave, CRYP_KeyInitTypeDef * CRYP_KeyInitStruct)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Saves the CRYP peripheral Context.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• CRYP_ContextSave: pointer to a CRYP_Context structure that contains the repository for current context.</td>
</tr>
<tr>
<td></td>
<td>• CRYP_KeyInitStruct: pointer to a CRYP_KeyInitTypeDef structure that contains the configuration information for the CRYP Keys.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• This function stops DMA transfer before to save the context. After restoring the context, you have to enable the DMA again (if the DMA was previously used).</td>
</tr>
</tbody>
</table>

6.2.7.2 CRYP_RestoreContext

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void CRYP_RestoreContext (CRYP_Context * CRYP_ContextRestore)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Restores the CRYP peripheral Context.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• CRYP_ContextRestore: pointer to a CRYP_Context structure that contains the repository for saved context.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• Since the DMA transfer is stopped in CRYP_SaveContext() function, after restoring the context, you have to enable the DMA again (if the DMA was previously used).</td>
</tr>
<tr>
<td></td>
<td>• The data that were saved during context saving must be rewrited into the IN FIFO.</td>
</tr>
</tbody>
</table>
6.2.8 CRYPTO DMA interface Configuration function

6.2.8.1 CRYP_DMACmd

Function Name: void CRYP_DMACmd ( uint8_t CRYP_DMAReq, FunctionalState NewState)

Function Description: Enables or disables the CRYP DMA interface.

Parameters:
- **CRYP_DMAReq**: specifies the CRYP DMA transfer request to be enabled or disabled. This parameter can be any combination of the following values:
  - **CRYP_DMAReq_DataOUT**: DMA for outgoing (Tx) data transfer
  - **CRYP_DMAReq_DataIN**: DMA for incoming (Rx) data transfer
- **NewState**: new state of the selected CRYP DMA transfer request. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.

6.2.9 Interrupt and flag management functions

6.2.9.1 CRYP_ITConfig

Function Name: void CRYP_ITConfig ( uint8_t CRYP_IT, FunctionalState NewState)

Function Description: Enables or disables the specified CRYP interrupts.

Parameters:
- **CRYP_IT**: specifies the CRYP interrupt source to be enabled or disabled. This parameter can be any combination of the following values:
  - **CRYP_IT_INI**: Input FIFO interrupt
  - **CRYP_IT_OUTI**: Output FIFO interrupt
- **NewState**: new state of the specified CRYP interrupt. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.
6.2.9.2 CRYP_GetITStatus

Function Name: ITStatus CRYP_GetITStatus (uint8_t CRYP_IT)
Function Description: Checks whether the specified CRYP interrupt has occurred or not.
Parameters:
- CRYP_IT: specifies the CRYP (masked) interrupt source to check. This parameter can be one of the following values:
  - CRYP_IT_INI: Input FIFO interrupt
  - CRYP_IT_OUTI: Output FIFO interrupt
Return values:
- The new state of CRYP_IT (SET or RESET).
Notes:
- This function checks the status of the masked interrupt (i.e. the interrupt should be previously enabled).

6.2.9.3 CRYP_GetFlagStatus

Function Name: FlagStatus CRYP_GetFlagStatus (uint8_t CRYP_FLAG)
Function Description: Checks whether the specified CRYP flag is set or not.
Parameters:
- CRYP_FLAG: specifies the CRYP flag to check. This parameter can be one of the following values:
  - CRYP_FLAG_IFEM: Input FIFO Empty flag.
  - CRYP_FLAG_IFNF: Input FIFO Not Full flag.
  - CRYP_FLAG_OFNE: Output FIFO Not Empty flag.
  - CRYP_FLAG_OFFU: Output FIFO Full flag.
  - CRYP_FLAG_BUSY: Busy flag.
  - CRYP_FLAG_OUTRIS: Output FIFO raw interrupt flag.
  - CRYP_FLAG_INRIS: Input FIFO raw interrupt flag.
Return values:
- The new state of CRYP_FLAG (SET or RESET).
Notes:
- None.

6.2.10 High Level AES functions

6.2.10.1 CRYP_AES_ECB

Function Name: ErrorStatus CRYP_AES_ECB (uint8_t Mode, uint8_t * Key, uint16_t Keysize, uint8_t * Input, uint32_t Ilength, uint8_t *
6.2.10.2 CRYP_AES_CBC

Function Name

```c
ErrorStatus CRYP_AES_CBC ( uint8_t Mode, uint8_t InitVectors, uint8_t * Key, uint16_t Keysize, uint8_t * Input, uint32_t Ilength, uint8_t * Output)
```

Function Description
Encrypt and decrypt using AES in CBC Mode.

Parameters
- **Mode**: encryption or decryption Mode. This parameter can be one of the following values:
  - **MODE_ENCRYPT**: Encryption
  - **MODE_DECRYPT**: Decryption
- **InitVectors**: Initialisation Vectors used for AES algorithm.
- **Key**: Key used for AES algorithm.
- **Keysize**: length of the Key, must be a 128, 192 or 256.
- **Input**: pointer to the Input buffer.
- **Ilength**: length of the Input buffer, must be a multiple of 16.
- **Output**: pointer to the returned buffer.

Return values
- An ErrorStatus enumeration value:
  - **SUCCESS**: Operation done
  - **ERROR**: Operation failed

Notes
- None.
6.2.10.3 CRYP_AES_CTR

**Function Name**

```
ErrorStatus CRYP_AES_CTR ( uint8_t Mode, uint8_t InitVectors, uint8_t * Key, uint16_t Keysize, uint8_t * Input, uint32_t Ilength, uint8_t * Output)
```

**Function Description**

Encrypt and decrypt using AES in CTR Mode.

**Parameters**

- **Mode**: encryption or decryption Mode. This parameter can be one of the following values:
  - `MODE_ENCRYPT`: Encryption
  - `MODE_DECRYPT`: Decryption
- **InitVectors**: Initialisation Vectors used for AES algorithm.
- **Key**: Key used for AES algorithm.
- **Keysize**: length of the Key, must be a 128, 192 or 256.
- **Input**: pointer to the Input buffer.
- **Ilength**: length of the Input buffer, must be a multiple of 16.
- **Output**: pointer to the returned buffer.

**Return values**

- An `ErrorStatus` enumeration value:
  - `SUCCESS`: Operation done
  - `ERROR`: Operation failed

**Notes**

- None.

6.2.11 High Level TDES functions

6.2.11.1 CRYP_TDES_ECB

**Function Name**

```
ErrorStatus CRYP_TDES_ECB ( uint8_t Mode, uint8_t Key, uint8_t * Input, uint32_t Ilength, uint8_t * Output)
```

**Function Description**

Encrypt and decrypt using TDES in ECB Mode.

**Parameters**

- **Mode**: encryption or decryption Mode. This parameter can be one of the following values:
  - `MODE_ENCRYPT`: Encryption
  - `MODE_DECRYPT`: Decryption
- **Key**: Key used for TDES algorithm.
- **Ilength**: length of the Input buffer, must be a multiple of 8.
- **Input**: pointer to the Input buffer.
- **Output**: pointer to the returned buffer.

**Return values**

- An `ErrorStatus` enumeration value:
  - `SUCCESS`: Operation done
  - `ERROR`: Operation failed

**Notes**

- None.
6.2.11.2 CRYP_TDES_CBC

Function Name: ErrorStatus CRYP_TDES_CBC (uint8_t Mode, uint8_t Key,
uint8_t InitVectors, uint8_t * Input, uint32_t Ilength, uint8_t * Output)

Function Description: Encrypt and decrypt using TDES in CBC Mode.

Parameters:
- Mode: encryption or decryption Mode. This parameter can be one of the following values:
  - MODE_ENCRYPT: Encryption
  - MODE_DECRYPT: Decryption
- Key: Key used for TDES algorithm.
- InitVectors: Initialisation Vectors used for TDES algorithm.
- Input: pointer to the Input buffer.
- Ilength: length of the Input buffer, must be a multiple of 8.
- Output: pointer to the returned buffer.

Return values:
- An ErrorStatus enumeration value:
  - SUCCESS: Operation done
  - ERROR: Operation failed

Notes:
- None.

6.2.12 High Level DES functions

6.2.12.1 CRYP_DES_ECB

Function Name: ErrorStatus CRYP_DES_ECB (uint8_t Mode, uint8_t Key,
uint8_t * Input, uint32_t Ilength, uint8_t * Output)

Function Description: Encrypt and decrypt using DES in ECB Mode.

Parameters:
- Mode: encryption or decryption Mode. This parameter can be one of the following values:
  - MODE_ENCRYPT: Encryption
  - MODE_DECRYPT: Decryption
- Key: Key used for DES algorithm.
- Ilength: length of the Input buffer, must be a multiple of 8.
- Input: pointer to the Input buffer.
- Output: pointer to the returned buffer.

Return values:
- An ErrorStatus enumeration value:
SUCCESS: Operation done
ERROR: Operation failed

Notes
• None.

6.2.12.2  CRYP_DES_CBC

Function Name
ErrorStatus CRYP_DES_CBC ( uint8_t Mode, uint8_t Key,
uint8_t InitVectors, uint8_t * Input, uint32_t Ilength, uint8_t * Output)

Function Description
Encrypt and decrypt using DES in CBC Mode.

Parameters
• Mode : encryption or decryption Mode. This parameter can be one of the following values:
  – MODE_ENCRYPT : Encryption
  – MODE_DECRYPT : Decryption
• Key : Key used for DES algorithm.
• InitVectors : Initialisation Vectors used for DES algorithm.
• Ilength : length of the Input buffer, must be a multiple of 8.
• Input : pointer to the Input buffer.
• Output : pointer to the returned buffer.

Return values
• An ErrorStatus enumeration value:
  – SUCCESS: Operation done
  – ERROR: Operation failed

Notes
• None.

6.3  CRYP Firmware driver defines

6.3.1  CRYP Firmware driver defines

CRYP

CRYP_Algorithm_Direction
• #define: CRYP_AlgoDir_Encrypt((uint16_t)0x0000)

• #define: CRYP_AlgoDir_Decrypt((uint16_t)0x0004)
Cryptographic processor (CRYP)

**CRYP_Algorithm_Mode**
- #define: `CRYP_AlgoMode_TDES_ECB((uint16_t)0x0000)
  < TDES Modes

- #define: `CRYP_AlgoMode_TDES_CBC((uint16_t)0x0008)
  DES Modes

- #define: `CRYP_AlgoMode_DES_ECB((uint16_t)0x0010)

- #define: `CRYP_AlgoMode_DES_CBC((uint16_t)0x0018)

- #define: `CRYP_AlgoMode_AES_ECB((uint16_t)0x0020)

- #define: `CRYP_AlgoMode_AES_CBC((uint16_t)0x0028)

- #define: `CRYP_AlgoMode_AES_CTR((uint16_t)0x0030)

- #define: `CRYP_AlgoMode_AES_Key((uint16_t)0x0038)

**CRYP_Data_Type**
- #define: `CRYP_DataType_32b((uint16_t)0x0000)

- #define: `CRYP_DataType_16b((uint16_t)0x0040)

- #define: `CRYP_DataType_8b((uint16_t)0x0080)

- #define: `CRYP_DataType_1b((uint16_t)0x00C0)
**CRYP_DMA_transfer_requests**

- #define: `CRYP_DMAReq_DataIN((uint8_t)0x01)`

- #define: `CRYP_DMAReq_DataOUT((uint8_t)0x02)`

**CRYP_Encryption_Decryption_modes_definition**

- #define: `MODE_ENCRYPT((uint8_t)0x01)`

- #define: `MODE_DECRYPT((uint8_t)0x00)`

**CRYP_flags_definition**

- #define: `CRYP_FLAG_BUSY((uint8_t)0x10)`
  The CRYP core is currently processing a block of data or a key preparation (for AES decryption).

- #define: `CRYP_FLAG_IFEM((uint8_t)0x01)`
  Input Fifo Empty

- #define: `CRYP_FLAG_IFNF((uint8_t)0x02)`
  Input Fifo is Not Full

- #define: `CRYP_FLAG_INRIS((uint8_t)0x22)`
  Raw interrupt pending

- #define: `CRYP_FLAG_OFNE((uint8_t)0x04)`
  Input Fifo service raw interrupt status

- #define: `CRYP_FLAG_OFFU((uint8_t)0x08)`
  Output Fifo is Full

- #define: `CRYP_FLAG_OUTRIS((uint8_t)0x21)`
CRYP_interruptions_definition
- #define: CRYP_IT_INI((uint8_t)0x01)
  IN Fifo Interrupt
- #define: CRYP_IT_OUTI((uint8_t)0x02)
  OUT Fifo Interrupt

CRYP_Key_Size_for_AES_only
- #define: CRYP_KeySize_128b((uint16_t)0x0000)
- #define: CRYP_KeySize_192b((uint16_t)0x0100)
- #define: CRYP_KeySize_256b((uint16_t)0x0200)

6.4 CRYP Programming Example
The example below explains how to use the Cryptographic processor to encrypt data using AES 128 in all modes: ECB, CBC and CTR. For more examples about CRYP configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\CRYP\/* Includes --------------------------------------------------------*/
#include "stm32f2xx.h"

/* Private define --------------------------------------------------*/
#define AES_TEXT_SIZE 64

/* Private variables ---------------------------------------------*/
uint8_t AES128key[16] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
uint8_t IV_1[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};
uint8_t Plaintext[AES_TEXT_SIZE] = {0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96, 0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a, 0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
uint8_t Encryptedtext[AES_TEXT_SIZE];

/**
 * @brief  Main program
 * @param  None
 * @retval None
 */

int main(void)
{
    /* Enable CRYP clock */
    RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_CRYP, ENABLE);

    /* Encrypt the plaintext message in ECB mode */
    CRYP_AES_ECB(MODE_ENCRYPT, AES128key, 128, Plaintext,
                 AES_TEXT_SIZE, Encryptedtext);

    /* Encrypt the plaintext message in CBC mode */
    CRYP_AES_CBC(MODE_ENCRYPT, IV_1, AES128key, 128, Plaintext,
                 AES_TEXT_SIZE, Encryptedtext);

    /* Encrypt the plaintext message in CTR mode */
    CRYP_AES_CTR(MODE_ENCRYPT, IV_1, AES128key, 128, Plaintext,
                 AES_TEXT_SIZE, Encryptedtext);

    while (1)
    {
    }
}

0x9e,0xb7,0x6f,0xac,0x45,0xaf,0x8e,0x51,
0x30,0xc8,0x1c,0x46,0xa3,0x5c,0xe4,0x11,
0xe5,0xfb,0xc1,0x19,0x1a,0xa0,0x52,0xef,
0xf6,0xf9f,0x24,0x45,0xdf,0xf4,0xf9b,0x17,
0xad,0x2b,0x41,0x7b,0xe6,0x6c,0x37,0x10;
7  Digital-to-analog converter (DAC)

7.1  DAC Firmware driver registers structures

7.1.1  DAC_TypeDef

DAC_TypeDef is defined in the stm32f2xx.h file and contains the DAC registers definition.

Data Fields

- __IO uint32_t CR
- __IO uint32_t SWTRIGR
- __IO uint32_t DHR12R1
- __IO uint32_t DHR12L1
- __IO uint32_t DHR8R1
- __IO uint32_t DHR12R2
- __IO uint32_t DHR12L2
- __IO uint32_t DHR8R2
- __IO uint32_t DHR12RD
- __IO uint32_t DHR12LD
- __IO uint32_t DHR8RD
- __IO uint32_t DOR1
- __IO uint32_t DOR2
- __IO uint32_t SR

Field Documentation

- __IO uint32_t DAC_TypeDef::CR
  - DAC control register, Address offset: 0x00
- __IO uint32_t DAC_TypeDef::SWTRIGR
  - DAC software trigger register, Address offset: 0x04
- __IO uint32_t DAC_TypeDef::DHR12R1
  - DAC channel1 12-bit right-aligned data holding register, Address offset: 0x08
- __IO uint32_t DAC_TypeDef::DHR12L1
  - DAC channel1 12-bit left aligned data holding register, Address offset: 0x0C
- __IO uint32_t DAC_TypeDef::DHR8R1
  - DAC channel1 8-bit right aligned data holding register, Address offset: 0x10
- __IO uint32_t DAC_TypeDef::DHR12R2
  - DAC channel2 12-bit right aligned data holding register, Address offset: 0x14
- __IO uint32_t DAC_TypeDef::DHR12L2
  - DAC channel2 12-bit left aligned data holding register, Address offset: 0x18
- __IO uint32_t DAC_TypeDef::DHR8R2
  - DAC channel2 8-bit right aligned data holding register, Address offset: 0x1C
- __IO uint32_t DAC_TypeDef::DHR12RD
  - Dual DAC 12-bit right-aligned data holding register, Address offset: 0x20
- __IO uint32_t DAC_TypeDef::DHR12LD
  - Dual DAC 12-bit left aligned data holding register, Address offset: 0x24
- __IO uint32_t DAC_TypeDef::DHR8RD
  - Dual DAC 8-bit right aligned data holding register, Address offset: 0x28
7.1.2 DAC_InitTypeDef

`DAC_InitTypeDef` is defined in the `stm32f2xx_dac.h` file and contains the DAC initialization parameters.

**Data Fields**

- `__IO uint32_t DAC_TypeDef::DOR1`  
  - DAC channel1 data output register, Address offset: 0x2C
- `__IO uint32_t DAC_TypeDef::DOR2`  
  - DAC channel2 data output register, Address offset: 0x30
- `__IO uint32_t DAC_TypeDef::SR`  
  - DAC status register, Address offset: 0x34

### Field Documentation

- `uint32_t DAC_InitTypeDef::DAC_Trigger`  
  - Specifies the external trigger for the selected DAC channel. This parameter can be a value of `DAC_trigger_selection`
- `uint32_t DAC_InitTypeDef::DAC_WaveGeneration`  
  - Specifies whether DAC channel noise waves or triangle waves are generated, or whether no wave is generated. This parameter can be a value of `DAC_wave_generation`
- `uint32_t DAC_InitTypeDef::DAC_LFSRUnmask_TriangleAmplitude`  
  - Specifies the LFSR mask for noise wave generation or the maximum amplitude triangle generation for the DAC channel. This parameter can be a value of `DAC_lfsrunmask_triangleamplitude`
- `uint32_t DAC_InitTypeDef::DAC_OutputBuffer`  
  - Specifies whether the DAC channel output buffer is enabled or disabled. This parameter can be a value of `DAC_output_buffer`

### DAC Firmware driver API description

The following section lists the various functions of the DAC library.

#### 7.2.1 DAC peripheral features

**DAC Channels**

The device integrates two 12-bit Digital Analog Converters that can be used independently or simultaneously (dual mode):

- DAC channel1 with DAC_OUT1 (PA4) as output
Digital-to-analog converter (DAC)

- DAC channel2 with DAC_OUT2 (PA5) as output

**DAC Triggers**

Digital to Analog conversion can be non-triggered using DAC_Trigger_None and DAC_OUT1/DAC_OUT2 is available once writing to DHRx register using DAC_SetChannel1Data() / DAC_SetChannel2Data() functions. Digital to Analog conversion can be triggered by:

- External event: EXTI Line 9 (any GPIOx_Pin9) using DAC_Trigger_Ext_IT9. The used pin (GPIOx_Pin9) must be configured in input mode.
- Timers TRGO: TIM2, TIM4, TIM5, TIM6, TIM7 and TIM8 (DAC_Trigger_T2_TRGO, DAC_Trigger_T4_TRGO...) The timer TRGO event should be selected using TIM_SelectOutputTrigger()
- Software using DAC_Trigger_Software

**DAC Buffer mode feature**

Each DAC channel integrates an output buffer that can be used to reduce the output impedance, and to drive external loads directly without having to add an external operational amplifier. To enable, the output buffer use DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Enable.

Refer to the device datasheet for more details about output impedance value with and without output buffer.

**DAC wave generation feature**

Both DAC channels can be used to generate:

- Noise wave using DAC_WaveGeneration_Noise
- Triangle wave using DAC_WaveGeneration_Triangle

Wave generation can be disabled using DAC_WaveGeneration_None.

**DAC data format**

The DAC data format can be:

- 8-bit right alignment using DAC_Align_8b_R
- 12-bit left alignment using DAC_Align_12b_L
- 12-bit right alignment using DAC_Align_12b_R

**DAC data value to voltage correspondence**

The analog output voltage on each DAC channel pin is determined by the following equation:

\[
\text{DAC\_OUTx} = \frac{\text{VREF} + \text{DOR}}{4095} \times 4095 \text{ with DOR is the Data Output Register VEF+ is the input voltage reference (refer to the device datasheet) e.g.}
\]

To set DAC_OUT1 to 0.7V, use DAC_SetChannel1Data(DAC_Align_12b_R, 868), assuming that VREF+ = 3.3V, DAC_OUT1 = (3.3 868) / 4095 = 0.7V

**DMA request**

A DMA1 request can be generated when an external trigger (but not a software trigger) occurs if DMA1 requests are enabled using DAC_DMACmd()
7.2.2 How to use this driver

1. Enable DAC APB clock to get write access to DAC registers using
   RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE)
2. Configure DAC_OUTx (DAC_OUT1: PA4, DAC_OUT2: PA5) in analog mode.
3. Configure the DAC channel using DAC_Init() function
4. Enable the DAC channel using DAC_Cmd() function

7.2.3 DAC channels configuration: trigger, output buffer, data format

- DAC_DeInit()
- DAC_Init()
- DAC_StructInit()
- DAC_Cmd()
- DAC_SoftwareTriggerCmd()
- DAC_DualSoftwareTriggerCmd()
- DAC_WaveGenerationCmd()
- DAC_SetChannel1Data()
- DAC_SetChannel2Data()
- DAC_SetDualChannelData()
- DAC_GetDataOutputValue()

7.2.4 DMA management

- DAC_DMACmd()

7.2.5 Interrupt and flag management

- DAC_ITConfig()
- DAC_GetFlagStatus()
- DAC_ClearFlag()
- DAC_GetITStatus()
- DAC_ClearITPendingBit()

7.2.6 DAC channels configuration

7.2.6.1 DAC_DeInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void DAC_DeInit ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Deinitializes the DAC peripheral registers to their default reset values.</td>
</tr>
</tbody>
</table>
7.2.6.2 DAC_Init

Function Name: void DAC_Init (uint32_t DAC_Channel, DAC_InitTypeDef * DAC_InitStruct)

Function Description: Initializes the DAC peripheral according to the specified parameters in the DAC_InitStruct.

Parameters:
- DAC_Channel: the selected DAC channel. This parameter can be one of the following values:
  - DAC_Channel_1: DAC Channel1 selected
  - DAC_Channel_2: DAC Channel2 selected
- DAC_InitStruct: pointer to a DAC_InitTypeDef structure that contains the configuration information for the specified DAC channel.

Return values:
- None.

Notes:
- None.

7.2.6.3 DAC_StructInit

Function Name: void DAC_StructInit (DAC_InitTypeDef * DAC_InitStruct)

Function Description: Fills each DAC_InitStruct member with its default value.

Parameters:
- DAC_InitStruct: pointer to a DAC_InitTypeDef structure which will be initialized.

Return values:
- None.

Notes:
- None.
7.2.6.4 DAC_Cmd

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void DAC_Cmd ( uint32_t DAC_Channel, FunctionalState NewState)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Enables or disables the specified DAC channel.</td>
</tr>
</tbody>
</table>
| Parameters | • DAC_Channel : The selected DAC channel. This parameter can be one of the following values:  
  - DAC_Channel_1 : DAC Channel1 selected  
  - DAC_Channel_2 : DAC Channel2 selected  
  • NewState : new state of the DAC channel. This parameter can be: ENABLE or DISABLE. |
| Return values | • None. |
| Notes | • When the DAC channel is enabled the trigger source can no more be modified. |

7.2.6.5 DAC_SoftwareTriggerCmd

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void DAC_SoftwareTriggerCmd ( uint32_t DAC_Channel, FunctionalState NewState)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Enables or disables the selected DAC channel software trigger.</td>
</tr>
</tbody>
</table>
| Parameters | • DAC_Channel : The selected DAC channel. This parameter can be one of the following values:  
  - DAC_Channel_1 : DAC Channel1 selected  
  - DAC_Channel_2 : DAC Channel2 selected  
  • NewState : new state of the selected DAC channel software trigger. This parameter can be: ENABLE or DISABLE. |
| Return values | • None. |
| Notes | • None. |

7.2.6.6 DAC_DualSoftwareTriggerCmd

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void DAC_DualSoftwareTriggerCmd ( FunctionalState NewState)</th>
</tr>
</thead>
</table>

DocID 18540 Rev 1

155/634
### 7.2.6.7 DAC_WaveGenerationCmd

**Function Name**
void DAC_WaveGenerationCmd ( uint32_t DAC_Channel, uint32_t DAC_Wave, FunctionalState NewState)

**Function Description**
Enables or disables the selected DAC channel wave generation.

**Parameters**
- **DAC_Channel**: The selected DAC channel. This parameter can be one of the following values:
  - `DAC_Channel_1`: DAC Channel1 selected
  - `DAC_Channel_2`: DAC Channel2 selected
- **DAC_Wave**: specifies the wave type to enable or disable. This parameter can be one of the following values:
  - `DAC_Wave_Noise`: noise wave generation
  - `DAC_Wave_Triangle`: triangle wave generation
- **NewState**: new state of the selected DAC channel wave generation. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.

**Notes**
- None.

### 7.2.6.8 DAC_SetChannel1Data

**Function Name**
void DAC_SetChannel1Data ( uint32_t DAC_Align, uint16_t Data)

**Function Description**
Set the specified data holding register value for DAC channel1.

**Parameters**
- **DAC_Align**: Specifies the data alignment for DAC channel1. This parameter can be one of the following values:
  - `DAC_Align_8b_R`: 8bit right data alignment selected
  - `DAC_Align_12b_L`: 12bit left data alignment selected
  - `DAC_Align_12b_R`: 12bit right data alignment selected
- **Data**: Data to be loaded in the selected data holding

**Return values**
- None.

**Notes**
- None.
7.2.6.9 DAC_SetChannel2Data

Function Name  void DAC_SetChannel2Data ( uint32_t DAC_Align, uint16_t Data)
Function Description  Set the specified data holding register value for DAC channel2.
Parameters  
  - DAC_Align : Specifies the data alignment for DAC channel2. This parameter can be one of the following values:
    - DAC_Align_8b_R : 8bit right data alignment selected
    - DAC_Align_12b_L : 12bit left data alignment selected
    - DAC_Align_12b_R : 12bit right data alignment selected
  - Data : Data to be loaded in the selected data holding register.
Return values  None.
Notes  None.

7.2.6.10 DAC_SetDualChannelData

Function Name  void DAC_SetDualChannelData ( uint32_t DAC_Align, uint16_t Data2, uint16_t Data1)
Function Description  Set the specified data holding register value for dual channel DAC.
Parameters  
  - DAC_Align : Specifies the data alignment for dual channel DAC. This parameter can be one of the following values:
    - DAC_Align_8b_R : 8bit right data alignment selected
    - DAC_Align_12b_L : 12bit left data alignment selected
    - DAC_Align_12b_R : 12bit right data alignment selected
  - Data2 : Data for DAC Channel2 to be loaded in the selected data holding register.
  - Data1 : Data for DAC Channel1 to be loaded in the selected data holding register.
Return values  None.
Notes
- In dual mode, a unique register access is required to write in both DAC channels at the same time.

7.2.6.11 DAC_GetDataOutputValue

Function Name: uint16_t DAC_GetDataOutputValue ( uint32_t DAC_Channel)
Function Description: Returns the last data output value of the selected DAC channel.
Parameters:
- **DAC_Channel**: The selected DAC channel. This parameter can be one of the following values:
  - **DAC_Channel_1**: DAC Channel1 selected
  - **DAC_Channel_2**: DAC Channel2 selected
Return values:
- The selected DAC channel data output value.
Notes:
- None.

7.2.7 DMA management function

7.2.7.1 DAC_DMACmd

Function Name: void DAC_DMACmd ( uint32_t DAC_Channel, FunctionalState NewState)
Function Description: Enables or disables the specified DAC channel DMA request.
Parameters:
- **DAC_Channel**: The selected DAC channel. This parameter can be one of the following values:
  - **DAC_Channel_1**: DAC Channel1 selected
  - **DAC_Channel_2**: DAC Channel2 selected
- **NewState**: new state of the selected DAC channel DMA request. This parameter can be: ENABLE or DISABLE.
Return values:
- None.
Notes:
- When enabled DMA1 is generated when an external trigger (EXTI Line9, TIM2, TIM4, TIM5, TIM6, TIM7 or TIM8 but not a software trigger) occurs.
- The DAC channel1 is mapped on DMA1 Stream 5 channel7 which must be already configured.
- The DAC channel2 is mapped on DMA1 Stream 6 channel7 which must be already configured.
7.2.8 Interrupt and flag management functions

7.2.8.1 DAC_ITConfig

Function Name: void DAC_ITConfig ( uint32_t DAC_Channel, uint32_t DAC_IT, FunctionalState NewState)

Function Description: Enables or disables the specified DAC interrupts.

Parameters:
- DAC_Channel: The selected DAC channel. This parameter can be one of the following values:
  - DAC_Channel_1: DAC Channel1 selected
  - DAC_Channel_2: DAC Channel2 selected
- DAC_IT: specifies the DAC interrupt sources to be enabled or disabled. This parameter can be the following values:
  - DAC_IT_DMAUDR: DMA underrun interrupt mask
- NewState: new state of the specified DAC interrupts. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes:
- The DMA underrun occurs when a second external trigger arrives before the acknowledgement for the first external trigger is received (first request).

7.2.8.2 DAC_GetFlagStatus

Function Name: FlagStatus DAC_GetFlagStatus ( uint32_t DAC_Channel, uint32_t DAC_FLAG)

Function Description: Checks whether the specified DAC flag is set or not.

Parameters:
- DAC_Channel: The selected DAC channel. This parameter can be one of the following values:
  - DAC_Channel_1: DAC Channel1 selected
  - DAC_Channel_2: DAC Channel2 selected
- DAC_FLAG: specifies the flag to check. This parameter can be only of the following value:
  - DAC_FLAG_DMAUDR: DMA underrun flag

Return values: The new state of DAC_FLAG (SET or RESET).

Notes:
- The DMA underrun occurs when a second external trigger arrives before the acknowledgement for the first external trigger.
trigger is received (first request).

7.2.8.3 DAC_ClearFlag

Function Name: void DAC_ClearFlag ( uint32_t DAC_Channel, uint32_t DAC_FLAG)

Function Description: Clears the DAC channel's pending flags.

Parameters:
- **DAC_Channel**: The selected DAC channel. This parameter can be one of the following values:
  - **DAC_Channel_1**: DAC Channel1 selected
  - **DAC_Channel_2**: DAC Channel2 selected
- **DAC_FLAG**: Specifies the flag to clear. This parameter can be of the following value:
  - **DAC_FLAG_DMAUDR**: DMA underrun flag

Return values:
- None.

Notes:
- The DMA underrun occurs when a second external trigger arrives before the acknowledgement for the first external trigger is received (first request).

7.2.8.4 DAC_GetITStatus

Function Name: ITStatus DAC_GetITStatus ( uint32_t DAC_Channel, uint32_t DAC_IT)

Function Description: Checks whether the specified DAC interrupt has occurred or not.

Parameters:
- **DAC_Channel**: The selected DAC channel. This parameter can be one of the following values:
  - **DAC_Channel_1**: DAC Channel1 selected
  - **DAC_Channel_2**: DAC Channel2 selected
- **DAC_IT**: Specifies the DAC interrupt source to check. This parameter can be the following values:
  - **DAC_IT_DMAUDR**: DMA underrun interrupt mask

Return values:
- The new state of **DAC_IT** (SET or RESET).

Notes:
- The DMA underrun occurs when a second external trigger arrives before the acknowledgement for the first external trigger is received (first request).
7.2.8.5 DAC_ClearITPendingBit

Function Name  void DAC_ClearITPendingBit ( uint32_t DAC_Channel, 
uint32_t DAC_IT)

Function Description  Clears the DAC channel's interrupt pending bits.

Parameters

- **DAC_Channel** : The selected DAC channel. This parameter can be one of the following values:
  - **DAC_Channel_1** : DAC Channel1 selected
  - **DAC_Channel_2** : DAC Channel2 selected
- **DAC_IT** : specifies the DAC interrupt pending bit to clear. This parameter can be the following values:
  - **DAC_IT_DMAUDR** : DMA underrun interrupt mask

Return values  None.

Notes

- The DMA underrun occurs when a second external trigger arrives before the acknowledgement for the first external trigger is received (first request).

7.3 DAC Firmware driver defines

7.3.1 DAC Firmware driver defines

DAC

**DAC_Channel_selection**

- #define: **DAC_Channel_1**((uint32_t)0x00000000)

- #define: **DAC_Channel_2**((uint32_t)0x00000010)

**DAC_data_alignement**

- #define: **DAC_Align_12b_R**((uint32_t)0x00000000)

- #define: **DAC_Align_12b_L**((uint32_t)0x00000004)
Digital-to-analog converter (DAC)

- `#define: DAC_Align_8b_R((uint32_t)0x00000008)`

**DAC_flags_definition**
- `#define: DAC_FLAG_DMAUDR((uint32_t)0x00002000)`

**DAC_interrupts_definition**
- `#define: DAC_IT_DMAUDR((uint32_t)0x00002000)`

**DAC_lfsrunmask_triangleamplitude**
- `#define: DAC_LFSRUnmask_Bit0((uint32_t)0x00000000)`
  Unmask DAC channel LFSR bit0 for noise wave generation
- `#define: DAC_LFSRUnmask_Bits1_0((uint32_t)0x00000100)`
  Unmask DAC channel LFSR bit[1:0] for noise wave generation
- `#define: DAC_LFSRUnmask_Bits2_0((uint32_t)0x00000200)`
  Unmask DAC channel LFSR bit[2:0] for noise wave generation
- `#define: DAC_LFSRUnmask_Bits3_0((uint32_t)0x00000300)`
  Unmask DAC channel LFSR bit[3:0] for noise wave generation
- `#define: DAC_LFSRUnmask_Bits4_0((uint32_t)0x00000400)`
  Unmask DAC channel LFSR bit[4:0] for noise wave generation
- `#define: DAC_LFSRUnmask_Bits5_0((uint32_t)0x00000500)`
  Unmask DAC channel LFSR bit[5:0] for noise wave generation
- `#define: DAC_LFSRUnmask_Bits6_0((uint32_t)0x00000600)`
  Unmask DAC channel LFSR bit[6:0] for noise wave generation
- `#define: DAC_LFSRUnmask_Bits7_0((uint32_t)0x00000700)`
  Unmask DAC channel LFSR bit[7:0] for noise wave generation
• #define: `DAC_LFSRUnmask_Bits8_0((uint32_t)0x00000000)`
  Unmask DAC channel LFSR bit[8:0] for noise wave generation

• #define: `DAC_LFSRUnmask_Bits9_0((uint32_t)0x00000000)`
  Unmask DAC channel LFSR bit[9:0] for noise wave generation

• #define: `DAC_LFSRUnmask_Bits10_0((uint32_t)0x00000000)`
  Unmask DAC channel LFSR bit[10:0] for noise wave generation

• #define: `DAC_LFSRUnmask_Bits11_0((uint32_t)0x00000000)`
  Unmask DAC channel LFSR bit[11:0] for noise wave generation

• #define: `DAC_TriangleAmplitude_1((uint32_t)0x00000000)`
  Select max triangle amplitude of 1

• #define: `DAC_TriangleAmplitude_3((uint32_t)0x00000000)`
  Select max triangle amplitude of 3

• #define: `DAC_TriangleAmplitude_7((uint32_t)0x00000000)`
  Select max triangle amplitude of 7

• #define: `DAC_TriangleAmplitude_15((uint32_t)0x00000000)`
  Select max triangle amplitude of 15

• #define: `DAC_TriangleAmplitude_31((uint32_t)0x00000000)`
  Select max triangle amplitude of 31

• #define: `DAC_TriangleAmplitude_63((uint32_t)0x00000000)`
  Select max triangle amplitude of 63

• #define: `DAC_TriangleAmplitude_127((uint32_t)0x00000000)`
  Select max triangle amplitude of 127

• #define: `DAC_TriangleAmplitude_255((uint32_t)0x00000000)`
  Select max triangle amplitude of 255
Digital-to-analog converter (DAC)

- **#define: DAC_TriangleAmplitude_511((uint32_t)0x00000800)**
  Select max triangle amplitude of 511

- **#define: DAC_TriangleAmplitude_1023((uint32_t)0x00000900)**
  Select max triangle amplitude of 1023

- **#define: DAC_TriangleAmplitude_2047((uint32_t)0x00000A00)**
  Select max triangle amplitude of 2047

- **#define: DAC_TriangleAmplitude_4095((uint32_t)0x00000B00)**
  Select max triangle amplitude of 4095

**DAC_output_buffer**
- **#define: DAC_OutputBuffer_Enable((uint32_t)0x00000000)**
- **#define: DAC_OutputBuffer_Disable((uint32_t)0x00000002)**

**DAC_trigger_selection**
- **#define: DAC_Trigger_None((uint32_t)0x00000000)**
  Conversion is automatic once the DAC1_DHRxxxx register has been loaded, and not by external trigger

  - **#define: DAC_Trigger_T2_TRGO((uint32_t)0x00000024)**
    TIM2 TRGO selected as external conversion trigger for DAC channel

  - **#define: DAC_Trigger_T4_TRGO((uint32_t)0x0000002C)**
    TIM4 TRGO selected as external conversion trigger for DAC channel

  - **#define: DAC_Trigger_T5_TRGO((uint32_t)0x0000001C)**
    TIM5 TRGO selected as external conversion trigger for DAC channel

  - **#define: DAC_Trigger_T6_TRGO((uint32_t)0x00000004)**
    TIM6 TRGO selected as external conversion trigger for DAC channel
- #define: **DAC_Trigger_T7_TRGO**((uint32_t)0x00000014)
  TIM7 TRGO selected as external conversion trigger for DAC channel

- #define: **DAC_Trigger_T8_TRGO**((uint32_t)0x0000000C)
  TIM8 TRGO selected as external conversion trigger for DAC channel

- #define: **DAC_Trigger_Ext_IT9**((uint32_t)0x00000034)
  EXTI Line9 event selected as external conversion trigger for DAC channel

- #define: **DAC_Trigger_Software**((uint32_t)0x0000003C)
  Conversion started by software trigger for DAC channel

**DAC_wave_generation**

- #define: **DAC_WaveGeneration_None**((uint32_t)0x00000000)

- #define: **DAC_WaveGeneration_Noise**((uint32_t)0x00000040)

- #define: **DAC_WaveGeneration_Triangle**((uint32_t)0x00000080)

- #define: **DAC_Wave_Noise**((uint32_t)0x00000040)

- #define: **DAC_Wave_Triangle**((uint32_t)0x00000080)

### 7.4 DAC Programming Example

The example below explains how to generate Triangle wave using the DAC (this example assumes that DAC channel2 pin and TIM6 are already configured). For more examples about DAC configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\DAC\.

```c
DAC_InitTypeDef  DAC_InitStruct;
/* Enable DAC clock */
```
RCC_APB1PeriphClockCmd(RCC_APB1Periph_DAC, ENABLE);

/* DAC channel2 Configuration ****************************************/
DAC_InitStruct.DAC_Trigger = DAC_Trigger_T6_TRGO;
// TIM6 TRGO signal is used to trigger the DAC
DAC_InitStruct.DAC_WaveGeneration = DAC_WaveGeneration_Triangle;
DAC_InitStruct.DAC_LFSRUnmask_TriangleAmplitude =
DAC_TriangleAmplitude_1023;
DAC_InitStruct.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
DAC_Init(DAC_Channel_2, &DAC_InitStruct);

/* Enable DAC Channel2 */
DAC_Cmd(DAC_Channel_2, ENABLE);

/* Set DAC channel2 DHR12RD register */
DAC_SetChannel2Data(DAC_Align_12b_R, 0x100);
8 Debug support (DBGMCU)

8.1 DBGMCU Firmware driver registers structures

8.1.1 DBGMCU_TypeDef

(DBGMCU_TypeDef) is defined in the stm32f2xx.h file and contains the DBGMCU registers definition.

Data Fields

- __IO uint32_t IDCODE
- __IO uint32_t CR
- __IO uint32_t APB1FZ
- __IO uint32_t APB2FZ

Field Documentation

- __IO uint32_t DBGMCU_TypeDef::IDCODE
  - MCU device ID code, Address offset: 0x00
- __IO uint32_t DBGMCU_TypeDef::CR
  - Debug MCU configuration register, Address offset: 0x04
- __IO uint32_t DBGMCU_TypeDef::APB1FZ
  - Debug MCU APB1 freeze register, Address offset: 0x08
- __IO uint32_t DBGMCU_TypeDef::APB2FZ
  - Debug MCU APB2 freeze register, Address offset: 0x0C

8.2 DBGMCU Firmware driver API description

The following section lists the various functions of the DBGMCU library.

Functions

- DBGMCU_GetREVID()
- DBGMCU_GetDEVID()
- DBGMCU_Config()
- DBGMCU_APB1PeriphConfig()
- DBGMCU_APB2PeriphConfig()

8.2.1 Functions

8.2.1.1 DBGMCU_GetREVID

Function Name

uint32_t DBGMCU_GetREVID ( void )
Function Description: Returns the device revision identifier.
Parameters: None.
Return values: Device revision identifier
Notes: None.

### 8.2.1.2 DBGMCU_GetDEVID

Function Name: uint32_t DBGMCU_GetDEVID ( void )
Function Description: Returns the device identifier.
Parameters: None.
Return values: Device identifier
Notes: None.

### 8.2.1.3 DBGMCU_Config

Function Name: void DBGMCU_Config ( uint32_t DBGMCU_Periph,
FunctionalState NewState)
Function Description: Configures low power mode behavior when the MCU is in Debug mode.
Parameters:
- **DBGMCU_Periph**: specifies the low power mode. This parameter can be any combination of the following values:
  - **DBGMCU_SLEEP**: Keep debugger connection during SLEEP mode
  - **DBGMCU_STOP**: Keep debugger connection during STOP mode
  - **DBGMCU_STANDBY**: Keep debugger connection during STANDBY mode
- **NewState**: new state of the specified low power mode in Debug mode. This parameter can be: ENABLE or DISABLE.

Return values: None.
Notes: None.
8.2.1.4 DBGMCU_APB1PeriphConfig

Function Name
void DBGMCU_APB1PeriphConfig ( uint32_t DBGMCU_Periph, FunctionalState NewState)

Function Description
Configures APB1 peripheral behavior when the MCU is in Debug mode.

Parameters
- **DBGMCU_Periph**: specifies the APB1 peripheral. This parameter can be any combination of the following values:
  - **DBGMCU_TIM2_STOP**: TIM2 counter stopped when Core is halted
  - **DBGMCU_TIM3_STOP**: TIM3 counter stopped when Core is halted
  - **DBGMCU_TIM4_STOP**: TIM4 counter stopped when Core is halted
  - **DBGMCU_TIM5_STOP**: TIM5 counter stopped when Core is halted
  - **DBGMCU_TIM6_STOP**: TIM6 counter stopped when Core is halted
  - **DBGMCU_TIM7_STOP**: TIM7 counter stopped when Core is halted
  - **DBGMCU_TIM9_STOP**: TIM9 counter stopped when Core is halted
  - **DBGMCU_TIM11_STOP**: TIM11 counter stopped when Core is halted
  - **DBGMCU_TIM12_STOP**: TIM12 counter stopped when Core is halted
  - **DBGMCU_TIM13_STOP**: TIM13 counter stopped when Core is halted
  - **DBGMCU_TIM14_STOP**: TIM14 counter stopped when Core is halted
  - **DBGMCU_RTC_STOP**: RTC Wakeup counter stopped when Core is halted.
  - **DBGMCU_WWDG_STOP**: Debug WWDG stopped when Core is halted
  - **DBGMCU_IWDG_STOP**: Debug IWDG stopped when Core is halted
  - **DBGMCU_I2C1_SMBUS_TIMEOUT**: I2C1 SMBUS timeout mode stopped when Core is halted
  - **DBGMCU_I2C2_SMBUS_TIMEOUT**: I2C2 SMBUS timeout mode stopped when Core is halted
  - **DBGMCU_I2C3_SMBUS_TIMEOUT**: I2C3 SMBUS timeout mode stopped when Core is halted
  - **DBGMCU_CAN2_STOP**: Debug CAN2 stopped when Core is halted
  - **DBGMCU_CAN1_STOP**: Debug CAN1 stopped when Core is halted

This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- None.
8.2.1.5 DBGMCU_APB2PeriphConfig

Function Name  void DBGMCU_APB2PeriphConfig ( uint32_t DBGMCU_Periph, FunctionalState NewState)

Function Description Configures APB2 peripheral behavior when the MCU is in Debug mode.

Parameters
- **DBGMCU_Periph**: specifies the APB2 peripheral. This parameter can be any combination of the following values:
  - **DBGMCU_TIM1_STOP**: TIM1 counter stopped when Core is halted
  - **DBGMCU_TIM8_STOP**: TIM8 counter stopped when Core is halted
  - **DBGMCU_TIM9_STOP**: TIM9 counter stopped when Core is halted
  - **DBGMCU_TIM10_STOP**: TIM10 counter stopped when Core is halted
  - **DBGMCU_TIM11_STOP**: TIM11 counter stopped when Core is halted

- **NewState**: new state of the specified peripheral in Debug mode. This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- None.

8.3 DBGMCU Firmware driver defines

DBGMCU

**DBGMCU_Exposed_Constants**

- `#define DBGMCU_SLEEP((uint32_t)0x00000001)

- `#define DBGMCU_STOP((uint32_t)0x00000002)

- `#define DBGMCU_STANDBY((uint32_t)0x00000004)

- `#define DBGMCU_TIM2_STOP((uint32_t)0x00000001)`
- #define: `DBGMCU_TIM3_STOP((uint32_t)0x00000002)`

- #define: `DBGMCU_TIM4_STOP((uint32_t)0x00000004)`

- #define: `DBGMCU_TIM5_STOP((uint32_t)0x00000008)`

- #define: `DBGMCU_TIM6_STOP((uint32_t)0x00000010)`

- #define: `DBGMCU_TIM7_STOP((uint32_t)0x00000020)`

- #define: `DBGMCU_TIM12_STOP((uint32_t)0x00000040)`

- #define: `DBGMCU_TIM13_STOP((uint32_t)0x00000080)`

- #define: `DBGMCU_TIM14_STOP((uint32_t)0x00000100)`

- #define: `DBGMCU_RTC_STOP((uint32_t)0x00000400)`

- #define: `DBGMCU_WWDG_STOP((uint32_t)0x00000800)`

- #define: `DBGMCU_IWDG_STOP((uint32_t)0x00001000)`

- #define: `DBGMCU_I2C1_SMBUS_TIMEOUT((uint32_t)0x00200000)`
- #define: `DBGMCU_I2C2_SMBUS_TIMEOUT((uint32_t)0x00400000)`

- #define: `DBGMCU_I2C3_SMBUS_TIMEOUT((uint32_t)0x00800000)`

- #define: `DBGMCU_CAN1_STOP((uint32_t)0x02000000)`

- #define: `DBGMCU_CAN2_STOP((uint32_t)0x04000000)`

- #define: `DBGMCU_TIM1_STOP((uint32_t)0x00000001)`

- #define: `DBGMCU_TIM8_STOP((uint32_t)0x00000002)`

- #define: `DBGMCU_TIM9_STOP((uint32_t)0x00010000)`

- #define: `DBGMCU_TIM10_STOP((uint32_t)0x00020000)`

- #define: `DBGMCU_TIM11_STOP((uint32_t)0x00040000)`


9 Digital camera interface (DCMI)

9.1 DCMI Firmware driver registers structures

9.1.1 DCMI_TypeDef

*DCMI_TypeDef* is defined in the stm32f2xx.h file and contains the DCMI registers definition.

Data Fields

- __IO uint32_t CR
- __IO uint32_t SR
- __IO uint32_t RISR
- __IO uint32_t IER
- __IO uint32_t MISR
- __IO uint32_t ICR
- __IO uint32_t ESCR
- __IO uint32_t ESUR
- __IO uint32_t CWSTRTR
- __IO uint32_t CWSIZER
- __IO uint32_t DR

Field Documentation

- __IO uint32_t DCMI_TypeDef::CR
  - DCMI control register 1, Address offset: 0x00
- __IO uint32_t DCMI_TypeDef::SR
  - DCMI status register, Address offset: 0x04
- __IO uint32_t DCMI_TypeDef::RISR
  - DCMI raw interrupt status register, Address offset: 0x08
- __IO uint32_t DCMI_TypeDef::IER
  - DCMI interrupt enable register, Address offset: 0x0C
- __IO uint32_t DCMI_TypeDef::MISR
  - DCMI masked interrupt status register, Address offset: 0x10
- __IO uint32_t DCMI_TypeDef::ICR
  - DCMI interrupt clear register, Address offset: 0x14
- __IO uint32_t DCMI_TypeDef::ESCR
  - DCMI embedded synchronization code register, Address offset: 0x18
- __IO uint32_t DCMI_TypeDef::ESUR
  - DCMI embedded synchronization unmask register, Address offset: 0x1C
- __IO uint32_t DCMI_TypeDef::CWSTRTR
  - DCMI crop window start, Address offset: 0x20
- __IO uint32_t DCMI_TypeDef::CWSIZER
  - DCMI crop window size, Address offset: 0x24
- __IO uint32_t DCMI_TypeDef::DR
  - DCMI data register, Address offset: 0x28
9.1.2 DCMI_InitTypeDef

`DCMI_InitTypeDef` is defined in the `stm32f2xx_dcmi.h` file and contains the DCMI common initialization parameters.

Data Fields

- `uint16_t DCMI_CaptureMode`
- `uint16_t DCMI_SynchroMode`
- `uint16_t DCMI_PCKPolarity`
- `uint16_t DCMI_VSPolarity`
- `uint16_t DCMI_HSPolarity`
- `uint16_t DCMI_CaptureRate`
- `uint16_t DCMI_ExtendedDataMode`

Field Documentation

- `uint16_t DCMI_InitTypeDef::DCMI_CaptureMode`
  - Specifies the Capture Mode: Continuous or Snapshot. This parameter can be a value of `DCMI_Capture_Mode`
- `uint16_t DCMI_InitTypeDef::DCMI_SynchroMode`
  - Specifies the Synchronization Mode: Hardware or Embedded. This parameter can be a value of `DCMI_Synchronization_Mode`
- `uint16_t DCMI_InitTypeDef::DCMI_PCKPolarity`
  - Specifies the Pixel clock polarity: Falling or Rising. This parameter can be a value of `DCMI_PIXCK_Polarity`
- `uint16_t DCMI_InitTypeDef::DCMI_VSPolarity`
  - Specifies the Vertical synchronization polarity: High or Low. This parameter can be a value of `DCMI_VSYNC_Polarity`
- `uint16_t DCMI_InitTypeDef::DCMI_HSPolarity`
  - Specifies the Horizontal synchronization polarity: High or Low. This parameter can be a value of `DCMI_HSYNC_Polarity`
- `uint16_t DCMI_InitTypeDef::DCMI_CaptureRate`
  - Specifies the frequency of frame capture: All, 1/2 or 1/4. This parameter can be a value of `DCMI_Capture_Rate`
- `uint16_t DCMI_InitTypeDef::DCMI_ExtendedDataMode`
  - Specifies the data width: 8-bit, 10-bit, 12-bit or 14-bit. This parameter can be a value of `DCMI_Extended_Data_Mode`

9.1.3 DCMI_CROPIntTypeDef

`DCMI_CROPIntTypeDef` is defined in the `stm32f2xx_dcmi.h` file and contains the DCMI's CROP mode initialization parameters.

Data Fields

- `uint16_t DCMI_VerticalStartLine`
- `uint16_t DCMI_HorizontalOffsetCount`
- `uint16_t DCMI_VerticalLineCount`
- `uint16_t DCMI_CaptureCount`
Field Documentation

- `uint16_t DCMI_CROPInitTypeDef::DCMI_VerticalStartLine`
  - Specifies the Vertical start line count from which the image capture will start. This parameter can be a value between 0x00 and 0x1FFF
- `uint16_t DCMI_CROPInitTypeDef::DCMI_HorizontalOffsetCount`
  - Specifies the number of pixel clocks to count before starting a capture. This parameter can be a value between 0x00 and 0x3FFF
- `uint16_t DCMI_CROPInitTypeDef::DCMI_VerticalLineCount`
  - Specifies the number of lines to be captured from the starting point. This parameter can be a value between 0x00 and 0x3FFF
- `uint16_t DCMI_CROPInitTypeDef::DCMI_CaptureCount`
  - Specifies the number of pixel clocks to be captured from the starting point on the same line. This parameter can be a value between 0x00 and 0x3FFF

9.1.4 DCMI_CodesInitTypeDef

`DCMI_CodesInitTypeDef` is defined in the `stm32f2xx_dcmi.h` file and contains the DCMI's embedded synchronization codes initialization parameters.

Data Fields

- `uint8_t DCMI_FrameStartCode`
- `uint8_t DCMI_LineStartCode`
- `uint8_t DCMI_LineEndCode`
- `uint8_t DCMI_FrameEndCode`

Field Documentation

- `uint8_t DCMI_CodesInitTypeDef::DCMI_FrameStartCode`
  - Specifies the code of the frame start delimiter.
- `uint8_t DCMI_CodesInitTypeDef::DCMI_LineStartCode`
  - Specifies the code of the line start delimiter.
- `uint8_t DCMI_CodesInitTypeDef::DCMI_LineEndCode`
  - Specifies the code of the line end delimiter.
- `uint8_t DCMI_CodesInitTypeDef::DCMI_FrameEndCode`
  - Specifies the code of the frame end delimiter.

9.2 DCMI Firmware driver API description

The following section lists the various functions of the DCMI library.

9.2.1 How to use this driver
The sequence below describes how to use this driver to capture image from a camera module connected to the DCMI Interface. This sequence does not take into account the configuration of the camera module, which should be made before to configure and enable the DCMI to capture images.

1. Enable the clock for the DCMI and associated GPIOs using the following functions:
   RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_DCMI, ENABLE);
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOx, ENABLE);
2. DCMI pins configuration
   a. Connect the involved DCMI pins to AF13 using the following function
      GPIO_PinAFConfig(GPIOx, GPIO_PinSourcex, GPIO_AF_DCMI);
   b. Configure these DCMI pins in alternate function mode by calling the function
      GPIO_Init();
3. Declare a DCMI_InitTypeDef structure, for example: DCMI_InitTypeDef
   DCMI_InitStructure; and fill the DCMI_InitStructure variable with the allowed values of
   the structure member.
4. Initialize the DCMI interface by calling the function DCMI_Init(&DCMI_InitStructure);
5. Configure the DMA2_Stream1 channel1 to transfer Data from DCMI DR register to the
   destination memory buffer.
6. Enable DCMI interface using the function DCMI_Cmd(ENABLE);
7. Start the image capture using the function DCMI_CaptureCmd(ENABLE);
8. At this stage the DCMI interface waits for the first start of frame, then a DMA request
   is generated continuously/once (depending on the mode used, Continuous/Snapshot)
   to transfer the received data into the destination memory.

If you need to capture only a rectangular window from the received image, you have to use the DCMI_CROPConfig() function to configure the coordinates and size of the window to be captured, then enable the Crop feature using
DCMI_CROPCmd(ENABLE); In this case, the Crop configuration should be made
before to enable and start the DCMI interface.

9.2.2 Initialization and configuration

- DCMI_DeInit()
- DCMI_Init()
- DCMI_StructInit()
- DCMI_CROPConfig()
- DCMI_CROPCmd()
- DCMI_SetEmbeddedSynchroCodes()
- DCMI_JPEGCmd()

9.2.3 Image capture

- DCMI_Cmd()
- DCMI_CaptureCmd()
- DCMI_ReadData()

9.2.4 Interrupt and flag management

- DCMI_ITConfig()
- DCMI_GetFlagStatus()
9.2.5 Initialization and configuration functions

9.2.5.1 DCMI_DeInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void DCMI_DeInit ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Deinitializes the DCMI registers to their default reset values.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• None.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>

9.2.5.2 DCMI_Init

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void DCMI_Init ( DCMI_InitTypeDef * DCMI_InitStruct)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Initializes the DCMI according to the specified parameters in the DCMI_InitStruct.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• DCMI_InitStruct : pointer to a DCMI_InitTypeDef structure that contains the configuration information for the DCMI.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>

9.2.5.3 DCMI_StructInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void DCMI_StructInit ( DCMI_InitTypeDef * DCMI_InitStruct)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Fills each DCMI_InitStruct member with its default value.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• DCMI_InitStruct : pointer to a DCMI_InitTypeDef structure which will be initialized.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
</tbody>
</table>
9.2.5.4 DCMI_CROPConfig

Function Name: void DCMI_CROPConfig (DCMI_CROPInitTypeDef *DCMI_CROPInitStruct)

Function Description: Initializes the DCMI peripheral CROP mode according to the specified parameters in the DCMI_CROPInitStruct.

Parameters:
- DCMI_CROPInitStruct: pointer to a DCMI_CROPInitTypeDef structure that contains the configuration information for the DCMI peripheral CROP mode.

Return values: None.

Notes:
- This function should be called before to enable and start the DCMI interface.

9.2.5.5 DCMI_CROPCmd

Function Name: void DCMI_CROPCmd (FunctionalState NewState)

Function Description: Enables or disables the DCMI Crop feature.

Parameters:
- NewState: new state of the DCMI Crop feature. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes:
- This function should be called before to enable and start the DCMI interface.

9.2.5.6 DCMI_SetEmbeddedSynchroCodes

Function Name: void DCMI_SetEmbeddedSynchroCodes (
Function Description
Sets the embedded synchronization codes.

Parameters
- **DCMI_CodesInitTypeDef**: pointer to a DCMI_CodesInitTypeDef structure that contains the embedded synchronization codes for the DCMI peripheral.

Return values
- None.

Notes
- None.

9.2.5.7 DCMI_JPEGCmd

Function Name
`void DCMI_JPEGCmd ( FunctionalState NewState)`

Function Description
Enables or disables the DCMI JPEG format.

Parameters
- **NewState**: new state of the DCMI JPEG format. This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- The Crop and Embedded Synchronization features cannot be used in this mode.

9.2.6 Image capture functions

9.2.6.1 DCMI_Cmd

Function Name
`void DCMI_Cmd ( FunctionalState NewState)`

Function Description
Enables or disables the DCMI interface.

Parameters
- **NewState**: new state of the DCMI interface. This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- None.
9.2.6.2  DCMI_CaptureCmd

Function Name  
void DCMI_CaptureCmd ( FunctionalState NewState)

Function Description  
Enables or disables the DCMI Capture.

Parameters  
- **NewState**: new state of the DCMI capture. This parameter can be: ENABLE or DISABLE.

Return values  
- None.

Notes  
- None.

9.2.6.3  DCMI_ReadData

Function Name  
uint32_t DCMI_ReadData ( void )

Function Description  
Reads the data stored in the DR register.

Parameters  
- None.

Return values  
- Data register value

Notes  
- None.

9.2.7  Interrupt and flag management functions

9.2.7.1  DCMI_ITConfig

Function Name  
void DCMI_ITConfig ( uint16_t DCMI_IT, FunctionalState NewState)

Function Description  
Enables or disables the DCMI interface interrupts.

Parameters  
- **DCMI_IT**: specifies the DCMI interrupt sources to be enabled or disabled. This parameter can be any combination of the following values:
  - **DCMI_IT_FRAME**: Frame capture complete interrupt mask
  - **DCMI_IT_OVF**: Overflow interrupt mask
  - **DCMI_IT_ERR**: Synchronization error interrupt mask
  - **DCMI_IT_VSYNC**: VSYNC interrupt mask
  - **DCMI_IT_LINE**: Line interrupt mask
9.2.7.2 DCMI_GetFlagStatus

Function Name: FlagStatus DCMI_GetFlagStatus ( uint16_t DCMI_FLAG)

Function Description: Checks whether the DCMI interface flag is set or not.

Parameters:
- **DCMI_FLAG**: specifies the flag to check. This parameter can be one of the following values:
  - `DCMI_FLAG_FRAMERI`: Frame capture complete Raw flag mask
  - `DCMI_FLAG_OVFRI`: Overflow Raw flag mask
  - `DCMI_FLAG_ERRRI`: Synchronization error Raw flag mask
  - `DCMI_FLAG_VSYNCRI`: VSYNC Raw flag mask
  - `DCMI_FLAG_LINERI`: Line Raw flag mask
  - `DCMI_FLAG_FRAMEMI`: Frame capture complete Masked flag mask
  - `DCMI_FLAG_OVFMI`: Overflow Masked flag mask
  - `DCMI_FLAG_ERRMI`: Synchronization error Masked flag mask
  - `DCMI_FLAG_VSYNCMI`: VSYNC Masked flag mask
  - `DCMI_FLAG_LINEMI`: Line Masked flag mask
  - `DCMI_FLAG_HSYNC`: HSYNC flag mask
  - `DCMI_FLAG_VSYNC`: VSYNC flag mask
  - `DCMI_FLAG_FNE`: Fifo not empty flag mask

Return values:
- The new state of DCMI_FLAG (SET or RESET).

Notes:
- None.

9.2.7.3 DCMI_ClearFlag

Function Name: void DCMI_ClearFlag ( uint16_t DCMI_FLAG)

Function Description: Clears the DCMI's pending flags.

Parameters:
- **DCMI_FLAG**: specifies the flag to clear. This parameter can
be any combination of the following values:
- `DCMI_FLAG_FRAMERI` : Frame capture complete Raw flag mask
- `DCMI_FLAG_OVFRI` : Overflow Raw flag mask
- `DCMI_FLAG_ERRRI` : Synchronization error Raw flag mask
- `DCMI_FLAG_VSYNCRI` : VSYNC Raw flag mask
- `DCMI_FLAG_LINERI` : Line Raw flag mask

**Return values**
- None.

**Notes**
- None.

### 9.2.7.4 DCMI_GetITStatus

**Function Name**
`ITStatus DCMI_GetITStatus ( uint16_t DCMI_IT)`

**Function Description**
Checks whether the DCMI interrupt has occurred or not.

**Parameters**
- `DCMI_IT` : specifies the DCMI interrupt source to check. This parameter can be one of the following values:
  - `DCMI_IT_FRAME` : Frame capture complete interrupt mask
  - `DCMI_IT_OVF` : Overflow interrupt mask
  - `DCMI_IT_ERR` : Synchronization error interrupt mask
  - `DCMI_IT_VSYNC` : VSYNC interrupt mask
  - `DCMI_IT_LINE` : Line interrupt mask

**Return values**
- The new state of `DCMI_IT` (SET or RESET).

**Notes**
- None.

### 9.2.7.5 DCMI_ClearITPendingBit

**Function Name**
`void DCMI_ClearITPendingBit ( uint16_t DCMI_IT)`

**Function Description**
Clears the DCMI's interrupt pending bits.

**Parameters**
- `DCMI_IT` : specifies the DCMI interrupt pending bit to clear. This parameter can be any combination of the following values:
  - `DCMI_IT_FRAME` : Frame capture complete interrupt mask
  - `DCMI_IT_OVF` : Overflow interrupt mask

**Notes**
- None.
- **DCMI_IT_ERR**: Synchronization error interrupt mask
- **DCMI_IT_VSYNC**: VSYNC interrupt mask
- **DCMI_IT_LINE**: Line interrupt mask

**Return values**
- None.

**Notes**
- None.

### 9.3 DCMI Firmware driver defines

#### 9.3.1 DCMI Firmware driver defines

**DCMI**

**DCML_Capture_Mode**

- #define: `DCMI_CaptureMode_Continuous((uint16_t)0x0000)`

  The received data are transferred continuously into the destination memory through the DMA.

- #define: `DCMI_CaptureMode_SnapShot((uint16_t)0x0002)`

  Once activated, the interface waits for the start of frame and then transfers a single frame through the DMA.

**DCML_Capture_Rate**

- #define: `DCMI_CaptureRate_All_Frame((uint16_t)0x0000)`

  All frames are captured.

- #define: `DCMI_CaptureRate_1of2_Frame((uint16_t)0x0100)`

  Every alternate frame captured.

- #define: `DCMI_CaptureRate_1of4_Frame((uint16_t)0x0200)`

  One frame in 4 frames captured.

**DCML_Extended_Data_Mode**

- #define: `DCMI_ExtendedDataMode_8b((uint16_t)0x0000)`

  Interface captures 8-bit data on every pixel clock.

- #define: `DCMI_ExtendedDataMode_10b((uint16_t)0x0400)`

  Interface captures 10-bit data on every pixel clock.
- #define: `DCMI_ExtendedDataMode_12b((uint16_t)0x0800)`
  Interface captures 12-bit data on every pixel clock

- #define: `DCMI_ExtendedDataMode_14b((uint16_t)0x0C00)`
  Interface captures 14-bit data on every pixel clock

**DCMI_Flags**

- #define: `DCMI_FLAG_HSYNC((uint16_t)0x2001)`

- #define: `DCMI_FLAG_VSYNC((uint16_t)0x2002)`

- #define: `DCMI_FLAG_FNE((uint16_t)0x2004)`

- #define: `DCMI_FLAG_FRAMERI((uint16_t)0x0001)`

- #define: `DCMI_FLAG_OVFRI((uint16_t)0x0002)`

- #define: `DCMI_FLAG_ERRRI((uint16_t)0x0004)`

- #define: `DCMI_FLAG_VSYNCRI((uint16_t)0x0008)`

- #define: `DCMI_FLAG_LINERI((uint16_t)0x0010)`

- #define: `DCMI_FLAG_FRAMEMI((uint16_t)0x1001)`

- #define: `DCMI_FLAG_OVFMI((uint16_t)0x1002)`
- #define: DCMI_FLAG_ERRMI((uint16_t)0x1004)

- #define: DCMI_FLAG_VSYNCMI((uint16_t)0x1008)

- #define: DCMI_FLAG_LINEMI((uint16_t)0x1010)

**DCMI_HSYNC_Polarity**

- #define: DCMI_HSPolarity_Low((uint16_t)0x0000)
  Horizontal synchronization active Low

- #define: DCMI_HSPolarity_High((uint16_t)0x0040)
  Horizontal synchronization active High

**DCMI_interrupt_sources**

- #define: DCMI_IT_FRAME((uint16_t)0x0001)

- #define: DCMI_IT_OVF((uint16_t)0x0002)

- #define: DCMI_IT_ERR((uint16_t)0x0004)

- #define: DCMI_IT_VSYNC((uint16_t)0x0008)

- #define: DCMI_IT_LINE((uint16_t)0x0010)

**DCMI_PIXCK_Polarity**

- #define: DCMI_PCKPolarity_Falling((uint16_t)0x0000)
  Pixel clock active on Falling edge
Digital camera interface (DCMI)

- #define: `DCMI_PCKPolarity_Rising((uint16_t)0x0020)
  
  Pixel clock active on Rising edge

**DCMI_Synchronization_Mode**
- #define: `DCMI_SynchroMode_Hardware((uint16_t)0x0000)
  
  Hardware synchronization data capture (frame/line start/stop) is synchronized with the HSYNC/VSYNC signals

- #define: `DCMI_SynchroMode_Embedded((uint16_t)0x0010)
  
  Embedded synchronization data capture is synchronized with synchronization codes embedded in the data flow

**DCMI_VSYNC_Polarity**
- #define: `DCMI_VSPolarity_Low((uint16_t)0x0000)
  
  Vertical synchronization active Low

- #define: `DCMI_VSPolarity_High((uint16_t)0x0080)
  
  Vertical synchronization active High

### 9.4 DCMI Programming Example

The example below explains how to configure the DCMI to capture continuous frame from a camera module. For more examples about DCMI configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project/STM32F2xx_StdPeriph_Examples/DCMI/

```c
DCMI_InitTypeDef DCMI_InitStruct;
/* Enable DCMI clock */
RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_DCMI, ENABLE);
/* DCMI configuration */
DCMI_InitStruct.DCMI_CaptureMode = DCMI_CaptureMode_Continuous;
DCMI_InitStruct.DCMI_SynchroMode = DCMI_SynchroMode_Hardware;
DCMI_InitStruct.DCMI_PCKPolarity = DCMI_PCKPolarity_Falling;
DCMI_InitStruct.DCMI_VSPolarity = DCMI_VSPolarity_High;
DCMI_InitStruct.DCMI_HSPolarity = DCMI_HSPolarity_High;
DCMI_InitStruct.DCMI_CaptureRate = DCMI_CaptureRate_All_Frame;
DCMI_InitStruct.DCMI_ExtendedDataMode = DCMI_ExtendedDataMode_8b;
DCMI_Init(&DCMI_InitStruct);
```
10 DMA controller (DMA)

10.1 DMA Firmware driver registers structures

10.1.1 DMA_TypeDef

DMA_TypeDef is defined in the stm32f2xx.h file and contains the DMA common registers definition.

Data Fields

- __IO uint32_t LISR
- __IO uint32_t HISR
- __IO uint32_t LIFCR
- __IO uint32_t HIFCR

Field Documentation

- __IO uint32_t DMA_TypeDef::LISR
  - DMA low interrupt status register, Address offset: 0x00
- __IO uint32_t DMA_TypeDef::HISR
  - DMA high interrupt status register, Address offset: 0x04
- __IO uint32_t DMA_TypeDef::LIFCR
  - DMA low interrupt flag clear register, Address offset: 0x08
- __IO uint32_t DMA_TypeDef::HIFCR
  - DMA high interrupt flag clear register, Address offset: 0x0C

10.1.2 DMA_Stream_TypeDef

DMA_Stream_TypeDef is defined in the stm32f2xx.h file and contains the DMA's stream registers definition.

Data Fields

- __IO uint32_t CR
- __IO uint32_t NDTR
- __IO uint32_t PAR
- __IO uint32_t M0AR
- __IO uint32_t M1AR
- __IO uint32_t FCR

Field Documentation

- __IO uint32_t DMA_Stream_TypeDef::CR
  - DMA stream x configuration register
- __IO uint32_t DMA_Stream_TypeDef::NDTR
DMA controller (DMA)

- DMA stream x number of data register
- `__IO uint32_t DMA_Stream_TypeDef::PAR`
  - DMA stream x peripheral address register
- `__IO uint32_t DMA_Stream_TypeDef::M0AR`
  - DMA stream x memory 0 address register
- `__IO uint32_t DMA_Stream_TypeDef::M1AR`
  - DMA stream x memory 1 address register
- `__IO uint32_t DMA_Stream_TypeDef::FCR`
  - DMA stream x FIFO control register

10.1.3 DMA_InitTypeDef

`DMA_InitTypeDef` is defined in the stm32f2xx_dma.h file and contains the DMA initialization parameters.

Data Fields

- `uint32_t DMA_Channel`
- `uint32_t DMA_PeripheralBaseAddr`
- `uint32_t DMA_Memory0BaseAddr`
- `uint32_t DMA_DIR`
- `uint32_t DMA_BufferSize`
- `uint32_t DMA_PeripheralInc`
- `uint32_t DMA_MemoryInc`
- `uint32_t DMA_PeripheralDataSize`
- `uint32_t DMA_MemoryDataSize`
- `uint32_t DMA_Mode`
- `uint32_t DMA_Priority`
- `uint32_t DMA_FIFOMode`
- `uint32_t DMA_FIFOThreshold`
- `uint32_t DMA_MemoryBurst`
- `uint32_t DMA_PeripheralBurst`

Field Documentation

- `uint32_t DMA_InitTypeDef::DMA_Channel`
  - Specifies the channel used for the specified stream. This parameter can be a value of `DMA_channel`
- `uint32_t DMA_InitTypeDef::DMA_PeripheralBaseAddr`
  - Specifies the peripheral base address for DMAy Streamx.
- `uint32_t DMA_InitTypeDef::DMA_Memory0BaseAddr`
  - Specifies the memory 0 base address for DMAy Streamx. This memory is the default memory used when double buffer mode is not enabled.
- `uint32_t DMA_InitTypeDef::DMA_DIR`
  - Specifies if the data will be transferred from memory to peripheral, from memory to memory or from peripheral to memory. This parameter can be a value of `DMA_data_transfer_direction`
- `uint32_t DMA_InitTypeDef::DMA_BufferSize`
10.2 DMA Firmware driver API description

The following section lists the various functions of the DMA library.

10.2.1 How to use this driver

1. Enable The DMA controller clock using
   RCC_AHB1PeriphResetCmd(RCC_AHB1Periph_DMA1, ENABLE) function for DMA1
   or using RCC_AHB1PeriphResetCmd(RCC_AHB1Periph_DMA2, ENABLE) function
   for DMA2.
2. Enable and configure the peripheral to be connected to the DMA Stream (except
   for internal SRAM / FLASH memories: no initialization is necessary).
3. For a given Stream, program the required configuration through following parameters:
   Source and Destination addresses, Transfer Direction, Transfer size, Source and
   Destination data formats, Circular or Normal mode, Stream Priority level, Source and
   Destination Incrementation mode, FIFO mode and its Threshold (if needed), Burst
   Transfer configuration for the peripheral transfers. It specifies
   the amount of data to be transferred in a single non interruptable transaction.
   This parameter can be a value of DMA_peripheral_burst

mode for Source and/or Destination (if needed) using the DMA_Init() function. To avoid filling un-necessary fields, you can call DMA_StructInit() function to initialize a given structure with default values (reset values), the modify only necessary fields (i.e. Source and Destination addresses, Transfer size and Data Formats).

4. Enable the NVIC and the corresponding interrupt(s) using the function DMA_ITConfig() if you need to use DMA interrupts.

5. Optionally, if the Circular mode is enabled, you can use the Double buffer mode by configuring the second Memory address and the first Memory to be used through the function DMA_DoubleBufferModeConfig(). Then enable the Double buffer mode through the function DMA_DoubleBufferModeCmd(). These operations must be done before step 6.

6. Enable the DMA stream using the DMA_Cmd() function.

7. Activate the needed Stream Request using PPP_DMACmd() function for any PPP peripheral except internal SRAM and FLASH (i.e. SPI, USART ...) The function allowing this operation is provided in each PPP peripheral driver (i.e. SPI_DMACmd for SPI peripheral). Once the Stream is enabled, it is not possible to modify its configuration unless the stream is stopped and disabled. After enabling the Stream, it is advised to monitor the EN bit status using the function DMA_GetCmdStatus(). In case of configuration errors or bus errors this bit will remain reset and all transfers on this Stream will remain on hold.

8. Optionally, you can configure the number of data to be transferred when the Stream is disabled (i.e. after each Transfer Complete event or when a Transfer Error occurs) using the function DMA_SetCurrDataCounter(). And you can get the number of remaining data to be transferred using the function DMA_GetCurrDataCounter() at run time (when the DMA Stream is enabled and running).

9. To control DMA events you can use one of the following two methods: After checking a flag you should clear it using DMA_ClearFlag() function. And after checking on an interrupt event you should clear it using DMA_ClearITPendingBit() function.
   a. Check on DMA Stream flags using the function DMA_GetFlagStatus().
   b. Use DMA interrupts through the function DMA_ITConfig() at initialization phase and DMA_GetITStatus() function into interrupt routines in communication phase.

10. Optionally, if Circular mode and Double Buffer mode are enabled, you can modify the Memory Addresses using the function DMA_MemoryTargetConfig(). Make sure that the Memory Address to be modified is not the one currently in use by DMA Stream. This condition can be monitored using the function DMA_GetCurrentMemoryTarget().

11. Optionally, Pause-Resume operations may be performed: The DMA_Cmd() function may be used to perform Pause-Resume operation. When a transfer is ongoing, calling this function to disable the Stream will cause the transfer to be paused. All configuration registers and the number of remaining data will be preserved. When calling again this function to re-enable the Stream, the transfer will be resumed from the point where it was paused.

Memory-to-Memory transfer is possible by setting the address of the memory into the Peripheral registers. In this mode, Circular mode and Double Buffer mode are not allowed.

The FIFO is used mainly to reduce bus usage and to allow data packing/unpacking: it is possible to set different Data Sizes for the Peripheral and the Memory (i.e. you can set Half-Word data size for the peripheral to access its data register and set Word data size for the Memory to gain in access time. Each two Half-words will be packed and written in a single access to a Word in the Memory).
When FIFO is disabled, it is not allowed to configure different Data Sizes for Source and Destination. In this case the Peripheral Data Size will be applied to both Source and Destination.

### 10.2.2 Initialization and configuration

This subsection provides functions allowing to initialize the DMA Stream source and destination addresses, incrementation and data sizes, transfer direction, buffer size, circular/normal mode selection, memory-to-memory mode selection and Stream priority value.

The DMA_Init() function follows the DMA configuration procedures as described in reference manual (RM0033) except the first point: waiting on EN bit to be reset. This condition should be checked by user application using the function DMA_GetCmdStatus() before calling the DMA_Init() function.

- DMA_DeInit()
- DMA_Init()
- DMA_StructInit()
- DMA_Cmd()
- DMA_PeriphIncOffsetSizeConfig()
- DMA_FlowControllerConfig()

#### Data Counter functions

This subsection describes the functions allowing to configure and read the buffer size (number of data to be transferred).

The DMA data counter can be written only when the DMA Stream is disabled (i.e. after transfer complete event).

The following function can be used to write the Stream data counter value:

- void DMA_SetCurrDataCounter(DMA_Stream_TypeDef DMAy_Streamx, uint16_t Counter);

It is advised to use this function rather than DMA_Init() in situations where only the Data buffer needs to be reloaded.

If the Source and Destination Data Sizes are different, then the value written in data counter, expressing the number of transfers, is relative to the number of transfers from the Peripheral point of view. i.e. If Memory data size is Word, Peripheral data size is Half-Words, then the value to be configured in the data counter is the number of Half-Words to be transferred from/to the peripheral.
The DMA data counter can be read to indicate the number of remaining transfers for the relative DMA Stream. This counter is decremented at the end of each data transfer and when the transfer is complete:
- If Normal mode is selected: the counter is set to 0.
- If Circular mode is selected: the counter is reloaded with the initial value (configured before enabling the DMA Stream).

The following function can be used to read the Stream data counter value:
- `uint16_t DMA_GetCurrDataCounter(DMA_Stream_TypeDef DMAy_Streamx);`
- `DMA_SetCurrDataCounter()`
- `DMA_GetCurrDataCounter()`

### Double Buffer mode functions

This subsection provides function allowing to configure and control the double buffer mode parameters.

The Double Buffer mode can be used only when Circular mode is enabled.

The Double Buffer mode cannot be used when transferring data from Memory to Memory.

The Double Buffer mode allows to set two different Memory addresses from/to which the DMA controller will access alternatively (after completing transfer to/from target memory 0, it will start transfer to/from target memory 1).

This allows to reduce software overhead for double buffering and reduce the CPU access time.

Two functions must be called before calling the DMA_Init() function:
- `void DMA_DoubleBufferModeConfig(DMA_Stream_TypeDef DMAy_Streamx, uint32_t Memory1BaseAddr, uint32_t DMA_CurrentMemory);`
- `void DMA_DoubleBufferModeCmd(DMA_Stream_TypeDef DMAy_Streamx, FunctionalState NewState);`

DMA_DoubleBufferModeConfig() is called to configure the Memory 1 base address and the first Memory target from/to which the transfer will start after enabling the DMA Stream. Then DMA_DoubleBufferModeCmd() must be called to enable the Double Buffer mode (or disable it when it should not be used). Two functions can be called dynamically when the transfer is ongoing (or when the DMA Stream is stopped) to modify on of the target Memories addresses or to check which Memory target is currently used:
- `void DMA_MemoryTargetConfig(DMA_Stream_TypeDef DMAy_Streamx, uint32_t MemoryBaseAddr, uint32_t DMA_MemoryTarget);`
- `uint32_t DMA_GetCurrentMemoryTarget(DMA_Stream_TypeDef DMAy_Streamx);`

DMA_MemoryTargetConfig() can be called to modify the base address of one of the two target Memories.

The Memory of which the base address will be modified must not be currently be used by the DMA Stream (i.e. if the DMA Stream is currently transferring from Memory 1 then you can only modify base address of target Memory 0 and vice versa). To check this condition, it is recommended to use the function DMA_GetCurrentMemoryTarget() which returns the index of the Memory target currently in use by the DMA Stream.

- `DMA_DoubleBufferModeConfig()`
- `DMA_DoubleBufferModeCmd()`
- `DMA_MemoryTargetConfig()`
- `DMA_GetCurrentMemoryTarget()`
10.2.3 Interrupt and flag management

This subsection provides functions allowing to:

- Check the DMA enable status
- Check the FIFO status
- Configure the DMA Interrupts sources and check or clear the flags or pending bits status.

**DMA Enable status**

After configuring the DMA Stream (DMA_Init() function) and enabling the stream, it is recommended to check (or wait until) the DMA Stream is effectively enabled.

A Stream may remain disabled if a configuration parameter is wrong.

After disabling a DMA Stream, it is also recommended to check (or wait until) the DMA Stream is effectively disabled. If a Stream is disabled while a data transfer is ongoing, the current data will be transferred and the Stream will be effectively disabled only after this data transfer completion.

To monitor this state it is possible to use the following function:

- FunctionalState DMA_GetCmdStatus(DMA_Stream_TypeDef DMAy_Streamx);

**FIFO Status**

It is possible to monitor the FIFO status when a transfer is ongoing using the following function:

- uint32_t DMA_GetFIFOStatus(DMA_Stream_TypeDef DMAy_Streamx);

**DMA Interrupts and Flags**

The user should identify which mode will be used in his application to manage the DMA controller events: Polling mode or Interrupt mode.

- **Polling Mode** Each DMA stream can be managed through 5 event Flags (x : DMA Stream number): DMA_FLAG_FEIFx : that indicates that a FIFO Mode Transfer Error event occurred. DMA_FLAG_DMEIFx : that indicates that a Direct Mode Transfer Error event occurred. DMA_FLAG_TEIFx : that indicates that a Transfer Error event occurred. DMA_FLAG_HTIFx : that indicates that a Half-Transfer Complete event occurred. DMA_FLAG_TCIFx : that indicates that a Transfer Complete event occurred.

  In this mode it is recommended to use the following functions: FlagStatus DMA_GetFlagStatus(DMA_Stream_TypeDef DMAy_Streamx, uint32_t DMA_FLAG); void DMA_ClearFlag(DMA_Stream_TypeDef DMAy_Streamx, uint32_t DMA_FLAG);

- **Interrupt Mode** Each DMA Stream can be managed through 5 Interrupts, depending on the Interrupt Source: DMA_IT_FEIFx : specifies the interrupt source for the FIFO Mode Transfer Error event. DMA_IT_DMEIFx : specifies the interrupt source for the Direct Mode Transfer Error event. DMA_IT_TEIFx : specifies the interrupt source for the Transfer Error event. DMA_IT_HTIFx : specifies the interrupt source for the Half-Transfer Complete event. DMA_IT_TCIFx : specifies the interrupt source for the a Transfer Complete event.

  In this Mode it is recommended to use the following functions: void DMA_ITConfig(DMA_Stream_TypeDef DMAy_Streamx, uint32_t DMA_IT, FunctionalState NewState); ITStatus DMA_GetITStatus(DMA_Stream_TypeDef...
DMA controller (DMA)

DMAy_Streamx, uint32_t DMA_IT);  void
DMA_ClearITPendingBit(DMA_Stream_TypeDef DMAy_Streamx, uint32_t DMA_IT);

As a summary, the functions allowing to manage the DMA interrupts and flags are the following:

- DMA_GetCmdStatus()
- DMA_GetFIFOStatus()
- DMA_GetFlagStatus()
- DMA_ClearFlag()
- DMA_ITConfig()
- DMA_GetITStatus()
- DMA_ClearITPendingBit()

10.2.4 Initialization and configuration functions

10.2.4.1 DMA_DeInit

Function Name: void DMA_DeInit (DMA_Stream_TypeDef * DMAy_Streamx)
Function Description: Deinitialize the DMAy Streamx registers to their default reset values.
Parameters:
- DMAy_Streamx: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
Return values:
- None.
Notes:
- None.

10.2.4.2 DMA_Init

Function Name: void DMA_Init (DMA_Stream_TypeDef * DMAy_Streamx, DMA_InitTypeDef * DMA_InitStruct)
Function Description: Initializes the DMAy Streamx according to the specified parameters in the DMA_InitStruct structure.
Parameters:
- DMAy_Streamx: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- DMA_InitStruct: pointer to a DMA_InitTypeDef structure that contains the configuration information for the specified DMA Stream.
Return values:
- None.
Notes:
- Before calling this function, it is recommended to check that the Stream is actually disabled using the function DMA_GetCmdStatus().
10.2.4.3 DMA_StructInit

Function Name: void DMA_StructInit (DMA_InitTypeDef * DMA_InitStruct)

Function Description: Fills each DMA_InitStruct member with its default value.

Parameters:
- DMA_InitStruct: pointer to a DMA_InitTypeDef structure which will be initialized.

Return values:
- None.

Notes:
- None.

10.2.4.4 DMA_Cmd

Function Name: void DMA_Cmd (DMA_Stream_TypeDef * DMAy_Streamx, FunctionalState NewState)

Function Description: Enables or disables the specified DMAy Streamx.

Parameters:
- DMAy_Streamx: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- NewState: new state of the DMAy Streamx. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- This function may be used to perform Pause-Resume operation. When a transfer is ongoing, calling this function to disable the Stream will cause the transfer to be paused. All configuration registers and the number of remaining data will be preserved. When calling again this function to re-enable the Stream, the transfer will be resumed from the point where it was paused.
- After configuring the DMA Stream (DMA_Init() function) and enabling the stream, it is recommended to check (or wait until) the DMA Stream is effectively enabled. A Stream may remain disabled if a configuration parameter is wrong. After disabling a DMA Stream, it is also recommended to check (or wait until) the DMA Stream is effectively disabled. If a Stream is disabled while a data transfer is ongoing, the current data will be transferred and the Stream will be effectively disabled only after the transfer of this single data is finished.
## 10.2.4.5 DMA_PeriphIncOffsetSizeConfig

**Function Name**

```c
void DMA_PeriphIncOffsetSizeConfig (DMA_Stream_TypeDef * DMAy_Streamx, uint32_t DMA_Pincos)
```

**Function Description**

Configures, when the PINC (Peripheral Increment address mode) bit is set, if the peripheral address should be incremented with the data size (configured with PSIZE bits) or by a fixed offset equal to 4 (32-bit aligned addresses).

**Parameters**

- **DMAy_Streamx**: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- **DMA_Pincos**: specifies the Peripheral increment offset size. This parameter can be one of the following values:
  - **DMA_PINCOS_Psize**: Peripheral address increment is done accordingly to PSIZE parameter.
  - **DMA_PINCOS_WordAligned**: Peripheral address increment offset is fixed to 4 (32-bit aligned addresses).

**Return values**

- None.

**Notes**

- This function has no effect if the Peripheral Increment mode is disabled.

## 10.2.4.6 DMA_FlowControllerConfig

**Function Name**

```c
void DMA_FlowControllerConfig (DMA_Stream_TypeDef * DMAy_Streamx, uint32_t DMA_FlowCtrl)
```

**Function Description**

Configures, when the DMAy Streamx is disabled, the flow controller for the next transactions (Peripheral or Memory).

**Parameters**

- **DMAy_Streamx**: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- **DMA_FlowCtrl**: specifies the DMA flow controller. This parameter can be one of the following values:
  - **DMA_FlowCtrl_Memory**: DMAy_Streamx transactions flow controller is the DMA controller.
  - **DMA_FlowCtrl_Peripheral**: DMAy_Streamx transactions flow controller is the peripheral.

**Return values**

- None.

**Notes**

- Before enabling this feature, check if the used peripheral supports the Flow Controller mode or not.
10.2.5 Data Counter functions

10.2.5.1 DMA_SetCurrDataCounter

Function Name: void DMA_SetCurrDataCounter (DMA_Stream_TypeDef *DMAy_Streamx, uint16_t Counter)

Function Description: Writes the number of data units to be transferred on the DMAy Streamx.

Parameters:
- DMAy_Streamx: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- Counter: Number of data units to be transferred (from 0 to 65535) Number of data items depends only on the Peripheral data format.

Return values:
- The number of remaining data units in the current DMAy Streamx transfer.

Notes:
- If Peripheral data format is Bytes: number of data units is equal to total number of bytes to be transferred.
- If Peripheral data format is Half-Word: number of data units is equal to total number of bytes to be transferred / 2.
- If Peripheral data format is Word: number of data units is equal to total number of bytes to be transferred / 4.
- In Memory-to-Memory transfer mode, the memory buffer pointed by DMAy_SxPAR register is considered as Peripheral.

10.2.5.2 DMA_GetCurrDataCounter

Function Name: uint16_t DMA_GetCurrDataCounter (DMA_Stream_TypeDef *DMAy_Streamx)

Function Description: Returns the number of remaining data units in the current DMAy Streamx transfer.

Parameters:
- DMAy_Streamx: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.

Return values:
- The number of remaining data units in the current DMAy Streamx transfer.

Notes:
- None.
10.2.6  Double Buffer mode functions

10.2.6.1  DMA_DoubleBufferModeConfig

Function Name  void DMA_DoubleBufferModeConfig (DMA_Stream_TypeDef * DMAy_Streamx, uint32_t Memory1BaseAddr, uint32_t DMA_CurrentMemory)

Function Description  Configures, when the DMAy Streamx is disabled, the double buffer mode and the current memory target.

Parameters
- DMAy_Streamx : where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- Memory1BaseAddr : the base address of the second buffer (Memory 1)
- DMA_CurrentMemory : specifies which memory will be first buffer for the transactions when the Stream will be enabled. This parameter can be one of the following values:
  - DMA_Memory_0 : Memory 0 is the current buffer.
  - DMA_Memory_1 : Memory 1 is the current buffer.

Return values
- None.

Notes
- Memory0BaseAddr is set by the DMA structure configuration in DMA_Init().

10.2.6.2  DMA_DoubleBufferModeCmd

Function Name  void DMA_DoubleBufferModeCmd (DMA_Stream_TypeDef * DMAy_Streamx, FunctionalState NewState)

Function Description  Enables or disables the double buffer mode for the selected DMA stream.

Parameters
- DMAy_Streamx : where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- NewState : new state of the DMAy Streamx double buffer mode. This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- This function can be called only when the DMA Stream is disabled.
10.2.6.3 DMA_MemoryTargetConfig

Function Name
void DMA_MemoryTargetConfig (DMA_Stream_TypeDef * DMAy_Streamx, uint32_t MemoryBaseAddr, uint32_t DMA_MemoryTarget)

Function Description
Configures the Memory address for the next buffer transfer in double buffer mode (for dynamic use).

Parameters
• DMAy_Streamx : where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
• MemoryBaseAddr : The base address of the target memory buffer
• DMA_MemoryTarget : Next memory target to be used. This parameter can be one of the following values:
  – DMA_Memory_0 : To use the memory address 0
  – DMA_Memory_1 : To use the memory address 1

Return values
• None.

Notes
• It is not allowed to modify the Base Address of a target Memory when this target is involved in the current transfer. ie. If the DMA Stream is currently transferring to/from Memory 1, then it is not possible to modify Base address of Memory 1, but it is possible to modify Base address of Memory 0. To know which Memory is currently used, you can use the function DMA_GetCurrentMemoryTarget().

10.2.6.4 DMA_GetCurrentMemoryTarget

Function Name
uint32_t DMA_GetCurrentMemoryTarget (DMA_Stream_TypeDef * DMAy_Streamx)

Function Description
Returns the current memory target used by double buffer transfer.

Parameters
• DMAy_Streamx : where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.

Return values
• The memory target number: 0 for Memory0 or 1 for Memory1.

Notes
• None.
10.2.7 Interrupt and flag management functions

10.2.7.1 DMA_GetCmdStatus

Function Name: FunctionalState DMA_GetCmdStatus (DMA_Stream_TypeDef * DMAy_Streamx)
Function Description: Returns the status of EN bit for the specified DMAy Streamx.
Parameters:
- DMAy_Streamx : where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
Return values:
- Current state of the DMAy Streamx (ENABLE or DISABLE).
Notes:
- After configuring the DMA Stream (DMA_Init() function) and enabling the stream, it is recommended to check (or wait until) the DMA Stream is effectively enabled. A Stream may remain disabled if a configuration parameter is wrong. After disabling a DMA Stream, it is also recommended to check (or wait until) the DMA Stream is effectively disabled. If a Stream is disabled while a data transfer is ongoing, the current data will be transferred and the Stream will be effectively disabled only after the transfer of this single data is finished.

10.2.7.2 DMA_GetFIFOStatus

Function Name: uint32_t DMA_GetFIFOStatus (DMA_Stream_TypeDef * DMAy_Streamx)
Function Description: Returns the current DMAy Streamx FIFO filled level.
Parameters:
- DMAy_Streamx : where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
Return values:
- The FIFO filling state.
  - DMA_FIFOStatus_Less1QuarterFull: when FIFO is less than 1 quarter-full and not empty.
  - DMA_FIFOStatus_1QuarterFull: if more than 1 quarter-full.
  - DMA_FIFOStatus_HalfFull: if more than 1 half-full.
  - DMA_FIFOStatus_3QuartersFull: if more than 3 quarters-full.
  - DMA_FIFOStatus_Empty: when FIFO is empty
  - DMA_FIFOStatus_Full: when FIFO is full
10.2.7.3 DMA_GetFlagStatus

Function Name  FlagStatus DMA_GetFlagStatus (DMA_Stream_TypeDef *DMAy_Streamx, uint32_t DMA_FLAG)

Function Description Checks whether the specified DMAy Streamx flag is set or not.

Parameters
- **DMAy_Streamx**: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- **DMA_FLAG**: specifies the flag to check. This parameter can be one of the following values:
  - **DMA_FLAG_TCIFx**: Streamx transfer complete flag
  - **DMA_FLAG_HTIFx**: Streamx half transfer complete flag
  - **DMA_FLAG_TEIFx**: Streamx transfer error flag
  - **DMA_FLAG_DMEIFx**: Streamx direct mode error flag
  - **DMA_FLAG_FEIFx**: Streamx FIFO error flag
Where x can be 0 to 7 to select the DMA Stream.

Return values
- The new state of DMA_FLAG (SET or RESET).

Notes
- None.

10.2.7.4 DMA_ClearFlag

Function Name  void DMA_ClearFlag (DMA_Stream_TypeDef *DMAy_Streamx, uint32_t DMA_FLAG)

Function Description Clears the DMAy Streamx's pending flags.

Parameters
- **DMAy_Streamx**: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- **DMA_FLAG**: specifies the flag to clear. This parameter can be any combination of the following values:
  - **DMA_FLAG_TCIFx**: Streamx transfer complete flag
  - **DMA_FLAG_HTIFx**: Streamx half transfer complete flag
  - **DMA_FLAG_TEIFx**: Streamx transfer error flag
  - **DMA_FLAG_DMEIFx**: Streamx direct mode error flag
  - **DMA_FLAG_FEIFx**: Streamx FIFO error flag
Where x can be 0 to 7 to select the DMA Stream.
10.2.7.5 DMA_ITConfig

Function Name: void DMA_ITConfig (DMA_Stream_TypeDef *DMAy_Streamx, uint32_t DMA_IT, FunctionalState NewState)

Function Description: Enables or disables the specified DMAy Streamx interrupts.

Parameters:
- **DMAy_Streamx**: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- **DMA_IT**: specifies the DMA interrupt sources to be enabled or disabled. This parameter can be any combination of the following values:
  - **DMA_IT_TC**: Transfer complete interrupt mask
  - **DMA_IT_HT**: Half transfer complete interrupt mask
  - **DMA_IT_TE**: Transfer error interrupt mask
  - **DMA_IT_FE**: FIFO error interrupt mask
- **NewState**: new state of the specified DMA interrupts. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

10.2.7.6 DMA_GetITStatus

Function Name: ITStatus DMA_GetITStatus (DMA_Stream_TypeDef *DMAy_Streamx, uint32_t DMA_IT)

Function Description: Checks whether the specified DMAy Streamx interrupt has occurred or not.

Parameters:
- **DMAy_Streamx**: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- **DMA_IT**: specifies the DMA interrupt source to check. This parameter can be one of the following values:
  - **DMA_IT_TCIFx**: Streamx transfer complete interrupt
  - **DMA_IT_HTIFx**: Streamx half transfer complete interrupt
  - **DMA_IT_TEIFx**: Streamx transfer error interrupt
  - **DMA_IT_DMEIFx**: Streamx direct mode error interrupt
10.2.7.7 DMA_ClearITPendingBit

Function Name: void DMA_ClearITPendingBit (DMA_Stream_TypeDef *DMAy_Streamx, uint32_t DMA_IT)

Function Description: Clears the DMAy Streamx's interrupt pending bits.

Parameters:
- DMAy_Streamx: where y can be 1 or 2 to select the DMA and x can be 0 to 7 to select the DMA Stream.
- DMA_IT: specifies the DMA interrupt pending bit to clear. This parameter can be any combination of the following values:
  - DMA_IT_TCIFx: Streamx transfer complete interrupt
  - DMA_IT_HTIFx: Streamx half transfer complete interrupt
  - DMA_IT_TEIFx: Streamx transfer error interrupt
  - DMA_IT_DMEIFx: Streamx direct mode error interrupt
  - DMA_IT_FEIFx: Streamx FIFO error interrupt Where x can be 0 to 7 to select the DMA Stream.

Return values: None.

Notes: None.
DMA controller (DMA)

- #define: `DMA_Channel_2((uint32_t)0x04000000)`

- #define: `DMA_Channel_3((uint32_t)0x06000000)`

- #define: `DMA_Channel_4((uint32_t)0x08000000)`

- #define: `DMA_Channel_5((uint32_t)0x0A000000)`

- #define: `DMA_Channel_6((uint32_t)0x0C000000)`

- #define: `DMA_Channel_7((uint32_t)0x0E000000)`

`DMA_circular_normal_mode`

- #define: `DMA_Mode_Normal((uint32_t)0x00000000)`

- #define: `DMA_Mode_Circular((uint32_t)0x00000100)`

`DMA_data_transfer_direction`

- #define: `DMA_DIR_PeripheralToMemory((uint32_t)0x00000000)`

- #define: `DMA_DIR_MemoryToPeripheral((uint32_t)0x00000040)`

- #define: `DMA_DIR_MemoryToMemory((uint32_t)0x00000080)`
DMA_fifo_direct_mode

- #define: DMA_FIFOMode_Disable((uint32_t)0x00000000)

- #define: DMA_FIFOMode_Enable((uint32_t)0x00000004)

DMA_fifo_status_level

- #define: DMA_FIFOStatus_Less1QuarterFull((uint32_t)0x00000000 << 3)

- #define: DMA_FIFOStatus_1QuarterFull((uint32_t)0x00000001 << 3)

- #define: DMA_FIFOStatus_HalfFull((uint32_t)0x00000002 << 3)

- #define: DMA_FIFOStatus_3QuartersFull((uint32_t)0x00000003 << 3)

- #define: DMA_FIFOStatus_Empty((uint32_t)0x00000004 << 3)

- #define: DMA_FIFOStatus_Full((uint32_t)0x00000005 << 3)

DMA_fifo_threshold_level

- #define: DMA_FIFOThreshold_1QuarterFull((uint32_t)0x00000000)

- #define: DMA_FIFOThreshold_HalfFull((uint32_t)0x00000001)

- #define: DMA_FIFOThreshold_3QuartersFull((uint32_t)0x00000002)
- #define: `DMA_FIFOThreshold_Full((uint32_t)0x00000003)`

`DMA_flags_definition`

- #define: `DMA_FLAG_FEIF0((uint32_t)0x10800001)`

- #define: `DMA_FLAG_DMEIF0((uint32_t)0x10800004)`

- #define: `DMA_FLAG_TEIF0((uint32_t)0x10000008)`

- #define: `DMA_FLAG_HTIF0((uint32_t)0x10000010)`

- #define: `DMA_FLAG_TCIF0((uint32_t)0x10000020)`

- #define: `DMA_FLAG_FEIF1((uint32_t)0x10000040)`

- #define: `DMA_FLAG_DMEIF1((uint32_t)0x10000100)`

- #define: `DMA_FLAG_TEIF1((uint32_t)0x10000200)`

- #define: `DMA_FLAG_HTIF1((uint32_t)0x10000400)`

- #define: `DMA_FLAG_TCIF1((uint32_t)0x10000800)`

- #define: `DMA_FLAG_FEIF2((uint32_t)0x10010000)`
- #define: `DMA_FLAG_DMEIF2((uint32_t)0x10040000)`
- #define: `DMA_FLAG_TEIF2((uint32_t)0x10080000)`
- #define: `DMA_FLAG_HTIF2((uint32_t)0x10100000)`
- #define: `DMA_FLAG_TCIF2((uint32_t)0x10200000)`
- #define: `DMA_FLAG_FEIF3((uint32_t)0x10400000)`
- #define: `DMA_FLAG_DMEIF3((uint32_t)0x11000000)`
- #define: `DMA_FLAG_TEIF3((uint32_t)0x12000000)`
- #define: `DMA_FLAG_HTIF3((uint32_t)0x14000000)`
- #define: `DMA_FLAG_TCIF3((uint32_t)0x18000000)`
- #define: `DMA_FLAG_FEIF4((uint32_t)0x20000001)`
- #define: `DMA_FLAG_DMEIF4((uint32_t)0x20000004)`
- #define: `DMA_FLAG_TEIF4((uint32_t)0x20000008)`
• #define: `DMA_FLAG_HTIF4((uint32_t)0x20000010)`

• #define: `DMA_FLAG_TCIF4((uint32_t)0x20000020)`

• #define: `DMA_FLAG_FEIF5((uint32_t)0x20000040)`

• #define: `DMA_FLAG_DMEIF5((uint32_t)0x20000100)`

• #define: `DMA_FLAG_TEIF5((uint32_t)0x20000200)`

• #define: `DMA_FLAG_HTIF5((uint32_t)0x20000400)`

• #define: `DMA_FLAG_TCIF5((uint32_t)0x20000800)`

• #define: `DMA_FLAG_FEIF6((uint32_t)0x20010000)`

• #define: `DMA_FLAG_DMEIF6((uint32_t)0x20040000)`

• #define: `DMA_FLAG_TEIF6((uint32_t)0x20080000)`

• #define: `DMA_FLAG_HTIF6((uint32_t)0x20100000)`

• #define: `DMA_FLAG_TCIF6((uint32_t)0x20200000)`
- #define: `DMA_FLAG_FEIF7((uint32_t)0x20400000)`

- #define: `DMA_FLAG_DMEIF7((uint32_t)0x21000000)`

- #define: `DMA_FLAG_TEIF7((uint32_t)0x22000000)`

- #define: `DMA_FLAG_HTIF7((uint32_t)0x24000000)`

- #define: `DMA_FLAG_TCIF7((uint32_t)0x28000000)`

**DMA_flow_controller_definitions**

- #define: `DMA_FlowCtrl_Memory((uint32_t)0x00000000)`

- #define: `DMA_FlowCtrl_Peripheral((uint32_t)0x00000020)`

**DMA_interrupts_definitions**

- #define: `DMA_IT_FEIF0((uint32_t)0x90000001)`

- #define: `DMA_IT_DMEIF0((uint32_t)0x10001004)`

- #define: `DMA_IT_TEIF0((uint32_t)0x10002008)`

- #define: `DMA_IT_HTIF0((uint32_t)0x10004010)`
• #define: `DMA_IT_TCIF0((uint32_t)0x10008020)`

• #define: `DMA_IT_FEIF1((uint32_t)0x90000040)`

• #define: `DMA_IT_DMEIF1((uint32_t)0x10001100)`

• #define: `DMA_IT_TEIF1((uint32_t)0x10002200)`

• #define: `DMA_IT_HTIF1((uint32_t)0x10004400)`

• #define: `DMA_IT_TCID1((uint32_t)0x10008800)`

• #define: `DMA_IT_FEIF2((uint32_t)0x90010000)`

• #define: `DMA_IT_DMEIF2((uint32_t)0x10041000)`

• #define: `DMA_IT_TEIF2((uint32_t)0x10082000)`

• #define: `DMA_IT_HTIF2((uint32_t)0x10104000)`

• #define: `DMA_IT_TCIF2((uint32_t)0x10208000)`

• #define: `DMA_IT_FEIF3((uint32_t)0x90400000)`
- #define: `DMA_IT_DMEIF3((uint32_t)0x11001000)`

- #define: `DMA_IT_TEIF3((uint32_t)0x12002000)`

- #define: `DMA_IT_HTIF3((uint32_t)0x14004000)`

- #define: `DMA_IT_TCIIF3((uint32_t)0x18008000)`

- #define: `DMA_IT_FEIF4((uint32_t)0xA0000001)`

- #define: `DMA_IT_DMEIF4((uint32_t)0x20001004)`

- #define: `DMA_IT_TEIF4((uint32_t)0x20002008)`

- #define: `DMA_IT_HTIF4((uint32_t)0x20004010)`

- #define: `DMA_IT_TCIIF4((uint32_t)0x20008020)`

- #define: `DMA_IT_FEIF5((uint32_t)0xA0000040)`

- #define: `DMA_IT_DMEIF5((uint32_t)0x20001100)`

- #define: `DMA_IT_TEIF5((uint32_t)0x20002200)`
- #define: DMA_IT_HTIF5((uint32_t)0x20004400)

- #define: DMA_IT_TCIF5((uint32_t)0x20008800)

- #define: DMA_IT_FEIF6((uint32_t)0xA0010000)

- #define: DMA_IT_DMEIF6((uint32_t)0x20041000)

- #define: DMA_IT_TEIF6((uint32_t)0x20082000)

- #define: DMA_IT_HTIF6((uint32_t)0x20104000)

- #define: DMA_IT_TCIF6((uint32_t)0x20208000)

- #define: DMA_IT_FEIF7((uint32_t)0xA0400000)

- #define: DMA_IT_DMEIF7((uint32_t)0x21001000)

- #define: DMA_IT_TEIF7((uint32_t)0x22002000)

- #define: DMA_IT_HTIF7((uint32_t)0x24004000)

- #define: DMA_IT_TCIF7((uint32_t)0x28008000)
DMA_interrupt_enable_definitions

- #define: DMA_IT_TC((uint32_t)0x00000010)

- #define: DMA_IT_HT((uint32_t)0x00000008)

- #define: DMA_IT_TE((uint32_t)0x00000004)

- #define: DMA_IT_DME((uint32_t)0x00000002)

- #define: DMA_IT_FE((uint32_t)0x00000080)

DMA_memory_burst

- #define: DMA_MemoryBurst_Single((uint32_t)0x00000000)

- #define: DMA_MemoryBurst_INC4((uint32_t)0x00800000)

- #define: DMA_MemoryBurst_INC8((uint32_t)0x01000000)

- #define: DMA_MemoryBurst_INC16((uint32_t)0x01800000)

DMA_memory_data_size

- #define: DMA_MemoryDataSize_Byte((uint32_t)0x00000000)

- #define: DMA_MemoryDataSize_HalfWord((uint32_t)0x00000200)

- #define: DMA_MemoryDataSize_Word((uint32_t)0x00000400)
DMA controller (DMA)

- #define: DMA_MemoryDataSize_Word((uint32_t)0x00004000)

DMA_memory_incremented_mode
- #define: DMA_MemoryInc_Enable((uint32_t)0x00000400)
- #define: DMA_MemoryInc_Disable((uint32_t)0x00000000)

DMA_memory_targets_definitions
- #define: DMA_Memory_0((uint32_t)0x00000000)
- #define: DMA_Memory_1((uint32_t)0x00080000)

DMA_peripheral_burst
- #define: DMA_PeripheralBurst_Single((uint32_t)0x00000000)
- #define: DMA_PeripheralBurst_INC4((uint32_t)0x00200000)
- #define: DMA_PeripheralBurst_INC8((uint32_t)0x00400000)
- #define: DMA_PeripheralBurst_INC16((uint32_t)0x00600000)

DMA_peripheral_data_size
- #define: DMA_PeripheralDataSize_Byte((uint32_t)0x00000000)
- #define: DMA_PeripheralDataSize_HalfWord((uint32_t)0x00000800)
• #define: `DMA_PeripheralDataSize_Word((uint32_t)0x00001000)`

`DMA_peripheral_incremented_mode`
• #define: `DMA_PeripheralInc_Enable((uint32_t)0x00000200)`

• #define: `DMA_PeripheralInc_Disable((uint32_t)0x00000000)`

`DMA_peripheral_increment_offset`
• #define: `DMA_PINCOS_Psize((uint32_t)0x00000000)`

• #define: `DMA_PINCOS_WordAligned((uint32_t)0x00008000)`

`DMA_priority_level`
• #define: `DMA_Priority_Low((uint32_t)0x00000000)`

• #define: `DMA_Priority_Medium((uint32_t)0x00010000)`

• #define: `DMA_Priority_High((uint32_t)0x00020000)`

• #define: `DMA_Priority_VeryHigh((uint32_t)0x00030000)`

### 10.4 DMA Programming Example

The example below explains how to configure the DMA to transfer continuously converted data from ADC1 to SRAM memory (this example assumes that the ADC is already configured). For more examples about DMA configuration and usage, please refer to the DMA examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\DMA\.
```c
#define ADC1_DR_ADDRESS ((uint32_t)0x4001204C)

uint16_t ADCConvertedValue = 0;
DMA_InitTypeDef DMA_InitStruct;

/* Enable DMA2's AHB1 interface clock */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE);

/* Configure DMA2 Stream0 channel0 to transfer, in circular mode,
the converted data from ADC1 DR register to the
ADCConvertedValue variable */
DMA_InitStruct.DMA_Channel = DMA_Channel_0;
DMA_InitStruct.DMA_PeripheralBaseAddr = ADC1_DR_ADDRESS;
DMA_InitStruct.DMA_DIR = DMA_DIR_PeripheralToMemory;
DMA_InitStruct.DMA_BufferSize = 1;
DMA_InitStruct.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
DMA_InitStruct.DMA_MemoryInc = DMA_MemoryInc_Disable;
DMA_InitStruct.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
DMA_InitStruct.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
DMA_InitStruct.DMA_Mode = DMA_Mode_Circular;
DMA_InitStruct.DMA_Priority = DMA_Priority_High;
DMA_InitStruct.DMA_FIFOMode = DMA_FIFOMode_Disable;
DMA_InitStruct.DMA_FIFOThreshold = DMA_FIFOThreshold_HalfFull;
DMA_InitStruct.DMA_MemoryBurst = DMA_MemoryBurst_Single;
DMA_InitStruct.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
DMA_Init(DMA2_Stream0, &DMA_InitStruct);

/* Enable DMA2 Stream0 */
DMA_Cmd(DMA2_Stream0, ENABLE);
```

11 External interrupt/event controller (EXTI)

11.1 EXTI Firmware driver registers structures

11.1.1 EXTI_TypeDef

`EXTI_TypeDef` is defined in the `stm32f2xx.h` file and contains the EXTI registers definition.

Data Fields

- `__IO uint32_t IMR`  
  - EXTI Interrupt mask register, Address offset: 0x00
- `__IO uint32_t EMR`  
  - EXTI Event mask register, Address offset: 0x04
- `__IO uint32_t RTSR`  
  - EXTI Rising trigger selection register, Address offset: 0x08
- `__IO uint32_t FTSR`  
  - EXTI Falling trigger selection register, Address offset: 0x0C
- `__IO uint32_t SWIER`  
  - EXTI Software interrupt event register, Address offset: 0x10
- `__IO uint32_t PR`  
  - EXTI Pending register, Address offset: 0x14

Field Documentation

- `__IO uint32_t EXTI_TypeDef::IMR`  
  - EXTI Interrupt mask register, Address offset: 0x00
- `__IO uint32_t EXTI_TypeDef::EMR`  
  - EXTI Event mask register, Address offset: 0x04
- `__IO uint32_t EXTI_TypeDef::RTSR`  
  - EXTI Rising trigger selection register, Address offset: 0x08
- `__IO uint32_t EXTI_TypeDef::FTSR`  
  - EXTI Falling trigger selection register, Address offset: 0x0C
- `__IO uint32_t EXTI_TypeDef::SWIER`  
  - EXTI Software interrupt event register, Address offset: 0x10
- `__IO uint32_t EXTI_TypeDef::PR`  
  - EXTI Pending register, Address offset: 0x14

11.1.2 EXTI_InitTypeDef

`EXTI_InitTypeDef` is defined in the `stm32f2xx_exti.h`

Data Fields

- `uint32_t EXTI_Line`  
  - EXTI Line
- `EXTIMode_TypeDef EXTI_Mode`  
  - EXTI Mode
- `EXTITrigger_TypeDef EXTI_Trigger`  
  - EXTI Trigger
- `FunctionalState EXTI_LineCmd`  
  - EXTI Line Command

Field Documentation

- `uint32_t EXTI_InitTypeDef::EXTI_Line`  
  - EXTI Line
11.2 EXTI Firmware driver API description

The following section lists the various functions of the EXTI library.

11.2.1 EXTI features

External interrupt/event lines are mapped as following:

- All available GPIO pins are connected to the 16 external interrupt/event lines from EXTI0 to EXTI15.
- EXTI line 16 is connected to the PVD Output
- EXTI line 17 is connected to the RTC Alarm event
- EXTI line 18 is connected to the USB OTG FS Wakeup from suspend event
- EXTI line 19 is connected to the Ethernet Wakeup event
- EXTI line 20 is connected to the USB OTG HS (configured in FS) Wakeup event
- EXTI line 21 is connected to the RTC Tamper and Time Stamp events
- EXTI line 22 is connected to the RTC Wakeup event

11.2.2 How to use this driver

In order to use an I/O pin as an external interrupt source, follow steps below:

1. Configure the I/O in input mode using GPIO_Init()
2. Select the input source pin for the EXTI line using SYSCFG_EXTILineConfig()
3. Select the mode (interrupt, event) and configure the trigger selection (Rising, falling or both) using EXTI_Init()
4. Configure NVIC IRQ channel mapped to the EXTI line using NVIC_Init()

SYSCFG APB clock must be enabled to get write access to SYSCFG_EXTICRx registers using RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

11.2.3 Initialization and configuration

- EXTI_DelInit()
- EXTI_Init()
- EXTI_StructInit()
11.2.4 Interrupt and flag management

- EXTI_GenerateSWInterrupt()
- EXTI_GetFlagStatus()
- EXTI_ClearFlag()
- EXTI_GetITStatus()
- EXTI_ClearITPendingBit()

11.2.5 Initialization and configuration functions

11.2.5.1 EXTI_DeInit

Function Name: void EXTI_DeInit (void)
Function Description: Deinitializes the EXTI peripheral registers to their default reset values.
Parameters: None.
Return values: None.
Notes: None.

11.2.5.2 EXTI_Init

Function Name: void EXTI_Init (EXTI_InitTypeDef * EXTI_InitStruct)
Function Description: Initializes the EXTI peripheral according to the specified parameters in the EXTI_InitStruct.
Parameters: EXTI_InitStruct: pointer to a EXTI_InitTypeDef structure that contains the configuration information for the EXTI peripheral.
Return values: None.
Notes: None.

11.2.5.3 EXTI_StructInit
### 11.2.5.4 EXTI_GenerateSWInterrupt

**Function Name**  
`void EXTI_GenerateSWInterrupt ( uint32_t EXTI_Line)`

**Function Description**  
Generates a Software interrupt on selected EXTI line.

**Parameters**  
- `EXTI_Line`: specifies the EXTI line on which the software interrupt will be generated. This parameter can be any combination of EXTI_Line where x can be (0..22)

**Return values**  
- None.

**Notes**  
- None.

---

### 11.2.6 Interrupt and flag management functions

#### 11.2.6.1 EXTI_GetFlagStatus

**Function Name**  
`FlagStatus EXTI_GetFlagStatus ( uint32_t EXTI_Line)`

**Function Description**  
Checks whether the specified EXTI line flag is set or not.

**Parameters**  
- `EXTI_Line`: specifies the EXTI line flag to check. This parameter can be EXTI_Line where x can be (0..22)

**Return values**  
- The new state of EXTI_Line (SET or RESET).

**Notes**  
- None.
11.2.6.2 EXTI_ClearFlag

Function Name: void EXTI_ClearFlag ( uint32_t EXTI_Line)
Function Description: Clears the EXTI's line pending flags.
Parameters:
- EXTI_Line: specifies the EXTI lines flags to clear. This parameter can be any combination of EXTI_Linex where x can be (0..22)
Return values:
- None.
Notes:
- None.

11.2.6.3 EXTI_GetITStatus

Function Name: ITStatus EXTI_GetITStatus ( uint32_t EXTI_Line)
Function Description: Checks whether the specified EXTI line is asserted or not.
Parameters:
- EXTI_Line: specifies the EXTI line to check. This parameter can be EXTI_Linex where x can be (0..22)
Return values:
- The new state of EXTI_Line (SET or RESET).
Notes:
- None.

11.2.6.4 EXTI_ClearITPendingBit

Function Name: void EXTI_ClearITPendingBit ( uint32_t EXTI_Line)
Function Description: Clears the EXTI's line pending bits.
Parameters:
- EXTI_Line: specifies the EXTI lines to clear. This parameter can be any combination of EXTI_Linex where x can be (0..22)
Return values:
- None.
Notes:
- None.
11.3 EXTI Firmware driver defines

11.3.1 EXTI Firmware driver defines

EXTI

**EXTI_Lines**

- #define: `EXTI_Line0((uint32_t)0x00001)`
  External interrupt line 0

- #define: `EXTI_Line1((uint32_t)0x00002)`
  External interrupt line 1

- #define: `EXTI_Line2((uint32_t)0x00004)`
  External interrupt line 2

- #define: `EXTI_Line3((uint32_t)0x00008)`
  External interrupt line 3

- #define: `EXTI_Line4((uint32_t)0x00010)`
  External interrupt line 4

- #define: `EXTI_Line5((uint32_t)0x00020)`
  External interrupt line 5

- #define: `EXTI_Line6((uint32_t)0x00040)`
  External interrupt line 6

- #define: `EXTI_Line7((uint32_t)0x00080)`
  External interrupt line 7

- #define: `EXTI_Line8((uint32_t)0x00100)`
  External interrupt line 8

- #define: `EXTI_Line9((uint32_t)0x00200)`
  External interrupt line 9

- #define: `EXTI_Line10((uint32_t)0x00400)`
  External interrupt line 10
• #define: `EXTI_Line11((uint32_t)0x00800)
  External interrupt line 11

• #define: `EXTI_Line12((uint32_t)0x01000)
  External interrupt line 12

• #define: `EXTI_Line13((uint32_t)0x02000)
  External interrupt line 13

• #define: `EXTI_Line14((uint32_t)0x04000)
  External interrupt line 14

• #define: `EXTI_Line15((uint32_t)0x08000)
  External interrupt line 15

• #define: `EXTI_Line16((uint32_t)0x10000)
  External interrupt line 16 Connected to the PVD Output

• #define: `EXTI_Line17((uint32_t)0x20000)
  External interrupt line 17 Connected to the RTC Alarm event

• #define: `EXTI_Line18((uint32_t)0x40000)
  External interrupt line 18 Connected to the USB OTG FS Wakeup from suspend event

• #define: `EXTI_Line19((uint32_t)0x80000)
  External interrupt line 19 Connected to the Ethernet Wakeup event

• #define: `EXTI_Line20((uint32_t)0x100000)
  External interrupt line 20 Connected to the USB OTG HS (configured in FS) Wakeup event

• #define: `EXTI_Line21((uint32_t)0x00200000)
  External interrupt line 21 Connected to the RTC Tamper and Time Stamp events

• #define: `EXTI_Line22((uint32_t)0x00400000)
  External interrupt line 22 Connected to the RTC Wakeup event
11.4 EXTI Programming Example

The example below shows how to configure PA0 pin to be used as EXTI Line0. For more examples about EXTI configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\EXTI\.

```c
EXTI_InitTypeDef   EXTI_InitStructure;
GPIO_InitTypeDef   GPIO_InitStructure;

/* Enable GPIOA's AHB interface clock */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

/* Enable SYSCFG's APB interface clock */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

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

/* Connect EXTI Line0 to PA0 pin */
SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0);

/* Configure EXTI line0 */
EXTI_InitTypeDef.EXTI_Line = EXTI_Line0;
EXTI_InitTypeDef.EXTI_Mode = EXTI_Mode_Interrupt;
EXTI_InitTypeDef.EXTI_Trigger = EXTI_Trigger_Falling;
EXTI_InitTypeDef.EXTI_LineCmd = ENABLE;
EXTI_Init(&EXTI_InitStructure);
```
12 FLASH Memory (FLASH)

12.1 FLASH Firmware driver registers structures

12.1.1 FLASH_TypeDef

FLASH_TypeDef is defined in the stm32f2xx.h file and contains the FLASH interface registers definition.

Data Fields

- __IO uint32_t ACR
- __IO uint32_t KEYR
- __IO uint32_t OPTKEYR
- __IO uint32_t SR
- __IO uint32_t CR
- __IO uint32_t OPTCR

Field Documentation

- __IO uint32_t FLASH_TypeDef::ACR
  - Flash access control register, Address offset: 0x00
- __IO uint32_t FLASH_TypeDef::KEYR
  - Flash key register, Address offset: 0x04
- __IO uint32_t FLASH_TypeDef::OPTKEYR
  - Flash option key register, Address offset: 0x08
- __IO uint32_t FLASH_TypeDef::SR
  - Flash status register, Address offset: 0x0C
- __IO uint32_t FLASH_TypeDef::CR
  - Flash control register, Address offset: 0x10
- __IO uint32_t FLASH_TypeDef::OPTCR
  - Flash option control register, Address offset: 0x14

12.2 FLASH Firmware driver API description

The following section lists the various functions of the FLASH library.

12.2.1 How to use this driver

This driver provides functions to configure and program the FLASH memory of all STM32F2xx devices. These functions are split in 4 groups:

- FLASH Interface configuration functions: this group includes the management of the following features:
  - Set the latency
  - Enable/Disable the prefetch buffer
  - Enable/Disable the Instruction cache and the Data cache
- Reset the Instruction cache and the Data cache
- FLASH Memory Programming functions: this group includes all needed functions to erase and program the main memory:
  - Lock and Unlock the FLASH interface
  - Erase function: Erase sector, erase all sectors
  - Program functions: byte, half word, word and double word
- Option Bytes Programming functions: this group includes all needed functions to manage the Option Bytes:
  - Set/Reset the write protection
  - Set the Read protection Level
  - Set the BOR level
  - Program the user Option Bytes
  - Launch the Option Bytes loader
- Interrupts and flags management functions: this group includes all needed functions to:
  - Enable/Disable the FLASH interrupt sources
  - Get flags status
  - Clear flags
  - Get FLASH operation status
  - Wait for last FLASH operation

12.2.2 FLASH interface configuration

This group includes the following functions:

```c
void FLASH_SetLatency(uint32_t FLASH_Latency)
```

To correctly read data from FLASH memory, the number of wait states (LATENCY) must be correctly programmed according to the frequency of the CPU clock (HCLK) and the supply voltage of the device.

<table>
<thead>
<tr>
<th>Wait states (WS) (latency)</th>
<th>HCLK clock frequency (MHz)</th>
<th>Voltage range 2.7 to 3.6 V</th>
<th>Voltage range 2.4 to 2.7 V</th>
<th>Voltage range 2.1 to 2.4 V</th>
<th>Voltage range 1.8 to 2.1 V^a</th>
</tr>
</thead>
<tbody>
<tr>
<td>0WS(1CPU cycle)</td>
<td></td>
<td>0 &lt; HCLK ≤ 30</td>
<td>0 &lt; HCLK ≤ 24</td>
<td>0 &lt; HCLK ≤ 18</td>
<td>0 &lt; HCLK ≤ 16</td>
</tr>
<tr>
<td>1WS(2CPU cycle)</td>
<td></td>
<td>30 &lt; HCLK ≤ 60</td>
<td>24 &lt; HCLK ≤ 48</td>
<td>18 &lt; HCLK ≤ 36</td>
<td>16 &lt; HCLK ≤ 32</td>
</tr>
<tr>
<td>2WS(3CPU cycle)</td>
<td></td>
<td>60 &lt; HCLK ≤ 90</td>
<td>48 &lt; HCLK ≤ 72</td>
<td>36 &lt; HCLK ≤ 54</td>
<td>32 &lt; HCLK ≤ 48</td>
</tr>
<tr>
<td>3WS(4CPU cycle)</td>
<td></td>
<td>90 &lt; HCLK ≤ 120</td>
<td>72 &lt; HCLK ≤ 96</td>
<td>54 &lt; HCLK ≤ 72</td>
<td>48 &lt; HCLK ≤ 64</td>
</tr>
<tr>
<td>4WS(5CPU cycle)</td>
<td></td>
<td>NA</td>
<td>96 &lt; HCLK ≤ 120</td>
<td>72 &lt; HCLK ≤ 90</td>
<td>64 &lt; HCLK ≤ 80</td>
</tr>
<tr>
<td>5WS(6CPU cycle)</td>
<td></td>
<td>NA</td>
<td>NA</td>
<td>90 &lt; HCLK ≤ 108</td>
<td>80 &lt; HCLK ≤ 96</td>
</tr>
<tr>
<td>6WS(7CPU cycle)</td>
<td></td>
<td>NA</td>
<td>NA</td>
<td>108 &lt; HCLK ≤ 120</td>
<td>96 &lt; HCLK ≤ 112</td>
</tr>
</tbody>
</table>

^a If IRROFF is set to VDD on STM32F20xx devices, this value can be lowered to 1.65 V when the device operates in a reduced temperature range.
The unlock sequence is not needed for these functions.

The FLASH interface configuration functions are the following:

- void FLASH_SetLatency(uint32_t FLASH_Latency)
- void FLASH_PrefetchBufferCmd(FunctionalState NewState)
- void FLASH_InstructionCacheCmd(FunctionalState NewState)
- void FLASH_DataCacheCmd(FunctionalState NewState)
- void FLASH_InstructionCacheReset(void) - void FLASH_DataCacheReset(void)

12.2.3 FLASH memory programming

This group includes the following functions:

- void FLASH_Unlock(void)
- void FLASH_Lock(void) - FLASH_Status FLASH_EraseSector(uint32_t FLASH_Sector, uint8_t VoltageRange)
- FLASH_Status FLASH_UnlockAllSectors(uint8_t VoltageRange)
- FLASH_Status FLASH_ProgramDoubleWord(uint32_t Address, uint64_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_ProgramByte(uint32_t Address, uint8_t Data)

Any operation of erase or program should follow these steps:

1. Call the FLASH_Unlock() function to enable the FLASH control register access
2. Call the desired function to erase sector(s) or program data

If IRROFF is set to V_{DD} on STM32F20xx devices, this value can be lowered to 1.65 V when the device operates in a reduced temperature range.
3. Call the `FLASH_Lock()` function to disable the FLASH control register access (recommended to protect the FLASH memory against possible unwanted operation)

The FLASH memory programming functions are the following:

- `FLASH_Unlock()`
- `FLASH_Lock()`
- `FLASH_EraseSector()`
- `FLASH_EraseAllSectors()`
- `FLASH_ProgramDoubleWord()`
- `FLASH_ProgramWord()`
- `FLASH_ProgramHalfWord()`
- `FLASH_ProgramByte()`

### 12.2.4 Option bytes programming

This group includes the following functions:

- `void FLASH_OB_Unlock(void)`
- `void FLASH_OB_Lock(void)`
- `void FLASH_OB_WRPConfig(uint32_t OB_WRP, FunctionalState NewState)`
- `void FLASH_OB_RDPConfig(uint8_t OB_RDP)`
- `void FLASH_OB_UserConfig(uint8_t OB_IWDG, uint8_t OB_STOP, uint8_t OB_STDBY)`
- `void FLASH_OB_BORConfig(uint8_t OB_BOR)`
- `FLASH_Status FLASH_ProgramOTP(uint32_t Address, uint32_t Data)`
- `void FLASH_OB_Launch(void)`
- `uint32_t FLASH_OB_GetUser(void)`
- `uint8_t FLASH_OB_GetWRP(void)`
- `uint8_t FLASH_OB_GetRDP(void)`
- `uint8_t FLASH_OB_GetBOR(void)`

Any operation of erase or program should follow these steps:

1. Call the `FLASH_OB_Unlock()` function to enable the FLASH option control register access
2. Call one or several functions to program the desired Option Bytes:
   - `void FLASH_OB_WRPConfig(uint32_t OB_WRP, FunctionalState NewState)` to Enable/Disable the desired sector write protection
   - `void FLASH_OB_RDPConfig(uint8_t OB_RDP)` to set the desired read Protection Level
   - `void FLASH_OB_UserConfig(uint8_t OB_IWDG, uint8_t OB_STOP, uint8_t OB_STDBY)` to configure the user Option Bytes.
   - `void FLASH_OB_BORConfig(uint8_t OB_BOR)` to set the BOR Level
3. Once all needed Option Bytes to be programmed are correctly written, call the `FLASH_OB_Launch()` function to launch the Option Bytes programming process. When changing the IWDG mode from HW to SW or from SW to HW, a system reset is needed to make the change effective.
4. Call the `FLASH_OB_Lock()` function to disable the FLASH option control register access (recommended to protect the Option Bytes against possible unwanted operations)

The functions that can be used to program the option bytes are the following:

- `FLASH_OB_Unlock()`
- `FLASH_OB_Lock()`
12.2.5 Interrupt and flag management

- `FLASH_ITConfig()`
- `FLASH_GetFlagStatus()`
- `FLASH_ClearFlag()`
- `FLASH_GetStatus()`
- `FLASH_WaitForLastOperation()`

12.2.6 FLASH interface configuration functions

12.2.6.1 FLASH_SetLatency

Function Name: `void FLASH_SetLatency ( uint32_t FLASH_Latency)`

Function Description: Sets the code latency value.

Parameters:
- `FLASH_Latency`: specifies the FLASH Latency value. This parameter can be one of the following values:
  - `FLASH_Latency_0`: FLASH Zero Latency cycle
  - `FLASH_Latency_1`: FLASH One Latency cycle
  - `FLASH_Latency_2`: FLASH Two Latency cycles
  - `FLASH_Latency_3`: FLASH Three Latency cycles
  - `FLASH_Latency_4`: FLASH Four Latency cycles
  - `FLASH_Latency_5`: FLASH Five Latency cycles
  - `FLASH_Latency_6`: FLASH Six Latency cycles
  - `FLASH_Latency_7`: FLASH Seven Latency cycles

Return values: None.

Notes: None.

12.2.6.2 FLASH_PrefetchBufferCmd

Function Name: `void FLASH_PrefetchBufferCmd ( FunctionalState NewState)`
Function Description
Enables or disables the Prefetch Buffer.

Parameters
- **NewState**: new state of the Prefetch Buffer. This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- None.

### 12.2.6.3 FLASH/InstructionCacheCmd

Function Name
`void FLASH/InstructionCacheCmd ( FunctionalState NewState)`

Function Description
Enables or disables the Instruction Cache feature.

Parameters
- **NewState**: new state of the Instruction Cache. This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- None.

### 12.2.6.4 FLASH/DataCacheCmd

Function Name
`void FLASH/DataCacheCmd ( FunctionalState NewState)`

Function Description
Enables or disables the Data Cache feature.

Parameters
- **NewState**: new state of the Data Cache. This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- None.

### 12.2.6.5 FLASH/InstructionCacheReset
Function Name: void FLASH_InstructionCacheReset (void)
Function Description: Resets the Instruction Cache.
Parameters: None.
Return values: None.
Notes: This function must be used only when the Instruction Cache is disabled.

12.2.6.6 FLASH_DataCacheReset

Function Name: void FLASH_DataCacheReset (void)
Function Description: Resets the Data Cache.
Parameters: None.
Return values: None.
Notes: This function must be used only when the Data Cache is disabled.

12.2.7 FLASH memory programming functions

12.2.7.1 FLASH_Unlock

Function Name: void FLASH_Unlock (void)
Function Description: Unlocks the FLASH control register access.
Parameters: None.
Return values: None.
Notes: None.

12.2.7.2 FLASH_Lock
Function Name: void FLASH_Lock ( void )

Function Description: Locks the FLASH control register access.

Parameters:
- None.

Return values:
- None.

Notes:
- None.

12.2.7.3  FLASH_EraseSector

Function Name: FLASH_Status FLASH_EraseSector ( uint32_t FLASH_Sector, uint8_t VoltageRange)

Function Description: Erases a specified FLASH Sector.

Parameters:
- **FLASH_Sector**: The Sector number to be erased. This parameter can be a value between FLASH_Sector_0 and FLASH_Sector_11.
- **VoltageRange**: The device voltage range which defines the erase parallelism. This parameter can be one of the following values:
  - **VoltageRange_1**: when the device voltage range is 1.8V to 2.1V, the operation will be done by byte (8-bit).
  - **VoltageRange_2**: when the device voltage range is 2.1V to 2.7V, the operation will be done by half word (16-bit).
  - **VoltageRange_3**: when the device voltage range is 2.7V to 3.6V, the operation will be done by word (32-bit).
  - **VoltageRange_4**: when the device voltage range is 2.7V to 3.6V + External Vpp, the operation will be done by double word (64-bit).

Return values:
- **FLASH Status**: The returned value can be: FLASH_BUSY, FLASH_ERROR_PROGRAM, FLASH_ERROR_WRP, FLASH_ERROR_OPERATION or FLASH_COMPLETE.

Notes:
- None.

12.2.7.4  FLASH_EraseAllSectors

Function Name: FLASH_Status FLASH_EraseAllSectors ( uint8_t
VoltageRange)

Function Description
Erases all FLASH Sectors.

Parameters
- **VoltageRange**: The device voltage range which defines the erase parallelism. This parameter can be one of the following values:
  - **VoltageRange_1**: when the device voltage range is 1.8V to 2.1V, the operation will be done by byte (8-bit)
  - **VoltageRange_2**: when the device voltage range is 2.1V to 2.7V, the operation will be done by half word (16-bit)
  - **VoltageRange_3**: when the device voltage range is 2.7V to 3.6V, the operation will be done by word (32-bit)
  - **VoltageRange_4**: when the device voltage range is 2.7V to 3.6V + External Vpp, the operation will be done by double word (64-bit)

Return values
- **FLASH Status**: The returned value can be: FLASH_BUSY, FLASH_ERROR_PROGRAM, FLASH_ERROR_WRP, FLASH_ERROR_OPERATION or FLASH_COMPLETE.

Notes
- None.

12.2.7.5 FLASH_ProgramDoubleWord

Function Name
FLASH_Status FLASH_ProgramDoubleWord ( uint32_t Address, uint64_t Data)

Function Description
Programs a double word (64-bit) at a specified address.

Parameters
- **Address**: specifies the address to be programmed.
- **Data**: specifies the data to be programmed.

Return values
- **FLASH Status**: The returned value can be: FLASH_BUSY, FLASH_ERROR_PROGRAM, FLASH_ERROR_WRP, FLASH_ERROR_OPERATION or FLASH_COMPLETE.

Notes
- This function must be used when the device voltage range is from 2.7V to 3.6V and an External Vpp is present.

12.2.7.6 FLASH_ProgramWord
12.2.7.7 FLASH_ProgramHalfWord

Function Name: FLASH_Status FLASH_ProgramHalfWord ( uint32_t Address, uint16_t Data )

Function Description: Programs a half word (16-bit) at a specified address.

Parameters:
- **Address**: specifies the address to be programmed. This parameter can be any address in Program memory zone or in OTP zone.
- **Data**: specifies the data to be programmed.

Return values:
- **FLASH Status**: The returned value can be: FLASH_BUSY, FLASH_ERROR_PROGRAM, FLASH_ERROR_WRP, FLASH_ERROR_OPERATION or FLASH_COMPLETE.

Notes:
- This function must be used when the device voltage range is from 2.1V to 3.6V.

12.2.7.8 FLASH_ProgramByte

Function Name: FLASH_Status FLASH_ProgramByte ( uint32_t Address, uint8_t Data )

Function Description: Programs a byte (8-bit) at a specified address.
### Parameters

- **Address**: specifies the address to be programmed. This parameter can be any address in Program memory zone or in OTP zone.
- **Data**: specifies the data to be programmed.

### Return values

- **FLASH Status**: The returned value can be: 
  - FLASH_BUSY
  - FLASH_ERROR_PROGRAM
  - FLASH_ERROR_WRP
  - FLASH_ERROR_OPERATION
  - FLASH_COMPLETE

### Notes

- This function can be used within all the device supply voltage ranges.

---

**12.2.8 Option bytes programming functions**

**12.2.8.1 FLASH_OB_Unlock**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void FLASH_OB_Unlock ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Unlocks the FLASH Option Control Registers access.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• None.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>

**12.2.8.2 FLASH_OB_Lock**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void FLASH_OB_Lock ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Locks the FLASH Option Control Registers access.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• None.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>
### 12.2.8.3 FLASH_OB_WRPConfig

**Function Name**  
void FLASH_OB_WRPConfig ( uint32_t OB_WRP, FunctionalState NewState)

**Function Description**  
Enables or disables the write protection of the desired sectors.

**Parameters**
- **OB_WRP**: specifies the sector(s) to be write protected or unprotected. This parameter can be one of the following values:
  - OB_WRP: A value between OB_WRP_Sector0 and OB_WRP_Sector11
  - OB_WRP_Sector_All
- **Newstate**: new state of the Write Protection. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.

**Notes**
- None.

### 12.2.8.4 FLASH_OB_RDPConfig

**Function Name**  
void FLASH_OB_RDPConfig ( uint8_t OB_RDP)

**Function Description**  
Sets the read protection level.

**Parameters**
- **OB_RDP**: specifies the read protection level. This parameter can be one of the following values:
  - OB_RDP_Level_0: No protection
  - OB_RDP_Level_1: Read protection of the memory
  - OB_RDP_Level_2: Full chip protection

**Return values**
- None.

**Notes**
- None.

### 12.2.8.5 FLASH_OB_UserConfig

**Function Name**  
void FLASH_OB_UserConfig ( uint8_t OB_IWDG, uint8_t OB_STOP, uint8_t OB_STDBY)

**Function Description**  
Programs the FLASH User Option Byte: IWDG_SW / RST_STOP
Parameters

- **OB_IWDG**: Selects the IWDG mode. This parameter can be one of the following values:
  - **OB_IWDG_SW**: Software IWDG selected
  - **OB_IWDG_HW**: Hardware IWDG selected

- **OB_STOP**: Reset event when entering STOP mode. This parameter can be one of the following values:
  - **OB_STOP_NoRST**: No reset generated when entering in STOP
  - **OB_STOP_RST**: Reset generated when entering in STOP

- **OB_STDBY**: Reset event when entering Standby mode. This parameter can be one of the following values:
  - **OB_STDBY_NoRST**: No reset generated when entering in STANDBY
  - **OB_STDBY_RST**: Reset generated when entering in STANDBY

Return values

- None.

Notes

- None.

### 12.2.8.6 FLASH_OB_BORConfig

**Function Name**

void FLASH_OB_BORConfig ( uint8_t OB_BOR)

**Function Description**

Sets the BOR Level.

**Parameters**

- **OB_BOR**: specifies the Option Bytes BOR Reset Level. This parameter can be one of the following values:
  - **OB_BOR_LEVEL3**: Supply voltage ranges from 2.7 to 3.6 V
  - **OB_BOR_LEVEL2**: Supply voltage ranges from 2.4 to 2.7 V
  - **OB_BOR_LEVEL1**: Supply voltage ranges from 2.1 to 2.4 V
  - **OB_BOR_OFF**: Supply voltage ranges from 1.62 to 2.1 V

**Return values**

- None.

**Notes**

- None.

### 12.2.8.7 FLASH_OB_Launch
Function Name: FLASH_Status FLASH_OB_Launch (void)
Function Description: Launch the option byte loading.
Parameters: None.
Return values: FLASH Status: The returned value can be: FLASH_BUSY, FLASH_ERROR_PROGRAM, FLASH_ERROR_WRP, FLASH_ERROR_OPERATION or FLASH_COMPLETE.
Notes: None.

12.2.8.8 FLASH_OB_GetUser

Function Name: uint8_t FLASH_OB_GetUser (void)
Function Description: Returns the FLASH User Option Bytes values.
Parameters: None.
Return values: The FLASH User Option Bytes values: IWDG_SW(Bit0), RST_STOP(Bit1) and RST_STDBY(Bit2).
Notes: None.

12.2.8.9 FLASH_OB_GetWRP

Function Name: uint16_t FLASH_OB_GetWRP (void)
Function Description: Returns the FLASH Write Protection Option Bytes value.
Parameters: None.
Return values: The FLASH Write Protection Option Bytes value
Notes: None.

12.2.8.10 FLASH_OB_GetRDP
Function Name: FlagStatus FLASH_OB_GetRDP (void)
Function Description: Returns the FLASH Read Protection level.
Parameters:
- None.
Return values:
- FLASH ReadOut Protection Status:
  - `SET`, when `OB_RDP_Level_1` or `OB_RDP_Level_2` is set
  - `RESET`, when `OB_RDP_Level_0` is set
Notes:
- None.

### 12.2.8.11 FLASH_OB_GetBOR

Function Name: uint8_t FLASH_OB_GetBOR (void)
Function Description: Returns the FLASH BOR level.
Parameters:
- None.
Return values:
- The FLASH BOR level:
  - `OB_BOR_LEVEL3`: Supply voltage ranges from 2.7 to 3.6 V
  - `OB_BOR_LEVEL2`: Supply voltage ranges from 2.4 to 2.7 V
  - `OB_BOR_LEVEL1`: Supply voltage ranges from 2.1 to 2.4 V
  - `OB_BOR_OFF`: Supply voltage ranges from 1.62 to 2.1 V
Notes:
- None.

### 12.2.9 Interrupt and flag management functions

#### 12.2.9.1 FLASH_ITConfig

Function Name: void FLASH_ITConfig (uint32_t FLASH_IT, FunctionalState NewState)
Function Description: Enables or disables the specified FLASH interrupts.
Parameters:
- `FLASH_IT`: specifies the FLASH interrupt sources to be enabled or disabled. This parameter can be any combination
of the following values:
- `FLASH_IT_ERR` : FLASH Error Interrupt
- `FLASH_IT_EOP` : FLASH end of operation Interrupt

Return values
- None.

Notes
- None.

### 12.2.9.2 FLASH_GetFlagStatus

**Function Name**
FlagStatus FLASH_GetFlagStatus ( uint32_t FLASH_FLAG)

**Function Description**
Checks whether the specified FLASH flag is set or not.

**Parameters**
- **FLASH_FLAG** : specifies the FLASH flag to check. This parameter can be one of the following values:
  - `FLASH_FLAG_EOP` : FLASH End of Operation flag
  - `FLASH_FLAG_OPERR` : FLASH operation Error flag
  - `FLASH_FLAG_WRPERR` : FLASH Write protected error flag
  - `FLASH_FLAG_PGAERR` : FLASH Programming Alignment error flag
  - `FLASH_FLAG_PGPERR` : FLASH Programming Parallelism error flag
  - `FLASH_FLAG_PGSERR` : FLASH Programming Sequence error flag
  - `FLASH_FLAG_BSY` : FLASH Busy flag

**Return values**
- The new state of FLASH_FLAG (SET or RESET).

**Notes**
- None.

### 12.2.9.3 FLASH_ClearFlag

**Function Name**
void FLASH_ClearFlag ( uint32_t FLASH_FLAG)

**Function Description**
Clears the FLASH's pending flags.

**Parameters**
- **FLASH_FLAG** : specifies the FLASH flags to clear. This parameter can be any combination of the following values:
  - `FLASH_FLAG_EOP` : FLASH End of Operation flag
  - `FLASH_FLAG_OPERR` : FLASH operation Error flag
  - `FLASH_FLAG_WRPERR` : FLASH Write protected error flag
- **FLASH_FLAG_PGAERR**: FLASH Programming Alignment error flag
- **FLASH_FLAG_PGPERR**: FLASH Programming Parallelism error flag
- **FLASH_FLAG_PGSERR**: FLASH Programming Sequence error flag

### Return values
- None.

### Notes
- None.

#### 12.2.9.4 FLASH_GetStatus

<table>
<thead>
<tr>
<th>Function Name</th>
<th>FLASH_Status FLASH_GetStatus ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Returns the FLASH Status.</td>
</tr>
<tr>
<td>Parameters</td>
<td>None.</td>
</tr>
<tr>
<td>Return values</td>
<td><strong>FLASH Status</strong>: The returned value can be: FLASH_BUSY, FLASH_ERROR_PROGRAM, FLASH_ERROR_WRP, FLASH_ERROR_OPERATION or FLASH_COMPLETE.</td>
</tr>
<tr>
<td>Notes</td>
<td>None.</td>
</tr>
</tbody>
</table>

#### 12.2.9.5 FLASH_WaitForLastOperation

<table>
<thead>
<tr>
<th>Function Name</th>
<th>FLASH_Status FLASH_WaitForLastOperation ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Waits for a FLASH operation to complete.</td>
</tr>
<tr>
<td>Parameters</td>
<td>None.</td>
</tr>
<tr>
<td>Return values</td>
<td><strong>FLASH Status</strong>: The returned value can be: FLASH_BUSY, FLASH_ERROR_PROGRAM, FLASH_ERROR_WRP, FLASH_ERROR_OPERATION or FLASH_COMPLETE.</td>
</tr>
<tr>
<td>Notes</td>
<td>None.</td>
</tr>
</tbody>
</table>
12.3 FLASH Firmware driver defines

12.3.1 FLASH Firmware driver defines

FLASH

FLASH_BOR_Reset_Level

- #define: OB_BOR_LEVEL3((uint8_t)0x00)
  Supply voltage ranges from 2.70 to 3.60 V

- #define: OB_BOR_LEVEL2((uint8_t)0x04)
  Supply voltage ranges from 2.40 to 2.70 V

- #define: OB_BOR_LEVEL1((uint8_t)0x08)
  Supply voltage ranges from 2.10 to 2.40 V

- #define: OB_BOR_OFF((uint8_t)0x0C)
  Supply voltage ranges from 1.62 to 2.10 V

FLASH_Exported_Constants

- #define: ACR_BYTE0_ADDRESS((uint32_t)0x40023C00)

- #define: OPTCR_BYTE0_ADDRESS((uint32_t)0x40023C14)

- #define: OPTCR_BYTE1_ADDRESS((uint32_t)0x40023C15)

- #define: OPTCR_BYTE2_ADDRESS((uint32_t)0x40023C16)

FLASH_Flags

- #define: FLASH_FLAG_EOP((uint32_t)0x00000001)
  FLASH End of Operation flag

- #define: FLASH_FLAG_OPERR((uint32_t)0x00000002)
  FLASH operation Error flag
FLASH Write protected error flag

FLASH Programming Alignment error flag

FLASH Programming Parallelism error flag

FLASH Programming Sequence error flag

FLASH Busy flag

End of FLASH Operation Interrupt source

Error Interrupt source

FLASH Keys

RDP_KEY(uint16_t)0x00A5

FLASH_KEY1(uint32_t)0x45670123

FLASH_KEY2(uint32_t)0xCDEF89AB

FLASH_OPT_KEY1(uint32_t)0x08192A3B

FLASH_OPT_KEY2(uint32_t)0x4C5D6E7F
Flash_Latency

- #define: FLASH_Latency_0((uint8_t)0x0000)  
  FLASH Zero Latency cycle

- #define: FLASH_Latency_1((uint8_t)0x0001)  
  FLASH One Latency cycle

- #define: FLASH_Latency_2((uint8_t)0x0002)  
  FLASH Two Latency cycles

- #define: FLASH_Latency_3((uint8_t)0x0003)  
  FLASH Three Latency cycles

- #define: FLASH_Latency_4((uint8_t)0x0004)  
  FLASH Four Latency cycles

- #define: FLASH_Latency_5((uint8_t)0x0005)  
  FLASH Five Latency cycles

- #define: FLASH_Latency_6((uint8_t)0x0006)  
  FLASH Six Latency cycles

- #define: FLASH_Latency_7((uint8_t)0x0007)  
  FLASH Seven Latency cycles

FLASH_Option_Bytes_IWatchdog

- #define: OB_IWDG_SW((uint8_t)0x20)  
  Software IWDG selected

- #define: OB_IWDG_HW((uint8_t)0x00)  
  Hardware IWDG selected

FLASH_Option_Bytes_nRST_STDBY

- #define: OB_STDBY_NoRST((uint8_t)0x80)
No reset generated when entering in STANDBY

- #define: OB_STDBY_RST((uint8_t)0x00)
  Reset generated when entering in STANDBY

**FLASH_Option_Beys_nRST_STOP**
- #define: OB_STOP_NoRST((uint8_t)0x40)
  No reset generated when entering in STOP

- #define: OB_STOP_RST((uint8_t)0x00)
  Reset generated when entering in STOP

**FLASH_Option_Beys_Read_Protection**
- #define: OB_RDP_Level_0((uint8_t)0xAA)

- #define: OB_RDP_Level_1((uint8_t)0x55)

**FLASH_Program_Parallelism**
- #define: FLASH_PSIZE_BYTE((uint32_t)0x00000000)

- #define: FLASH_PSIZE_HALF_WORD((uint32_t)0x00000100)

- #define: FLASH_PSIZE_WORD((uint32_t)0x00000200)

- #define: FLASH_PSIZE_DOUBLE_WORD((uint32_t)0x00000300)

- #define: CR_PSIZE_MASK((uint32_t)0xFFFFFCFF)

**FLASH_Sectors**
• #define: \texttt{FLASH\_Sector\_0((uint16\_t)0x0000)}
  Sector Number 0

• #define: \texttt{FLASH\_Sector\_1((uint16\_t)0x0008)}
  Sector Number 1

• #define: \texttt{FLASH\_Sector\_2((uint16\_t)0x0010)}
  Sector Number 2

• #define: \texttt{FLASH\_Sector\_3((uint16\_t)0x0018)}
  Sector Number 3

• #define: \texttt{FLASH\_Sector\_4((uint16\_t)0x0020)}
  Sector Number 4

• #define: \texttt{FLASH\_Sector\_5((uint16\_t)0x0028)}
  Sector Number 5

• #define: \texttt{FLASH\_Sector\_6((uint16\_t)0x0030)}
  Sector Number 6

• #define: \texttt{FLASH\_Sector\_7((uint16\_t)0x0038)}
  Sector Number 7

• #define: \texttt{FLASH\_Sector\_8((uint16\_t)0x0040)}
  Sector Number 8

• #define: \texttt{FLASH\_Sector\_9((uint16\_t)0x0048)}
  Sector Number 9

• #define: \texttt{FLASH\_Sector\_10((uint16\_t)0x0050)}
  Sector Number 10

• #define: \texttt{FLASH\_Sector\_11((uint16\_t)0x0058)}
  Sector Number 11
**FLASH_Voltage_Range**

- **#define**: `VoltageRange_1((uint8_t)0x00)`
  Device operating range: 1.8V to 2.1V

- **#define**: `VoltageRange_2((uint8_t)0x01)`
  Device operating range: 2.1V to 2.7V

- **#define**: `VoltageRange_3((uint8_t)0x02)`
  Device operating range: 2.7V to 3.6V

- **#define**: `VoltageRange_4((uint8_t)0x03)`
  Device operating range: 2.7V to 3.6V + External Vpp

### 12.4 FLASH Programming Example

The example below explains how to program the FLASH. For more examples about FLASH configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project/STM32F2xx_SnapPeriph_Examples/FLASH:

```c
/* Includes ---------------------------------------------------*/
#include "stm32f2xx.h"

/* Private define ---------------------------------------------*/
#define BUFFER_SIZE 10

/* Private variables ------------------------------------------*/
uint32_t pBuffer[BUFFER_SIZE]={0x20212223,0x24252627,
                              0x28292a2b,0x2c2d2e2f,
                              0xFFEEDDCC,0xBBAA9988,
                              0x77665544,0x33221100,
                              0x10000000,0x20000000};

/* Private function prototypes -------------------------------*/
uint32_t FLASH_If_Write(uint32_t* FlashAddress, uint32_t* Data, uint16_t DataLength);

/* Private functions ------------------------------------------*/

/**
 * @brief  Main program
 * @param  None
 * @retval None
 */
int main(void)
{
    uint32_t WriteAddress = 0x08008000; //Base address of Sector 2
```
/* Unlock the Flash to enable the flash control register access */
FLASH_Unlock();

/* Device voltage range supposed to be [2.7V to 3.6V], the operation will be done by word */

/* Erase Sector 2 */
if (FLASH_EraseSector(FLASH_Sector_2, VoltageRange_3) == FLASH_COMPLETE)
{
    /* Write pBuffer content to Sector 2 */
    if (FLASH_If_Write(&WriteAddress, pBuffer, BUFFER_SIZE) != 0)
    {
        /* Error occurred while Flash write.
         * User can add here some code to deal with this error */
    }
    else
    {
        /* Error occurred while Flash write.
         * User can add here some code to deal with this error */
    }

    /* Lock the Flash to disable the flash control register access
     * (recommended to protect the FLASH memory against possible unwanted operation) */
    FLASH_Lock();
    while (1)
    {
    }
}

/**
 * @brief This function writes a data buffer in flash
 * (data are 32-bit aligned).
 * @note After writing data buffer, the flash content is checked.
 * @param FlashAddress: start address for writing data buffer
 * @param Data: pointer on data buffer
 * @param DataLength: length of data buffer(unit is 32-bit word)
 * @retval 0: Data successfully written to Flash memory
 * 1: Error occurred while writing data in Flash memory
 * 2: Written Data in flash memory is different from expected one
 */
uint32_t FLASH_If_Write(uint32_t* FlashAddress, uint32_t* Data, uint16_t DataLength)
{
    uint32_t i = 0;
    for (i = 0; i < DataLength; i++)
    {
/ Device voltage range supposed to be [2.7V to 3.6V], the operation will be done by word */
if (FLASH_ProgramWord(*FlashAddress, *(uint32_t*)(Data+i)) == FLASH_COMPLETE)
{
    /* Check the written value */
    if (*((uint32_t)*FlashAddress) != *(uint32_t)(Data+i))
    {
        /* Flash content doesn't match SRAM content */
        return(2);
    }
    /* Increment FLASH destination address */
    *FlashAddress += 4;
}
else
{
    /* Error occurred while writing data in Flash memory */
    return (1);
}
return (0);
13 Flexible static memory controller (FSMC)

13.1 FSMC Firmware driver registers structures

13.1.1 FSMC_Bank1_TypeDef

*FSMC_Bank1_TypeDef* is defined in the stm32f2xx.h file and contains the FSMC Bank1 configuration registers definition.

Data Fields

- __IO uint32_t BTCR

Field Documentation

- __IO uint32_t FSMC_Bank1_TypeDef::BTCR[8]
  - NOR/PSRAM chip-select control register (BCR) and chip-select timing register (BTR), Address offset: 0x00-1C

13.1.2 FSMC_Bank1E_TypeDef

*FSMC_Bank1E_TypeDef* is defined in the stm32f2xx.h file and contains the FSMC Bank1 write timing registers definition.

Data Fields

- __IO uint32_t BWTR

Field Documentation

- __IO uint32_t FSMC_Bank1E_TypeDef::BWTR[7]
  - NOR/PSRAM write timing registers, Address offset: 0x104-0x11C

13.1.3 FSMC_Bank2_TypeDef

*FSMC_Bank2_TypeDef* is defined in the stm32f2xx.h file and contains the FSMC Bank2 configuration registers definition.

Data Fields

- __IO uint32_t PCR2
- __IO uint32_t SR2
- __IO uint32_t PMEM2
- __IO uint32_t PATT2
- uint32_t RESERVED0
Field Documentation

- **__IO uint32_t ECCR2**

Field Documentation

- **__IO uint32_t FSMC_Bank2_TypeDef::PCR2**
  - NAND Flash control register 2, Address offset: 0x60
- **__IO uint32_t FSMC_Bank2_TypeDef::SR2**
  - NAND Flash FIFO status and interrupt register 2, Address offset: 0x64
- **__IO uint32_t FSMC_Bank2_TypeDef::PMEM2**
  - NAND Flash Common memory space timing register 2, Address offset: 0x68
- **__IO uint32_t FSMC_Bank2_TypeDef::PATT2**
  - NAND Flash Attribute memory space timing register 2, Address offset: 0x6C
- **uint32_t FSMC_Bank2_TypeDef::RESERVED0**
  - Reserved, 0x70
- **__IO uint32_t FSMC_Bank2_TypeDef::ECCR2**
  - NAND Flash ECC result registers 2, Address offset: 0x74

13.1.4 FSMC_Bank3_TypeDef

`FSMC_Bank3_TypeDef` is defined in the stm32f2xx.h file and contains the FSMC Bank3 configuration registers definition.

Data Fields

- **__IO uint32_t PCR3**
- **__IO uint32_t SR3**
- **__IO uint32_t PMEM3**
- **__IO uint32_t PATT3**
- **uint32_t RESERVED0**
- **__IO uint32_t ECCR3**

Field Documentation

- **__IO uint32_t FSMC_Bank3_TypeDef::PCR3**
  - NAND Flash control register 3, Address offset: 0x80
- **__IO uint32_t FSMC_Bank3_TypeDef::SR3**
  - NAND Flash FIFO status and interrupt register 3, Address offset: 0x84
- **__IO uint32_t FSMC_Bank3_TypeDef::PMEM3**
  - NAND Flash Common memory space timing register 3, Address offset: 0x88
- **__IO uint32_t FSMC_Bank3_TypeDef::PATT3**
  - NAND Flash Attribute memory space timing register 3, Address offset: 0x8C
- **uint32_t FSMC_Bank3_TypeDef::RESERVED0**
  - Reserved, 0x90
- **__IO uint32_t FSMC_Bank3_TypeDef::ECCR3**
  - NAND Flash ECC result registers 3, Address offset: 0x94
13.1.5   FSMC_Bank4_TypeDef

*FSMC_Bank4_TypeDef* is defined in the stm32f2xx.h file and contains the FSMC Bank4 configuration registers definition.

**Data Fields**

- __IO uint32_t PCR4
- __IO uint32_t SR4
- __IO uint32_t PMEM4
- __IO uint32_t PATT4
- __IO uint32_t PIO4

**Field Documentation**

- __IO uint32_t FSMC_Bank4_TypeDef::PCR4
  - PC Card control register 4, Address offset: 0xA0
- __IO uint32_t FSMC_Bank4_TypeDef::SR4
  - PC Card FIFO status and interrupt register 4, Address offset: 0xA4
- __IO uint32_t FSMC_Bank4_TypeDef::PMEM4
  - PC Card Common memory space timing register 4, Address offset: 0xA8
- __IO uint32_t FSMC_Bank4_TypeDef::PATT4
  - PC Card Attribute memory space timing register 4, Address offset: 0xAC
- __IO uint32_t FSMC_Bank4_TypeDef::PIO4
  - PC Card I/O space timing register 4, Address offset: 0xB0

13.1.6   FSMC_NORSRAMTimingInitTypeDef

*FSMC_NORSRAMTimingInitTypeDef* is defined in the stm32f2xx_fsmc.h file and contains the NOR/SRAM timing initialization parameters.

**Data Fields**

- uint32_t FSMC_AddressSetupTime
- uint32_t FSMC_AddressHoldTime
- uint32_t FSMC_DataSetupTime
- uint32_t FSMC_BusTurnAroundDuration
- uint32_t FSMC_CLKDivision
- uint32_t FSMC_DataLatency
- uint32_t FSMC_AccessMode

**Field Documentation**

- uint32_t FSMC_NORSRAMTimingInitTypeDef::FSMC_AddressSetupTime
  - Defines the number of HCLK cycles to configure the duration of the address setup time. This parameter can be a value between 0 and 0xF. This parameter is not used with synchronous NOR Flash memories.
- uint32_t FSMC_NORSRAMTimingInitTypeDef::FSMC_AddressHoldTime
Flexible static memory controller (FSMC)

- Defines the number of HCLK cycles to configure the duration of the address hold time. This parameter can be a value between 0 and 0xF. This parameter is not used with synchronous NOR Flash memories.

- **uint32_t FSMC_NORSRAMTimingInitTypeDef::FSMC_DataSetupTime**
  - Defines the number of HCLK cycles to configure the duration of the data setup time. This parameter can be a value between 0 and 0xFF. This parameter is used for SRAMs, ROMs and asynchronous multiplexed NOR Flash memories.

- **uint32_t FSMC_NORSRAMTimingInitTypeDef::FSMC_BusTurnAroundDuration**
  - Defines the number of HCLK cycles to configure the duration of the bus turnaround. This parameter can be a value between 0 and 0xF. This parameter is only used for multiplexed NOR Flash memories.

- **uint32_t FSMC_NORSRAMTimingInitTypeDef::FSMC_CLKDivision**
  - Defines the period of CLK clock output signal, expressed in number of HCLK cycles. This parameter can be a value between 1 and 0xF. This parameter is not used for asynchronous NOR Flash, SRAM or ROM accesses.

- **uint32_t FSMC_NORSRAMTimingInitTypeDef::FSMC_DataLatency**
  - Defines the number of memory clock cycles to issue to the memory before getting the first data. The parameter value depends on the memory type as shown below: It must be set to 0 in case of a CRAM. It is don't care in asynchronous NOR, SRAM or ROM accesses. It may assume a value between 0 and 0xF in NOR Flash memories with synchronous burst mode enable.

- **uint32_t FSMC_NORSRAMTimingInitTypeDef::FSMC_AccessMode**
  - Specifies the asynchronous access mode. This parameter can be a value of FSMC_Access_Mode.

### 13.1.7 FSMC_NORSRAMInitTypeDef

*FSMC_NORSRAMInitTypeDef* is defined in the stm32f2xx_fsmc.h file and contains the NOR/SRAM common initialization parameters.

#### Data Fields

- **uint32_t FSMC_Bank**
- **uint32_t FSMC_DataAddressMux**
- **uint32_t FSMC_MemoryType**
- **uint32_t FSMC_MemoryDataWidth**
- **uint32_t FSMC_BurstAccessMode**
- **uint32_t FSMC_AsynchronousWait**
- **uint32_t FSMC_WaitSignalPolarity**
- **uint32_t FSMC_WrapMode**
- **uint32_t FSMC_WaitSignalActive**
- **uint32_t FSMC_WriteOperation**
- **uint32_t FSMC_WaitSignal**
- **uint32_t FSMC_ExtendedMode**
- **uint32_t FSMC_WriteBurst**
- **FSMC_NORSRAMTimingInitTypeDef * FSMC_ReadWriteTimingStruct**
- **FSMC_NORSRAMTimingInitTypeDef * FSMC_WriteTimingStruct**

#### Field Documentation
- `uint32_t FSMC_NORSRAMInitTypeDef::FSMC_Bank`  
  Specifies the NOR/SRAM memory bank that will be used. This parameter can be a value of `FSMC_NORSRAM_Bank`

- `uint32_t FSMC_NORSRAMInitTypeDef::FSMC_DataAddressMux`  
  Specifies whether the address and data values are multiplexed on the databus or not. This parameter can be a value of `FSMC_Data_Address_Bus_Multiplexing`

- `uint32_t FSMC_NORSRAMInitTypeDef::FSMC_MemoryType`  
  Specifies the type of external memory attached to the corresponding memory bank. This parameter can be a value of `FSMC_Memory_Type`

- `uint32_t FSMC_NORSRAMInitTypeDef::FSMC_MemoryDataWidth`  
  Specifies the external memory device width. This parameter can be a value of `FSMC_Data_Width`

- `uint32_t FSMC_NORSRAMInitTypeDef::FSMC_BurstAccessMode`  
  Enables or disables the burst access mode for Flash memory, valid only with synchronous burst Flash memories. This parameter can be a value of `FSMC_Burst_Access_Mode`

- `uint32_t FSMC_NORSRAMInitTypeDef::FSMC_AsynchronousWait`  
  Enables or disables wait signal during asynchronous transfers, valid only with asynchronous Flash memories. This parameter can be a value of `FSMC_AsynchronousWait`

- `uint32_t FSMC_NORSRAMInitTypeDef::FSMC_WaitSignalPolarity`  
  Specifies the wait signal polarity, valid only when accessing the Flash memory in burst mode. This parameter can be a value of `FSMC_Wait_Signal_Polarity`

- `uint32_t FSMC_NORSRAMInitTypeDef::FSMC_WrapMode`  
  Enables or disables the wrapped burst access mode for Flash memory, valid only when accessing Flash memories in burst mode. This parameter can be a value of `FSMC_Wrap_Mode`

- `uint32_t FSMC_NORSRAMInitTypeDef::FSMC_WaitSignalActive`  
  Specifies if the wait signal is asserted by the memory one clock cycle before the wait state or during the wait state, valid only when accessing memories in burst mode. This parameter can be a value of `FSMC_Wait_Timing`

- `uint32_t FSMC_NORSRAMInitTypeDef::FSMC_WriteOperation`  
  Enables or disables the write operation in the selected bank by the FSMC. This parameter can be a value of `FSMC_Write_Operation`

- `uint32_t FSMC_NORSRAMInitTypeDef::FSMC_WriteBurst`  
  Enables or disables the write burst operation. This parameter can be a value of `FSMC_Write_Burst`

- `FSMC_NORSRAMTimingInitTypeDef* FSMC_NORSRAMInitTypeDef::FSMC_ReadWriteTimingStruct`  
  Timing Parameters for write and read access if the ExtendedMode is not used

- `FSMC_NORSRAMTimingInitTypeDef* FSMC_NORSRAMInitTypeDef::FSMC_WriteTimingStruct`  
  Timing Parameters for write access if the ExtendedMode is used
13.1.8 FSMC_NAND_PCCARDTimingInitTypeDef

`FSMC_NAND_PCCARDTimingInitTypeDef` is defined in the `stm32f2xx_fsmc.h` file and contains the NAND/PCCARD timing initialization parameters.

**Data Fields**

- `uint32_t FSMC_SetupTime`
- `uint32_t FSMC_WaitSetupTime`
- `uint32_t FSMC_HoldSetupTime`
- `uint32_t FSMC_HiZSetupTime`

**Field Documentation**

- `uint32_t FSMC_NAND_PCCARDTimingInitTypeDef::FSMC_SetupTime`
  - Defines the number of HCLK cycles to setup address before the command assertion for NAND-Flash read or write access to common/Attribute or I/O memory space (depending on the memory space timing to be configured). This parameter can be a value between 0 and 0xFF.
- `uint32_t FSMC_NAND_PCCARDTimingInitTypeDef::FSMC_WaitSetupTime`
  - Defines the minimum number of HCLK cycles to assert the command for NAND-Flash read or write access to common/Attribute or I/O memory space (depending on the memory space timing to be configured). This parameter can be a number between 0x00 and 0xFF.
- `uint32_t FSMC_NAND_PCCARDTimingInitTypeDef::FSMC_HoldSetupTime`
  - Defines the number of HCLK clock cycles to hold address (and data for write access) after the command deassertion for NAND-Flash read or write access to common/Attribute or I/O memory space (depending on the memory space timing to be configured). This parameter can be a number between 0x00 and 0xFF.
- `uint32_t FSMC_NAND_PCCARDTimingInitTypeDef::FSMC_HiZSetupTime`
  - Defines the number of HCLK clock cycles during which the databus is kept in HiZ after the start of a NAND-Flash write access to common/Attribute or I/O memory space (depending on the memory space timing to be configured). This parameter can be a number between 0x00 and 0xFF.

13.1.9 FSMC_NANDInitTypeDef

`FSMC_NANDInitTypeDef` is defined in the `stm32f2xx_fsmc.h` file and contains the NAND common initialization parameters.

**Data Fields**

- `uint32_t FSMC_Bank`
- `uint32_t FSMC_Waitfeature`
- `uint32_t FSMC_MemoryDataWidth`
- `uint32_t FSMC_ECC`
- `uint32_t FSMC_ECCPageSize`
- `uint32_t FSMC_TCLRSetupTime`
- `uint32_t FSMC_TARSetupTime`
- `FSMC_NAND_PCCARDTimingInitTypeDef * FSMC_CommonSpaceTimingStruct`
Flexible static memory controller (FSMC)

- `FSMC_NAND_PCCARDTimingInitTypeDef * FSMC_AttributeSpaceTimingStruct`

Field Documentation

- `uint32_t FSMC_NANDInitTypeDef::FSMC_Bank`
  - Specifies the NAND memory bank that will be used. This parameter can be a value of `FSMC_NAND_Bank`

- `uint32_t FSMC_NANDInitTypeDef::FSMC_Waitfeature`
  - Enables or disables the Wait feature for the NAND Memory Bank. This parameter can be any value of `FSMC_Wait_feature`

- `uint32_t FSMC_NANDInitTypeDef::FSMC_MemoryDataWidth`
  - Specifies the external memory device width. This parameter can be any value of `FSMC_Data_Width`

- `uint32_t FSMC_NANDInitTypeDef::FSMC_ECC`
  - Enables or disables the ECC computation. This parameter can be any value of `FSMC_ECC`

- `uint32_t FSMC_NANDInitTypeDef::FSMC_ECCPageSize`
  - Defines the page size for the extended ECC. This parameter can be any value of `FSMC_ECC_Page_Size`

- `uint32_t FSMC_NANDInitTypeDef::FSMC_TCLRSetupTime`
  - Defines the number of HCLK cycles to configure the delay between CLE low and RE low. This parameter can be a value between 0 and 0xFF.

- `uint32_t FSMC_NANDInitTypeDef::FSMC_TARSetupTime`
  - Defines the number of HCLK cycles to configure the delay between ALE low and RE low. This parameter can be a number between 0x0 and 0xFF

- `FSMC_NAND_PCCARDTimingInitTypeDef* FSMC_CommonSpaceTimingStruct`
  - FSMC Common Space Timing

- `FSMC_NAND_PCCARDTimingInitTypeDef* FSMC_AttributeSpaceTimingStruct`
  - FSMC Attribute Space Timing

13.1.10 `FSMC_PCCARDInitTypeDef`

`FSMC_PCCARDInitTypeDef` is defined in the stm32f2xx_fsmc.h file and contains the PCCARD common initialization parameters.

Data Fields

- `uint32_t FSMC_Waitfeature`
- `uint32_t FSMC_TCLRSetupTime`
- `uint32_t FSMC_TARSetupTime`
- `FSMC_NAND_PCCARDTimingInitTypeDef * FSMC_CommonSpaceTimingStruct`
- `FSMC_NAND_PCCARDTimingInitTypeDef * FSMC_AttributeSpaceTimingStruct`
- `FSMC_NAND_PCCARDTimingInitTypeDef * FSMC_IOSpaceTimingStruct`

Field Documentation
13.2 FSMC Firmware driver API description

The following section lists the various functions of the FSMC library.

13.2.1 NOR/SRAM controller

The following sequence should be followed to configure the FSMC to interface with SRAM, PSRAM, NOR or OneNAND memory connected to the NOR/SRAM Bank:

1. Enable the clock for the FSMC and associated GPIOs using the following functions:
   RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC, ENABLE);
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOx, ENABLE);

2. FSMC pins configuration
   - Connect the involved FSMC pins to AF12 using the following function
     GPIO_PinAFConfig(GPIOx, GPIO_PinSourcex, GPIO_AF_FSMC);
   - Configure these FSMC pins in alternate function mode by calling the function
     GPIO_Init();

3. Declare a FSMC_NORSRAMInitStruct structure, for example:
   FSMC_NORSRAMInitStruct FSMC_NORSRAMInitStructure; and fill the
   FSMC_NORSRAMInitStruct variable with the allowed values of the structure
   member.

4. Initialize the NOR/SRAM Controller by calling the function
   FSMC_NORSRAMInit(&FSMC_NORSRAMInitStruct);

5. Then enable the NOR/SRAM Bank, for example:
   FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM2, ENABLE);

6. At this stage you can read/write from/to the memory connected to the NOR/SRAM
   Bank.

The NOR/SRAM Controller functions are the following:

- `FSMC_NORSRAMDelInit()`
- `FSMC_NORSRAMInit()`
- `FSMC_NORSRAMStructInit()`
- `FSMC_NORSRAMCmd()`
### 13.2.2 NAND controller

The following sequence should be followed to configure the FSMC to interface with 8-bit or 16-bit NAND memory connected to the NAND Bank:

1. Enable the clock for the FSMC and associated GPIOs using the following functions:
   - RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC, ENABLE);
   - RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOx, ENABLE);
2. FSMC pins configuration
   - Connect the involved FSMC pins to AF12 using the following function
     GPIO_PinAFConfig(GPIOx, GPIO_PinSourcex, GPIO_AF_FSMC);
   - Configure these FSMC pins in alternate function mode by calling the function
     GPIO_Init();
3. Declare a FSMC_NANDInitTypeDef structure, for example: FSMC_NANDInitTypeDef FSMC_NANDInitStructure; and fill the FSMC_NANDInitStructure variable with the allowed values of the structure member.
4. Initialize the NAND Controller by calling the function
   FSMC_NANDInit(&FSMC_NANDInitStructure);
5. Then enable the NAND Bank, for example: FSMC_NANDCmd(FSMC_Bank3_NAND, ENABLE);
6. At this stage you can read/write from/to the memory connected to the NAND Bank.

To enable the Error Correction Code (ECC), use the function
FSMC_NANDECCCmd(FSMC_Bank3_NAND, ENABLE); and to get the current ECC value
use the function ECCval = FSMC_GetECC(FSMC_Bank3_NAND);

NAND Controller functions are the following:

- `FSMC_NANDEnInit()`
- `FSMC_NANDInit()`
- `FSMC_NANDStructInit()`
- `FSMC_NANDCmd()`
- `FSMC_NANDECCCmd()`
- `FSMC_GetECC()`

### 13.2.3 PCCARD controller

The following sequence should be followed to configure the FSMC to interface with 16-bit PC Card compatible memory connected to the PCCARD Bank:

1. Enable the clock for the FSMC and associated GPIOs using the following functions:
   - RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC, ENABLE);
   - RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOx, ENABLE);
2. FSMC pins configuration
   - Connect the involved FSMC pins to AF12 using the following function
     GPIO_PinAFConfig(GPIOx, GPIO_PinSourcex, GPIO_AF_FSMC);
   - Configure these FSMC pins in alternate function mode by calling the function
     GPIO_Init();
3. Declare a FSMC_PCCARDInitTypeDef structure, for example:
   FSMC_PCCARDInitTypeDef FSMC_PCCARDInitStructure; and fill the FSMC_PCCARDInitStructure variable with the allowed values of the structure member.
4. Initialize the PCCARD Controller by calling the function
   FSMC_PCCARDInit(&FSMC_PCCARDInitStructure);
5. Then enable the PCCARD Bank: FSMC_PCCARDCmd(ENABLE);
6. At this stage you can read/write from/to the memory connected to the PCCARD Bank.

The PCCARD Controller functions are:

- FSMC_PCCARDDeInit()
- FSMC_PCCARDInit()
- FSMC_PCCARDStructInit()
- FSMC_PCCARDCmd()

13.2.4 Interrupt and flag management

- FSMC_ITConfig()
- FSMC_GetFlagStatus()
- FSMC_ClearFlag()
- FSMC_GetITStatus()
- FSMC_ClearITPendingBit()

13.2.5 NOR_SRAM controller functions

13.2.5.1 FSMC_NORSRAMDeInit

Function Name: void FSMC_NORSRAMDeInit ( uint32_t FSMC_Bank)
Function Description: Deinitializes the FSMC NOR/SRAM Banks registers to their
default reset values.
Parameters:
- FSMC_Bank: specifies the FSMC Bank to be used This parameter can be one of the following values:
  - FSMC_Bank1_NORSRAM1: FSMC Bank1 NOR/SRAM1
  - FSMC_Bank1_NORSRAM2: FSMC Bank1 NOR/SRAM2
  - FSMC_Bank1_NORSRAM3: FSMC Bank1 NOR/SRAM3
  - FSMC_Bank1_NORSRAM4: FSMC Bank1 NOR/SRAM4

Return values:
- None.

Notes:
- None.

13.2.5.2 FSMC_NORSRAMInit
Function Name: `void FSMC_NORSRAMInit (FSMC_NORSRAMInitTypeDef *FSMC_NORSRAMInitStruct)`

Function Description: Initializes the FSMC NOR/SRAM Banks according to the specified parameters in the FSMC_NORSRAMInitStruct.

Parameters:
- `FSMC_NORSRAMInitStruct`: pointer to a FSMC_NORSRAMInitTypeDef structure that contains the configuration information for the FSMC NOR/SRAM specified Banks.

Return values:
- None.

Notes:
- None.

---

### 13.2.5.3 FSMC_NORSRAMStructInit

Function Name: `void FSMC_NORSRAMStructInit (FSMC_NORSRAMInitStruct)`

Function Description: Fills each FSMC_NORSRAMInitStruct member with its default value.

Parameters:
- `FSMC_NORSRAMInitStruct`: pointer to a FSMC_NORSRAMInitStruct structure which will be initialized.

Return values:
- None.

Notes:
- None.

---

### 13.2.5.4 FSMC_NORSRAMCmd

Function Name: `void FSMC_NORSRAMCmd (uint32_t FSMC_Bank, FunctionalState NewState)`

Function Description: Enables or disables the specified NOR/SRAM Memory Bank.

Parameters:
- `FSMC_Bank`: specifies the FSMC Bank to be used. This parameter can be one of the following values:
  - `FSMC_Bank1_NORSRAM1`: FSMC Bank1 NOR/SRAM1
  - `FSMC_Bank1_NORSRAM2`: FSMC Bank1 NOR/SRAM2
  - `FSMC_Bank1_NORSRAM3`: FSMC Bank1 NOR/SRAM3
13.2.6 NAND controller functions

13.2.6.1 FSMC_NANDDeInit

Function Name: void FSMC_NANDDeInit ( uint32_t FSMC_Bank)
Function Description: Deinitializes the FSMC NAND Banks registers to their default reset values.
Parameters:
- FSMC_Bank: specifies the FSMC Bank to be used. This parameter can be one of the following values:
  - FSMC_Bank2_NAND: FSMC Bank2 NAND
  - FSMC_Bank3_NAND: FSMC Bank3 NAND

Return values:
- None.

Notes:
- None.

13.2.6.2 FSMC_NANDInit

Function Name: void FSMC_NANDInit ( FSMC_NANDInitTypeDef * FSMC_NANDInitStruct)
Function Description: Initializes the FSMC NAND Banks according to the specified parameters in the FSMC_NANDInitStruct.
Parameters:
- FSMC_NANDInitStruct: pointer to a FSMC_NANDInitTypeDef structure that contains the configuration information for the FSMC NAND specified Banks.

Return values:
- None.

Notes:
- None.
13.2.6.3  **FSMC_NANDStructInit**

**Function Name**  
void FSMC_NANDStructInit ( FSMC_NANDInitTypeDef * FSMC_NANDInitStruct)

**Function Description**  
Fills each FSMC_NANDInitStruct member with its default value.

**Parameters**  
- **FSMC_NANDInitStruct** : pointer to a FSMC_NANDInitTypeDef structure which will be initialized.

**Return values**  
- None.

**Notes**  
- None.

13.2.6.4  **FSMC_NANDCmd**

**Function Name**  
void FSMC_NANDCmd ( uint32_t FSMC_Bank, FunctionalState NewState)

**Function Description**  
Enables or disables the specified NAND Memory Bank.

**Parameters**  
- **FSMC_Bank** : specifies the FSMC Bank to be used This parameter can be one of the following values:
  - **FSMC_Bank2_NAND** : FSMC Bank2 NAND
  - **FSMC_Bank3_NAND** : FSMC Bank3 NAND
- **NewState** : new state of the FSMC_Bank. This parameter can be: ENABLE or DISABLE.

**Return values**  
- None.

**Notes**  
- None.

13.2.6.5  **FSMC_NANDECCCmd**

**Function Name**  
void FSMC_NANDECCCmd ( uint32_t FSMC_Bank, FunctionalState NewState)

**Function Description**  
Enables or disables the FSMC NAND ECC feature.
Parameters • **FSMC_Bank** : specifies the FSMC Bank to be used This parameter can be one of the following values:
  - **FSMC_Bank2_NAND** : FSMC Bank2 NAND
  - **FSMC_Bank3_NAND** : FSMC Bank3 NAND
• **NewState** : new state of the FSMC NAND ECC feature. This parameter can be: ENABLE or DISABLE.

Return values • None.
Notes • None.

### 13.2.6.6 FSMC_GetECC

**Function Name** `uint32_t FSMC_GetECC ( uint32_t FSMC_Bank)`

**Function Description** Returns the error correction code register value.

**Parameters** • **FSMC_Bank** : specifies the FSMC Bank to be used This parameter can be one of the following values:
  - **FSMC_Bank2_NAND** : FSMC Bank2 NAND
  - **FSMC_Bank3_NAND** : FSMC Bank3 NAND

**Return values** • The Error Correction Code (ECC) value.

**Notes** • None.

### 13.2.7 PCCARD controller functions

#### 13.2.7.1 FSMC_PCCARDDelInit

**Function Name** `void FSMC_PCCARDDelInit ( void )`

**Function Description** Deinitializes the FSMC PCCARD Bank registers to their default reset values.

**Parameters** • None.
**Return values** • None.
**Notes** • None.
## 13.2.7.2 FSMC_PCCARDInit

**Function Name**

void FSMC_PCCARDInit (FSMC_PCCARDInitTypeDef *FSMC_PCCARDInitStruct)

**Function Description**

Initializes the FSMC PCCARD Bank according to the specified parameters in the FSMC_PCCARDInitStruct.

**Parameters**

- **FSMC_PCCARDInitStruct**: pointer to a FSMC_PCCARDInitTypeDef structure that contains the configuration information for the FSMC PCCARD Bank.

**Return values**

- None.

**Notes**

- None.

## 13.2.7.3 FSMC_PCCARDStructInit

**Function Name**

void FSMC_PCCARDStructInit (FSMC_PCCARDInitStructTypeDef *FSMC_PCCARDInitStruct)

**Function Description**

Fills each FSMC_PCCARDInitStruct member with its default value.

**Parameters**

- **FSMC_PCCARDInitStruct**: pointer to a FSMC_PCCARDInitStructTypeDef structure which will be initialized.

**Return values**

- None.

**Notes**

- None.

## 13.2.7.4 FSMC_PCCARDCmd

**Function Name**

void FSMC_PCCARDCmd (FunctionalState NewState)

**Function Description**

Enables or disables the PCCARD Memory Bank.

**Parameters**

- **NewState**: new state of the PCCARD Memory Bank. This parameter can be: ENABLE or DISABLE.

**Return values**

- None.

**Notes**

- None.
13.2.8 Interrupt and flag management functions

13.2.8.1 FSMC_ITConfig

Function Name: void FSMC_ITConfig ( uint32_t FSMC_Bank, uint32_t FSMC_IT, FunctionalState NewState)

Function Description: Enables or disables the specified FSMC interrupts.

Parameters:
- **FSMC_Bank**: specifies the FSMC Bank to be used. This parameter can be one of the following values:
  - `FSMC_Bank2_NAND`: FSMC Bank2 NAND
  - `FSMC_Bank3_NAND`: FSMC Bank3 NAND
  - `FSMC_Bank4_PCCARD`: FSMC Bank4 PCCARD
- **FSMC_IT**: specifies the FSMC interrupt sources to be enabled or disabled. This parameter can be any combination of the following values:
  - `FSMC_IT_RisingEdge`: Rising edge detection interrupt.
  - `FSMC_IT_Level`: Level edge detection interrupt.
  - `FSMC_IT_FallingEdge`: Falling edge detection interrupt.
- **NewState**: new state of the specified FSMC interrupts. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

13.2.8.2 FSMC_GetFlagStatus

Function Name: FlagStatus FSMC_GetFlagStatus ( uint32_t FSMC_Bank, uint32_t FSMC_FLAG)

Function Description: Checks whether the specified FSMC flag is set or not.

Parameters:
- **FSMC_Bank**: specifies the FSMC Bank to be used. This parameter can be one of the following values:
  - `FSMC_Bank2_NAND`: FSMC Bank2 NAND
  - `FSMC_Bank3_NAND`: FSMC Bank3 NAND
  - `FSMC_Bank4_PCCARD`: FSMC Bank4 PCCARD
- **FSMC_FLAG**: specifies the flag to check. This parameter can be one of the following values:
### 13.2.8.3 FSMC_ClearFlag

**Function Name**  
`void FSMC_ClearFlag ( uint32_t FSMC_Bank, uint32_t FSMC_FLAG)`

**Function Description**  
Clears the FSMC’s pending flags.

**Parameters**
- `FSMC_Bank` : specifies the FSMC Bank to be used. This parameter can be one of the following values:
  - `FSMC_Bank2_NAND` : FSMC Bank2 NAND
  - `FSMC_Bank3_NAND` : FSMC Bank3 NAND
  - `FSMC_Bank4_PCCARD` : FSMC Bank4 PCCARD
- `FSMC_FLAG` : specifies the flag to clear. This parameter can be any combination of the following values:
  - `FSMC_FLAG_RisingEdge` : Rising edge detection Flag.
  - `FSMC_FLAG_Level` : Level detection Flag.
  - `FSMC_FLAG_FallingEdge` : Falling edge detection Flag.
  - `FSMC_FLAG_FEMPT` : FIFO empty Flag.

**Return values**
- None.

**Notes**
- None.

### 13.2.8.4 FSMC_GetITStatus

**Function Name**  
`ITStatus FSMC_GetITStatus ( uint32_t FSMC_Bank, uint32_t FSMC_IT)`

**Function Description**  
Checks whether the specified FSMC interrupt has occurred or not.

**Parameters**
- `FSMC_Bank` : specifies the FSMC Bank to be used. This parameter can be one of the following values:
- **FSMC_Bank2_NAND**: FSMC Bank2 NAND
- **FSMC_Bank3_NAND**: FSMC Bank3 NAND
- **FSMC_Bank4_PCCARD**: FSMC Bank4 PCCARD

- **FSMC_IT**: specifies the FSMC interrupt source to check. This parameter can be one of the following values:
  - **FSMC_IT_RisingEdge**: Rising edge detection interrupt.
  - **FSMC_IT_Level**: Level edge detection interrupt.
  - **FSMC_IT_FallingEdge**: Falling edge detection interrupt.

**Return values**: The new state of FSMC_IT (SET or RESET).

**Notes**: None.

### 13.2.8.5 FSMC_ClearITPendingBit

**Function Name**: void FSMC_ClearITPendingBit ( uint32_t FSMC_Bank, uint32_t FSMC_IT)

**Function Description**: Clears the FSMC's interrupt pending bits.

**Parameters**:
- **FSMC_Bank**: specifies the FSMC Bank to be used. This parameter can be one of the following values:
  - **FSMC_Bank2_NAND**: FSMC Bank2 NAND
  - **FSMC_Bank3_NAND**: FSMC Bank3 NAND
  - **FSMC_Bank4_PCCARD**: FSMC Bank4 PCCARD
- **FSMC_IT**: specifies the interrupt pending bit to clear. This parameter can be any combination of the following values:
  - **FSMC_IT_RisingEdge**: Rising edge detection interrupt.
  - **FSMC_IT_Level**: Level edge detection interrupt.
  - **FSMC_IT_FallingEdge**: Falling edge detection interrupt.

**Return values**: None.

**Notes**: None.

### 13.3 FSMC Firmware driver defines

#### 13.3.1 FSMC Firmware driver defines

`FSMC`  

`FSMC_Access_Mode`
- `#define: FSMC_AccessMode_A((uint32_t)0x00000000)`

- `#define: FSMC_AccessMode_B((uint32_t)0x10000000)`

- `#define: FSMC_AccessMode_C((uint32_t)0x20000000)`

- `#define: FSMC_AccessMode_D((uint32_t)0x30000000)`

`FSMC_AsynchronousWait`  
- `#define: FSMC_AsynchronousWait_Disable((uint32_t)0x00000000)`

- `#define: FSMC_AsynchronousWait_Enable((uint32_t)0x00008000)`

`FSMC_Burst_Access_Mode`  
- `#define: FSMC_BurstAccessMode_Disable((uint32_t)0x00000000)`

- `#define: FSMC_BurstAccessMode_Enable((uint32_t)0x00000100)`

`FSMC_Data_Address_Bus_Multiplexing`  
- `#define: FSMC_DataAddressMux_Disable((uint32_t)0x00000000)`

- `#define: FSMC_DataAddressMux_Enable((uint32_t)0x00000002)`

`FSMC_Data_Width`  
- `#define: FSMC_MemoryDataWidth_8b((uint32_t)0x00000000)`
- **#define:** `FSMC_MemoryDataWidth_16b((uint32_t)0x00000010)`

**FSMC_ECC**
- **#define:** `FSMC_ECC_Disable((uint32_t)0x00000000)`

- **#define:** `FSMC_ECC_Enable((uint32_t)0x00000040)`

**FSMC_ECC_Page_Size**
- **#define:** `FSMC_ECCPageSize_256Bytes((uint32_t)0x00000000)`

- **#define:** `FSMC_ECCPageSize_512Bytes((uint32_t)0x00020000)`

- **#define:** `FSMC_ECCPageSize_1024Bytes((uint32_t)0x00040000)`

- **#define:** `FSMC_ECCPageSize_2048Bytes((uint32_t)0x00060000)`

- **#define:** `FSMC_ECCPageSize_4096Bytes((uint32_t)0x00080000)`

- **#define:** `FSMC_ECCPageSize_8192Bytes((uint32_t)0x000A0000)`

**FSMC_Extended_Mode**
- **#define:** `FSMC_ExtendedMode_Disable((uint32_t)0x00000000)`

- **#define:** `FSMC_ExtendedMode_Enable((uint32_t)0x00004000)`
FSMC_Flags
- #define: SMSM_FLAG_RisingEdge((uint32_t)0x00000001)
- #define: SMSM_FLAG_Level((uint32_t)0x00000002)
- #define: SMSM_FLAG_FallingEdge((uint32_t)0x00000004)
- #define: SMSM_FLAG_FEMPT((uint32_t)0x00000040)

FSMC_Interrupt_sources
- #define: SMSM_IT_RisingEdge((uint32_t)0x00000008)
- #define: SMSM_IT_Level((uint32_t)0x00000010)
- #define: SMSM_IT_FallingEdge((uint32_t)0x00000020)

FSMC_Memory_Type
- #define: SMSM_MemoryType_SRAM((uint32_t)0x00000000)
- #define: SMSM_MemoryType_PSRAM((uint32_t)0x00000004)
- #define: SMSM_MemoryType_NOR((uint32_t)0x00000008)

FSMC_NAND_Bank
- #define: SMSM_Bank2_NAND((uint32_t)0x00000010)
- `#define FSMC_Bank3_NAND(uint32_t)0x00000100`

**FSMC_NORSRAM_Bank**
- `#define FSMC_Bank1_NORSRAM1(uint32_t)0x00000000`
- `#define FSMC_Bank1_NORSRAM2(uint32_t)0x00000002`
- `#define FSMC_Bank1_NORSRAM3(uint32_t)0x00000004`
- `#define FSMC_Bank1_NORSRAM4(uint32_t)0x00000006`

**FSMC_PCCARD_Bank**
- `#define FSMC_Bank4_PCCARD(uint32_t)0x00001000`

**FSMC_Wait_feature**
- `#define FSMC_Waitfeature_Disable(uint32_t)0x00000000`
- `#define FSMC_Waitfeature_Enable(uint32_t)0x00002000`

**FSMC_Wait_Signal**
- `#define FSMC_WaitSignal_Disable(uint32_t)0x00000000`
- `#define FSMC_WaitSignal_Enable(uint32_t)0x00002000`
**FSMC_Wait_Signal_Polarity**

- #define:  
  ```c
  FSMC_WaitSignalPolarity_Low((uint32_t)0x00000000)
  ```

- #define:  
  ```c
  FSMC_WaitSignalPolarity_High((uint32_t)0x00000200)
  ```

**FSMC_Wait_Timing**

- #define:  
  ```c
  FSMC_WaitSignalActive_BeforeWaitState((uint32_t)0x00000000)
  ```

- #define:  
  ```c
  FSMC_WaitSignalActive_DuringWaitState((uint32_t)0x00000800)
  ```

**FSMC_Wrap_Mode**

- #define:  
  ```c
  FSMC_WrapMode_Disable((uint32_t)0x00000000)
  ```

- #define:  
  ```c
  FSMC_WrapMode_Enable((uint32_t)0x00000400)
  ```

**FSMC_Write_Burst**

- #define:  
  ```c
  FSMC_WriteBurst_Disable((uint32_t)0x00000000)
  ```

- #define:  
  ```c
  FSMC_WriteBurst_Enable((uint32_t)0x00080000)
  ```

**FSMC_Write_Operation**

- #define:  
  ```c
  FSMC_WriteOperation_Disable((uint32_t)0x00000000)
  ```

- #define:  
  ```c
  FSMC_WriteOperation_Enable((uint32_t)0x00001000)
  ```
13.4 FSMC Programming Example

The example below explains how to configure the FSMC to interface with an external SRAM (connected to Bank1_SRAM2 Bank). For more examples about FSMC configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\FSMC\.

```c
FSMC_NORSRAMInitTypeDef SRAMInitS;
FSMC_NORSRAMTimingInitTypeDef p;

/* Enable FSMC clock */
RCC_AHB3PeriphClockCmd(RCC_AHB3Periph_FSMC, ENABLE);

/* FSMC Configuration */
p.FSMC_AddressSetupTime = 0;
p.FSMC_AddressHoldTime = 0;
p.FSMC_DataSetupTime = 4;
p.FSMC_BusTurnAroundDuration = 1;
p.FSMC_CLKDivision = 0;
p.FSMC_DataLatency = 0;
p.FSMC_AccessMode = FSMC_AccessMode_A;

SRAMInitS.FSMC_Bank = FSMC_Bank1_NORSRAM2;
SRAMInitS.FSMC_DataAddressMux = FSMC_DataAddressMux_Disable;
SRAMInitS.FSMC_MemoryType = FSMC_MemoryType_PSRAM;
SRAMInitS.FSMC_DataAddressWidth = FSMC_DataAddressWidth_16b;
SRAMInitS.FSMC_BurstAccessMode = FSMC_BurstAccessMode_Disable;
SRAMInitS.FSMC_AsynchronousWait = FSMC_AsynchronousWait_Disable;
SRAMInitS.FSMC_WaitSignalPolarity = FSMC_WaitSignalPolarity_Low;
SRAMInitS.FSMC_WrapMode = FSMC_WrapMode_Disable;
SRAMInitS.FSMC_WriteSignalActive = FSMC_WriteSignalActive_BeforeWaitState;
SRAMInitS.FSMC_WriteOperation = FSMC_WriteOperation_Enable;
SRAMInitS.FSMC_WriteSignal = FSMC_WriteSignal_Disable;
SRAMInitS.FSMC_ExtendedMode = FSMC_ExtendedMode_Disable;
SRAMInitS.FSMC_WriteBurst = FSMC_WriteBurst_Disable;
SRAMInitS.FSMC_ReadWriteTimingStruct = &p;
SRAMInitS.FSMC_WriteTimingStruct = &p;

FSMC_NORSRAMInit(&SRAMInitS);

/* Enable FSMC Bank1_SRAM2 Bank */
FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM2, ENABLE);
```
14 General-purpose I/Os (GPIO)

14.1 GPIO Firmware driver registers structures

14.1.1 GPIO_TypeDef

`GPIO_TypeDef` is defined in the `stm32f2xx.h` file and contains the GPIO registers definition.

Data Fields

- `__IO uint32_t MODER` - GPIO port mode register, Address offset: 0x00
- `__IO uint32_t OTYPER` - GPIO port output type register, Address offset: 0x04
- `__IO uint32_t OSPEEDR` - GPIO port output speed register, Address offset: 0x08
- `__IO uint32_t PUPDR` - GPIO port pull-up/pull-down register, Address offset: 0x0C
- `__IO uint32_t IDR` - GPIO port input data register, Address offset: 0x10
- `__IO uint32_t ODR` - GPIO port output data register, Address offset: 0x14
- `__IO uint16_t BSRRL` - GPIO port bit set/reset low register, Address offset: 0x18
- `__IO uint16_t BSRRH` - GPIO port bit set/reset high register, Address offset: 0x1A
- `__IO uint32_t LCKR` - GPIO port configuration lock register, Address offset: 0x1C
- `__IO uint32_t AFR` - GPIO alternate function registers, Address offset: 0x24-0x28

Field Documentation

- `__IO uint32_t GPIO_TypeDef::MODER` - GPIO port mode register, Address offset: 0x00
- `__IO uint32_t GPIO_TypeDef::OTYPER` - GPIO port output type register, Address offset: 0x04
- `__IO uint32_t GPIO_TypeDef::OSPEEDR` - GPIO port output speed register, Address offset: 0x08
- `__IO uint32_t GPIO_TypeDef::PUPDR` - GPIO port pull-up/pull-down register, Address offset: 0x0C
- `__IO uint32_t GPIO_TypeDef::IDR` - GPIO port input data register, Address offset: 0x10
- `__IO uint32_t GPIO_TypeDef::ODR` - GPIO port output data register, Address offset: 0x14
- `__IO uint16_t GPIO_TypeDef::BSRRL` - GPIO port bit set/reset low register, Address offset: 0x18
- `__IO uint16_t GPIO_TypeDef::BSRRH` - GPIO port bit set/reset high register, Address offset: 0x1A
- `__IO uint32_t GPIO_TypeDef::LCKR` - GPIO port configuration lock register, Address offset: 0x1C
- `__IO uint32_t GPIO_TypeDef::AFR[2]` - GPIO alternate function registers, Address offset: 0x24-0x28

14.1.2 GPIO_InitTypeDef

`GPIO_InitTypeDef` is defined in the `stm32f2xx_gpio.h`
Data Fields

- `uint32_t GPIO_Pin`
- `GPIOMode_TypeDef GPIO_Mode`
- `GPIOSpeed_TypeDef GPIO_Speed`
- `GPIOOType_TypeDef GPIO_OType`
- `GPIOPuPd_TypeDef GPIO_PuPd`

Field Documentation

- `uint32_t GPIO_InitTypeDef::GPIO_Pin`
  - Specifies the GPIO pins to be configured. This parameter can be any value of `GPIO_pins_define`
- `GPIOMode_TypeDef GPIO_InitTypeDef::GPIO_Mode`
  - Specifies the operating mode for the selected pins. This parameter can be a value of `GPIOMode_TypeDef`
- `GPIOSpeed_TypeDef GPIO_InitTypeDef::GPIO_Speed`
  - Specifies the speed for the selected pins. This parameter can be a value of `GPIOSpeed_TypeDef`
- `GPIOOType_TypeDef GPIO_InitTypeDef::GPIO_OType`
  - Specifies the operating output type for the selected pins. This parameter can be a value of `GPIOOType_TypeDef`
- `GPIOPuPd_TypeDef GPIO_InitTypeDef::GPIO_PuPd`
  - Specifies the operating Pull-up/Pull down for the selected pins. This parameter can be a value of `GPIOPuPd_TypeDef`

14.2 GPIO Firmware driver API description

The following section lists the various functions of the GPIO library.

14.2.1 How to use this driver

1. Enable the GPIO AHB clock using the following function
   
   ```c
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOx, ENABLE);
   ```

2. Configure the GPIO pin(s) using `GPIO_Init()`. Four possible configuration are available for each pin:
   - Input: Floating, Pull-up, Pull-down.
   - Output: Push-Pull (Pull-up, Pull-down or no Pull). Open Drain (Pull-up, Pull-down or no Pull). In output mode, the speed is configurable: 2 MHz, 25 MHz, 50 MHz or 100 MHz.
   - Alternate Function: Push-Pull (Pull-up, Pull-down or no Pull). Open Drain (Pull-up, Pull-down or no Pull).
   - Analog: required mode when a pin is to be used as ADC channel or DAC output.

3. Peripherals alternate function:
   - For ADC and DAC, configure the desired pin in analog mode using `GPIO_InitStruct->GPIO_Mode = GPIO_Mode_AN;`
   - For other peripherals (TIM, USART...):
Connect the pin to the desired peripherals' Alternate Function (AF) using GPIO_PinAFConfig() function - Configure the desired pin in alternate function mode using GPIO_InitStruct->GPIO_Mode = GPIO_Mode_AF - Select the type, pull-up/pull-down and output speed via GPIO_PuPd, GPIO_OType and GPIO_Speed members - Call GPIO_Init() function

4. To get the level of a pin configured in input mode use GPIO_ReadInputDataBit()
5. To set/reset the level of a pin configured in output mode use GPIO_SetBits()/GPIO_ResetBits()
6. During and just after reset, the alternate functions are not active and the GPIO pins are configured in input floating mode (except JTAG pins).
7. The LSE oscillator pins OSC32_IN and OSC32_OUT can be used as general-purpose (PC14 and PC15, respectively) when the LSE oscillator is off. The LSE has priority over the GPIO function.
8. The HSE oscillator pins OSC_IN/OSC_OUT can be used as general-purpose PH0 and PH1, respectively, when the HSE oscillator is off. The HSE has priority over the GPIO function.

14.2.2 Initialization and configuration

- GPIO_DeInit()
- GPIO_Init()
- GPIO_StructInit()
- GPIO_PinLockConfig()

14.2.3 GPIO Read and Write

- GPIO_ReadInputDataBit()
- GPIO_ReadInputData()
- GPIO_ReadOutputDataBit()
- GPIO_ReadOutputData()
- GPIO_SetBits()
- GPIO_ResetBits()
- GPIO_WriteBit()
- GPIO_Write()
- GPIO_ToggleBits()

14.2.4 GPIO Alternate functions configuration

- GPIO_PinAFConfig()

14.2.5 Initialization and configuration functions

14.2.5.1 GPIO_DeInit

Function Name: void GPIO_DeInit (GPIO_TypeDef * GPIOx)
Function Description: Deinitializes the GPIOx peripheral registers to their default reset values.
### 14.2.5.2 GPIO_Init

**Function Name**

```c
void GPIO_Init ( GPIO_TypeDef * GPIOx, GPIO_InitTypeDef * GPIO_InitStruct)
```

**Function Description**

Initializes the GPIOx peripheral according to the specified parameters in the GPIO_InitStruct.

**Parameters**

- **GPIOx**: where x can be (A..I) to select the GPIO peripheral.
- **GPIO_InitStruct**: pointer to a GPIO_InitTypeDef structure that contains the configuration information for the specified GPIO peripheral.

**Return values**

- None.

**Notes**

- None.

### 14.2.5.3 GPIO_StructInit

**Function Name**

```c
void GPIO_StructInit ( GPIO_InitTypeDef * GPIO_InitStruct)
```

**Function Description**

Fills each GPIO_InitStruct member with its default value.

**Parameters**

- **GPIO_InitStruct**: pointer to a GPIO_InitTypeDef structure which will be initialized.

**Return values**

- None.

**Notes**

- None.

### 14.2.5.4 GPIO_PinLockConfig
Function Name: void GPIO_PinLockConfig (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)

Function Description: Locks GPIO Pins configuration registers.

Parameters:
- GPIOx : where x can be (A..I) to select the GPIO peripheral.
- GPIO_Pin : specifies the port bit to be locked. This parameter can be any combination of GPIO_Pin_x where x can be (0..15).

Return values:
- None.

Notes:
- The locked registers are GPIOx_MODER, GPIOx_OTYPER, GPIOx_OSPEEDR, GPIOx_PUPDR, GPIOx_AFRL and GPIOx_AFRH.
- The configuration of the locked GPIO pins can no longer be modified until the next reset.

14.2.6 GPIO Read and Write functions

14.2.6.1 GPIO_ReadInputDataBit

Function Name: uint8_t GPIO_ReadInputDataBit (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)

Function Description: Reads the specified input port pin.

Parameters:
- GPIOx : where x can be (A..I) to select the GPIO peripheral.
- GPIO_Pin : specifies the port bit to read. This parameter can be GPIO_Pin_x where x can be (0..15).

Return values:
- The input port pin value.

Notes:
- None.

14.2.6.2 GPIO_ReadInputData

Function Name: uint16_t GPIO_ReadInputData (GPIO_TypeDef * GPIOx)

Function Description: Reads the specified GPIO input data port.

Parameters:
- GPIOx : where x can be (A..I) to select the GPIO peripheral.

Return values:
- GPIO input data port value.

Notes:
- None.
14.2.6.3  GPIO_ReadOutputDataBit

Function Name  uint8_t GPIO_ReadOutputDataBit ( GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)  
Function Description  Reads the specified output data port bit.  
Parameters  - GPIOx : where x can be (A..I) to select the GPIO peripheral.  
            - GPIO_Pin : specifies the port bit to read. This parameter can be GPIO_Pin_x where x can be (0..15).  
Return values  - The output port pin value.  
Notes  - None.

14.2.6.4  GPIO_ReadOutputData

Function Name  uint16_t GPIO_ReadOutputData ( GPIO_TypeDef * GPIOx)  
Function Description  Reads the specified GPIO output data port.  
Parameters  - GPIOx : where x can be (A..I) to select the GPIO peripheral.  
Return values  - GPIO output data port value.  
Notes  - None.

14.2.6.5  GPIO_SetBits

Function Name  void GPIO_SetBits ( GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)  
Function Description  Sets the selected data port bits.  
Parameters  - GPIOx : where x can be (A..I) to select the GPIO peripheral.  
            - GPIO_Pin : specifies the port bits to be written. This parameter can be any combination of GPIO_Pin_x where x can be (0..15).
### 14.2.6.6 GPIO_ResetBits

**Function Name**

```c
void GPIO_ResetBits ( GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)
```

**Function Description**
Cleans the selected data port bits.

**Parameters**
- **GPIOx**: where x can be (A..I) to select the GPIO peripheral.
- **GPIO_Pin**: specifies the port bits to be written. This parameter can be any combination of GPIO_Pin_x where x can be (0..15).

**Return values**
- None.

**Notes**
- This function uses GPIOx_BSRR register to allow atomic read/modify accesses. In this way, there is no risk of an IRQ occurring between the read and the modify access.

### 14.2.6.7 GPIO_WriteBit

**Function Name**

```c
void GPIO_WriteBit ( GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin, BitAction BitVal)
```

**Function Description**
Sets or clears the selected data port bit.

**Parameters**
- **GPIOx**: where x can be (A..I) to select the GPIO peripheral.
- **GPIO_Pin**: specifies the port bit to be written. This parameter can be one of GPIO_Pin_x where x can be (0..15).
- **BitVal**: specifies the value to be written to the selected bit. This parameter can be one of the BitAction enum values:
  - `Bit_RESET`: to clear the port pin
  - `Bit_SET`: to set the port pin

**Return values**
- None.

**Notes**
- None.
14.2.6.8  GPIO_Write

Function Name  void GPIO_Write ( GPIO_TypeDef * GPIOx, uint16_t PortVal)
Function Description  Writes data to the specified GPIO data port.
Parameters  
  -  GPIOx : where x can be (A..I) to select the GPIO peripheral.
  -  PortVal : specifies the value to be written to the port output data register.
Return values  
  -  None.
Notes  
  -  None.

14.2.6.9  GPIO_ToggleBits

Function Name  void GPIO_ToggleBits ( GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)
Function Description  Toggles the specified GPIO pins.
Parameters  
  -  GPIOx : where x can be (A..I) to select the GPIO peripheral.
  -  GPIO_Pin : Specifies the pins to be toggled.
Return values  
  -  None.
Notes  
  -  None.

14.2.7  GPIO Alternate functions configuration function
14.2.7.1  GPIO_PinAFConfig

Function Name  void GPIO_PinAFConfig ( GPIO_TypeDef * GPIOx, uint16_t GPIO_PinSource, uint8_t GPIO_AF)
Function Description  Changes the mapping of the specified pin.
Parameters  
  -  GPIOx : where x can be (A..I) to select the GPIO peripheral.
  -  GPIO_PinSource : specifies the pin for the Alternate function. This parameter can be GPIO_PinSource_x where x can be (0..15).
• **GPIO_AFSelection**: selects the pin to used as Alternate function. This parameter can be one of the following values:
  - **GPIO_AF_RTC_50Hz**: Connect RTC_50Hz pin to AF0 (default after reset)
  - **GPIO_AF_MCO**: Connect MCO pin (MCO1 and MCO2) to AF0 (default after reset)
  - **GPIO_AF_TAMPER**: Connect TAMPER pins (TAMPER_1 and TAMPER_2) to AF0 (default after reset)
  - **GPIO_AF_SWJ**: Connect SWJ pins (SWD and JTAG) to AF0 (default after reset)
  - **GPIO_AF_TRACE**: Connect TRACE pins to AF0 (default after reset)
  - **GPIO_AF_TIM1**: Connect TIM1 pins to AF1
  - **GPIO_AF_TIM2**: Connect TIM2 pins to AF1
  - **GPIO_AF_TIM3**: Connect TIM3 pins to AF2
  - **GPIO_AF_TIM4**: Connect TIM4 pins to AF2
  - **GPIO_AF_TIM5**: Connect TIM5 pins to AF2
  - **GPIO_AF_TIM8**: Connect TIM8 pins to AF3
  - **GPIO_AF_TIM9**: Connect TIM9 pins to AF3
  - **GPIO_AF_TIM10**: Connect TIM10 pins to AF3
  - **GPIO_AF_TIM11**: Connect TIM11 pins to AF3
  - **GPIO_AF_I2C1**: Connect I2C1 pins to AF4
  - **GPIO_AF_I2C2**: Connect I2C2 pins to AF4
  - **GPIO_AF_I2C3**: Connect I2C3 pins to AF4
  - **GPIO_AF_SPI1**: Connect SPI1 pins to AF5
  - **GPIO_AF_SPI2**: Connect SPI2/I2S2 pins to AF5
  - **GPIO_AF_SPI3**: Connect SPI3/I2S3 pins to AF6
  - **GPIO_AF_USART1**: Connect USART1 pins to AF7
  - **GPIO_AF_USART2**: Connect USART2 pins to AF7
  - **GPIO_AF_USART3**: Connect USART3 pins to AF7
  - **GPIO_AF_UART4**: Connect UART4 pins to AF8
  - **GPIO_AF_UART5**: Connect UART5 pins to AF8
  - **GPIO_AF_USART6**: Connect USART6 pins to AF8
  - **GPIO_AF_CAN1**: Connect CAN1 pins to AF9
  - **GPIO_AF_CAN2**: Connect CAN2 pins to AF9
  - **GPIO_AF_TIM12**: Connect TIM12 pins to AF9
  - **GPIO_AF_TIM13**: Connect TIM13 pins to AF9
  - **GPIO_AF_TIM14**: Connect TIM14 pins to AF9
  - **GPIO_AF_OTG_FS**: Connect OTG_FS pins to AF10
  - **GPIO_AF_OTG_HS**: Connect OTG_HS pins to AF10
  - **GPIO_AF_ETH**: Connect ETHERNET pins to AF11
  - **GPIO_AF_FSMC**: Connect FSMC pins to AF12
  - **GPIO_AF_OTG_HS_FS**: Connect OTG HS (configured in FS) pins to AF12
  - **GPIO_AF_SDIO**: Connect SDIO pins to AF12
  - **GPIO_AF_DCM**: Connect DCM pins to AF13
  - **GPIO_AF_EVENTOUT**: Connect EVENTOUT pins to AF15

Return values
• None.

Notes
• None.
14.3 GPIO Firmware driver defines

14.3.1 GPIO Firmware driver defines

GPIO

(GPIO_Alartern_function_selection_define

- #define: GPIO_AF_RTC_50Hz((uint8_t)0x00)

- #define: GPIO_AF_MCO((uint8_t)0x00)

- #define: GPIO_AF_TAMPER((uint8_t)0x00)

- #define: GPIO_AF_SWJ((uint8_t)0x00)

- #define: GPIO_AF_TRACE((uint8_t)0x00)

- #define: GPIO_AF_TIM1((uint8_t)0x01)

- #define: GPIO_AF_TIM2((uint8_t)0x01)

- #define: GPIO_AF_TIM3((uint8_t)0x02)

- #define: GPIO_AF_TIM4((uint8_t)0x02)

- #define: GPIO_AF_TIM5((uint8_t)0x02)
- `#define GPIO_AF_TIM8((uint8_t)0x03)`

- `#define GPIO_AF_TIM9((uint8_t)0x03)`

- `#define GPIO_AF_TIM10((uint8_t)0x03)`

- `#define GPIO_AF_TIM11((uint8_t)0x03)`

- `#define GPIO_AF_I2C1((uint8_t)0x04)`

- `#define GPIO_AF_I2C2((uint8_t)0x04)`

- `#define GPIO_AF_I2C3((uint8_t)0x04)`

- `#define GPIO_AF_SPI1((uint8_t)0x05)`

- `#define GPIO_AF_SPI2((uint8_t)0x05)`

- `#define GPIO_AF_SPI3((uint8_t)0x06)`

- `#define GPIO_AF_USART1((uint8_t)0x07)`

- `#define GPIO_AF_USART2((uint8_t)0x07)`
- #define: `GPIO_AF_USART3((uint8_t)0x07)`

- #define: `GPIO_AF_UART4((uint8_t)0x08)`

- #define: `GPIO_AF_UART5((uint8_t)0x08)`

- #define: `GPIO_AF_USART6((uint8_t)0x08)`

- #define: `GPIO_AF_CAN1((uint8_t)0x09)`

- #define: `GPIO_AF_CAN2((uint8_t)0x09)`

- #define: `GPIO_AF_TIM12((uint8_t)0x09)`

- #define: `GPIO_AF_TIM13((uint8_t)0x09)`

- #define: `GPIO_AF_TIM14((uint8_t)0x09)`

- #define: `GPIO_AF_OTG_FS((uint8_t)0xA)`

- #define: `GPIO_AF_OTG_HS((uint8_t)0xA)`

- #define: `GPIO_AF_ETH((uint8_t)0x0B)`
- #define: `GPIO_AF_FSMC((uint8_t)0xC)`

- #define: `GPIO_AF_OTG_HS_FS((uint8_t)0xC)`

- #define: `GPIO_AF_SDIO((uint8_t)0xC)`

- #define: `GPIO_AF_DCMI((uint8_t)0x0D)`

- #define: `GPIO_AF_EVENTOUT((uint8_t)0x0F)`

**GPIO_Legacy**

- #define: `GPIO_Mode_AINGPIO_Mode_AN`

- #define: `GPIO_AF_OTG1_FSGPIO_AF_OTG_FS`

- #define: `GPIO_AF_OTG2_HSGPIO_AF_OTG_HS`

- #define: `GPIO_AF_OTG2_FSGPIO_AF_OTG_HS_FS`

**GPIO_pins_define**

- #define: `GPIO_Pin_0((uint16_t)0x0001)`

- #define: `GPIO_Pin_1((uint16_t)0x0002)`
- #define: `GPIO_Pin_2((uint16_t)0x0004)`

- #define: `GPIO_Pin_3((uint16_t)0x0008)`

- #define: `GPIO_Pin_4((uint16_t)0x0010)`

- #define: `GPIO_Pin_5((uint16_t)0x0020)`

- #define: `GPIO_Pin_6((uint16_t)0x0040)`

- #define: `GPIO_Pin_7((uint16_t)0x0080)`

- #define: `GPIO_Pin_8((uint16_t)0x0100)`

- #define: `GPIO_Pin_9((uint16_t)0x0200)`

- #define: `GPIO_Pin_10((uint16_t)0x0400)`

- #define: `GPIO_Pin_11((uint16_t)0x0800)`

- #define: `GPIO_Pin_12((uint16_t)0x1000)`

- #define: `GPIO_Pin_13((uint16_t)0x2000)`
• **#define:** `GPIO_Pin_14((uint16_t)0x4000)`

• **#define:** `GPIO_Pin_15((uint16_t)0x8000)`

• **#define:** `GPIO_Pin_All((uint16_t)0xFFFF)`

**GPIO_Pin_sources**

• **#define:** `GPIO_PinSource0((uint8_t)0x00)`

• **#define:** `GPIO_PinSource1((uint8_t)0x01)`

• **#define:** `GPIO_PinSource2((uint8_t)0x02)`

• **#define:** `GPIO_PinSource3((uint8_t)0x03)`

• **#define:** `GPIO_PinSource4((uint8_t)0x04)`

• **#define:** `GPIO_PinSource5((uint8_t)0x05)`

• **#define:** `GPIO_PinSource6((uint8_t)0x06)`

• **#define:** `GPIO_PinSource7((uint8_t)0x07)`

• **#define:** `GPIO_PinSource8((uint8_t)0x08)`
• #define: `GPIO_PinSource9((uint8_t)0x09)`

• #define: `GPIO_PinSource10((uint8_t)0x0A)`

• #define: `GPIO_PinSource11((uint8_t)0x0B)`

• #define: `GPIO_PinSource12((uint8_t)0x0C)`

• #define: `GPIO_PinSource13((uint8_t)0x0D)`

• #define: `GPIO_PinSource14((uint8_t)0x0E)`

• #define: `GPIO_PinSource15((uint8_t)0x0F)`

14.4 GPIO Programming Example

The example below explains how to configure the different GPIO modes. For more examples about GPIO configuration and usage, please refer to the GPIO examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\

Output mode

The example below shows how to configure an I/O in output mode (for example PG6 to drive a led)

```c
GPIO_InitTypeDef GPIO_InitStructure;
/* Enable GPIOG's AHB interface clock */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG, ENABLE);

/* Configure PG6 in output pushpull mode */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
```
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
GPIO_Init(GPIOG, &GPIO_InitStructure);

/* Set PG6 to high level */
GPIO_SetBits(GPIOG, GPIO_Pin_6);

**Input mode**

The example below shows how to configure an I/O in input mode (for example PG6 to be used as an EXTI line)

GPIO_InitTypeDef  GPIO_InitStructure;
/* Enable GPIOG's AHB interface clock */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG, ENABLE);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOG, &GPIO_InitStructure);

**Analog mode**

The example below shows how to configure an I/O in analog mode (for example PA4 to be used as ADC input or DAC output)

GPIO_InitTypeDef  GPIO_InitStructure;
/* Enable GPIOA's AHB interface clock */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
GPIO_Init(GPIOA, &GPIO_InitStructure);

**Alternate function mode**

The example below shows how to configure USART2 Tx/Rx I/Os on PD5/PD6 pins

GPIO_InitTypeDef  GPIO_InitStructure;
/* Enable GPIOD's AHB interface clock */
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOD, ENABLE);
/* Connect PD5 to USART2_Tx */
GPIO_PinAFConfig(GPIOD, GPIO_PinSource5, GPIO_AF_USART2);
/* Connect PD6 to USART2_Rx*/
GPIO_PinAFConfig(GPIOD, GPIO_PinSource6, GPIO_AF_USART2);
/* Configure USART2_Tx and USART2_Rx as alternate function */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6;
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(GPIOD,&GPIO_InitStructure);
15 Hash processor (HASH)

15.1 HASH Firmware driver registers structures

15.1.1 HASH_TypeDef

`HASH_TypeDef` is defined in the `stm32f2xx.h` file and contains the HASH registers definition.

**Data Fields**

- `__IO uint32_t CR`
- `__IO uint32_t DIN`
- `__IO uint32_t STR`
- `__IO uint32_t HR`
- `__IO uint32_t IMR`
- `__IO uint32_t SR`
- `uint32_t RESERVED`
- `__IO uint32_t CSR`

**Field Documentation**

- `__IO uint32_t HASH_TypeDef::CR`  
  - HASH control register, Address offset: 0x00
- `__IO uint32_t HASH_TypeDef::DIN`  
  - HASH data input register, Address offset: 0x04
- `__IO uint32_t HASH_TypeDef::STR`  
  - HASH start register, Address offset: 0x08
- `__IO uint32_t HASH_TypeDef::HR[5]`  
  - HASH digest registers, Address offset: 0x0C-0x1C
- `__IO uint32_t HASH_TypeDef::IMR`  
  - HASH interrupt enable register, Address offset: 0x20
- `__IO uint32_t HASH_TypeDef::SR`  
  - HASH status register, Address offset: 0x24
- `uint32_t HASH_TypeDef::RESERVED[52]`  
  - Reserved, 0x28-0xF4
- `__IO uint32_t HASH_TypeDef::CSR[51]`  
  - HASH context swap registers, Address offset: 0x0F8-0x1C0

15.1.2 HASH_InitTypeDef

`HASH_InitTypeDef` is defined in the `stm32f2xx_hash.h` file and contains the HASH common initialization parameters.

**Data Fields**

- `uint32_t HASH_AlgoSelection`
Field Documentation

- `uint32_t HASH_AlgoMode`
- `uint32_t HASH_DataType`
- `uint32_t HASH_HMACKeyType`

Field Documentation

- `uint32_t HASH_InitTypeDef::HASH_AlgoSelection`
  - SHA-1 or MD5. This parameter can be a value of `HASH_Algo_Selection`
- `uint32_t HASH_InitTypeDef::HASH_AlgoMode`
  - HASH or HMAC. This parameter can be a value of `HASH_processor_Algorithm_Mode`
- `uint32_t HASH_InitTypeDef::HASH_DataType`
  - 32-bit data, 16-bit data, 8-bit data or bit-string. This parameter can be a value of `HASH_Data_Type`
- `uint32_t HASH_InitTypeDef::HASH_HMACKeyType`
  - HMAC Short key or HMAC Long Key. This parameter can be a value of `HASH_HMAC_Long_key_only_for_HMAC_mode`

15.1.3 HASH_MsgDigest

`HASH_MsgDigest` is defined in the stm32f2xx_hash.h

Data Fields

- `uint32_t Data`

Field Documentation

- `uint32_t HASH_MsgDigest::Data[5]`
  - Message digest result : 5x 32bit words for SHA1 or 4x 32bit words for MD5

15.1.4 HASH_Context

`HASH_Context` is defined in the stm32f2xx_hash.h

Data Fields

- `uint32_t HASH_IMR`
- `uint32_t HASH_STR`
- `uint32_t HASH_CR`
- `uint32_t HASH_CSR`

Field Documentation

- `uint32_t HASH_Context::HASH_IMR`
15.2 HASH Firmware driver API description

The following section lists the various functions of the HASH library.

15.2.1 How to use this driver

HASH operation

1. Enable the HASH controller clock using 
   `RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_HASH, ENABLE)` function.
2. Initialize the HASH using `HASH_Init()` function.
3. Reset the HASH processor core, so that the HASH will be ready to compute he 
   message digest of a new message by using `HASH_Reset()` function.
4. Enable the HASH controller using the `HASH_Cmd()` function.
5. if using DMA for Data input transfer, Activate the DMA Request using 
   `HASH_DMACmd()` function
6. if DMA is not used for data transfer, use `HASH_DataIn()` function to enter data to IN 
   FIFO.
7. Configure the Number of valid bits in last word of the message using 
   `HASH_SetLastWordValidBitsNbr()` function.
8. If the message length is not an exact multiple of 512 bits, then the function 
   `HASH_StartDigest()` must be called to launch the computation of the final digest.
9. Once computed, the digest can be read using `HASH_GetDigest()` function.
10. To control HASH events you can use one of the following two methods: After 
    checking a flag you should clear it using `HASH_ClearFlag()` function. And after 
    checking on an interrupt event you should clear it using `HASH_ClearITPendingBit()` 
    function.
    - Check on HASH flags using the `HASH_GetFlagStatus()` function.
    - Use HASH interrupts through the function `HASH_ITConfig()` at initialization phase 
      and `HASH_GetITStatus()` function into interrupt routines in hashing phase.
11. Save and restore hash processor context using `HASH_SaveContext()` and 
    `HASH_RestoreContext()` functions.

HMAC operation

The HMAC algorithm is used for message authentication, by irreversibly binding the 
message being processed to a key chosen by the user. For HMAC specifications, refer to 
"HMAC: keyed-hashing for message authentication, H. Krawczyk, M. Bellare, R. Canetti, 
February 1997".

Basically, the HMAC algorithm consists of two nested hash operations:

\[
\text{HMAC}(\text{message}) = \text{Hash}([\text{key} \mid \text{pad} \text{ XOR } 0x5C] \mid \text{Hash}([\text{key} \mid \text{pad} \text{ XOR } 0x36] \mid \text{message}])
\]

where:

- "pad" is a sequence of zeroes needed to extend the key to the length of the underlying 
  hash function data block (that is 512 bits for both the SHA-1 and MD5 hash 
  algorithms)
- "|" represents the concatenation operator
To compute the HMAC, four different phases are required:

1. Initialize the HASH using HASH_Init() function to do HMAC operation.
2. The key (to be used for the inner hash function) is then given to the core. This operation follows the same mechanism as the one used to send the message in the hash operation (that is, by HASH_DataIn() function and, finally, HASH_StartDigest() function.
3. Once the last word has been entered and computation has started, the hash processor elaborates the key. It is then ready to accept the message text using the same mechanism as the one used to send the message in the hash operation.
4. After the first hash round, the hash processor returns "ready" to indicate that it is ready to receive the key to be used for the outer hash function (normally, this key is the same as the one used for the inner hash function). When the last word of the key is entered and computation starts, the HMAC result is made available using HASH_GetDigest() function.

15.2.2 Initialization and configuration

This section provides functions allowing to:
- Initialize the HASH peripheral
- Configure the HASH Processor
- MD5/SHA1
- HASH/HMAC
- datatype
- HMAC Key (if mode = HMAC)
- Reset the HASH Processor

The initialization and configuration functions are the following:

- HASH_DeInit()
- HASH_Init()
- HASH_StructInit()
- HASH_Reset()

15.2.3 Message Digest generation

This section provides functions allowing the generation of message digest:

- Push data in the IN FIFO : using HASH_DataIn()
- Get the number of words set in IN FIFO, use HASH_GetInFIFOWordsNbr()
- set the last word valid bits number using HASH_SetLastWordValidBitsNbr()
- start digest calculation : using HASH_StartDigest()
- Get the Digest message : using HASH_GetDigest()

The message digest functions are the following:

- HASH_SetLastWordValidBitsNbr()
- HASH_DataIn()
- HASH_GetInFIFOWordsNbr()
- HASH_GetDigest()
- HASH_StartDigest()
15.2.4 Context swapping

This section provides functions allowing to save and store HASH Context.

It is possible to interrupt a HASH/HMAC process to perform another processing with a higher priority, and to complete the interrupted process later on, when the higher priority task is complete. To do so, the context of the interrupted task must be saved from the HASH registers to memory, and then be restored from memory to the HASH registers.

- To save the current context, use HASH_SaveContext() function
- To restore the saved context, use HASH_RestoreContext() function

15.2.5 Initialization and configuration

This section provides functions allowing to

- Initialize the HASH peripheral
- Configure the HASH Processor
- MD5/SHA1
- HASH/HMAC
- datatype
- HMAC Key (if mode = HMAC)
- Reset the HASH Processor

The initialization and configuration functions are the following:

- HASH_DeInit()
- HASH_Init()
- HASH_StructInit()
- HASH_Reset()

15.2.6 Interrupt and flag management

This section provides functions allowing to configure the HASH Interrupts and to get the status and clear flags and Interrupts pending bits.

The HASH provides 5 Flags and 2 Interrupts sources:

- Flags
  - HASH_FLAG_DINIS : set when 16 locations are free in the Data IN FIFO which means that a new block (512 bit) can be entered into the input buffer.
  - HASH_FLAG_DCIS : set when Digest calculation is complete
  - HASH_FLAG_DMAS : set when HASH's DMA interface is enabled (DMAE=1) or a transfer is ongoing. This Flag is cleared only by hardware.
  - HASH_FLAG_BUSY : set when The hash core is processing a block of data This Flag is cleared only by hardware.
  - HASH_FLAG_DINNE : set when Data IN FIFO is not empty which means that the Data IN FIFO contains at least one word of data. This Flag is cleared only by hardware.
Interrupts

- HASH_IT_DINI: if enabled, this interrupt source is pending when 16 locations are free in the Data IN FIFO which means that a new block (512 bit) can be entered into the input buffer. This interrupt source is cleared using HASH_ClearITPendingBit(HASH_IT_DINI) function.
- HASH_IT_DCI: if enabled, this interrupt source is pending when Digest calculation is complete. This interrupt source is cleared using HASH_ClearITPendingBit(HASH_IT_DCI) function.

Managing the HASH controller events The user should identify which mode will be used in his application to manage the HASH controller events: Polling mode or Interrupt mode. In the Polling Mode it is advised to use the following functions:
  - HASH_GetFlagStatus(): to check if flags events occur. HASH_ClearFlag(): to clear the flags events. In the Interrupt Mode it is advised to use the following functions:
    - fnc: to enable or disable the interrupt source.
    - HASH_GetITStatus(): to check if Interrupt occurs.
    - HASH_ClearITPendingBit(): to clear the Interrupt pending Bit (corresponding Flag).

- fnc
- HASH_GetFlagStatus()
- HASH_ClearFlag()
- HASH_GetITStatus()
- HASH_ClearITPendingBit()

### 15.2.7 High level functions

#### High Level SHA1 functions

- HASH_SHA1()
- HMAC_SHA1()

#### High Level MD5 functions

- HASH_MD5()
- HMAC_MD5()

### 15.2.8 Initialization and configuration functions

#### 15.2.8.1 HASH_DeInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void HASH_DeInit ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Deinitializes the HASH peripheral registers to their default reset values.</td>
</tr>
<tr>
<td>Parameters</td>
<td>None.</td>
</tr>
<tr>
<td>Return values</td>
<td>None.</td>
</tr>
<tr>
<td>Notes</td>
<td>None.</td>
</tr>
</tbody>
</table>
15.2.8.2 HASH_Init

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void HASH_Init (HASH_InitTypeDef * HASH_InitStruct)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Initializes the HASH peripheral according to the specified parameters in the HASH_InitStruct structure.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• HASH_InitStruct : pointer to a HASH_InitTypeDef structure that contains the configuration information for the HASH peripheral.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
</tbody>
</table>
| Notes | • the hash processor is reset when calling this function so that the HASH will be ready to compute the message digest of a new message. There is no need to call HASH_Reset() function.  
  • The field HASH_HMACKeyType in HASH_InitTypeDef must be filled only if the algorithm mode is HMAC. |

15.2.8.3 HASH_StructInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void HASH_StructInit (HASH_InitTypeDef * HASH_InitStruct)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Fills each HASH_InitStruct member with its default value.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• HASH_InitStruct : pointer to a HASH_InitTypeDef structure which will be initialized.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• The default values set are: Processor mode is HASH, Algorithm selected is SHA1, Data type selected is 32b and HMAC Key Type is short key.</td>
</tr>
</tbody>
</table>

15.2.8.4 HASH_Reset

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void HASH_Reset (void)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Resets the HASH processor core, so that the HASH will be ready to compute the message digest of a new message.</td>
</tr>
</tbody>
</table>
15.2.9 Message Digest generation functions

15.2.9.1 HASH_SetLastWordValidBitsNbr

Function Name: void HASH_SetLastWordValidBitsNbr ( uint16_t ValidNumber)
Function Description: Configure the Number of valid bits in last word of the message.
Parameters:
- **ValidNumber**: Number of valid bits in last word of the message. This parameter must be a number between 0 and 0x1F.
  - **0x00**: All 32 bits of the last data written are valid
  - **0x01**: Only bit [0] of the last data written is valid
  - **0x02**: Only bits[1:0] of the last data written are valid
  - **0x03**: Only bits[2:0] of the last data written are valid
  - ... (up to 0x1F)
  - **0x1F**: Only bits[30:0] of the last data written are valid

Return values: None.
Notes: The Number of valid bits must be set before to start the message digest competition (in Hash and HMAC) and key treatment (in HMAC).

15.2.9.2 HASH_DataIn

Function Name: void HASH_DataIn ( uint32_t Data)
Function Description: Writes data in the Data Input FIFO.
Parameters:
- **Data**: new data of the message to be processed.

Return values: None.
Notes: None.
15.2.9.3 HASH_GetInFIFOWordsNbr

Function Name  
uint8_t HASH_GetInFIFOWordsNbr ( void )

Function Description  
Returns the number of words already pushed into the IN FIFO.

Parameters  
- None.

Return values  
- The value of words already pushed into the IN FIFO.

Notes  
- None.

15.2.9.4 HASH_GetDigest

Function Name  
void HASH_GetDigest ( HASH_MsgDigest * HASH_MessageDigest)

Function Description  
Provides the message digest result.

Parameters  
- HASH_MessageDigest : pointer to a HASH_MsgDigest structure which will hold the message digest result

Return values  
- None.

Notes  
- In MD5 mode, Data[4] filed of HASH_MsgDigest structure is not used and is read as zero.

15.2.9.5 HASH_StartDigest

Function Name  
void HASH_StartDigest ( void )

Function Description  
Starts the message padding and calculation of the final message.

Parameters  
- None.

Return values  
- None.

Notes  
- None.
15.2.10  Context swapping functions

15.2.10.1  HASH_SaveContext

Function Name      void HASH_SaveContext ( HASH_Context * HASH_ContextSave)
Function Description  Save the Hash peripheral Context.
Parameters
  •  HASH_ContextSave: pointer to a HASH_Context structure that contains the repository for current context.
Return values
  •  None.
Notes
  •  The context can be saved only when no block is currently being processed. So user must wait for DINIS = 1 (the last block has been processed and the input FIFO is empty) or NBW ! = 0 (the FIFO is not full and no processing is ongoing).

15.2.10.2  HASH_RestoreContext

Function Name      void HASH_RestoreContext ( HASH_Context * HASH_ContextRestore)
Function Description  Restore the Hash peripheral Context.
Parameters
  •  HASH_ContextRestore: pointer to a HASH_Context structure that contains the repository for saved context.
Return values
  •  None.
Notes
  •  After calling this function, user can restart the processing from the point where it has been interrupted.

15.2.11  HASH DMA interface Configuration function

15.2.11.1  HASH_DMACmd
Function Name: void HASH_DMACmd (FunctionalState NewState)

Function Description: Enables or disables the HASH DMA interface.

Parameters:
- **NewState**: new state of the selected HASH DMA transfer request. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- The DMA is disabled by hardware after the end of transfer.

15.2.12 Interrupt and flag management functions

15.2.12.1 HASH_ITConfig

Function Name: void HASH_ITConfig (uint8_t HASH_IT, FunctionalState NewState)

Function Description: Enables or disables the specified HASH interrupts.

Parameters:
- **HASH_IT**: specifies the HASH interrupt source to be enabled or disabled. This parameter can be any combination of the following values:
  - **HASH_IT_DINI**: Data Input interrupt
  - **HASH_IT_DCI**: Digest Calculation Completion Interrupt
- **NewState**: new state of the specified HASH interrupt. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.

15.2.12.2 HASH_GetFlagStatus

Function Name: FlagStatus HASH_GetFlagStatus (uint16_t HASH_FLAG)

Function Description: Checks whether the specified HASH flag is set or not.

Parameters:
- **HASH_FLAG**: specifies the HASH flag to check. This parameter can be one of the following values:
  - **HASH_FLAG_DINIS**: Data input interrupt status flag
  - **HASH_FLAG_DCIS**: Digest calculation completion interrupt status flag
  - **HASH_FLAG_BUSY**: Busy flag
  - **HASH_FLAG_DMAS**: DMAS Status flag
15.2.12.3 HASH_ClearFlag

Function Name: void HASH_ClearFlag ( uint16_t HASH_FLAG)
Function Description: Clears the HASH flags.
Parameters:
- HASH_FLAG: specifies the flag to clear. This parameter can be any combination of the following values:
  - HASH_FLAG_DINIS: Data Input Flag
  - HASH_FLAG_DCIS: Digest Calculation Completion Flag

Return values:
- None.

Notes:
- None.

15.2.12.4 HASH_GetITStatus

Function Name: ITStatus HASH_GetITStatus ( uint8_t HASH_IT)
Function Description: Checks whether the specified HASH interrupt has occurred or not.
Parameters:
- HASH_IT: specifies the HASH interrupt source to check. This parameter can be one of the following values:
  - HASH_IT_DINI: Data Input interrupt
  - HASH_IT_DCI: Digest Calculation Completion Interrupt

Return values:
- The new state of HASH_IT (SET or RESET).

Notes:
- None.

15.2.12.5 HASH_ClearITPendingBit
Function Name: void HASH_ClearITPendingBit (uint8_t HASH_IT)

Function Description: Clears the HASH interrupt pending bit(s).

Parameters:
- **HASH_IT**: specifies the HASH interrupt pending bit(s) to clear. This parameter can be any combination of the following values:
  - **HASH_IT_DINI**: Data Input interrupt
  - **HASH_IT_DCI**: Digest Calculation Completion Interrupt

Return values: None.

Notes: None.

15.2.13 High Level SHA1 functions

15.2.13.1 HASH_SHA1

Function Name: ErrorStatus HASH_SHA1 (uint8_t * Input, uint32_t Ilen, uint8_t Output)

Function Description: Compute the HASH SHA1 digest.

Parameters:
- **Input**: pointer to the Input buffer to be treated.
- **Ilen**: length of the Input buffer.
- **Output**: the returned digest

Return values: An ErrorStatus enumeration value:
  - **SUCCESS**: digest computation done
  - **ERROR**: digest computation failed

Notes: None.

15.2.13.2 HMAC_SHA1

Function Name: ErrorStatus HMAC_SHA1 (uint8_t * Key, uint32_t Keylen, uint8_t * Input, uint32_t Ilen, uint8_t Output)

Function Description: Compute the HMAC SHA1 digest.

Parameters:
- **Key**: pointer to the Key used for HMAC.
- **Keylen**: length of the Key used for HMAC.
- **Input**: pointer to the Input buffer to be treated.
- **Ilen**: length of the Input buffer.
15.2.14 High Level MD5 functions

15.2.14.1 HASH_MD5

Function Name

ErrorStatus HASH_MD5 ( uint8_t * Input, uint32_t Ilen, uint8_t Output)

Function Description
Compute the HASH MD5 digest.

Parameters

- Input : pointer to the Input buffer to be treated.
- Ilen : length of the Input buffer.
- Output : the returned digest

Return values

- An ErrorStatus enumeration value:
  - SUCCESS: digest computation done
  - ERROR: digest computation failed

Notes

- None.

15.2.14.2 HMAC_MD5

Function Name

ErrorStatus HMAC_MD5 ( uint8_t * Key, uint32_t Keylen, uint8_t Input, uint32_t Ilen, uint8_t Output)

Function Description
Compute the HMAC MD5 digest.

Parameters

- Key : pointer to the Key used for HMAC.
- Keylen : length of the Key used for HMAC.
- Input : pointer to the Input buffer to be treated.
- Ilen : length of the Input buffer.
- Output : the returned digest

Return values

- An ErrorStatus enumeration value:
  - SUCCESS: digest computation done
  - ERROR: digest computation failed

Notes

- None.
15.3 HASH Firmware driver defines

15.3.1 HASH Firmware driver defines

HASH

HASH_Algo_Selection
- #define: HASH_AlgoSelection_SHA1((uint16_t)0x0000)
  HASH function is SHA1

- #define: HASH_AlgoSelection_MD5((uint16_t)0x0080)
  HASH function is MD5

HASH_Data_Type
- #define: HASH_DataType_32b((uint16_t)0x0000)

- #define: HASH_DataType_16b((uint16_t)0x0010)

- #define: HASH_DataType_8b((uint16_t)0x0020)

- #define: HASH_DataType_1b((uint16_t)0x0030)

HASH_flags_definition
- #define: HASH_FLAG_DINIS((uint16_t)0x0001)
  16 locations are free in the DIN : A new block can be entered into the input buffer.

- #define: HASH_FLAG_DCIS((uint16_t)0x0002)
  Digest calculation complete

- #define: HASH_FLAG_DMAS((uint16_t)0x0004)
DMA interface is enabled (DMAE=1) or a transfer is ongoing

- #define: HASH_FLAG_BUSY((uint16_t)0x0008)
The hash core is Busy: processing a block of data

- #define: HASH_FLAG_DINNE((uint16_t)0x1000)
DIN not empty: The input buffer contains at least one word of data

**HASH_HMAC_Long_key_only_for_HMAC_mode**

- #define: HASH_HMACKeyType_ShortKey((uint32_t)0x00000000)
HMAC Key is <= 64 bytes

- #define: HASH_HMACKeyType_LongKey((uint32_t)0x00010000)
HMAC Key is > 64 bytes

**HASH_interrupts_definition**

- #define: HASH_IT_DINI((uint8_t)0x01)
A new block can be entered into the input buffer (DIN)

- #define: HASH_IT_DCI((uint8_t)0x02)
Digest calculation complete

**HASH_processor_Algorithm_Mode**

- #define: HASH_AlgoMode_HASH((uint16_t)0x0000)
Algorithm is HASH

- #define: HASH_AlgoMode_HMAC((uint16_t)0x0040)
Algorithm is HMAC

### 15.4 HASH Programming Example

The example below explains how to use the HASH peripheral to hash data using HMAC SHA-1 Algorithm. For more examples about HASH configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\HASH:

```c
/* Includes -----------------------------------------------*/
#include "stm32f2xx.h"
```
```c
/* Private define ---------------------------------------------*/
#define INPUT_TAB_SIZE    ((uint32_t)40)
#define KEY_TAB_SIZE      ((uint32_t)40)
/* Private variables ------------------------------------------*/
uint8_t pBuff[INPUT_TAB_SIZE] =
    {0x54,0x68,0x65,0x20,0x68,0x61,0x73,0x68,0x20,0x70,0x72,0x6f,0x63,0x65,0x73,0x73,0x6f,0x72,0x20,0x69,0x73,0x20,0x61,0x20,0x66,0x75,0x6c,0x6c,0x79,0x20,0x63,0x6f,0x6d,0x70,0x6c,0x69,0x61,0x6e,0x74,0x20};
uint8_t pKey[KEY_TAB_SIZE] =
    {0x54,0x68,0x65,0x20,0x68,0x61,0x73,0x68,0x20,0x70,0x72,0x6f,0x63,0x65,0x73,0x73,0x6f,0x72,0x20,0x69,0x73,0x20,0x61,0x20,0x66,0x75,0x6c,0x6c,0x79,0x20,0x63,0x6f,0x6d,0x70,0x6c,0x69,0x61,0x6e,0x74,0x20};
uint8_t Sha1Output[20];
/* Private functions ------------------------------------------*/
/**
 * @brief Main program
 * @param None
 * @retval None
 */
int main(void)
{
    /* Enable HASH clock */
    RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_HASH, ENABLE);

    /* HMAC SHA-1 Digest Computation */
    HMAC_SHA1(pKey, KEY_TAB_SIZE, pBuff, INPUT_TAB_SIZE, Sha1Output);

    while (1)
    {
    }
}
```
16 Inter-integrated circuit interface (I2C)

16.1 I2C Firmware driver registers structures

16.1.1 I2C_TypeDef

*I2C_TypeDef* is defined in the stm32f2xx.h file and contains the I2C registers definition.

**Data Fields**

- `__IO uint16_t CR1`
- `uint16_t RESERVED0`
- `__IO uint16_t CR2`
- `uint16_t RESERVED1`
- `__IO uint16_t OAR1`
- `uint16_t RESERVED2`
- `__IO uint16_t OAR2`
- `uint16_t RESERVED3`
- `__IO uint16_t DR`
- `uint16_t RESERVED4`
- `__IO uint16_t SR1`
- `uint16_t RESERVED5`
- `__IO uint16_t SR2`
- `uint16_t RESERVED6`
- `__IO uint16_t CCR`
- `uint16_t RESERVED7`
- `__IO uint16_t TRISE`
- `uint16_t RESERVED8`

**Field Documentation**

- `__IO uint16_t I2C_TypeDef::CR1`
  - I2C Control register 1, Address offset: 0x00
- `uint16_t I2C_TypeDef::RESERVED0`
  - Reserved, 0x02
- `__IO uint16_t I2C_TypeDef::CR2`
  - I2C Control register 2, Address offset: 0x04
- `uint16_t I2C_TypeDef::RESERVED1`
  - Reserved, 0x06
- `__IO uint16_t I2C_TypeDef::OAR1`
  - I2C Own address register 1, Address offset: 0x08
- `uint16_t I2C_TypeDef::RESERVED2`
  - Reserved, 0x0A
- `__IO uint16_t I2C_TypeDef::OAR2`
  - I2C Own address register 2, Address offset: 0x0C
- `uint16_t I2C_TypeDef::RESERVED3`
  - Reserved, 0x0E
- `__IO uint16_t I2C_TypeDef::DR`
  - I2C Data register, Address offset: 0x10
16.1.2 I2C_InitTypeDef

`I2C_InitTypeDef` is defined in the `stm32f2xx_i2c.h` file and contains the I2C initialization parameters.

### Data Fields

- `uint32_t I2C_ClockSpeed`
- `uint16_t I2C_Mode`
- `uint16_t I2C_DutyCycle`
- `uint16_t I2C_OwnAddress1`
- `uint16_t I2C_Ack`
- `uint16_t I2C_AcknowledgedAddress`

### Field Documentation

- `uint32_t I2C_InitTypeDef::I2C_ClockSpeed`
  - Specifies the clock frequency. This parameter must be set to a value lower than 400kHz
- `uint16_t I2C_InitTypeDef::I2C_Mode`
  - Specifies the I2C mode. This parameter can be a value of `I2C_mode`
- `uint16_t I2C_InitTypeDef::I2C_DutyCycle`
  - Specifies the I2C fast mode duty cycle. This parameter can be a value of `I2C_duty_cycle_in_fast_mode`
- `uint16_t I2C_InitTypeDef::I2C_OwnAddress1`
  - Specifies the first device own address. This parameter can be a 7-bit or 10-bit address.
- `uint16_t I2C_InitTypeDef::I2C_Ack`
  - Enables or disables the acknowledgement. This parameter can be a value of `I2C_acknowledgement`
- `uint16_t I2C_InitTypeDef::I2C_AcknowledgedAddress`
– Specifies if 7-bit or 10-bit address is acknowledged. This parameter can be a value of `I2C_acknowledged_address`.

### 16.2 I2C Firmware driver API description

The following section lists the various functions of the I2C library.

#### 16.2.1 How to use this driver

1. Enable peripheral clock using `RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2Cx, ENABLE)` function for I2C1, I2C2 or I2C3.
2. Enable SDA, SCL and SMBA (when used) GPIO clocks using `RCC_AHBPeriphClockCmd()` function.
3. Peripherals alternate function:
   - Connect the pin to the desired peripheral Alternate Function (AF) using `GPIO_PinAFConfig()` function
   - Configure the desired pin in alternate function by: `GPIO_InitStruct->GPIO_Mode = GPIO_Mode_AF`
   - Select the type, pull-up/pull-down and output speed via `GPIO_PuPd`, `GPIO_OType` and `GPIO_Speed` members
   - Call `GPIO_Init()` function. Recommended configuration is Push-Pull, Pull-up, Open-Drain. Add an external pull up if necessary (typically 4.7 KOhm).
4. Program the Mode, duty cycle, Own address, Ack, Speed and Acknowledged Address using the `I2C_Init()` function.
5. Optionally you can enable/configure the following parameters without re-initialization (i.e there is no need to call again `I2C_Init()` function):
   - Enable the acknowledge feature using `I2C_AcknowledgeConfig()` function
   - Enable the dual addressing mode using `I2C_DualAddressCmd()` function
   - Enable the general call using the `I2C_GeneralCallCmd()` function
   - Enable the clock stretching using `I2C_StretchClockCmd()` function
   - Enable the fast mode duty cycle using the `I2C_FastModeDutyCycleConfig()` function.
   - Configure the NACK position for Master Receiver mode in case of 2 bytes reception using the function `I2C_NACKPositionConfig()`.
   - Enable the PEC Calculation using `I2C_CalculatePEC()` function
   - For SMBus Mode:
     - Enable the Address Resolution Protocol (ARP) using `I2C_ARPCmd()` function
     - Configure the SMBusAlert pin using `I2C_SMBusAlertConfig()` function
6. Enable the NVIC and the corresponding interrupt using the function `I2C_ITConfig()` if you need to use interrupt mode.
7. When using the DMA mode When using DMA mode, I2C interrupts can be used at the same time to control the communication flow (Start/Stop/Ack... events and errors).
   - Configure the DMA using `DMA_Init()` function
   - Activate the needed channel Request using `I2C_DMAInit()` or `I2C_DMAStartTransferCmd()` function.
8. Enable the I2C using the `I2C_Cmd()` function.
9. Enable the DMA using the `DMA_Cmd()` function when using DMA mode in the transfers.
16.2.2 Initialization and configuration

- `I2C_DeInit()`
- `I2C_Init()`
- `I2C_StructInit()`
- `I2C_Cmd()`
- `I2C_GenerateSTART()`
- `I2C_GenerateSTOP()`
- `I2C_Send7bitAddress()`
- `I2C_AcknowledgeConfig()`
- `I2C_OwnAddress2Config()`
- `I2C_DualAddressCmd()`
- `I2C_GeneralCallCmd()`
- `I2C_SoftwareResetCmd()`
- `I2C.StretchClockCmd()`
- `I2C_FastModeDutyCycleConfig()`
- `I2C_NACKPositionConfig()`
- `I2C_SMBusAlertConfig()`
- `I2C_ARPCmd()`

16.2.3 Data transfers

- `I2C_SendData()`
- `I2C_ReceiveData()`

16.2.4 PEC management

- `I2C_TransmitPEC()`
- `I2C_PECPositionConfig()`
- `I2C_CalculatePEC()`
- `I2C_GetPEC()`

16.2.5 DMA transfers management

This section provides functions allowing to configure the I2C DMA channels requests.

- `I2C_DMACmd()`
- `I2C_DMALastTransferCmd()`

16.2.6 Interrupt event and flag management

This section provides functions allowing to configure the I2C Interrupts sources and check or clear the flags or pending bits status. The user should identify which mode will be used in his application to manage the communication: Polling mode, Interrupt mode or DMA mode.
I2C State Monitoring Functions

This I2C driver provides three different ways for I2C state monitoring depending on the application requirements and constraints:

- **Basic state monitoring (Using I2C_CheckEvent() function)** It compares the status registers (SR1 and SR2) content to a given event (can be the combination of one or more flags). It returns SUCCESS if the current status includes the given flags and returns ERROR if one or more flags are missing in the current status.
  
  This function is suitable for most applications as well as for startup activity since the events are fully described in the product reference manual (RM0033). It is also suitable for users who need to define their own events.

  **Limitations:**
  
  If an error occurs (ie. error flags are set besides to the monitored flags), the I2C_CheckEvent() function may return SUCCESS despite the communication hold or corrupted real state. In this case, it is advised to use error interrupts to monitor the error events and handle them in the interrupt IRQ handler. For error management, it is advised to use the following functions:

  - I2C_ITConfig() to configure and enable the error interrupts (I2C_IT_ERR).
  - I2Cx_ER_IRQHandler() which is called when the error interrupt occurs. Where x is the peripheral instance (I2C1, I2C2 ...)
  - I2C_GetFlagStatus() or I2C_GetITStatus() to be called into the I2Cx_ER_IRQHandler() function in order to determine which error occurred.
  - I2C_ClearFlag() or I2C_ClearITPendingBit() and/or I2C_SoftwareResetCmd() and/or I2C_GenerateStop() in order to clear the error flag and source and return to correct communication status.

- **Advanced state monitoring (Using the function I2C_GetLastEvent())** It makes use of the function I2C_GetLastEvent() which returns the image of both status registers in a single word (uint32_t) (Status Register 2 value is shifted left by 16 bits and concatenated to Status Register 1).
  
  This function is suitable for the same applications as above but it allows to overcome the mentioned limitation of I2C_GetFlagStatus() function.

  The returned value could be compared to events already defined in the library (stm32f2xx_i2c.h) or to custom values defined by user. This function is suitable when multiple flags are monitored at the same time.

  At the opposite of I2C_CheckEvent() function, this function allows user to choose when an event is accepted (when all events flags are set and no other flags are set or just when the needed flags are set like I2C_CheckEvent() function).

  **Limitations:**
  
  User may need to define his own events. The same comment concerning the error management is applicable for this function if the user decides to check only regular communication flags (and ignores error flags).

- **Flag-based state monitoring (Using the function I2C_GetFlagStatus())** It makes use of the function I2C_GetFlagStatus() which simply returns the status of one single flag (ie. I2C_FLAG_RXNE ...).

  This function could be used for specific applications or in debug phase.

  It is suitable when only one flag checking is needed (most I2C events are monitored through multiple flags).

  **Limitations:**
  
  When calling this function, the Status register is accessed. Some flags are cleared when the status register is accessed. So checking the status of one Flag, may clear other ones.

  This function may need to be called twice or more in order to monitor one single event.

  For detailed description of Events, please refer to section I2C_Events in stm32f2xx_i2c.h file.

- I2C_ReadRegister()
### 16.2.7 Initialization and configuration functions

#### 16.2.7.1 I2C_DeInit

**Function Name**

```c
void I2C_DeInit (I2C_TypeDef *I2Cx)
```

**Function Description**

Deinitialize the I2Cx peripheral registers to their default reset values.

**Parameters**

- `I2Cx`: where x can be 1, 2 or 3 to select the I2C peripheral.

**Return values**

- None.

**Notes**

- None.

#### 16.2.7.2 I2C_Init

**Function Name**

```c
void I2C_Init (I2C_TypeDef *I2Cx, I2C_InitTypeDef *I2C_InitStruct)
```

**Function Description**

Initializes the I2Cx peripheral according to the specified parameters in the I2C_InitStruct.

**Parameters**

- `I2Cx`: where x can be 1, 2 or 3 to select the I2C peripheral.
- `I2C_InitStruct`: pointer to a I2C_InitStruct structure that contains the configuration information for the specified I2C peripheral.

**Return values**

- None.

**Notes**

- To use the I2C at 400 KHz (in fast mode), the PCLK1 frequency (I2C peripheral input clock) must be a multiple of 10 MHz.

#### 16.2.7.3 I2C_StructInit
Function Name: void I2C_StructInit (I2C_InitTypeDef * I2C_InitStruct)

Function Description: Fills each I2C_InitStruct member with its default value.

Parameters:
- I2C_InitStruct: pointer to an I2C_InitTypeDef structure which will be initialized.

Return values:
- None.

Notes:
- None.

16.2.7.4 I2C_Cmd

Function Name: void I2C_Cmd (I2C_TypeDef * I2Cx, FunctionalState NewState)

Function Description: Enables or disables the specified I2C peripheral.

Parameters:
- I2Cx: where x can be 1, 2 or 3 to select the I2C peripheral.
- NewState: new state of the I2C peripheral. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.

16.2.7.5 I2C_GenerateSTART

Function Name: void I2C_GenerateSTART (I2C_TypeDef * I2Cx, FunctionalState NewState)

Function Description: Generates I2Cx communication START condition.

Parameters:
- I2Cx: where x can be 1, 2 or 3 to select the I2C peripheral.
- NewState: new state of the I2C START condition generation. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.
16.2.7.6  I2C_GenerateSTOP

Function Name     void I2C_GenerateSTOP ( I2C_TypeDef * I2Cx, FunctionalState NewState)
Function Description  Generates I2Cx communication STOP condition.
Parameters
  •  I2Cx : where x can be 1, 2 or 3 to select the I2C peripheral.
  •  NewState : new state of the I2C STOP condition generation. This parameter can be: ENABLE or DISABLE.
Return values
  •  None.
Notes
  •  None.

16.2.7.7  I2C_Send7bitAddress

Function Name     void I2C_Send7bitAddress ( I2C_TypeDef * I2Cx, uint8_t Address, uint8_t I2C_Direction)
Function Description  Transmits the address byte to select the slave device.
Parameters
  •  I2Cx : where x can be 1, 2 or 3 to select the I2C peripheral.
  •  Address : specifies the slave address which will be transmitted
  •  I2C_Direction : specifies whether the I2C device will be a Transmitter or a Receiver. This parameter can be one of the following values
    –  I2C_Direction_Transmitter : Transmitter mode
    –  I2C_Direction_Receiver : Receiver mode
Return values
  •  None.
Notes
  •  None.

16.2.7.8  I2C_AcknowledgeConfig

Function Name     void I2C_AcknowledgeConfig ( I2C_TypeDef * I2Cx, FunctionalState NewState)
Function Description  Enables or disables the specified I2C acknowledge feature.
### Parameters
- **I2Cx**: where x can be 1, 2 or 3 to select the I2C peripheral.
- **NewState**: new state of the I2C Acknowledgement. This parameter can be: ENABLE or DISABLE.

### Return values
- **None**.

### Notes
- None.

#### 16.2.7.9 I2C_OwnAddress2Config

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void I2C_OwnAddress2Config (I2C_TypeDef * I2Cx, uint8_t Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Configures the specified I2C own address2.</td>
</tr>
</tbody>
</table>
| Parameters                     | • **I2Cx**: where x can be 1, 2 or 3 to select the I2C peripheral.  
                                 | • **Address**: specifies the 7bit I2C own address2.             |
| Return values                  | • None.                                                       |
| Notes                          | • None.                                                       |

#### 16.2.7.10 I2C_DualAddressCmd

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void I2C_DualAddressCmd (I2C_TypeDef * I2Cx, FunctionalState NewState)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Enables or disables the specified I2C dual addressing mode.</td>
</tr>
</tbody>
</table>
| Parameters                     | • **I2Cx**: where x can be 1, 2 or 3 to select the I2C peripheral.  
                                 | • **NewState**: new state of the I2C dual addressing mode. This parameter can be: ENABLE or DISABLE. |
| Return values                  | • None.                                                           |
| Notes                          | • None.                                                           |

#### 16.2.7.11 I2C_GeneralCallCmd
16.2.7.12 I2C_SoftwareResetCmd

Function Name: void I2C_SoftwareResetCmd (I2C_TypeDef * I2Cx, FunctionalState NewState)

Function Description: Enables or disables the specified I2C software reset.

Parameters:
- I2Cx: where x can be 1, 2 or 3 to select the I2C peripheral.
- NewState: new state of the I2C software reset. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- When software reset is enabled, the I2C IOs are released (this can be useful to recover from bus errors).

16.2.7.13 I2C_StretchClockCmd

Function Name: void I2C_StretchClockCmd (I2C_TypeDef * I2Cx, FunctionalState NewState)

Function Description: Enables or disables the specified I2C Clock stretching.

Parameters:
- I2Cx: where x can be 1, 2 or 3 to select the I2C peripheral.
- NewState: new state of the I2C Clock stretching. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.
16.2.7.14 I2C_FastModeDutyCycleConfig

Function Name
void I2C_FastModeDutyCycleConfig ( I2C_TypeDef * I2Cx, uint16_t I2C_DutyCycle)

Function Description
Selects the specified I2C fast mode duty cycle.

Parameters
- I2Cx : where x can be 1, 2 or 3 to select the I2C peripheral.
- I2C_DutyCycle : specifies the fast mode duty cycle. This parameter can be one of the following values:
  - I2C_DutyCycle_2 : I2C fast mode Tlow/Thigh = 2
  - I2C_DutyCycle_16_9 : I2C fast mode Tlow/Thigh = 16/9

Return values
- None.

Notes
- None.

16.2.7.15 I2C_NACKPositionConfig

Function Name
void I2C_NACKPositionConfig ( I2C_TypeDef * I2Cx, uint16_t I2C_NACKPosition)

Function Description
Selects the specified I2C NACK position in master receiver mode.

Parameters
- I2Cx : where x can be 1, 2 or 3 to select the I2C peripheral.
- I2C_NACKPosition : specifies the NACK position. This parameter can be one of the following values:
  - I2C_NACKPosition_Next : indicates that the next byte will be the last received byte.
  - I2C_NACKPosition_Current : indicates that current byte is the last received byte.

Return values
- None.

Notes
- This function is useful in I2C Master Receiver mode when the number of data to be received is equal to 2. In this case, this function should be called (with parameter I2C_NACKPosition_Next) before data reception starts, as described in the 2-byte reception procedure recommended in Reference Manual in Section: Master receiver.
- This function configures the same bit (POS) as I2C_PECPositionConfig() but is intended to be used in I2C mode while I2C_PECPositionConfig() is intended to used in SMBUS mode.
16.2.7.16 I2C_SMBusAlertConfig

Function Name: void I2C_SMBusAlertConfig (I2C_TypeDef *I2Cx, uint16_t I2C_SMBusAlert)

Function Description: Drives the SMBusAlert pin high or low for the specified I2C.

Parameters:
- I2Cx: where x can be 1, 2 or 3 to select the I2C peripheral.
- I2C_SMBusAlert: specifies SMBAlert pin level. This parameter can be one of the following values:
  - I2C_SMBusAlert_Low: SMBAlert pin driven low
  - I2C_SMBusAlert_High: SMBAlert pin driven high

Return values:
- None.

Notes:
- None.

16.2.7.17 I2C_ARPCmd

Function Name: void I2C_ARPCmd (I2C_TypeDef *I2Cx, FunctionalState NewState)

Function Description: Enables or disables the specified I2C ARP.

Parameters:
- I2Cx: where x can be 1, 2 or 3 to select the I2C peripheral.
- NewState: new state of the I2Cx ARP. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.

16.2.8 Data transfers functions

16.2.8.1 I2C_SendData

Function Name: void I2C_SendData (I2C_TypeDef *I2Cx, uint8_t Data)
16.2.8.2 I2C_ReceiveData

Function Name  uint8_t I2C_ReceiveData ( I2C_TypeDef * I2Cx)
Function Description Returns the most recent received data by the I2Cx peripheral.
Parameters  • I2Cx : where x can be 1, 2 or 3 to select the I2C peripheral.
Return values  • The value of the received data.
Notes  • None.

16.2.9 PEC management functions

16.2.9.1 I2C_TransmitPEC

Function Name  void I2C_TransmitPEC ( I2C_TypeDef * I2Cx, FunctionalState NewState)
Function Description Enables or disables the specified I2C PEC transfer.
Parameters  • I2Cx : where x can be 1, 2 or 3 to select the I2C peripheral.
  • NewState : new state of the I2C PEC transmission. This parameter can be: ENABLE or DISABLE.
Return values  • None.
Notes  • None.

16.2.9.2 I2C_PECPositionConfig
Function Name: void I2C_PECPositionConfig (I2C_TypeDef * I2Cx, uint16_t I2C_PECPosition)

Function Description: Selects the specified I2C PEC position.

Parameters:
- I2Cx: where x can be 1, 2 or 3 to select the I2C peripheral.
- I2C_PECPosition: specifies the PEC position. This parameter can be one of the following values:
  - I2C_PECPosition_Next: indicates that the next byte is PEC
  - I2C_PECPosition_Current: indicates that current byte is PEC

Return values: None.

Notes:
- This function configures the same bit (POS) as I2C_NACKPositionConfig() but is intended to be used in SMBUS mode while I2C_NACKPositionConfig() is intended to be used in I2C mode.

16.2.9.3 I2C_CalculatePEC

Function Name: void I2C_CalculatePEC (I2C_TypeDef * I2Cx, FunctionalState NewState)

Function Description: Enables or disables the PEC value calculation of the transferred bytes.

Parameters:
- I2Cx: where x can be 1, 2 or 3 to select the I2C peripheral.
- NewState: new state of the I2Cx PEC value calculation. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

16.2.9.4 I2C_GetPEC

Function Name: uint8_t I2C_GetPEC (I2C_TypeDef * I2Cx)

Function Description: Returns the PEC value for the specified I2C.

Parameters:
- I2Cx: where x can be 1, 2 or 3 to select the I2C peripheral.
16.2.10 DMA transfers management functions

16.2.10.1 I2C_DMACmd

Function Name: void I2C_DMACmd (I2C_TypeDef * I2Cx, FunctionalState NewState)
Function Description: Enables or disables the specified I2C DMA requests.
Parameters:
- I2Cx: where x can be 1, 2 or 3 to select the I2C peripheral.
- NewState: new state of the I2C DMA transfer. This parameter can be: ENABLE or DISABLE.
Return values: None.
Notes: None.

16.2.10.2 I2C_DMALastTransferCmd

Function Name: void I2C_DMALastTransferCmd (I2C_TypeDef * I2Cx, FunctionalState NewState)
Function Description: Specifies that the next DMA transfer is the last one.
Parameters:
- I2Cx: where x can be 1, 2 or 3 to select the I2C peripheral.
- NewState: new state of the I2C DMA last transfer. This parameter can be: ENABLE or DISABLE.
Return values: None.
Notes: None.

16.2.11 Interrupt event and flag management functions

16.2.11.1 I2C_ReadRegister

Return values: The PEC value.
Notes: None.
Function Name: `uint16_t I2C_ReadRegister (I2C_TypeDef * I2Cx, uint8_t I2C_Register)`

Function Description: Reads the specified I2C register and returns its value.

Parameters:
- `I2C_Register`: specifies the register to read. This parameter can be one of the following values:
  - `I2C_Register_CR1`: CR1 register.
  - `I2C_Register_CR2`: CR2 register.
  - `I2C_Register_OAR1`: OAR1 register.
  - `I2C_Register_OAR2`: OAR2 register.
  - `I2C_Register_DR`: DR register.
  - `I2C_Register_SR1`: SR1 register.
  - `I2C_Register_SR2`: SR2 register.
  - `I2C_Register_CCR`: CCR register.
  - `I2C_Register_TRISE`: TRISE register.

Return values:
- The value of the read register.

Notes:
- None.

### 16.2.11.2 I2C_ITConfig

Function Name: `void I2C_ITConfig (I2C_TypeDef * I2Cx, uint16_t I2C_IT, FunctionalState NewState)`

Function Description: Enables or disables the specified I2C interrupts.

Parameters:
- `I2Cx`: where x can be 1, 2 or 3 to select the I2C peripheral.
- `I2C_IT`: specifies the I2C interrupts sources to be enabled or disabled. This parameter can be any combination of the following values:
  - `I2C_IT_BUF`: Buffer interrupt mask
  - `I2C_IT_EVT`: Event interrupt mask
  - `I2C_IT_ERR`: Error interrupt mask
- `NewState`: new state of the specified I2C interrupts. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.

### 16.2.11.3 I2C_CheckEvent
### Function Name

**ErrorStatus I2C_CheckEvent (I2C_TypeDef * I2Cx, uint32_t I2C_EVENT)**

Checks whether the last I2Cx Event is equal to the one passed as parameter.

### Parameters

- **I2Cx**: where x can be 1, 2 or 3 to select the I2C peripheral.
- **I2C_EVENT**: specifies the event to be checked. This parameter can be one of the following values:
  - `I2C_EVENT_SLAVE_TRANSMITTER_ADDRESS_MATCHED`: EV1
  - `I2C_EVENT_SLAVE_RECEIVER_ADDRESS_MATCHED`: EV1
  - `I2C_EVENT_SLAVE_TRANSMITTER_SECONDADDRESS_MAPPED`: EV1
  - `I2C_EVENT_SLAVE_RECEIVER_SECONDADDRESS_MAPPED`: EV1
  - `I2C_EVENT_SLAVE_GENERALCALLADDRESS_MAPPED`: EV1
  - `I2C_EVENT_SLAVE_BYTE_RECEIVED`: EV2
  - `I2C_EVENT_SLAVE_BYTE_RECEIVED | I2C_FLAG_DUALF`: EV2
  - `I2C_EVENT_SLAVE_BYTE_RECEIVED | I2C_FLAG_GENCALL`: EV2
  - `I2C_EVENT_SLAVE_BYTE_TRANSMITTED`: EV3
  - `I2C_EVENT_SLAVE_BYTE_TRANSMITTED | I2C_FLAG_DUALF`: EV3
  - `I2C_EVENT_SLAVE_BYTE_TRANSMITTED | I2C_FLAG_GENCALL`: EV3
  - `I2C_EVENT_SLAVE_ACK_FAILURE`: EV3_2
  - `I2C_EVENT_SLAVE_STOP_DETECTED`: EV4
  - `I2C_EVENT_MASTER_MODE_SELECT`: EV5
  - `I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED`: EV6
  - `I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED`: EV6
  - `I2C_EVENT_MASTER_BYTE_RECEIVED`: EV7
  - `I2C_EVENT_MASTER_BYTE_TRANSMITTING`: EV8
  - `I2C_EVENT_MASTER_BYTE_TRANSMITTED`: EV8_2
  - `I2C_EVENT_MASTER_MODE_ADDRESS10`: EV9

### Return values

- An ErrorStatus enumeration value:
  - **SUCCESS**: Last event is equal to the I2C_EVENT
  - **ERROR**: Last event is different from the I2C_EVENT

### Notes

- For detailed description of Events, please refer to section I2C_Events in stm32f2xx_i2c.h file.

---

16.2.11.4 **I2C_GetLastEvent**
### 16.2.11.5 I2C_GetFlagStatus

**Function Name**  
`FlagStatus I2C_GetFlagStatus (I2C_TypeDef * I2Cx, uint32_t I2C_FLAG)`

**Function Description**  
Checks whether the specified I2C flag is set or not.

**Parameters**  
- `I2Cx` : where x can be 1, 2 or 3 to select the I2C peripheral.
- `I2C_FLAG` : specifies the flag to check. This parameter can be one of the following values:
  - `I2C_FLAG_DUALF` : Dual flag (Slave mode)
  - `I2C_FLAG_SMBHOST` : SMBus host header (Slave mode)
  - `I2C_FLAG_SMBDEFAULT` : SMBus default header (Slave mode)
  - `I2C_FLAG_GENCALL` : General call header flag (Slave mode)
  - `I2C_FLAG_TRA` : Transmitter/Receiver flag
  - `I2C_FLAG_BUSY` : Bus busy flag
  - `I2C_FLAG_MSL` : Master/Slave flag
  - `I2C_FLAG_SMBALERT` : SMBus Alert flag
  - `I2C_FLAG_TIMEOUT` : Timeout or Tlow error flag
  - `I2C_FLAG_PECERR` : PEC error in reception flag
  - `I2C_FLAG_OVR` : Overrun/Underrun flag (Slave mode)
  - `I2C_FLAG_AF` : Acknowledge failure flag
  - `I2C_FLAG_ARLO` : Arbitration lost flag (Master mode)
  - `I2C_FLAG_BERR` : Bus error flag
  - `I2C_FLAG_TXE` : Data register empty flag (Transmitter)
  - `I2C_FLAG_RXNE` : Data register not empty (Receiver) flag
  - `I2C_FLAG_STOPF` : Stop detection flag (Slave mode)
  - `I2C_FLAG_ADD10` : 10-bit header sent flag (Master mode)
  - `I2C_FLAG_BTF` : Byte transfer finished flag
  - `I2C_FLAG_ADDR` : Address sent flag (Master mode)

**Return values**  
- The new state of `I2C_FLAG` (SET or RESET).
16.2.11.6 I2C_ClearFlag

Function Name: `void I2C_ClearFlag (I2C_TypeDef *I2Cx, uint32_t I2C_FLAG)`

Function Description: Clears the I2Cx’s pending flags.

Parameters:
- `I2Cx`: where x can be 1, 2 or 3 to select the I2C peripheral.
- `I2C_FLAG`: specifies the flag to clear. This parameter can be any combination of the following values:
  - `I2C_FLAG_SMBALERT`: SMBus Alert flag
  - `I2C_FLAG_TIMEOUT`: Timeout or Tlow error flag
  - `I2C_FLAG_PECERR`: PEC error in reception flag
  - `I2C_FLAG_OVR`: Overrun/Underrun flag (Slave mode)
  - `I2C_FLAG_AF`: Acknowledge failure flag
  - `I2C_FLAG_ARLO`: Arbitration lost flag (Master mode)
  - `I2C_FLAG_BERR`: Bus error flag

Return values:
- None.

Notes:
- STOPF (STOP detection) is cleared by software sequence: a read operation to I2C_SR1 register (`I2C_GetFlagStatus()`) followed by a write operation to I2C_CR1 register (`I2C_Cmd()` to re-enable the I2C peripheral).
- ADD10 (10-bit header sent) is cleared by software sequence: a read operation to I2C_SR1 (`I2C_GetFlagStatus()`), followed by writing the second byte of the address in DR register.
- BTF (Byte Transfer Finished) is cleared by software sequence: a read operation to I2C_SR1 register (`I2C_GetFlagStatus()`) followed by a read/write to I2C_DR register (`I2C_SendData()`).
- ADDR (Address sent) is cleared by software sequence: a read operation to I2C_SR1 register (`I2C_GetFlagStatus()`) followed by a read operation to I2C_SR2 register (`((void)(I2Cx->SR2))`).
- SB (Start Bit) is cleared software sequence: a read operation to I2C_SR1 register (`I2C_GetFlagStatus()`) followed by a write operation to I2C_DR register (`I2C_SendData()`).

16.2.11.7 I2C_GetITStatus
Function Name: ITStatus I2C_GetIStatus (I2C_TypeDef * I2Cx, uint32_t I2C_IT)

Function Description: Checks whether the specified I2C interrupt has occurred or not.

Parameters:
- **I2Cx**: where x can be 1, 2 or 3 to select the I2C peripheral.
- **I2C_IT**: specifies the interrupt source to check. This parameter can be one of the following values:
  - I2C_IT_SMBALERT: SMBus Alert flag
  - I2C_IT_TIMEOUT: Timeout or Tlow error flag
  - I2C_IT_PECERR: PEC error in reception flag
  - I2C_IT_OVR: Overrun/Underrun flag (Slave mode)
  - I2C_IT_AF: Acknowledge failure flag
  - I2C_IT_ARLO: Arbitration lost flag (Master mode)
  - I2C_IT_BERR: Bus error flag
  - I2C_IT_TXE: Data register empty flag (Transmitter)
  - I2C_IT_RXNE: Data register not empty (Receiver) flag
  - I2C_IT_STOPF: Stop detection flag (Slave mode)
  - I2C_IT_ADD10: 10-bit header sent flag (Master mode)
  - I2C_IT_BTF: Byte transfer finished flag
  - I2C_IT_ADDR: Address sent flag (Master mode) "ADSL" Address matched flag (Slave mode) "ENDAD"
  - I2C_IT_SB: Start bit flag (Master mode)

Return values:
- The new state of I2C_IT (SET or RESET).

Notes:
- None.

16.2.11.8 I2C_ClearITPendingBit

Function Name: void I2C_ClearITPendingBit (I2C_TypeDef * I2Cx, uint32_t I2C_IT)

Function Description: Clears the I2Cx's interrupt pending bits.

Parameters:
- **I2Cx**: where x can be 1, 2 or 3 to select the I2C peripheral.
- **I2C_IT**: specifies the interrupt pending bit to clear. This parameter can be any combination of the following values:
  - I2C_IT_SMBALERT: SMBus Alert interrupt
  - I2C_IT_TIMEOUT: Timeout or Tlow error interrupt
  - I2C_IT_PECERR: PEC error in reception interrupt
  - I2C_IT_OVR: Overrun/Underrun interrupt (Slave mode)
  - I2C_IT_AF: Acknowledge failure interrupt
  - I2C_IT_ARLO: Arbitration lost interrupt (Master mode)
  - I2C_IT_BERR: Bus error interrupt

Return values:
- None.

Notes:
- STOPF (STOP detection) is cleared by software sequence: a read operation to I2C_SR1 register (I2C_GetITStatus()) followed by a write operation to I2C_CR1 register (I2C_Cmd())

328/634 DocID 18540 Rev 1
to re-enable the I2C peripheral).
- ADD10 (10-bit header sent) is cleared by software sequence: a read operation to I2C_SR1 (I2C_GetITStatus()) followed by writing the second byte of the address in I2C_DR register.
- BTF (Byte Transfer Finished) is cleared by software sequence: a read operation to I2C_SR1 register (I2C_GetITStatus()) followed by a read/write to I2C_DR register (I2C_SendData()).
- ADDR (Address sent) is cleared by software sequence: a read operation to I2C_SR1 register (I2C_GetITStatus()) followed by a read operation to I2C_SR2 register ((void)(I2Cx->SR2)).
- SB (Start Bit) is cleared by software sequence: a read operation to I2C_SR1 register (I2C_GetITStatus()) followed by a write operation to I2C_DR register (I2C_SendData()).

16.3 I2C Firmware driver defines

16.3.1 I2C Firmware driver defines

I2C

I2C_acknowledged_address
- #define: I2C_AcknowledgedAddress_7bit((uint16_t)0x4000)

- #define: I2C_AcknowledgedAddress_10bit((uint16_t)0xC000)

I2C_acknowledgement
- #define: I2C_Ack_Enable((uint16_t)0x0400)

- #define: I2C_Ack_Disable((uint16_t)0x0000)

I2C_duty_cycle_in_fast_mode
- #define: I2C_DutyCycle_16_9((uint16_t)0x4000)

I2C fast mode Tlow/Thigh = 16/9

- #define: I2C_DutyCycle_2((uint16_t)0xBFFF)
Inter-integrated circuit interface (I2C)

I2C fast mode Tlow/Thigh = 2

I2C_Events

- #define: `I2C_EVENT_MASTER_MODE_SELECT((uint32_t)0x00030001)`
  Communication start

- #define: `I2C_EVENT_MASTER_TRANSMITTER_MODE_SELECTED((uint32_t)0x00070082)`
  After checking on EV5 (start condition correctly released on the bus), the master sends the address of the slave(s) with which it will communicate (I2C_Send7bitAddress() function, it also determines the direction of the communication: Master transmitter or Receiver). Then the master has to wait that a slave acknowledges his address. If an acknowledge is sent on the bus, one of the following events will be set:

  - #define: `I2C_EVENT_MASTER_RECEIVER_MODE_SELECTED((uint32_t)0x00030002)`
  
  - #define: `I2C_EVENT_MASTER_MODE_ADDRESS10((uint32_t)0x00030008)`

  - #define: `I2C_EVENT_MASTER_BYTE_RECEIVED((uint32_t)0x00030040)`
    If a communication is established (START condition generated and slave address acknowledged) then the master has to check on one of the following events for communication procedures:

    - #define: `I2C_EVENT_MASTER_BYTE_TRANSMITTING((uint32_t)0x00070080)`
    
    - #define: `I2C_EVENT_MASTER_BYTE_TRANSMITTED((uint32_t)0x00070084)`

    - #define: `I2C_EVENT_SLAVE_RECEIVER_ADDRESS_MATCHED((uint32_t)0x00020002)`
      Communication start events

    - #define: `I2C_EVENT_SLAVE_TRANSMITTER_ADDRESS_MATCHED((uint32_t)0x00060008)`
```c
#define I2C_EVENT_SLAVE_RECEIVER_SECONDADDRESS_MATCHED((uint32_t)0x00820000)

#define I2C_EVENT_SLAVE_TRANSMITTER_SECONDADDRESS_MATCHED((uint32_t)0x00860080)

#define I2C_EVENT_SLAVE_GENERALCALLADDRESS_MATCHED((uint32_t)0x00120000)

#define I2C_EVENT_SLAVE_BYTE_RECEIVED((uint32_t)0x00020040)
Wait on one of these events when EV1 has already been checked and:

#define I2C_EVENT_SLAVE_STOP_DETECTED((uint32_t)0x00000010)

#define I2C_EVENT_SLAVE_BYTE_TRANSMITTED((uint32_t)0x00060084)

#define I2C_EVENT_SLAVE_BYTE_TRANSMITTING((uint32_t)0x00060080)

#define I2C_EVENT_SLAVE_ACK_FAILURE((uint32_t)0x00000400)

i2c_flags_definition

#define I2C_FLAG_DUALF((uint32_t)0x00800000)

#define I2C_FLAG_SMBHOST((uint32_t)0x00400000)
```
• \texttt{#define I2C\_FLAG\_SMBDEFAULT((uint32\_t)0x00200000)}

• \texttt{#define I2C\_FLAG\_GENCALL((uint32\_t)0x00100000)}

• \texttt{#define I2C\_FLAG\_TRA((uint32\_t)0x00040000)}

• \texttt{#define I2C\_FLAG\_BUSY((uint32\_t)0x00020000)}

• \texttt{#define I2C\_FLAG\_MSL((uint32\_t)0x00010000)}

• \texttt{#define I2C\_FLAG\_SMBALERT((uint32\_t)0x10008000)}

• \texttt{#define I2C\_FLAG\_TIMEOUT((uint32\_t)0x10004000)}

• \texttt{#define I2C\_FLAG\_PECERR((uint32\_t)0x10001000)}

• \texttt{#define I2C\_FLAG\_OVR((uint32\_t)0x10000800)}

• \texttt{#define I2C\_FLAG\_AF((uint32\_t)0x10000400)}

• \texttt{#define I2C\_FLAG\_ARLO((uint32\_t)0x10000200)}

• \texttt{#define I2C\_FLAG\_BERR((uint32\_t)0x10000100)}
• `#define I2C_FLAG_TXE((uint32_t)0x10000080)`

• `#define I2C_FLAG_RXNE((uint32_t)0x10000040)`

• `#define I2C_FLAG_STOPF((uint32_t)0x10000010)`

• `#define I2C_FLAG_ADD10((uint32_t)0x10000008)`

• `#define I2C_FLAG_BTF((uint32_t)0x10000004)`

• `#define I2C_FLAG_ADDR((uint32_t)0x10000002)`

• `#define I2C_FLAG_SB((uint32_t)0x10000001)`

I2C_interrupts_definition

• `#define I2C_IT_BUF((uint16_t)0x0400)`

• `#define I2C_IT_EVT((uint16_t)0x0200)`

• `#define I2C_IT_ERR((uint16_t)0x0100)`

• `#define I2C_IT_SMBALERT((uint32_t)0x01008000)`

• `#define I2C_IT_TIMEOUT((uint32_t)0x01004000)`

• `#define I2C_IT_TIMEOUT((uint32_t)0x01004000)`
#define: `I2C_IT_PECERR((uint32_t)0x01001000)`

#define: `I2C_IT_OVR((uint32_t)0x01000800)`

#define: `I2C_IT_AF((uint32_t)0x01000400)`

#define: `I2C_IT_ARLO((uint32_t)0x01000200)`

#define: `I2C_IT_BERR((uint32_t)0x01000100)`

#define: `I2C_IT_TXE((uint32_t)0x06000080)`

#define: `I2C_IT_RXNE((uint32_t)0x06000040)`

#define: `I2C_IT_STOPF((uint32_t)0x02000010)`

#define: `I2C_IT_ADD10((uint32_t)0x02000008)`

#define: `I2C_IT_BTF((uint32_t)0x02000004)`

#define: `I2C_IT_ADDR((uint32_t)0x02000002)`

#define: `I2C_IT_SB((uint32_t)0x02000001)`
I2C_mode
- #define: I2C_Mode_I2C((uint16_t)0x0000)
- #define: I2C_Mode_SMBusDevice((uint16_t)0x0002)
- #define: I2C_Mode_SMBusHost((uint16_t)0x000A)

I2C_NACK_position
- #define: I2C_NACKPosition_Next((uint16_t)0x0800)
- #define: I2C_NACKPosition_Current((uint16_t)0xF7FF)

I2C_PEC_position
- #define: I2C_PECPosition_Next((uint16_t)0x0800)
- #define: I2C_PECPosition_Current((uint16_t)0xF7FF)

I2C_registers
- #define: I2C_Register_CR1((uint8_t)0x00)
- #define: I2C_Register_CR2((uint8_t)0x04)
- #define: I2C_Register_OAR1((uint8_t)0x08)
- #define: I2C_Register_OAR2((uint8_t)0x0C)
• #define: \textit{I2C\_Register\_DR}((uint8\_t)0x10)

• #define: \textit{I2C\_Register\_SR1}((uint8\_t)0x14)

• #define: \textit{I2C\_Register\_SR2}((uint8\_t)0x18)

• #define: \textit{I2C\_Register\_CCR}((uint8\_t)0x1C)

• #define: \textit{I2C\_Register\_TRISE}((uint8\_t)0x20)

\textit{I2C\_SMBus\_alert\_pin\_level}

• #define: \textit{I2C\_SMBus\_Alert\_Low}((uint16\_t)0x2000)

• #define: \textit{I2C\_SMBus\_Alert\_High}((uint16\_t)0xDFFF)

\textit{I2C\_transfer\_direction}

• #define: \textit{I2C\_Direction\_Transmitter}((uint8\_t)0x00)

• #define: \textit{I2C\_Direction\_Receiver}((uint8\_t)0x01)

16.4 \textbf{I2C Programming Example}

The example below provides a typical configuration of the I2C peripheral. For more examples about I2C configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx\StdPeriph_Examples\I2C\
/* To use the I2C at 400 KHz (in fast mode), the PCLK1 frequency (I2C peripheral input clock) must be a multiple of 10 MHz */

#ifndef FAST_I2C_MODE
    #define I2C_SPEED      340000
    #define I2C_DUTYCYCLE  I2C_DutyCycle_16_9
#else
    #define I2C_SPEED      100000
    #define I2C_DUTYCYCLE  I2C_DutyCycle_2
#endif /* FAST_I2C_MODE*/

I2C_InitTypeDef I2C_InitStructure;

/* Enable I2C1 clock */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_I2C1, ENABLE);

/* CODEC_I2C peripheral configuration */
I2C_InitStructure.I2C_Mode = I2C_Mode_I2C;
I2C_InitStructure.I2C_DutyCycle = I2C_DUTYCYCLE;
I2C_InitStructure.I2C_OwnAddress1 = 0x33;
I2C_InitStructure.I2C_Ack = I2C_Ack_Enable;
I2C_InitStructure.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
I2C_InitStructure.I2C_ClockSpeed = I2C_SPEED;

/* Enable the I2C peripheral */
I2C_Cmd(I2C1, ENABLE);
I2C_Init(I2C1, &I2C_InitStructure);
17 Independent watchdog (IWDG)

17.1 IWDG Firmware driver registers structures

17.1.1 IWDG_TypeDef

*IWDG_TypeDef* is defined in the stm32f2xx.h file and contains the IWDG registers definition.

Data Fields

- __IO uint32_t KR
- __IO uint32_t PR
- __IO uint32_t RLR
- __IO uint32_t SR

Field Documentation

- __IO uint32_t IWDG_TypeDef::KR
  - IWDG Key register, Address offset: 0x00
- __IO uint32_t IWDG_TypeDef::PR
  - IWDG Prescaler register, Address offset: 0x04
- __IO uint32_t IWDG_TypeDef::RLR
  - IWDG Reload register, Address offset: 0x08
- __IO uint32_t IWDG_TypeDef::SR
  - IWDG Status register, Address offset: 0x0C

17.2 IWDG Firmware driver API description

The following section lists the various functions of the IWDG library.

IWDG features

The IWDG can be started by either software or hardware (configurable through option byte).

The IWDG is clocked by its own dedicated low-speed clock (LSI) and thus stays active even if the main clock fails. Once the IWDG is started, the LSI is forced ON and cannot be disabled (LSI cannot be disabled too), and the counter starts counting down from the reset value of 0xFFF. When it reaches the end of count value (0x000) a system reset is generated. The IWDG counter should be reloaded at regular intervals to prevent an MCU reset.

The IWDG is implemented in the VDD voltage domain that is still functional in STOP and STANDBY mode (IWDG reset can wake-up from STANDBY).

IWDGRST flag in RCC_CSR register can be used to inform when a IWDG reset occurs.

Min-max timeout value @32KHz (LSI): ~125us / ~32.7s The IWDG timeout may vary due to LSI frequency dispersion. STM32F2xx devices provide the capability to measure the LSI frequency (LSI clock connected internally to TIM5 CH4 input capture). The measured value
can be used to have an IWDG timeout with an acceptable accuracy. For more information, please refer to the STM32F2xx Reference manual.

How to use this driver
1. Enable write access to IWDG_PR and IWDG_RLR registers using IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable) function
2. Configure the IWDG prescaler using IWDG_SetPrescaler() function
3. Configure the IWDG counter value using IWDG_SetReload() function. This value will be loaded in the IWDG counter each time the counter is reloaded, then the IWDG will start counting down from this value.
4. Start the IWDG using IWDG_Enable() function, when the IWDG is used in software mode (no need to enable the LSI, it will be enabled by hardware).
5. Then the application program must reload the IWDG counter at regular intervals during normal operation to prevent an MCU reset, using IWDG_ReloadCounter() function.

Prescaler and Counter configuration functions
- IWDG_WriteAccessCmd()
- IWDG_SetPrescaler()
- IWDG_SetReload()
- IWDG_ReloadCounter()

IWDG activation function
- IWDG_Enable()

Flag management function
- IWDG_GetFlagStatus()

17.2.1 Prescaler and Counter configuration functions
17.2.1.1 IWDG_WriteAccessCmd

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void IWDG_WriteAccessCmd( uint16_t IWDG_WriteAccess)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Enables or disables write access to IWDG_PR and IWDG_RLR registers.</td>
</tr>
<tr>
<td>Parameters</td>
<td>IWDG_WriteAccess : new state of write access to IWDG_PR and IWDG_RLR registers. This parameter can be one of the following values:</td>
</tr>
<tr>
<td></td>
<td>- IWDG_WriteAccess_Enable : Enable write access to IWDG_PR and IWDG_RLR registers</td>
</tr>
<tr>
<td></td>
<td>- IWDG_WriteAccess_Disable : Disable write access to IWDG_PR and IWDG_RLR registers</td>
</tr>
<tr>
<td>Return values</td>
<td>None.</td>
</tr>
<tr>
<td>Notes</td>
<td>None.</td>
</tr>
</tbody>
</table>
17.2.1.2 IWDG_SetPrescaler

**Function Name**
void IWDG_SetPrescaler ( uint8_t IWDG_Prescaler)

**Function Description**
Sets IWDG Prescaler value.

**Parameters**
- **IWDG_Prescaler** : specifies the IWDG Prescaler value. This parameter can be one of the following values:
  - **IWDG_Prescaler_4** : IWDG prescaler set to 4
  - **IWDG_Prescaler_8** : IWDG prescaler set to 8
  - **IWDG_Prescaler_16** : IWDG prescaler set to 16
  - **IWDG_Prescaler_32** : IWDG prescaler set to 32
  - **IWDG_Prescaler_64** : IWDG prescaler set to 64
  - **IWDG_Prescaler_128** : IWDG prescaler set to 128
  - **IWDG_Prescaler_256** : IWDG prescaler set to 256

**Return values**
- None.

**Notes**
- None.

17.2.1.3 IWDG_SetReload

**Function Name**
void IWDG_SetReload ( uint16_t Reload)

**Function Description**
Sets IWDG Reload value.

**Parameters**
- **Reload** : specifies the IWDG Reload value. This parameter must be a number between 0 and 0x0FFF.

**Return values**
- None.

**Notes**
- None.

17.2.1.4 IWDG_ReloadCounter

**Function Name**
void IWDG_ReloadCounter ( void )

**Function Description**
Reloads IWDG counter with value defined in the reload register (write access to IWDG_PR and IWDG_RLR registers disabled).
17.2.2 IWDG activation function

17.2.2.1 IWDG_Enable

Function Name: void IWDG_Enable ( void )
Function Description: Enables IWDG (write access to IWDG_PR and IWDG_RLR registers disabled).
Parameters: None.
Return values: None.
Notes: None.

17.2.3 Flag management function

17.2.3.1 IWDG_GetFlagStatus

Function Name: FlagStatus IWDG_GetFlagStatus ( uint16_t IWDG_FLAG)
Function Description: Checks whether the specified IWDG flag is set or not.
Parameters: IWDG_FLAG: specifies the flag to check. This parameter can be one of the following values:
  - IWDG_FLAG_PVU: Prescaler Value Update on going
  - IWDG_FLAG_RVU: Reload Value Update on going
Return values: The new state of IWDG_FLAG (SET or RESET).
Notes: None.

17.3 IWDG Firmware driver defines

IWDG
Independent watchdog (IWDG)

**IWDG_Flag**
- `#define: IWDG_FLAG_PVU((uint16_t)0x0001)`

- `#define: IWDG_FLAG_RVU((uint16_t)0x0002)`

**IWDG_prescaler**
- `#define: IWDG_Prescaler_4((uint8_t)0x00)`

- `#define: IWDG_Prescaler_8((uint8_t)0x01)`

- `#define: IWDG_Prescaler_16((uint8_t)0x02)`

- `#define: IWDG_Prescaler_32((uint8_t)0x03)`

- `#define: IWDG_Prescaler_64((uint8_t)0x04)`

- `#define: IWDG_Prescaler_128((uint8_t)0x05)`

- `#define: IWDG_Prescaler_256((uint8_t)0x06)`

**IWDG_WriteAccess**
- `#define: IWDG_WriteAccess_Enable((uint16_t)0x5555)`

- `#define: IWDG_WriteAccess_Disable((uint16_t)0x0000)`
17.4 IWDG Programming Example

The example below explains how to configure the IWDG to have a timeout of 250 ms (the timeout may vary due to LSI frequency dispersion). For more examples about IWDG configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\IWDG\:

```c
/* IWDG timeout equal to 250ms.
   The timeout may vary due to LSI frequency dispersion, the
   LSE value is centred around 32 KHz */

/* Enable write access to IWDG_PR and IWDG_RLR registers */
IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);

/* IWDG counter clock: LSI/32 */
IWDG_SetPrescaler(IWDG_Prescaler_32);

/* Set counter reload value to obtain 250ms IWDG TimeOut. 
   Counter Reload Value = 250ms/IWDG counter clock period 
   = 250ms / (LSI/32) 
   = 0.25s / (32 KHz /32) 
   = 250 */
IWDG_SetReload(250);

/* Reload IWDG counter */
IWDG_ReloadCounter();

/* Enable IWDG (LSI oscillator will be enabled by hardware) */
IWDG_Enable();
```
18 Power control (PWR)

18.1 PWR Firmware driver registers structures

18.1.1 PWR_TypeDef

*PWR_TypeDef* is defined in the *stm32f2xx.h* file and contains the PWR registers definition.

*Data Fields*

- __IO uint32_t CR
- __IO uint32_t CSR

*Field Documentation*

- __IO uint32_t PWR_TypeDef::CR
  - PWR power control register, Address offset: 0x00
- __IO uint32_t PWR_TypeDef::CSR
  - PWR power control/status register, Address offset: 0x04

18.2 PWR Firmware driver API description

The following section lists the various functions of the PWR library.

18.2.1 Backup Domain access

After reset, the backup domain (RTC registers, RTC backup data registers and backup SRAM) is protected against possible unwanted write accesses. To enable access to the RTC Domain and RTC registers, proceed as follows:

- Enable the Power Controller (PWR) APB1 interface clock using the RCC_APB1PeriphClockCmd() function.
- Enable access to RTC domain using the PWR_BackupAccessCmd() function.
- PWR_DeInit()
- PWR_BackupAccessCmd()

18.2.2 Power control configuration

*PVD Configuration functions*

The PVD is used to monitor the VDD power supply by comparing it to a threshold selected by the PVD Level (PLS[2:0] bits in the PWR_CR).

A PVDO flag is available to indicate if VDD/VDDA is higher or lower than the PVD threshold. This event is internally connected to the EXTI line16 and can generate an interrupt if enabled through the EXTI registers.

The PVD is stopped in Standby mode.
WakeUp pin configuration functions

The WakeUp pin is used to wake up the system from Standby mode. This pin is forced in input pull down configuration and is active on rising edges.

There is only one WakeUp pin: WakeUp Pin 1 on PA.00.

Backup Regulator configuration functions

The backup domain includes 4 Kbytes of backup SRAM accessible only from the CPU, and address in 32-bit, 16-bit or 8-bit mode. Its content is retained even in Standby or VBAT mode when the low power backup regulator is enabled. It can be considered as an internal EEPROM when VBAT is always present. You can use the PWR_BackupRegulatorCmd() function to enable the low power backup regulator and use the PWR_GetFlagStatus(PWR_FLAG_BRR) to check if it is ready or not.

When the backup domain is supplied by VDD (analog switch connected to VDD) the backup SRAM is powered from VDD which replaces the VBAT power supply to save battery life.

The backup SRAM is not mass erased by a tamper event. It is read protected to prevent confidential data, such as cryptographic private key, from being accessed. The backup SRAM can be erased only through the Flash interface when a protection level change from level 1 to level 0 is requested. Refer to the description of Read protection (RDP) in the Flash programming manual.

FLASH Power Down configuration functions

By setting the FPDS bit in the PWR_CR register by using the PWR_FlashPowerDownCmd() function, the Flash memory also enters power down mode when the device enters Stop mode. When the Flash memory is in power down mode, an additional startup delay is incurred when waking up from Stop mode.

Low Power modes configuration functions

The devices feature 3 low-power modes:

- Sleep mode: Cortex-M3 core stopped, peripherals kept running.
- Stop mode: all clocks are stopped, regulator running, regulator in low power mode
- Standby mode: 1.2V domain powered off.

Sleep mode
- Entry: The Sleep mode is entered by using the __WFI() or __WFE() functions.
- Exit: Any peripheral interrupt acknowledged by the nested vectored interrupt controller (NVIC) can wake up the device from Sleep mode.

Stop mode
In Stop mode, all clocks in the 1.2V domain are stopped, the PLL, the HSI, and the HSE RC oscillators are disabled. Internal SRAM and register contents are preserved. The voltage regulator can be configured either in normal or low-power mode. To minimize the consumption in Stop mode, FLASH can be powered off before entering the Stop mode.
can be switched on again by software after exiting the Stop mode using the 
PWR_FlashPowerDownCmd() function.

- Entry: The Stop mode is entered using the 
PWR_EnterSTOPMode(PWR_Regulator_LowPower,) function with regulator in 
LowPower or with Regulator ON.
- Exit: Any EXTI Line (Internal or External) configured in Interrupt/Event mode.

Standby mode

The Standby mode allows to achieve the lowest power consumption. It is based on the 
Cortex-M3 deepsleep mode, with the voltage regulator disabled. The 1.2V domain is 
consequently powered off. The PLL, the HSI oscillator and the HSE oscillator are also 
switched off. SRAM and register contents are lost except for the RTC registers, RTC 
backup registers, backup SRAM and Standby circuitry. The voltage regulator is OFF.

- Entry: The Standby mode is entered using the PWR_EnterSTANDBYMode() function.
- Exit: WKUP pin rising edge, RTC alarm (Alarm A and Alarm B), RTC wakeup, tamper 
event, time-stamp event, external reset in NRST pin, IWDG reset.

Auto-wakeup (AWU) from low-power mode

The MCU can be woken up from low-power mode by an RTC Alarm event, an RTC 
Wakeup event, a tamper event, a time-stamp event, or a comparator event, without 
depending on an external interrupt (Auto-wakeup mode):

- RTC auto-wakeup (AWU) from the Stop mode.
  - To wake up from the Stop mode with an RTC alarm event, it is necessary to:
    a. Configure the EXTI Line 17 to be sensitive to rising edges (Interrupt or Event 
       modes) using the EXTI_Init() function.
    b. Enable the RTC Alarm Interrupt using the RTC_ITConfig() function
    c. Configure the RTC to generate the RTC alarm using the RTC_SetAlarm() 
       and RTC_AlarmCmd() functions.
  - To wake up from the Stop mode with an RTC Tamper or time stamp event, it is 
    necessary to:
    a. Configure the EXTI Line 21 to be sensitive to rising edges (Interrupt or Event 
       modes) using the EXTI_Init() function.
    b. Enable the RTC Tamper or time stamp Interrupt using the RTC_ITConfig() 
       function
    c. Configure the RTC to detect the tamper or time stamp event using the 
       RTC_TimeStampConfig(), RTC_TamperTriggerConfig() and 
       RTC_TamperCmd() functions.
  - To wake up from the Stop mode with an RTC WakeUp event, it is necessary to:
    a. Configure the EXTI Line 22 to be sensitive to rising edges (Interrupt or Event 
       modes) using the EXTI_Init() function.
    b. Enable the RTC WakeUp Interrupt using the RTC_ITConfig() function
    c. Configure the RTC to generate the RTC WakeUp event using the 
       RTC_WakeUpClockConfig(), RTC_SetWakeUpCounter() and 
       RTC_WakeUpCmd() functions.

- RTC auto-wakeup (AWU) from the Standby mode
  - To wake up from the Standby mode with an RTC alarm event, it is necessary to:
    a. Enable the RTC Alarm Interrupt using the RTC_ITConfig() function
    b. Configure the RTC to generate the RTC alarm using the RTC_SetAlarm() 
       and RTC_AlarmCmd() functions.
  - To wake up from the Standby mode with an RTC Tamper or time stamp event, it is 
    necessary to:
    a. Enable the RTC Tamper or time stamp Interrupt using the RTC_ITConfig() 
       function
b. Configure the RTC to detect the tamper or time stamp event using the
   `RTC_TimeStampConfig()`, `RTC_TamperTriggerConfig()` and
   `RTC_TamperCmd()` functions.
   - To wake up from the Standby mode with an RTC WakeUp event, it is necessary
e   to:
     a. Enable the RTC WakeUp Interrupt using the `RTC_ITConfig()` function
     b. Configure the RTC to generate the RTC WakeUp event using the
        `RTC_WakeUpClockConfig()`, `RTC_SetWakeUpCounter()` and
        `RTC_WakeUpCmd()` functions.

   - `PWR_EnterSTOPMode()`
   - `PWR_EnterSTANDBYMode()`

18.2.3 Backup Domain Access function

18.2.3.1 PWR_DeInit

Function Name: `void PWR_DeInit ( void )`
Function Description: Deinitializes the PWR peripheral registers to their default reset values.
Parameters:
- None.
Return values:
- None.
Notes:
- None.

18.2.3.2 PWR_BackupAccessCmd

Function Name: `void PWR_BackupAccessCmd ( FunctionalState NewState)`
Function Description: Enables or disables access to the backup domain (RTC registers,
   RTC backup data registers and backup SRAM).
Parameters:
- `NewState`: new state of the access to the backup domain.
  This parameter can be: ENABLE or DISABLE.
Return values:
- None.
Notes:
- If the HSE divided by 2, 3, ..31 is used as the RTC clock, the
  Backup Domain Access should be kept enabled.
18.2.4 PVD configuration functions

18.2.4.1 PWR_PVDLevelConfig

Function Name: void PWR_PVDLevelConfig ( uint32_t PWR_PVDLevel)
Function Description: Configures the voltage threshold detected by the Power Voltage Detector (PVD).
Parameters:
- **PWR_PVDLevel**: specifies the PVD detection level. This parameter can be one of the following values:
  - **PWR_PVDLevel_0**: PVD detection level set to 2.0V
  - **PWR_PVDLevel_1**: PVD detection level set to 2.2V
  - **PWR_PVDLevel_2**: PVD detection level set to 2.3V
  - **PWR_PVDLevel_3**: PVD detection level set to 2.5V
  - **PWR_PVDLevel_4**: PVD detection level set to 2.7V
  - **PWR_PVDLevel_5**: PVD detection level set to 2.8V
  - **PWR_PVDLevel_6**: PVD detection level set to 2.9V
  - **PWR_PVDLevel_7**: PVD detection level set to 3.0V

Return values:
- None.

Notes:
- Refer to the electrical characteristics of your device datasheet for more details.

18.2.4.2 PWR_PVDCmd

Function Name: void PWR_PVDCmd ( FunctionalState NewState)
Function Description: Enables or disables the Power Voltage Detector (PVD).
Parameters:
- **NewState**: new state of the PVD. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.

18.2.5 Wakeup pin configuration function

18.2.5.1 PWR_WakeUpPinCmd
### Function Name: PWR_WakeUpPinCmd

**Function Description:** Enables or disables the WakeUp Pin functionality.

**Parameters:**
- **NewState**: new state of the WakeUp Pin functionality. This parameter can be: ENABLE or DISABLE.

**Return values:**
- None.

**Notes:**
- None.

---

### 18.2.6 Backup Regulator configuration function

#### 18.2.6.1 PWR_BackupRegulatorCmd

**Function Name:** void PWR_BackupRegulatorCmd ( FunctionalState NewState)

**Function Description:** Enables or disables the Backup Regulator.

**Parameters:**
- **NewState**: new state of the Backup Regulator. This parameter can be: ENABLE or DISABLE.

**Return values:**
- None.

**Notes:**
- None.

---

### 18.2.7 FLASH Power Down configuration function

#### 18.2.7.1 PWR_FlashPowerDownCmd

**Function Name:** void PWR_FlashPowerDownCmd ( FunctionalState NewState)

**Function Description:** Enables or disables the Flash Power Down in STOP mode.

**Parameters:**
- **NewState**: new state of the Flash power mode. This parameter can be: ENABLE or DISABLE.

**Return values:**
- None.

**Notes:**
- None.
18.2.8 Low Power modes configuration functions

18.2.8.1 PWR_EnterSTOPMode

Function Name void PWR_EnterSTOPMode ( uint32_t PWR_Regulator, uint8_t PWR_STOPEntry)

Function Description Enters STOP mode.

Parameters
- **PWR_Regulator**: specifies the regulator state in STOP mode. This parameter can be one of the following values:
  - **PWR_Regulator_ON**: STOP mode with regulator ON
  - **PWR_Regulator_LowPower**: STOP mode with regulator in low power mode
- **PWR_STOPEntry**: specifies if STOP mode in entered with WFI or WFE instruction. This parameter can be one of the following values:
  - **PWR_STOPEntry_WFI**: enter STOP mode with WFI instruction
  - **PWR_STOPEntry_WFE**: enter STOP mode with WFE instruction

Return values
- None.

Notes
- In Stop mode, all I/O pins keep the same state as in Run mode.
- When exiting Stop mode by issuing an interrupt or a wakeup event, the HSI RC oscillator is selected as system clock.
- When the voltage regulator operates in low power mode, an additional startup delay is incurred when waking up from Stop mode. By keeping the internal regulator ON during Stop mode, the consumption is higher although the startup time is reduced.

18.2.8.2 PWR_EnterSTANDBYMode

Function Name void PWR_EnterSTANDBYMode ( void )

Function Description Enters STANDBY mode.

Parameters
- None.

Return values
- None.

Notes
- In Standby mode, all I/O pins are high impedance except for:
  - Reset pad (still available)
  - RTC_AF1 pin (PC13) if configured for tamper, time-stamp, RTC Alarm out, or RTC clock calibration out.
  - RTC_AF2 pin (PI8) if configured for tamper or time-
18.2.9 Flags management functions

18.2.9.1 PWR_GetFlagStatus

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Function Description</th>
<th>Parameters</th>
</tr>
</thead>
</table>
| FlagStatus PWR_GetFlagStatus (uint32_t PWR_FLAG) | Checks whether the specified PWR flag is set or not. | **PWR_FLAG** : specifies the flag to check. This parameter can be one of the following values:

- **PWR_FLAG_WU**: Wake Up flag. This flag indicates that a wakeup event was received from the WKUP pin or from the RTC alarm (Alarm A or Alarm B), RTC Tamper event, RTC TimeStamp event or RTC Wakeup. An additional wakeup event is detected if the WKUP pin is enabled (by setting the EWUP bit) when the WKUP pin level is already high.
- **PWR_FLAG_SB**: StandBy flag. This flag indicates that the system was resumed from StandBy mode.
- **PWR_FLAG_PVDO**: PVD Output. This flag is valid only if PVD is enabled by the PWR_PVDCmd() function. The PVD is stopped by Standby mode. For this reason, this bit is equal to 0 after Standby or reset until the PVDE bit is set.
- **PWR_FLAG_BRR**: Backup regulator ready flag. This bit is not reset when the device wakes up from Standby mode or by a system reset or power reset. |
| Return values | **The new state of PWR_FLAG (SET or RESET).** |
| Notes | **None.** |

18.2.9.2 PWR_ClearFlag

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Function Description</th>
<th>Parameters</th>
</tr>
</thead>
</table>
| void PWR_ClearFlag ( uint32_t PWR_FLAG) | Clears the PWR's pending flags. | **PWR_FLAG** : specifies the flag to clear. This parameter can be one of the following values:

- **PWR_FLAG_WU**: Wake Up flag |

---

- WKUP pin 1 (PA0) if enabled.
PWR control (PWR)

- `PWR_FLAG_SB`: StandBy flag

<table>
<thead>
<tr>
<th>Return values</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>None.</td>
<td>None.</td>
</tr>
</tbody>
</table>

18.3 PWR Firmware driver defines

18.3.1 PWR Firmware driver defines

PWR

`PWR_Flag`

- `#define`: `PWR_FLAG_WUPWR_CSR_WUF`

- `#define`: `PWR_FLAG_SBPWR_CSR_SBF`

- `#define`: `PWR_FLAG_PVDOPWR_CSR_PVDO`

- `#define`: `PWR_FLAG_BRRPWR_CSR_BRR`

`PWR_PVD_detection_level`

- `#define`: `PWR_PVDLevel_0PWR_CR_PLS_LEV0`

- `#define`: `PWR_PVDLevel_1PWR_CR_PLS_LEV1`

- `#define`: `PWR_PVDLevel_2PWR_CR_PLS_LEV2`

- `#define`: `PWR_PVDLevel_3PWR_CR_PLS_LEV3`
PWR_PVDLevel_4PWR_CR_PLS_LEV4

PWR_PVDLevel_5PWR_CR_PLS_LEV5

PWR_PVDLevel_6PWR_CR_PLS_LEV6

PWR_PVDLevel_7PWR_CR_PLS_LEV7

PWR_Regulator_state_in_STOP_mode
• #define: PWR_Regulator_ON((uint32_t)0x00000000)

PWR_Regulator_LowPowerPWR_CR_LPDS

PWR_STOP_mode_entry
• #define: PWR_STOPEntry_WFI((uint8_t)0x01)

• #define: PWR_STOPEntry_WFE((uint8_t)0x02)

18.4 PWR Programming Example
The example below explains how to enter the system to STANDBY mode and wake-up from this mode using the WKUP pin(PA0). For more examples about PWR configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\PWR

/* Enable PWR Clock */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);

/* Enable WKUP pin 1 */
PWR_WakeUpPinCmd(ENABLE);

/* Request to enter STANDBY mode (Wake Up flag is cleared in
PWR_EnterSTANDBYMode();
19 Reset and clock control (RCC)

19.1 RCC Firmware driver registers structures

19.1.1 RCC_TypeDef

*RCC_TypeDef* is defined in the stm32f2xx.h file and contains the RCC registers definition.

**Data Fields**

- __IO uint32_t CR
- __IO uint32_t PLLCFGR
- __IO uint32_t CFGR
- __IO uint32_t CIR
- __IO uint32_t AHB1RSTR
- __IO uint32_t AHB2RSTR
- __IO uint32_t AHB3RSTR
- uint32_t RESERVED0
- __IO uint32_t APB1RSTR
- __IO uint32_t APB2RSTR
- uint32_t RESERVED1
- __IO uint32_t AHB1ENR
- __IO uint32_t AHB2ENR
- __IO uint32_t AHB3ENR
- uint32_t RESERVED2
- __IO uint32_t APB1ENR
- __IO uint32_t APB2ENR
- uint32_t RESERVED3
- __IO uint32_t AHB1LPENR
- __IO uint32_t AHB2LPENR
- __IO uint32_t AHB3LPENR
- uint32_t RESERVED4
- __IO uint32_t APB1LPENR
- __IO uint32_t APB2LPENR
- uint32_t RESERVED5
- __IO uint32_t BDCR
- __IO uint32_t CSR
- uint32_t RESERVED6
- __IO uint32_t SSCGR
- __IO uint32_t PLLI2SCFGR

**Field Documentation**

- __IO uint32_t RCC_TypeDef::CR
  - RCC clock control register, Address offset: 0x00
- __IO uint32_t RCC_TypeDef::PLLCFGR
  - RCC PLL configuration register, Address offset: 0x04
- __IO uint32_t RCC_TypeDef::CFGR
  - RCC clock configuration register, Address offset: 0x08
### Reset and clock control (RCC)

**__IO uint32_t RCC_TypeDef::CIR**
- RCC clock interrupt register, Address offset: 0x0C

**__IO uint32_t RCC_TypeDef::AHB1RSTR**
- RCC AHB1 peripheral reset register, Address offset: 0x10

**__IO uint32_t RCC_TypeDef::AHB2RSTR**
- RCC AHB2 peripheral reset register, Address offset: 0x14

**__IO uint32_t RCC_TypeDef::AHB3RSTR**
- RCC AHB3 peripheral reset register, Address offset: 0x18

**uint32_t RCC_TypeDef::RESERVED0**
- Reserved, 0x1C

**__IO uint32_t RCC_TypeDef::APB1RSTR**
- RCC APB1 peripheral reset register, Address offset: 0x20

**__IO uint32_t RCC_TypeDef::APB2RSTR**
- RCC APB2 peripheral reset register, Address offset: 0x24

**uint32_t RCC_TypeDef::RESERVED1[2]**
- Reserved, 0x28-0x2C

**__IO uint32_t RCC_TypeDef::AHB1ENR**
- RCC AHB1 peripheral clock register, Address offset: 0x30

**__IO uint32_t RCC_TypeDef::AHB2ENR**
- RCC AHB2 peripheral clock register, Address offset: 0x34

**__IO uint32_t RCC_TypeDef::AHB3ENR**
- RCC AHB3 peripheral clock register, Address offset: 0x38

**uint32_t RCC_TypeDef::RESERVED2**
- Reserved, 0x3C

**__IO uint32_t RCC_TypeDef::APB1ENR**
- RCC APB1 peripheral clock enable register, Address offset: 0x40

**__IO uint32_t RCC_TypeDef::APB2ENR**
- RCC APB2 peripheral clock enable register, Address offset: 0x44

**uint32_t RCC_TypeDef::RESERVED3[2]**
- Reserved, 0x48-0x4C

**__IO uint32_t RCC_TypeDef::AHB1LPENR**
- RCC AHB1 peripheral clock enable in low power mode register, Address offset: 0x50

**__IO uint32_t RCC_TypeDef::AHB2LPENR**
- RCC AHB2 peripheral clock enable in low power mode register, Address offset: 0x54

**__IO uint32_t RCC_TypeDef::AHB3LPENR**
- RCC AHB3 peripheral clock enable in low power mode register, Address offset: 0x58

**uint32_t RCC_TypeDef::RESERVED4**
- Reserved, 0x5C

**__IO uint32_t RCC_TypeDef::APB1LPENR**
- RCC APB1 peripheral clock enable in low power mode register, Address offset: 0x60

**__IO uint32_t RCC_TypeDef::APB2LPENR**
- RCC APB2 peripheral clock enable in low power mode register, Address offset: 0x64

**uint32_t RCC_TypeDef::RESERVED5[2]**
- Reserved, 0x68-0x6C

**__IO uint32_t RCC_TypeDef::BDCR**
- RCC Backup domain control register, Address offset: 0x70

**__IO uint32_t RCC_TypeDef::CSR**
- RCC clock control & status register, Address offset: 0x74
• `uint32_t RCC_TypeDef::RESERVED6[2]`
  – Reserved, 0x78-0x7C
• `__IO uint32_t RCC_TypeDef::SSCGR`
  – RCC spread spectrum clock generation register, Address offset: 0x80
• `__IO uint32_t RCC_TypeDef::PLLI2SCFGR`
  – RCC PLLI2S configuration register, Address offset: 0x84

19.1.2 RCC_ClocksTypeDef

**RCC_ClocksTypeDef** is defined in the stm32f2xx_rcc.h file and will hold the clocks frequencies.

Data Fields

- `uint32_t SYSCLK_Frequency`
- `uint32_t HCLK_Frequency`
- `uint32_t PCLK1_Frequency`
- `uint32_t PCLK2_Frequency`

Field Documentation

- `uint32_t RCC_ClocksTypeDef::SYSCLK_Frequency`
  – SYSCLK clock frequency expressed in Hz
- `uint32_t RCC_ClocksTypeDef::HCLK_Frequency`
  – HCLK clock frequency expressed in Hz
- `uint32_t RCC_ClocksTypeDef::PCLK1_Frequency`
  – PCLK1 clock frequency expressed in Hz
- `uint32_t RCC_ClocksTypeDef::PCLK2_Frequency`
  – PCLK2 clock frequency expressed in Hz

19.2 RCC Firmware driver API description

The following section lists the various functions of the RCC library.

19.2.1 RCC specific features

After reset the device is running from Internal High Speed oscillator (HSI 16MHz) with Flash 0 wait state, Flash prefetch buffer, D-Cache and I-Cache are disabled, and all peripherals are off except internal SRAM, Flash and JTAG.

- There is no prescaler on High speed (AHB) and Low speed (APB) busses; all peripherals mapped on these busses are running at HSI speed.
- The clock for all peripherals is switched off, except the SRAM and FLASH.
- All GPIOs are in input floating state, except the JTAG pins which are assigned to be used for debug purpose.
Once the device started from reset, the user application has to:

- Configure the clock source to be used to drive the System clock (if the application needs higher frequency/performance)
- Configure the System clock frequency and Flash settings
- Configure the AHB and APB busses prescalers
- Enable the clock for the peripheral(s) to be used
- Configure the clock source(s) for peripherals which clocks are not derived from the System clock (I2S, RTC, ADC, USB OTG FS/SDIO/RNG)

19.2.2 Internal and external clocks, PLL, CSS and MCO configuration

This section provides functions allowing to configure the internal/external clocks, PLLs, CSS and MCO pins.

- HSI (high-speed internal), 16 MHz factory-trimmed RC used directly or through the PLL as System clock source.
- LSI (low-speed internal), 32 KHz low consumption RC used as IWDG and/or RTC clock source.
- HSE (high-speed external), 4 to 26 MHz crystal oscillator used directly or through the PLL as System clock source. Can be used also as RTC clock source.
- LSE (low-speed external), 32 KHz oscillator used as RTC clock source.
- PLL (clocked by HSI or HSE), featuring two different output clocks: - The first output is used to generate the high speed system clock (up to 120 MHz) - The second output is used to generate the clock for the USB OTG FS (48 MHz), the random analog generator (<=48 MHz) and the SDIO (<= 48 MHz).
- PLLI2S (clocked by HSI or HSE), used to generate an accurate clock to achieve high-quality audio performance on the I2S interface.
- CSS (Clock security system), once enable and if a HSE clock failure occurs (HSE used directly or through PLL as System clock source), the System clock is automatically switched to HSI and an interrupt is generated if enabled. The interrupt is linked to the Cortex-M3 NMI (Non-Maskable Interrupt) exception vector.
- MCO1 (microcontroller clock output), used to output HSI, LSE, HSE or PLL clock (through a configurable prescaler) on PA8 pin. 9. MCO2 (microcontroller clock output), used to output HSE, PLL, SYSCLK or PLLI2S clock (through a configurable prescaler) on PC9 pin.

Below is the list of functions used to configure the internal and external clocks, PLLs, CSS and MCO pins:

- RCC_DeInit()
- RCC_HSEConfig()
- RCC_WaitForHSEStartUp()
- RCC_AdjustHSICalibrationValue()
- RCC_HSCIcmd()
- RCC_LSEConfig()
- RCC_LSCIcmd()
- RCC_PLLConfig()
- RCC_PLLCmd()
- RCC_PLLI2SConfig()
- RCC_PLLI2SCmd()
- RCC_ClockSecuritySystemCmd()
- RCC_MCO1Config()
- RCC_MCO2Config()
19.2.3 System AHB and APB busses clocks configuration

This section provides functions allowing to configure the System, AHB, APB1 and APB2 busses clocks.

**Several clock sources can be used to drive the System clock (SYSCLK): HSI, HSE and PLL.**

The AHB clock (HCLK) is derived from System clock through configurable prescaler and used to clock the CPU, memory and peripherals mapped on AHB bus (DMA, GPIO...). APB1 (PCLK1) and APB2 (PCLK2) clocks are derived from AHB clock through configurable prescalers and used to clock the peripherals mapped on these busses. You can use "RCC_GetClocksFreq()" function to retrieve the frequencies of these clocks. All the peripheral clocks are derived from the System clock (SYSCLK) except:

- I2S: the I2S clock can be derived either from a specific PLL (PLLI2S) or from an external clock mapped on the I2S_CKIN pin. You have to use RCC_I2SCLKConfig() function to configure this clock.
- RTC: the RTC clock can be derived either from the LSI, LSE or HSE clock divided by 2 to 31. You have to use RCC_RTCCLKConfig() and RCC_RTCCLKCmd() functions to configure this clock.
- USB OTG FS, SDIO and RTC: USB OTG FS require a frequency equal to 48 MHz to work correctly, while the SDIO require a frequency equal or lower than to 48. This clock is derived of the main PLL through PLLQ divider.
- IWDG clock which is always the LSI clock.

The maximum frequency of the SYSCLK and HCLK is 120 MHz, PCLK2 60 MHz and PCLK1 30 MHz.

Depending on the device voltage range, the maximum frequency should be adapted accordingly:

<table>
<thead>
<tr>
<th>Wait states (WS) (latency)</th>
<th>HCLK clock frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Voltage range 2.7 to 3.6 V</td>
</tr>
<tr>
<td>0WS(1CPU cycle)</td>
<td>0 &lt; HCLK ≤ 30</td>
</tr>
<tr>
<td>1WS(2CPU cycle)</td>
<td>30 &lt; HCLK ≤ 60</td>
</tr>
<tr>
<td>2WS(3CPU cycle)</td>
<td>60 &lt; HCLK ≤ 90</td>
</tr>
<tr>
<td>3WS(4CPU cycle)</td>
<td>90 &lt; HCLK ≤ 120</td>
</tr>
<tr>
<td>4WS(5CPU)</td>
<td>NA</td>
</tr>
</tbody>
</table>

*If IRROFF is set to VDD on STM32F20xx devices, this value can be lowered to 1.65 V when the device operates in a reduced temperature range.*
Wait states (WS) (latency) | HCLK clock frequency (MHz) |
|-------------------------|--------------------------|
|                        | Voltage range 2.7 to 3.6 V | Voltage range 2.4 to 2.7 V | Voltage range 2.1 to 2.4 V | Voltage range 1.8 to 2.1 V
| 5WS/6CPU cycle         | NA                        | NA                        | 90 < HCLK ≤ 108          | 80 < HCLK ≤ 96
| 6WS/7CPU cycle         | NA                        | NA                        | 108 < HCLK ≤ 120         | 96 < HCLK ≤ 112
| 7WS/8CPU cycle         | NA                        | NA                        | NA                        | 12 < HCLK ≤ 120

- **RCC_SYSCLKConfig()**
- **RCC_GetSYSCLKSource()**
- **RCC_HCLKConfig()**
- **RCC_PCLK1Config()**
- **RCC_PCLK2Config()**
- **RCC_GetClocksFreq()**

### 19.2.4 Peripheral clocks configuration

This section provides functions allowing to configure the Peripheral clocks.

- The RTC clock which is derived from the LSI, LSE or HSE clock divided by 2 to 31.
- After restart from Reset or wakeup from STANDBY, all peripherals are off except internal SRAM, Flash and JTAG. Before to start using a peripheral you have to enable its interface clock. You can do this using RCC_AHBPeriphClockCmd(), RCC_APB2PeriphClockCmd() and RCC_APB1PeriphClockCmd() functions.
- To reset the peripherals configuration (to the default state after device reset) you can use RCC_AHBPeriphResetCmd(), RCC_APB2PeriphResetCmd() and RCC_APB1PeriphResetCmd() functions.
- To further reduce power consumption in SLEEP mode the peripheral clocks can be disabled prior to executing the WFI or WFE instructions. You can do this using RCC_AHBPeriphClockLPModeCmd(), RCC_APB2PeriphClockLPModeCmd() and RCC_APB1PeriphClockLPModeCmd() functions.

Below is the list of functions used to configure peripheral clocks:

- **RCC_RTCCLKConfig()**
- **RCC_RTCCLKCmd()**
- **RCC_BackupResetCmd()**
- **RCC_I2SCLKConfig()**
- **RCC_AHB1PeriphClockCmd()**
- **RCC_AHB2PeriphClockCmd()**
- **RCC_AHB3PeriphClockCmd()**
- **RCC_APB1PeriphClockCmd()**
- **RCC_APB2PeriphClockCmd()**
- **RCC_AHB1PeriphResetCmd()**
- **RCC_AHB2PeriphResetCmd()**
- **RCC_AHB3PeriphResetCmd()**
- **RCC_APB1PeriphResetCmd()**
- **RCC_APB2PeriphResetCmd()**
- **RCC_AHB1PeriphClockLPModeCmd()**
19.2.5 Interrupt and flag management functions

Below is the list of functions to manage interrupts and flags:

- RCC_ITConfig()
- RCC_GetFlagStatus()
- RCC_ClearFlag()
- RCC_GetITStatus()
- RCC_ClearITPendingBit()

19.2.6 Internal and external clocks, PLL, CSS and MCO configuration functions

19.2.6.1 RCC_DeInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void RCC_DeInit ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Resets the RCC clock configuration to the default reset state.</td>
</tr>
<tr>
<td>Parameters</td>
<td>None.</td>
</tr>
<tr>
<td>Return values</td>
<td>None.</td>
</tr>
<tr>
<td>Notes</td>
<td>The default reset state of the clock configuration is given below: HSI ON and used as system clock source, HSE, PLL and PLLI2S OFF, AHB, APB1 and APB2 prescaler set to 1, CSS, MCO1 and MCO2 OFF, all interrupts disabled. This function doesn't modify the configuration of the Peripheral clocks LSI, LSE and RTC clocks.</td>
</tr>
</tbody>
</table>

19.2.6.2 RCC_HSEConfig

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void RCC_HSEConfig ( uint8_t RCC_HSE )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Configures the External High Speed oscillator (HSE).</td>
</tr>
<tr>
<td>Parameters</td>
<td>RCC_HSE : specifies the new state of the HSE. This parameter can be one of the following values:</td>
</tr>
<tr>
<td></td>
<td>- RCC_HSE_OFF : turn OFF the HSE oscillator, HSERDY flag goes low after 6 HSE oscillator clock cycles.</td>
</tr>
<tr>
<td></td>
<td>- RCC_HSE_ON : turn ON the HSE oscillator</td>
</tr>
</tbody>
</table>
Reset and clock control (RCC)

- **RCC_HSE_Bypass** : HSE oscillator bypassed with external clock

Return values
- None.

Notes
- After enabling the HSE (RCC_HSE_ON or RCC_HSE_Bypass), the application software should wait on HSERDY flag to be set indicating that HSE clock is stable and can be used to clock the PLL and/or system clock.
- HSE state can not be changed if it is used directly or through the PLL as system clock. In this case, you have to select another source of the system clock then change the HSE state (ex. disable it).
- The HSE is stopped by hardware when entering STOP and STANDBY modes.
- This function reset the CSSON bit, so if the Clock security system (CSS) was previously enabled you have to enable it again after calling this function.

**19.2.6.3 RCC_WaitForHSEStartUp**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>ErrorStatus RCC_WaitForHSEStartUp ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Waits for HSE start-up.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• None.</td>
</tr>
</tbody>
</table>
| Return values | • An ErrorStatus enumeration value:  
  - SUCCESS: HSE oscillator is stable and ready to use  
  - ERROR: HSE oscillator not yet ready |
| Notes | • This function waits on HSERDY flag to be set and return SUCCESS if this flag is set, otherwise returns ERROR if the timeout is reached and this flag is not set. The timeout value is defined by the constant HSE_STARTUP_TIMEOUT in stm32f2xx.h file. You can tailor it depending on the HSE crystal used in your application. |

**19.2.6.4 RCC_AdjustHSICalibrationValue**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void RCC_AdjustHSICalibrationValue ( uint8_t HSI CalibrationValue )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Adjusts the Internal High Speed oscillator (HSI) calibration value.</td>
</tr>
</tbody>
</table>
### Parameters
- **HSICalibrationValue**: specifies the calibration trimming value. This parameter must be a number between 0 and 0x1F.

### Return values
- None.

### Notes
- The calibration is used to compensate for the variations in voltage and temperature that influence the frequency of the internal HSI RC.

#### 19.2.6.5 RCC_HSICmd

**Function Name**: void RCC_HSICmd ( FunctionalState NewState)

**Function Description**: Enables or disables the Internal High Speed oscillator (HSI).

**Parameters**
- **NewState**: new state of the HSI. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.

**Notes**
- The HSI is stopped by hardware when entering STOP and STANDBY modes. It is used (enabled by hardware) as system clock source after startup from Reset, wakeup from STOP and STANDBY mode, or in case of failure of the HSE used directly or indirectly as system clock (if the Clock Security System CSS is enabled).
- HSI can not be stopped if it is used as system clock source. In this case, you have to select another source of the system clock then stop the HSI.
- After enabling the HSI, the application software should wait on HSIRDY flag to be set indicating that HSI clock is stable and can be used as system clock source.
- When the HSI is stopped, HSIRDY flag goes low after 6 HSI oscillator clock cycles.

#### 19.2.6.6 RCC_LSEConfig

**Function Name**: void RCC_LSEConfig ( uint8_t RCC_LSE)

**Function Description**: Configures the External Low Speed oscillator (LSE).

**Parameters**
- **RCC_LSE**: specifies the new state of the LSE. This parameter can be one of the following values:
  - **RCC_LSE_OFF**: turn OFF the LSE oscillator, LSERDY
flag goes low after 6 LSE oscillator clock cycles.

- **RCC_LSE_ON**: turn ON the LSE oscillator
- **RCC_LSE_Bypass**: LSE oscillator bypassed with external clock

**Return values**
- None.

**Notes**
- As the LSE is in the Backup domain and write access is denied to this domain after reset, you have to enable write access using `PWR_BackupAccessCmd(ENABLE)` function before to configure the LSE (to be done once after reset).
- After enabling the LSE (RCC_LSE_ON or RCC_LSE_Bypass), the application software should wait on LSERDY flag to be set indicating that LSE clock is stable and can be used to clock the RTC.

### 19.2.6.7 RCC_LSICmd

**Function Name**

`void RCC_LSICmd ( FunctionalState NewState)`

**Function Description**
Enables or disables the Internal Low Speed oscillator (LSI).

**Parameters**
- **NewState**: new state of the LSI. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.

**Notes**
- After enabling the LSI, the application software should wait on LSIRDY flag to be set indicating that LSI clock is stable and can be used to clock the IWDG and/or the RTC.
- LSI can not be disabled if the IWDG is running.
- When the LSI is stopped, LSIRDY flag goes low after 6 LSI oscillator clock cycles.

### 19.2.6.8 RCC_PLLConfig

**Function Name**

`void RCC_PLLConfig ( uint32_t RCC_PLLSource, uint32_t PLLM, uint32_t PLLN, uint32_t PLLP, uint32_t PLLQ)`

**Function Description**
Configures the main PLL clock source, multiplication and division factors.

**Parameters**
- **RCC_PLLSource**: specifies the PLL entry clock source. This parameter can be one of the following values:
Reset and clock control (RCC)

- **RCC_PLLSource_HSI**: HSI oscillator clock selected as PLL clock entry
- **RCC_PLLSource_HSE**: HSE oscillator clock selected as PLL clock entry

**Parameters**
- **PLLM**: specifies the division factor for PLL VCO input clock. This parameter must be a number between 0 and 63.
- **PLLN**: specifies the multiplication factor for PLL VCO output clock. This parameter must be a number between 192 and 432.
- **PLLP**: specifies the division factor for main system clock (SYSCLK). This parameter must be a number in the range {2, 4, 6, or 8}.
- **PLLQ**: specifies the division factor for OTG FS, SDIO and RNG clocks. This parameter must be a number between 4 and 15.

**Return values**
- None.

**Notes**
- This function must be used only when the main PLL is disabled.
- This clock source (RCC_PLLSource) is common for the main PLL and PLLI2S.
- You have to set the PLLM parameter correctly to ensure that the VCO input frequency ranges from 1 to 2 MHz. It is recommended to select a frequency of 2 MHz to limit PLL jitter.
- You have to set the PLLN parameter correctly to ensure that the VCO output frequency is between 192 and 432 MHz.
- You have to set the PLLP parameter correctly to not exceed 120 MHz on the System clock frequency.
- If the USB OTG FS is used in your application, you have to set the PLLQ parameter correctly to have 48 MHz clock for the USB. However, the SDIO and RNG need a frequency lower than or equal to 48 MHz to work correctly.

### 19.2.6.9 RCC_PLLCmd

**Function Name**

`void RCC_PLLCmd ( FunctionalState NewState)`

**Function Description**

Enables or disables the main PLL.

**Parameters**
- **NewState**: new state of the main PLL. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.

**Notes**
- After enabling the main PLL, the application software should wait on PLLRDY flag to be set indicating that PLL clock is
stable and can be used as system clock source.
- The main PLL can not be disabled if it is used as system clock source
- The main PLL is disabled by hardware when entering STOP and STANDBY modes.

19.2.6.10 RCC_PLLI2SWconfig

Function Name
void RCC_PLLI2SWconfig ( uint32_t PLLI2SN, uint32_t PLLI2SR)

Function Description
Configures the PLLI2S clock multiplication and division factors.

Parameters
- PLLI2SN : specifies the multiplication factor for PLLI2S VCO output clock This parameter must be a number between 192 and 432.
- PLLI2SR : specifies the division factor for I2S clock This parameter must be a number between 2 and 7.

Return values
- None.

Notes
- PLLI2S is available only in Silicon RevisionB and RevisionY.
- This function must be used only when the PLLI2S is disabled.
- PLLI2S clock source is common with the main PLL (configured in RCC_PLLConfig function )
- You have to set the PLLI2SN parameter correctly to ensure that the VCO output frequency is between 192 and 432 MHz.
- You have to set the PLLI2SR parameter correctly to not exceed 192 MHz on the I2S clock frequency.

19.2.6.11 RCC_PLLI2SCmd

Function Name
void RCC_PLLI2SCmd ( FunctionalState NewState)

Function Description
Enables or disables the PLLI2S.

Parameters
- NewState : new state of the PLLI2S. This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- PLLI2S is available only in RevisionB and RevisionY
- The PLLI2S is disabled by hardware when entering STOP
19.2.6.12 RCC_ClockSecuritySystemCmd

Function Name: void RCC_ClockSecuritySystemCmd ( FunctionalState NewState)

Function Description: Enables or disables the Clock Security System.

Parameters:
- **NewState**: new state of the Clock Security System. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- If a failure is detected on the HSE oscillator clock, this oscillator is automatically disabled and an interrupt is generated to inform the software about the failure (Clock Security System Interrupt, CSSI), allowing the MCU to perform rescue operations. The CSSI is linked to the Cortex-M3 NMI (Non-Maskable Interrupt) exception vector.

19.2.6.13 RCC_MCO1Config

Function Name: void RCC_MCO1Config ( uint32_t RCC_MCO1Source, uint32_t RCC_MCO1Div)

Function Description: Selects the clock source to output on MCO1 pin(PA8).

Parameters:
- **RCC_MCO1Source**: specifies the clock source to output. This parameter can be one of the following values:
  - **RCC_MCO1Source_HSI**: HSI clock selected as MCO1 source
  - **RCC_MCO1Source_LSE**: LSE clock selected as MCO1 source
  - **RCC_MCO1Source_HSE**: HSE clock selected as MCO1 source
  - **RCC_MCO1Source_PLLCLK**: main PLL clock selected as MCO1 source
- **RCC_MCO1Div**: specifies the MCO1 prescaler. This parameter can be one of the following values:
  - **RCC_MCO1Div_1**: no division applied to MCO1 clock
  - **RCC_MCO1Div_2**: division by 2 applied to MCO1 clock
  - **RCC_MCO1Div_3**: division by 3 applied to MCO1 clock
19.2.6.14 RCC_MCO2Config

Function Name: void RCC_MCO2Config (uint32_t RCC_MCO2Source, uint32_t RCC_MCO2Div)

Function Description: Selects the clock source to output on MCO2 pin (PC9).

Parameters:
- **RCC_MCO2Source**: specifies the clock source to output. This parameter can be one of the following values:
  - **RCC_MCO2Source_SYSCLK**: System clock (SYSCLK) selected as MCO2 source
  - **RCC_MCO2Source_PLLI2SCLK**: PLLI2S clock selected as MCO2 source
  - **RCC_MCO2Source_HSE**: HSE clock selected as MCO2 source
  - **RCC_MCO2Source_PLLCLK**: main PLL clock selected as MCO2 source

- **RCC_MCO2Div**: specifies the MCO2 prescaler. This parameter can be one of the following values:
  - **RCC_MCO2Div_1**: no division applied to MCO2 clock
  - **RCC_MCO2Div_2**: division by 2 applied to MCO2 clock
  - **RCC_MCO2Div_3**: division by 3 applied to MCO2 clock
  - **RCC_MCO2Div_4**: division by 4 applied to MCO2 clock
  - **RCC_MCO2Div_5**: division by 5 applied to MCO2 clock

Return values: None.

Notes: PC9 should be configured in alternate function mode.
Reset and clock control (RCC)

Parameters

- **RCC_SYSCLKSource**: specifies the clock source used as system clock. This parameter can be one of the following values:
  - **RCC_SYSCLKSource_HSI**: HSI selected as system clock source
  - **RCC_SYSCLKSource_HSE**: HSE selected as system clock source
  - **RCC_SYSCLKSource_PLLCLK**: PLL selected as system clock source

Return values

- None.

Notes

- The HSI is used (enabled by hardware) as system clock source after startup from Reset, wake-up from STOP and STANDBY mode, or in case of failure of the HSE used directly or indirectly as system clock (if the Clock Security System CSS is enabled).
- A switch from one clock source to another occurs only if the target clock source is ready (clock stable after startup delay or PLL locked). If a clock source which is not yet ready is selected, the switch will occur when the clock source will be ready. You can use `RCC_GetSYSCLKSource()` function to know which clock is currently used as system clock source.

### 19.2.7.2 RCC_GetSYSCLKSource

**Function Name**

uint8_t RCC_GetSYSCLKSource ( void )

**Function Description**

Returns the clock source used as system clock.

**Parameters**

- None.

**Return values**

- The clock source used as system clock. The returned value can be one of the following:
  - 0x00: HSI used as system clock
  - 0x04: HSE used as system clock
  - 0x08: PLL used as system clock

**Notes**

- None.

### 19.2.7.3 RCC_HCLKConfig

**Function Name**

void RCC_HCLKConfig ( uint32_t RCC_SYSCLK )
### 19.2.7.4 RCC_PCLK1Config

**Function Name**
void RCC_PCLK1Config ( uint32_t RCC_HCLK)

**Function Description**
Configures the Low Speed APB clock (PCLK1).

**Parameters**
- **RCC_HCLK**: defines the APB1 clock divider. This clock is derived from the AHB clock (HCLK). This parameter can be one of the following values:
  - **RCC_HCLK_Div1**: APB1 clock = HCLK
  - **RCC_HCLK_Div2**: APB1 clock = HCLK/2
  - **RCC_HCLK_Div4**: APB1 clock = HCLK/4
  - **RCC_HCLK_Div8**: APB1 clock = HCLK/8
  - **RCC_HCLK_Div16**: APB1 clock = HCLK/16
  - **RCC_HCLK_Div64**: APB1 clock = HCLK/64
  - **RCC_HCLK_Div128**: APB1 clock = HCLK/128
  - **RCC_HCLK_Div256**: APB1 clock = HCLK/256
  - **RCC_HCLK_Div512**: APB1 clock = HCLK/512

**Return values**
- None.

**Notes**
- Depending on the device voltage range, the software has to set correctly these bits to ensure that HCLK not exceed the maximum allowed frequency (for more details refer to section above "CPU, AHB and APB busses clocks configuration functions")
Function Name: void RCC_PCLK2Config (uint32_t RCC_HCLK)

Function Description: Configures the High Speed APB clock (PCLK2).

Parameters:
- RCC_HCLK: defines the APB2 clock divider. This clock is derived from the AHB clock (HCLK). This parameter can be one of the following values:
  - RCC_HCLK_Div1: APB2 clock = HCLK
  - RCC_HCLK_Div2: APB2 clock = HCLK/2
  - RCC_HCLK_Div4: APB2 clock = HCLK/4
  - RCC_HCLK_Div8: APB2 clock = HCLK/8
  - RCC_HCLK_Div16: APB2 clock = HCLK/16

Return values:
- None.

Notes:
- None.

19.2.7.6 RCC_GetClocksFreq

Function Name: void RCC_GetClocksFreq (RCC_ClocksTypeDef *RCC_Clocks)

Function Description: Returns the frequencies of different on chip clocks; SYSCLK, HCLK, PCLK1 and PCLK2.

Parameters:
- RCC_Clocks: pointer to a RCC_ClocksTypeDef structure which will hold the clocks frequencies.

Return values:
- None.

Notes:
- The system frequency computed by this function is not the real frequency in the chip. It is calculated based on the predefined constant and the selected clock source:
  - If SYSCLK source is HSI, function returns values based on HSI_VALUE(*)
  - If SYSCLK source is HSE, function returns values based on HSE_VALUE(**)
  - If SYSCLK source is PLL, function returns values based on HSE_VALUE(**) or HSI_VALUE(*) multiplied/divided by the PLL factors.
  - (*) HSI_VALUE is a constant defined in stm32f2xx.h file (default value 16 MHz) but the real value may vary depending on the variations in voltage and temperature.
  - (**) HSE_VALUE is a constant defined in stm32f2xx.h file (default value 25 MHz), user has to ensure that HSE_VALUE is same as the real frequency of the crystal used. Otherwise, this function may have wrong result.
  - The result of this function could be not correct when using fractional value for HSE crystal.
  - This function can be used by the user application to compute the baudrate for the communication peripherals or configure...
other parameters.

- Each time SYSCLK, HCLK, PCLK1 and/or PCLK2 clock changes, this function must be called to update the structure's field. Otherwise, any configuration based on this function will be incorrect.

19.2.8 Peripheral clocks configuration functions

19.2.8.1 RCC_RTCCLKConfig

Function Name: void RCC_RTCCLKConfig ( uint32_t RCC_RTCCLKSource)

Function Description: Configures the RTC clock (RTTCLK).

Parameters:

- RCC_RTCCLKSource : specifies the RTC clock source. This parameter can be one of the following values:
  - RCC_RTCCLKSource_LSE : LSE selected as RTC clock
  - RCC_RTCCLKSource_LSI : LSI selected as RTC clock
  - RCC_RTCCLKSource_HSE_Divx : HSE clock divided by x selected as RTC clock, where x:[2,31]

Return values:

- None.

Notes:

- As the RTC clock configuration bits are in the Backup domain and write access is denied to this domain after reset, you have to enable write access using PWR_BackupAccessCmd(ENABLE) function before to configure the RTC clock source (to be done once after reset).
- Once the RTC clock is configured it can't be changed unless the Backup domain is reset using RCC_BackupResetCmd() function, or by a Power On Reset (POR).
- If the LSE or LSI is used as RTC clock source, the RTC continues to work in STOP and STANDBY modes, and can be used as wakeup source. However, when the HSE clock is used as RTC clock source, the RTC cannot be used in STOP and STANDBY modes.
- The maximum input clock frequency for RTC is 1MHz (when using HSE as RTC clock source).

19.2.8.2 RCC_RTCCLKCmd
Function Name: void RCC_RTCCLKCmd (FunctionalState NewState)  
Function Description: Enables or disables the RTC clock.  
Parameters:
- **NewState**: new state of the RTC clock. This parameter can be: ENABLE or DISABLE.  
Return values:
- None.  
Notes:
- This function must be used only after the RTC clock source was selected using the RCC_RTCCLKConfig function.

### 19.2.8.3 RCC_BackupResetCmd

Function Name: void RCC_BackupResetCmd (FunctionalState NewState)  
Function Description: Forces or releases the Backup domain reset.  
Parameters:
- **NewState**: new state of the Backup domain reset. This parameter can be: ENABLE or DISABLE.  
Return values:
- None.  
Notes:
- This function resets the RTC peripheral (including the backup registers) and the RTC clock source selection in RCC_CSR register.  
- The BKPSRAM is not affected by this reset.

### 19.2.8.4 RCC_I2SCLKConfig

Function Name: void RCC_I2SCLKConfig (uint32_t RCC_I2SCLKSource)  
Function Description: Configures the I2S clock source (I2SCLK).  
Parameters:
- **RCC_I2SCLKSource**: specifies the I2S clock source. This parameter can be one of the following values:
  - **RCC_I2S2CLKSource_PLLI2S**: PLLI2S clock used as I2S clock source  
  - **RCC_I2S2CLKSource_Ext**: External clock mapped on the I2S_CKIN pin used as I2S clock source  
Return values:
- None.  
Notes:
- This function must be called before enabling the I2S APB clock.  
- This function applies only to Silicon RevisionB and RevisionY.
19.2.8.5 RCC_AHB1PeriphClockCmd

Function Name: void RCC_AHB1PeriphClockCmd ( uint32_t RCC_AHB1Periph, FunctionalState NewState)

Function Description: Enables or disables the AHB1 peripheral clock.

Parameters:
- **RCC_AHBPeriph**: specifies the AHB1 peripheral to gates its clock. This parameter can be any combination of the following values:
  - RCC_AHB1Periph_GPIOA: GPIOA clock
  - RCC_AHB1Periph_GPIOB: GPIOB clock
  - RCC_AHB1Periph_GPIOC: GPIOC clock
  - RCC_AHB1Periph_GPIOD: GPIOD clock
  - RCC_AHB1Periph_GPIOE: GPIOE clock
  - RCC_AHB1Periph_GPIOF: GPIOF clock
  - RCC_AHB1Periph_GPIOG: GPIOG clock
  - RCC_AHB1Periph_GPIOH: GPIOH clock
  - RCC_AHB1Periph_CRC: CRC clock
  - RCC_AHB1Periph_BKPSRAM: BKPSRAM interface clock
  - RCC_AHB1Periph_DMA1: DMA1 clock
  - RCC_AHB1Periph_DMA2: DMA2 clock
  - RCC_AHB1Periph_ETH_MAC: Ethernet MAC clock
  - RCC_AHB1Periph_ETH_MAC_Tx: Ethernet Transmission clock
  - RCC_AHB1Periph_ETH_MAC_Rx: Ethernet Reception clock
  - RCC_AHB1Periph_ETH_MAC_PTP: Ethernet PTP clock
  - RCC_AHB1Periph_OTG_HS: USB OTG HS clock
  - RCC_AHB1Periph_OTG_HS_ULPI: USB OTG HS ULPI clock

- **NewState**: new state of the specified peripheral clock. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes:
- After reset, the peripheral clock (used for registers read/write access) is disabled and the application software has to enable this clock before using it.
Function Name: void RCC_AHB2PeriphClockCmd (uint32_t RCC_AHB2Periph, FunctionalState NewState)

Function Description: Enables or disables the AHB2 peripheral clock.

Parameters:
- RCC_AHBPeriph: specifies the AHB2 peripheral to gates its clock. This parameter can be any combination of the following values:
  - RCC_AHB2Periph_DCMI: DCMI clock
  - RCC_AHB2Periph_CRYP: CRYP clock
  - RCC_AHB2Periph_HASH: HASH clock
  - RCC_AHB2Periph_RNG: RNG clock
  - RCC_AHB2Periph_OTG_FS: USB OTG FS clock
- NewState: new state of the specified peripheral clock. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes:
- After reset, the peripheral clock (used for registers read/write access) is disabled and the application software has to enable this clock before using it.

Function Name: void RCC_AHB3PeriphClockCmd (uint32_t RCC_AHB3Periph, FunctionalState NewState)

Function Description: Enables or disables the AHB3 peripheral clock.

Parameters:
- RCC_AHBPeriph: specifies the AHB3 peripheral to gates its clock. This parameter must be: RCC_AHB3Periph_FSMC
- NewState: new state of the specified peripheral clock. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes:
- After reset, the peripheral clock (used for registers read/write access) is disabled and the application software has to enable this clock before using it.

Function Name: void RCC_APB1PeriphClockCmd

Function Description: Enables or disables the APB1 peripheral clock.
Function Name: void RCC_APB1PeriphClockCmd ( uint32_t RCC_APB1Periph, FunctionalState NewState)

Function Description: Enables or disables the Low Speed APB (APB1) peripheral clock.

Parameters:
- RCC_APB1Periph: specifies the APB1 peripheral to gates its clock. This parameter can be any combination of the following values:
  - RCC_APB1Periph_TIM2: TIM2 clock
  - RCC_APB1Periph_TIM3: TIM3 clock
  - RCC_APB1Periph_TIM4: TIM4 clock
  - RCC_APB1Periph_TIM5: TIM5 clock
  - RCC_APB1Periph_TIM6: TIM6 clock
  - RCC_APB1Periph_TIM7: TIM7 clock
  - RCC_APB1Periph_TIM12: TIM12 clock
  - RCC_APB1Periph_TIM13: TIM13 clock
  - RCC_APB1Periph_TIM14: TIM14 clock
  - RCC_APB1Periph_WWDG: WWDG clock
  - RCC_APB1Periph_SPI2: SPI2 clock
  - RCC_APB1Periph_SPI3: SPI3 clock
  - RCC_APB1Periph_USART2: USART2 clock
  - RCC_APB1Periph_USART3: USART3 clock
  - RCC_APB1Periph_UART4: UART4 clock
  - RCC_APB1Periph_UART5: UART5 clock
  - RCC_APB1Periph_I2C1: I2C1 clock
  - RCC_APB1Periph_I2C2: I2C2 clock
  - RCC_APB1Periph_I2C3: I2C3 clock
  - RCC_APB1Periph_CAN1: CAN1 clock
  - RCC_APB1Periph_CAN2: CAN2 clock
  - RCC_APB1Periph_PWR: PWR clock
  - RCC_APB1Periph_DAC: DAC clock

- NewState: new state of the specified peripheral clock. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes:
- After reset, the peripheral clock (used for registers read/write access) is disabled and the application software has to enable this clock before using it.

19.2.8.9 RCC_APB2PeriphClockCmd

Function Name: void RCC_APB2PeriphClockCmd ( uint32_t RCC_APB2Periph, FunctionalState NewState)

Function Description: Enables or disables the High Speed APB (APB2) peripheral clock.

Parameters:
- RCC_APB2Periph: specifies the APB2 peripheral to gates
its clock. This parameter can be any combination of the following values:

- `RCC_APB2Periph_TIM1` : TIM1 clock
- `RCC_APB2Periph_TIM8` : TIM8 clock
- `RCC_APB2Periph_USART1` : USART1 clock
- `RCC_APB2Periph_USART6` : USART6 clock
- `RCC_APB2Periph_ADC1` : ADC1 clock
- `RCC_APB2Periph_ADC2` : ADC2 clock
- `RCC_APB2Periph_ADC3` : ADC3 clock
- `RCC_APB2Periph_SDIO` : SDIO clock
- `RCC_APB2Periph_SPI1` : SPI1 clock
- `RCC_APB2Periph_SYSCFG` : SYSCFG clock
- `RCC_APB2Periph_TIM9` : TIM9 clock
- `RCC_APB2Periph_TIM10` : TIM10 clock
- `RCC_APB2Periph_TIM11` : TIM11 clock

- **NewState** : new state of the specified peripheral clock. This parameter can be: ENABLE or DISABLE.

**Return values**

- None.

**Notes**

- After reset, the peripheral clock (used for registers read/write access) is disabled and the application software has to enable this clock before using it.

### 19.2.8.10 RCC_AHB1PeriphResetCmd

**Function Name**

```c
void RCC_AHB1PeriphResetCmd ( uint32_t RCC_AHB1Periph, FunctionalState NewState)
```

**Function Description**

Forces or releases AHB1 peripheral reset.

**Parameters**

- **RCC_AHB1Periph** : specifies the AHB1 peripheral to reset. This parameter can be any combination of the following values:
  - `RCC_AHB1Periph_GPIOA` : GPIOA clock
  - `RCC_AHB1Periph_GPIOB` : GPIOB clock
  - `RCC_AHB1Periph_GPIOC` : GPIOC clock
  - `RCC_AHB1Periph_GPIOD` : GPIOD clock
  - `RCC_AHB1Periph_GPIOE` : GPIOE clock
  - `RCC_AHB1Periph_GPIOF` : GPIOF clock
  - `RCC_AHB1Periph_GPIOG` : GPIOG clock
  - `RCC_AHB1Periph_GPIOH` : GPIOH clock
  - `RCC_AHB1Periph_CRC` : CRC clock
  - `RCC_AHB1Periph_DMA1` : DMA1 clock
  - `RCC_AHB1Periph_DMA2` : DMA2 clock
  - `RCC_AHB1Periph_ETH_MAC` : Ethernet MAC clock
  - `RCC_AHB1Periph_OTG_HS` : USB OTG HS clock
19.2.8.11 RCC_AHB2PeriphResetCmd

Function Name: void RCC_AHB2PeriphResetCmd ( uint32_t RCC_AHB2Periph, FunctionalState NewState)

Function Description: Forces or releases AHB2 peripheral reset.

Parameters:
- **RCC_AHB2Periph**: specifies the AHB2 peripheral to reset. This parameter can be any combination of the following values:
  - RCC_AHB2Periph_DCMI: DCMI clock
  - RCC_AHB2Periph_CRYP: CRYP clock
  - RCC_AHB2Periph_HASH: HASH clock
  - RCC_AHB2Periph_RNG: RNG clock
  - RCC_AHB2Periph_OTG_FS: USB OTG FS clock
- **NewState**: new state of the specified peripheral reset. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

19.2.8.12 RCC_AHB3PeriphResetCmd

Function Name: void RCC_AHB3PeriphResetCmd ( uint32_t RCC_AHB3Periph, FunctionalState NewState)

Function Description: Forces or releases AHB3 peripheral reset.

Parameters:
- **RCC_AHB3Periph**: specifies the AHB3 peripheral to reset. This parameter must be: RCC_AHB3Periph_FSMC
- **NewState**: new state of the specified peripheral reset. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.
19.2.8.13  RCC_APB1PeriphResetCmd

**Function Name**  
void RCC_APB1PeriphResetCmd ( uint32_t RCC_APB1Periph,  
FunctionalState NewState)

**Function Description**  
Forces or releases Low Speed APB (APB1) peripheral reset.

**Parameters**  
- **RCC_APB1Periph**: specifies the APB1 peripheral to reset. This parameter can be any combination of the following values:
  - RCC_APB1Periph_TIM2: TIM2 clock
  - RCC_APB1Periph_TIM3: TIM3 clock
  - RCC_APB1Periph_TIM4: TIM4 clock
  - RCC_APB1Periph_TIM5: TIM5 clock
  - RCC_APB1Periph_TIM6: TIM6 clock
  - RCC_APB1Periph_TIM7: TIM7 clock
  - RCC_APB1Periph_TIM12: TIM12 clock
  - RCC_APB1Periph_TIM13: TIM13 clock
  - RCC_APB1Periph_TIM14: TIM14 clock
  - RCC_APB1Periph_WWDG: WWDG clock
  - RCC_APB1Periph_SPI2: SPI2 clock
  - RCC_APB1Periph_SPI3: SPI3 clock
  - RCC_APB1Periph_USART2: USART2 clock
  - RCC_APB1Periph_USART3: USART3 clock
  - RCC_APB1Periph_UART4: UART4 clock
  - RCC_APB1Periph_UART5: UART5 clock
  - RCC_APB1Periph_I2C1: I2C1 clock
  - RCC_APB1Periph_I2C2: I2C2 clock
  - RCC_APB1Periph_I2C3: I2C3 clock
  - RCC_APB1Periph_CAN1: CAN1 clock
  - RCC_APB1Periph_CAN2: CAN2 clock
  - RCC_APB1Periph_PWR: PWR clock
  - RCC_APB1Periph_DAC: DAC clock

- **NewState**: new state of the specified peripheral reset. This parameter can be: ENABLE or DISABLE.

**Return values**  
- None.

**Notes**  
- None.

19.2.8.14  RCC_APB2PeriphResetCmd
Function Name: void 
RCC_APB2PeriphResetCmd ( uint32_t 
RCC_APB2Periph, 
FunctionalState NewState)

Function Description: Forces or releases High Speed APB (APB2) peripheral reset.

Parameters:
- **RCC_APB2Periph**: specifies the APB2 peripheral to reset. This parameter can be any combination of the following values:
  - RCC_APB2Periph_TIM1: TIM1 clock
  - RCC_APB2Periph_TIM8: TIM8 clock
  - RCC_APB2Periph_USART1: USART1 clock
  - RCC_APB2Periph_USART6: USART6 clock
  - RCC_APB2Periph_ADC1: ADC1 clock
  - RCC_APB2Periph_ADC2: ADC2 clock
  - RCC_APB2Periph_ADC3: ADC3 clock
  - RCC_APB2Periph_SDIO: SDIO clock
  - RCC_APB2Periph_SPI1: SPI1 clock
  - RCC_APB2Periph_SYSCFG: SYSCFG clock
  - RCC_APB2Periph_TIM9: TIM9 clock
  - RCC_APB2Periph_TIM10: TIM10 clock
  - RCC_APB2Periph_TIM11: TIM11 clock
- **NewState**: new state of the specified peripheral reset. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.

**19.2.8.15 RCC_AHB1PeriphClockLPModeCmd**

Function Name: void 
RCC_AHB1PeriphClockLPModeCmd ( uint32_t 
RCC_AHB1Periph, 
FunctionalState NewState)

Function Description: Enables or disables the AHB1 peripheral clock during Low Power (Sleep) mode.

Parameters:
- **RCC_AHBPeriph**: specifies the AHB1 peripheral to gates its clock. This parameter can be any combination of the following values:
  - RCC_AHB1Periph_GPIOA: GPIOA clock
  - RCC_AHB1Periph_GPIOB: GPIOB clock
  - RCC_AHB1Periph_GPIOC: GPIOC clock
  - RCC_AHB1Periph_GPIOD: GPIOD clock
  - RCC_AHB1Periph_GPIOE: GPIOE clock
  - RCC_AHB1Periph_GPIOF: GPIOF clock
  - RCC_AHB1Periph_GPIOG: GPIOG clock
  - RCC_AHB1Periph_GPIOH: GPIOH clock
  - RCC_AHB1Periph_CRC: CRC clock
  - RCC_AHB1Periph_BKPSRAM: BKPSRAM interface

clock
- `RCC_AHB1Periph_DMA1`: DMA1 clock
- `RCC_AHB1Periph_DMA2`: DMA2 clock
- `RCC_AHB1Periph_ETH_MAC`: Ethernet MAC clock
- `RCC_AHB1Periph_ETH_MAC_Tx`: Ethernet Transmission clock
- `RCC_AHB1Periph_ETH_MAC_Rx`: Ethernet Reception clock
- `RCC_AHB1Periph_ETH_MAC_PTP`: Ethernet PTP clock
- `RCC_AHB1Periph_OTG_HS`: USB OTG HS clock
- `RCC_AHB1Periph_OTG_HS_ULPI`: USB OTG HS ULPI clock

- **NewState**: new state of the specified peripheral clock. This parameter can be: ENABLE or DISABLE.

### Notes
- Peripheral clock gating in SLEEP mode can be used to further reduce power consumption.
- After wakeup from SLEEP mode, the peripheral clock is enabled again.
- By default, all peripheral clocks are enabled during SLEEP mode.

#### 19.2.8.16 RCC_AHB2PeriphClockLPModeCmd

**Function Name**
void RCC_AHB2PeriphClockLPModeCmd ( uint32_t RCC_AHB2Periph, FunctionalState NewState)

**Function Description**
Enables or disables the AHB2 peripheral clock during Low Power (Sleep) mode.

**Parameters**
- **RCC_AHBPeripheral**: specifies the AHB2 peripheral to gates its clock. This parameter can be any combination of the following values:
  - `RCC_AHB2Periph_DCI`: DCMI clock
  - `RCC_AHB2Periph_CRYP`: CRYP clock
  - `RCC_AHB2Periph_HASH`: HASH clock
  - `RCC_AHB2Periph_RNG`: RNG clock
  - `RCC_AHB2Periph_OTG_FS`: USB OTG FS clock

- **NewState**: new state of the specified peripheral clock. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.

**Notes**
- Peripheral clock gating in SLEEP mode can be used to further reduce power consumption.
- After wakeup from SLEEP mode, the peripheral clock is enabled again.
• By default, all peripheral clocks are enabled during SLEEP mode.

19.2.8.17 RCC_AHB3PeriphClockLPModeCmd

Function Name: void RCC_AHB3PeriphClockLPModeCmd (uint32_t RCC_AHB3Periph, FunctionalState NewState)

Function Description: Enables or disables the AHB3 peripheral clock during Low Power (Sleep) mode.

Parameters:
- **RCC_AHBPeriph**: specifies the AHB3 peripheral to gate its clock. This parameter must be: RCC_AHB3Periph_FSMC
- **NewState**: new state of the specified peripheral clock. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes:
- Peripheral clock gating in SLEEP mode can be used to further reduce power consumption.
- After wakeup from SLEEP mode, the peripheral clock is enabled again.
- By default, all peripheral clocks are enabled during SLEEP mode.

19.2.8.18 RCC_APB1PeriphClockLPModeCmd

Function Name: void RCC_APB1PeriphClockLPModeCmd (uint32_t RCC_APB1Periph, FunctionalState NewState)

Function Description: Enables or disables the APB1 peripheral clock during Low Power (Sleep) mode.

Parameters:
- **RCC_APB1Periph**: specifies the APB1 peripheral to gate its clock. This parameter can be any combination of the following values:
  - **RCC_APB1Periph_TIM2**: TIM2 clock
  - **RCC_APB1Periph_TIM3**: TIM3 clock
  - **RCC_APB1Periph_TIM4**: TIM4 clock
  - **RCC_APB1Periph_TIM5**: TIM5 clock
  - **RCC_APB1Periph_TIM6**: TIM6 clock
  - **RCC_APB1Periph_TIM7**: TIM7 clock
  - **RCC_APB1Periph_TIM12**: TIM12 clock
Reset and clock control (RCC)

- RCC_APB1Periph_TIM13 : TIM13 clock
- RCC_APB1Periph_TIM14 : TIM14 clock
- RCC_APB1Periph_WWDG : WWDG clock
- RCC_APB1Periph_SPI2 : SPI2 clock
- RCC_APB1Periph_SPI3 : SPI3 clock
- RCC_APB1Periph_USART2 : USART2 clock
- RCC_APB1Periph_USART3 : USART3 clock
- RCC_APB1Periph_UART4 : UART4 clock
- RCC_APB1Periph_UART5 : UART5 clock
- RCC_APB1Periph_I2C1 : I2C1 clock
- RCC_APB1Periph_I2C2 : I2C2 clock
- RCC_APB1Periph_I2C3 : I2C3 clock
- RCC_APB1Periph_CAN1 : CAN1 clock
- RCC_APB1Periph_CAN2 : CAN2 clock
- RCC_APB1Periph_PWR : PWR clock
- RCC_APB1Periph_DAC : DAC clock

- NewState : new state of the specified peripheral clock. This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- Peripheral clock gating in SLEEP mode can be used to further reduce power consumption.
- After wakeup from SLEEP mode, the peripheral clock is enabled again.
- By default, all peripheral clocks are enabled during SLEEP mode.

19.2.8.19 RCC_APB2PeriphClockLPModeCmd

Function Name
void RCC_APB2PeriphClockLPModeCmd ( uint32_t RCC_APB2Periph, FunctionalState NewState)

Function Description
Enables or disables the APB2 peripheral clock during Low Power (Sleep) mode.

Parameters
- RCC_APB2Periph : specifies the APB2 peripheral to gates its clock. This parameter can be any combination of the following values:
  - RCC_APB2Periph_TIM1 : TIM1 clock
  - RCC_APB2Periph_TIM8 : TIM8 clock
  - RCC_APB2Periph_USART1 : USART1 clock
  - RCC_APB2Periph_USART6 : USART6 clock
  - RCC_APB2Periph_ADC1 : ADC1 clock
  - RCC_APB2Periph_ADC2 : ADC2 clock
  - RCC_APB2Periph_ADC3 : ADC3 clock
  - RCC_APB2Periph_SDIO : SDIO clock
  - RCC_APB2Periph_SPI1 : SPI1 clock
Reset and clock control (RCC)

- **RCC_APB2Periph_SYSCFG**: SYSCFG clock
- **RCC_APB2Periph_TIM9**: TIM9 clock
- **RCC_APB2Periph_TIM10**: TIM10 clock
- **RCC_APB2Periph_TIM11**: TIM11 clock

- **NewState**: new state of the specified peripheral clock. This parameter can be: ENABLE or DISABLE.

### 19.2.9 Interrupt and flag management functions

#### 19.2.9.1 RCC_ITConfig

**Function Name**: void RCC_ITConfig ( uint8_t RCC_IT, FunctionalState NewState)

**Function Description**: Enables or disables the specified RCC interrupts.

**Parameters**

- **RCC_IT**: specifies the RCC interrupt sources to be enabled or disabled. This parameter can be any combination of the following values:
  - **RCC_IT_LSIRDY**: LSI ready interrupt
  - **RCC_IT_LSERDY**: LSE ready interrupt
  - **RCC_IT_HSIRDY**: HSI ready interrupt
  - **RCC_IT_HSERDY**: HSE ready interrupt
  - **RCC_IT_PLLRDY**: main PLL ready interrupt
  - **RCC_IT_PLLI2SRDY**: PLLI2S ready interrupt

- **NewState**: new state of the specified RCC interrupts. This parameter can be: ENABLE or DISABLE.

**Return values**

- None.

**Notes**

- Peripheral clock gating in SLEEP mode can be used to further reduce power consumption.
- After wakeup from SLEEP mode, the peripheral clock is enabled again.
- By default, all peripheral clocks are enabled during SLEEP mode.

#### 19.2.9.2 RCC_GetFlagStatus
Function Name: FlagStatus RCC_GetFlagStatus ( uint8_t RCC_FLAG)
Function Description: Checks whether the specified RCC flag is set or not.
Parameters:
- RCC_FLAG: specifies the flag to check. This parameter can be one of the following values:
  - RCC_FLAG_HSIRDY: HSI oscillator clock ready
  - RCC_FLAG_HSERDY: HSE oscillator clock ready
  - RCC_FLAG_PLLRDY: main PLL clock ready
  - RCC_FLAG_PLLI2SRDY: PLLI2S clock ready
  - RCC_FLAG_LSERDY: LSE oscillator clock ready
  - RCC_FLAG_LSIRDY: LSI oscillator clock ready
  - RCC_FLAG_BORRST: POR/PDR or BOR reset
  - RCC_FLAG_PINRST: Pin reset
  - RCC_FLAG_PORRST: POR/PDR reset
  - RCC_FLAG_SFTRST: Software reset
  - RCC_FLAG_IWDGRST: Independent Watchdog reset
  - RCC_FLAG_WWDGRST: Window Watchdog reset
  - RCC_FLAG_LPWRRST: Low Power reset

Return values:
- The new state of RCC_FLAG (SET or RESET).

Notes:
- None.

### 19.2.9.3 RCC_ClearFlag

Function Name: void RCC_ClearFlag ( void )
Function Description: Clears the RCC reset flags.
Parameters:
- None.

Return values:
- None.

Notes:
- None.

### 19.2.9.4 RCC_GetITStatus

Function Name: ITStatus RCC_GetITStatus ( uint8_t RCC_IT)
Function Description: Checks whether the specified RCC interrupt has occurred or not.
Parameters:
- RCC_IT: specifies the RCC interrupt source to check. This parameter can be one of the following values:
  - RCC_IT_LSIRDY: LSI ready interrupt
19.2.9.5 RCC_ClearITPendingBit

Function Name

void RCC_ClearITPendingBit ( uint8_t RCC_IT)

Function Description

Clears the RCC's interrupt pending bits.

Parameters

- RCC_IT : specifies the interrupt pending bit to clear. This parameter can be any combination of the following values:
  - RCC_IT_LSIRDY : LSI ready interrupt
  - RCC_IT_LSERDY : LSE ready interrupt
  - RCC_IT_HSIRDY : HSI ready interrupt
  - RCC_IT_HSERDY : HSE ready interrupt
  - RCC_IT_PLLRDY : main PLL ready interrupt
  - RCC_IT_PLLI2SRDY : PLLI2S ready interrupt
  - RCC_IT_CSS : Clock Security System interrupt

Return values

- None.

Notes

- None.
- #define: `RCC_AHB1Periph_GPIOC((uint32_t)0x00000004)`

- #define: `RCC_AHB1Periph_GPIOD((uint32_t)0x00000008)`

- #define: `RCC_AHB1Periph_GPIOE((uint32_t)0x00000010)`

- #define: `RCC_AHB1Periph_GPIOF((uint32_t)0x00000020)`

- #define: `RCC_AHB1Periph_GPIOG((uint32_t)0x00000040)`

- #define: `RCC_AHB1Periph_GPIOH((uint32_t)0x00000080)`

- #define: `RCC_AHB1Periph_GPIOI((uint32_t)0x00000100)`

- #define: `RCC_AHB1Periph_CRC((uint32_t)0x00001000)`

- #define: `RCC_AHB1Periph_FLITF((uint32_t)0x00008000)`

- #define: `RCC_AHB1Periph_SRAM1((uint32_t)0x00010000)`

- #define: `RCC_AHB1Periph_SRAM2((uint32_t)0x00020000)`

- #define: `RCC_AHB1Periph_BKPSRAM((uint32_t)0x00040000)`
Reset and clock control (RCC)

- `#define RCC_AHB1Periph_DMA1((uint32_t)0x00200000)`

- `#define RCC_AHB1Periph_DMA2((uint32_t)0x00400000)`

- `#define RCC_AHB1Periph_ETH_MAC((uint32_t)0x02000000)`

- `#define RCC_AHB1Periph_ETH_MAC_Tx((uint32_t)0x04000000)`

- `#define RCC_AHB1Periph_ETH_MAC_Rx((uint32_t)0x08000000)`

- `#define RCC_AHB1Periph_ETH_MAC_PTP((uint32_t)0x10000000)`

- `#define RCC_AHB1Periph_OTG_HS((uint32_t)0x20000000)`

- `#define RCC_AHB1Periph_OTG_HS_ULPI((uint32_t)0x40000000)`

- `#define RCC_AHB2Periph_DCMI((uint32_t)0x00000001)`

- `#define RCC_AHB2Periph_CRYP((uint32_t)0x00000010)`

- `#define RCC_AHB2Periph_HASH((uint32_t)0x00000020)`

- `#define RCC_AHB2Periph_RNG((uint32_t)0x00000040)`

**RCC_AHB2_Peripherals**

- `#define RCC_AHB2Periph_DCMI((uint32_t)0x00000001)`

- `#define RCC_AHB2Periph_CRYP((uint32_t)0x00000010)`

- `#define RCC_AHB2Periph_HASH((uint32_t)0x00000020)`

- `#define RCC_AHB2Periph_RNG((uint32_t)0x00000040)`
- `#define: RCC_AHB2Periph_OTG_FS((uint32_t)0x00000080)`

**RCC_AHB3_Peripherals**
- `#define: RCC_AHB3Periph_FSMC((uint32_t)0x00000001)`

**RCC_AHB_Clock_Source**
- `#define: RCC_SYSCLK_Div1((uint32_t)0x00000000)`
- `#define: RCC_SYSCLK_Div2((uint32_t)0x00000080)`
- `#define: RCC_SYSCLK_Div4((uint32_t)0x00000090)`
- `#define: RCC_SYSCLK_Div8((uint32_t)0x000000A0)`
- `#define: RCC_SYSCLK_Div16((uint32_t)0x000000B0)`
- `#define: RCC_SYSCLK_Div64((uint32_t)0x000000C0)`
- `#define: RCC_SYSCLK_Div128((uint32_t)0x000000D0)`
- `#define: RCC_SYSCLK_Div256((uint32_t)0x000000E0)`
- `#define: RCC_SYSCLK_Div512((uint32_t)0x000000F0)`
**RCC_APB1_APB2_Clock_Source**

- `#define: RCC_HCLK_Div1((uint32_t)0x00000000)`

- `#define: RCC_HCLK_Div2((uint32_t)0x00001000)`

- `#define: RCC_HCLK_Div4((uint32_t)0x00001400)`

- `#define: RCC_HCLK_Div8((uint32_t)0x00001800)`

- `#define: RCC_HCLK_Div16((uint32_t)0x00001C00)`

**RCC_APB1_Peripherals**

- `#define: RCC_APB1Periph_TIM2((uint32_t)0x00000001)`

- `#define: RCC_APB1Periph_TIM3((uint32_t)0x00000002)`

- `#define: RCC_APB1Periph_TIM4((uint32_t)0x00000004)`

- `#define: RCC_APB1Periph_TIM5((uint32_t)0x00000008)`

- `#define: RCC_APB1Periph_TIM6((uint32_t)0x00000010)`

- `#define: RCC_APB1Periph_TIM7((uint32_t)0x00000020)`
- #define: `RCC_APB1Periph_TIM12((uint32_t)0x00000040)`

- #define: `RCC_APB1Periph_TIM13((uint32_t)0x00000080)`

- #define: `RCC_APB1Periph_TIM14((uint32_t)0x00000100)`

- #define: `RCC_APB1Periph_WWDG((uint32_t)0x00000800)`

- #define: `RCC_APB1Periph_SPI2((uint32_t)0x00004000)`

- #define: `RCC_APB1Periph_SPI3((uint32_t)0x00008000)`

- #define: `RCC_APB1Periph_USART2((uint32_t)0x00020000)`

- #define: `RCC_APB1Periph_USART3((uint32_t)0x00040000)`

- #define: `RCC_APB1Periph_UART4((uint32_t)0x00080000)`

- #define: `RCC_APB1Periph_UART5((uint32_t)0x00100000)`

- #define: `RCC_APB1Periph_I2C1((uint32_t)0x00200000)`

- #define: `RCC_APB1Periph_I2C2((uint32_t)0x00400000)`
Reset and clock control (RCC)

- #define: `RCC_APB1Periph_I2C3((uint32_t)0x00800000)`

- #define: `RCC_APB1Periph_CAN1((uint32_t)0x02000000)`

- #define: `RCC_APB1Periph_CAN2((uint32_t)0x04000000)`

- #define: `RCC_APB1Periph_PWR((uint32_t)0x10000000)`

- #define: `RCC_APB1Periph_DAC((uint32_t)0x20000000)`

- #define: `RCC_APB2Periph_TIM1((uint32_t)0x00000001)`

- #define: `RCC_APB2Periph_TIM8((uint32_t)0x00000002)`

- #define: `RCC_APB2Periph_USART1((uint32_t)0x00000010)`

- #define: `RCC_APB2Periph_USART6((uint32_t)0x00000020)`

- #define: `RCC_APB2Periph_ADC((uint32_t)0x00000100)`

- #define: `RCC_APB2Periph_ADC1((uint32_t)0x00000100)`

- #define: `RCC_APB2Periph_ADC2((uint32_t)0x00000200)`

RCC_APB2_Peripherals

- #define: `RCC_APB2Periph_TIM1((uint32_t)0x00000001)`

- #define: `RCC_APB2Periph_TIM8((uint32_t)0x00000002)`

- #define: `RCC_APB2Periph_USART1((uint32_t)0x00000010)`

- #define: `RCC_APB2Periph_USART6((uint32_t)0x00000020)`

- #define: `RCC_APB2Periph_ADC((uint32_t)0x00000100)`

- #define: `RCC_APB2Periph_ADC1((uint32_t)0x00000100)`

- #define: `RCC_APB2Periph_ADC2((uint32_t)0x00000200)`
• #define: `RCC_APB2Periph_ADC3((uint32_t)0x00000400)`

• #define: `RCC_APB2Periph_SDIO((uint32_t)0x00000800)`

• #define: `RCC_APB2Periph_SPI1((uint32_t)0x00001000)`

• #define: `RCC_APB2Periph_SYSCFG((uint32_t)0x00004000)`

• #define: `RCC_APB2Periph_TIM9((uint32_t)0x00010000)`

• #define: `RCC_APB2Periph_TIM10((uint32_t)0x00020000)`

• #define: `RCC_APB2Periph_TIM11((uint32_t)0x00040000)`

\[\text{RCC\_Flag}\]

• #define: `RCC_FLAG_HSIRDY((uint8_t)0x21)`

• #define: `RCC_FLAG_HSERDY((uint8_t)0x31)`

• #define: `RCC_FLAG_PLLRDY((uint8_t)0x39)`

• #define: `RCC_FLAG_PLLI2SRDY((uint8_t)0x3B)`

• #define: `RCC_FLAG_LSERDY((uint8_t)0x41)`
- #define: `RCC_FLAG_LSIRDY((uint8_t)0x61)`

- #define: `RCC_FLAG_BORRST((uint8_t)0x79)`

- #define: `RCC_FLAG_PINRST((uint8_t)0x7A)`

- #define: `RCC_FLAG_PORRST((uint8_t)0x7B)`

- #define: `RCC_FLAG_SFTRST((uint8_t)0x7C)`

- #define: `RCC_FLAG_IWDGRST((uint8_t)0x7D)`

- #define: `RCC_FLAG_WWDGRST((uint8_t)0x7E)`

- #define: `RCC_FLAG_LPWRRST((uint8_t)0x7F)`

*RCC_HSE_configuration*

- #define: `RCC_HSE_OFF((uint8_t)0x00)`

- #define: `RCC_HSE_ON((uint8_t)0x01)`

- #define: `RCC_HSE_Bypass((uint8_t)0x05)`
RCC_I2S_Clock_Source
- #define: RCC_I2S2CLKSource_PLLI2S((uint8_t)0x00)
- #define: RCC_I2S2CLKSource_Ext((uint8_t)0x01)

RCC_Interrupt_Source
- #define: RCC_IT_LSIRDY((uint8_t)0x01)
- #define: RCC_IT_LSERDY((uint8_t)0x02)
- #define: RCC_IT_HSIRDY((uint8_t)0x04)
- #define: RCC_IT_HSERDY((uint8_t)0x08)
- #define: RCC_IT_PLLRDY((uint8_t)0x10)
- #define: RCC_IT_PLLI2SRDY((uint8_t)0x20)
- #define: RCC_IT_CSS((uint8_t)0x80)

RCC_LSE_Configuration
- #define: RCC_LSE_OFF((uint8_t)0x00)
- #define: RCC_LSE_ON((uint8_t)0x01)
- #define: \texttt{RCC\_LSE\_Bypass((uint8\_t)0x04)}

\textit{RCC\_MCO1\_Clock\_Source\_Prescaler}
- #define: \texttt{RCC\_MCO1Source\_HSI((uint32\_t)0x0000000)}
- #define: \texttt{RCC\_MCO1Source\_LSE((uint32\_t)0x00200000)}
- #define: \texttt{RCC\_MCO1Source\_HSE((uint32\_t)0x00400000)}
- #define: \texttt{RCC\_MCO1Source\_PLL\_CLK((uint32\_t)0x00600000)}
- #define: \texttt{RCC\_MCO1Div\_1((uint32\_t)0x00000000)}
- #define: \texttt{RCC\_MCO1Div\_2((uint32\_t)0x04000000)}
- #define: \texttt{RCC\_MCO1Div\_3((uint32\_t)0x05000000)}
- #define: \texttt{RCC\_MCO1Div\_4((uint32\_t)0x06000000)}
- #define: \texttt{RCC\_MCO1Div\_5((uint32\_t)0x07000000)}

\textit{RCC\_MCO2\_Clock\_Source\_Prescaler}
- #define: \texttt{RCC\_MCO2Source\_SYS\_CLK((uint32\_t)0x00000000)}
- #define: \texttt{RCC\_MCO2Source\_PLL\_I\_2\_CLK((uint32\_t)0x40000000)}
- #define: \texttt{RCC\_MCO2Source\_HSE((uint32\_t)0x80000000)}

- #define: \texttt{RCC\_MCO2Source\_PLLCLK((uint32\_t)0xC0000000)}

- #define: \texttt{RCC\_MCO2Div\_1((uint32\_t)0x00000000)}

- #define: \texttt{RCC\_MCO2Div\_2((uint32\_t)0x20000000)}

- #define: \texttt{RCC\_MCO2Div\_3((uint32\_t)0x28000000)}

- #define: \texttt{RCC\_MCO2Div\_4((uint32\_t)0x30000000)}

- #define: \texttt{RCC\_MCO2Div\_5((uint32\_t)0x38000000)}

\texttt{RCC\_PLL\_Clock\_Source}

- #define: \texttt{RCC\_PLLSource\_HSI((uint32\_t)0x00000000)}

- #define: \texttt{RCC\_PLLSource\_HSE((uint32\_t)0x00400000)}

\texttt{RCC\_RTC\_Clock\_Source}

- #define: \texttt{RCC\_RTCCLKSource\_LSE((uint32\_t)0x00000100)}

- #define: \texttt{RCC\_RTCCLKSource\_LSI((uint32\_t)0x00000200)}
- #define: `RCC_RTCCLKSource_HSE_Div2((uint32_t)0x00020300)`

- #define: `RCC_RTCCLKSource_HSE_Div3((uint32_t)0x00030300)`

- #define: `RCC_RTCCLKSource_HSE_Div4((uint32_t)0x00040300)`

- #define: `RCC_RTCCLKSource_HSE_Div5((uint32_t)0x00050300)`

- #define: `RCC_RTCCLKSource_HSE_Div6((uint32_t)0x00060300)`

- #define: `RCC_RTCCLKSource_HSE_Div7((uint32_t)0x00070300)`

- #define: `RCC_RTCCLKSource_HSE_Div8((uint32_t)0x00080300)`

- #define: `RCC_RTCCLKSource_HSE_Div9((uint32_t)0x00090300)`

- #define: `RCC_RTCCLKSource_HSE_Div10((uint32_t)0x000A0300)`

- #define: `RCC_RTCCLKSource_HSE_Div11((uint32_t)0x000B0300)`

- #define: `RCC_RTCCLKSource_HSE_Div12((uint32_t)0x000C0300)`

- #define: `RCC_RTCCLKSource_HSE_Div13((uint32_t)0x000D0300)`
- #define: `RCC_RTCCLKSource_HSE_Div14((uint32_t)0x000E0300)`

- #define: `RCC_RTCCLKSource_HSE_Div15((uint32_t)0x000F0300)`

- #define: `RCC_RTCCLKSource_HSE_Div16((uint32_t)0x00100300)`

- #define: `RCC_RTCCLKSource_HSE_Div17((uint32_t)0x00110300)`

- #define: `RCC_RTCCLKSource_HSE_Div18((uint32_t)0x00120300)`

- #define: `RCC_RTCCLKSource_HSE_Div19((uint32_t)0x00130300)`

- #define: `RCC_RTCCLKSource_HSE_Div20((uint32_t)0x00140300)`

- #define: `RCC_RTCCLKSource_HSE_Div21((uint32_t)0x00150300)`

- #define: `RCC_RTCCLKSource_HSE_Div22((uint32_t)0x00160300)`

- #define: `RCC_RTCCLKSource_HSE_Div23((uint32_t)0x00170300)`

- #define: `RCC_RTCCLKSource_HSE_Div24((uint32_t)0x00180300)`

- #define: `RCC_RTCCLKSource_HSE_Div25((uint32_t)0x00190300)`
• #define: \texttt{RCC\_RTCCLKSource\_HSE\_Div26((uint32\_t)0x001A0300)}

• #define: \texttt{RCC\_RTCCLKSource\_HSE\_Div27((uint32\_t)0x001B0300)}

• #define: \texttt{RCC\_RTCCLKSource\_HSE\_Div28((uint32\_t)0x001C0300)}

• #define: \texttt{RCC\_RTCCLKSource\_HSE\_Div29((uint32\_t)0x001D0300)}

• #define: \texttt{RCC\_RTCCLKSource\_HSE\_Div30((uint32\_t)0x001E0300)}

• #define: \texttt{RCC\_RTCCLKSource\_HSE\_Div31((uint32\_t)0x001F0300)}

\texttt{RCC\_System\_Clock\_Source}

• #define: \texttt{RCC\_SYSCLKSource\_HSI((uint32\_t)0x00000000)}

• #define: \texttt{RCC\_SYSCLKSource\_HSE((uint32\_t)0x00000001)}

• #define: \texttt{RCC\_SYSCLKSource\_PLLCLK((uint32\_t)0x00000002)}

### 19.4 RCC Programming Example

The example below explains how to use the RCC driver to configure the system clock to 120 MHz using the PLL as clock source (you can tailor the parameters PLL\_M, PLL\_N and PLL\_P to have different system clock settings). For more examples about RCC configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx\StdPeriph_Examples\RCC\
/* PLL_VCO = (HSE_VALUE or HSI_VALUE / PLL_M) * PLL_N */
#define PLL_M  25 /* For HSE value equal to 25 MHz */
#define PLL_N  240

/* SYSCLK = PLL_VCO / PLL_P */
#define PLL_P  2

/* USB OTG FS, SDIO and RNG Clock = PLL_VCO / PLLQ */
#define PLL_Q  5

/* In this example: PLL_VCO = 240 MHz SYSCLK = 120 MHz */

/*******************************************************************************/
/* PLL (clocked by HSE) used as System clock(SYSCLK) source */
/*******************************************************************************/
__IO uint32_t StartUpCounter = 0, HSEStartUpStatus = 0;

/* Enable HSE */
RCC_HSEConfig(RCC_HSE_ON);

/* Wait till HSE is ready */
HSEStartUpStatus = RCC_WaitForHSEStartUp();

if (HSEStartUpStatus == SUCCESS)
{
    /* Flash 3 wait state, prefetch buffer and cache ON */
    FLASH_SetLatency(FLASH_Latency_3);
    FLASH_PrefetchBufferCmd(ENABLE);
    FLASH_InstructionCacheCmd(ENABLE);
    FLASH_DataCacheCmd(ENABLE);

    /* HCLK = SYSCLK */
    RCC_HCLKConfig(RCC_SYSCLK_Div1);

    /* PCLK2 = HCLK/2 */
    RCC_PCLK2Config(RCC_HCLK_Div2);

    /* PCLK1 = HCLK/4 */
    RCC_PCLK1Config(RCC_HCLK_Div4);

    /* Configure the main PLL clock to 120 MHz */
    RCC_PLLConfig(RCC_PLLSource_HSE, PLL_M, PLL_N, PLL_P, PLL_Q);

    /* Enable the main PLL */
    RCC_PLLCmd(ENABLE);

    /* Wait till the main PLL is ready */
    while (RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET) {} 

    /* Select the main PLL as system clock source */
    RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
/ * Wait till the main PLL is used as system clock source */
  while (RCC_GetSYSCLKSource() != RCC_CFGR_SWS_PLL)
  {
  }
else
{ /* If HSE fails to start-up, user can add here some code
to deal with this error */
}
20 Random number generator (RNG)

20.1 RNG Firmware driver registers structures

20.1.1 RNG_TypeDef

RNG_TypeDef is defined in the stm32f2xx.h file and contains the RNG registers definition.

Data Fields

- __IO uint32_t CR
- __IO uint32_t SR
- __IO uint32_t DR

Field Documentation

- __IO uint32_t RNG_TypeDef::CR
  - RNG control register, Address offset: 0x00
- __IO uint32_t RNG_TypeDef::SR
  - RNG status register, Address offset: 0x04
- __IO uint32_t RNG_TypeDef::DR
  - RNG data register, Address offset: 0x08

20.2 RNG Firmware driver API description

The following section lists the various functions of the RNG library.

20.2.1 How to use this driver

1. Enable The RNG controller clock using 
   RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_RNG, ENABLE) function.
2. Activate the RNG peripheral using RNG_Cmd() function.
3. Wait until the 32 bit Random number Generator contains a valid random data (using 
   polling/interrupt mode). For more details, refer to Section 20.2.4: “Interrupt and flag 
   management” module description.
5. To get another 32 bit Random number, go to step 3.

20.2.2 Initialization and configuration

This section provides functions allowing to:

- Initialize the RNG peripheral
- Enable or disable the RNG peripheral

Below is list of functions to initialize and configure RNG:
20.2.3 Getting 32-bit Random number

This section provides a function allowing to get the 32 bit Random number.

Before to call this function you have to wait till DRDY flag is set, using RNG_GetFlagStatus(RNG_FLAG_DRDY) function.

RNG_GetRandomNumber()

20.2.4 Interrupt and flag management

This section provides functions allowing to configure the RNG Interrupts and to get the status and clear flags and Interrupts pending bits.

The RNG provides 3 Flags and 3 Interrupts sources:

Flags :
- RNG_FLAG_DRDY : In the case of the RNG_DR register contains valid random data. it is cleared by reading the valid data (using RNG_GetRandomNumber() function).
- RNG_FLAG_CECS : In the case of a seed error detection.
- RNG_FLAG_SECS : In the case of a clock error detection.

Interrupts :
- if enabled, an RNG interrupt is pending :
  - In the case of the RNG_DR register contains valid random data. This interrupt source is cleared once the RNG_DR register has been read (using RNG_GetRandomNumber() function) until a new valid value is computed.
  - In the case of a seed error : One of the following faulty sequences has been detected:
    - More than 64 consecutive bits at the same value (0 or 1)
    - More than 32 consecutive alternance of 0 and 1 (0101010101...01) This interrupt source is cleared using RNG_ClearITPendingBit(RNG_IT_SEI) function.
  - In the case of a clock error : the PLL48CLK (RNG peripheral clock source) was not correctly detected (fPLL48CLK< fHCLK/16). This interrupt source is cleared using RNG_ClearITPendingBit(RNG_IT_CEI) function.

In this case, User have to check that the clock controller is correctly configured to provide the RNG clock.

Managing the RNG controller events

The user should identify which mode will be used in his application to manage the RNG controller events: Polling mode or Interrupt mode.
In the Polling Mode it is advised to use the following functions: RNG_FLAG_DRDY can not be cleared by RNG_ClearFlag(). It is cleared only by reading the Random number data.
- RNG_GetFlagStatus() : to check if flags events occur.
- RNG_ClearFlag() : to clear the flags events.

In the Interrupt Mode it is advised to use the following functions:
- RNG_ITConfig() : to enable or disable the interrupt source.
- RNG_GetITStatus() : to check if Interrupt occurs.
- RNG_ClearITPendingBit() : to clear the Interrupt pending Bit (corresponding Flag).

Below is the list of functions to manage RNG controller event:

- RNG_ITConfig()
- RNG_GetFlagStatus()
- RNG_ClearFlag()
- RNG_GetITStatus()
- RNG_ClearITPendingBit()

### 20.2.5 Initialization and configuration functions

#### 20.2.5.1 RNG_DeInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void RNG_DeInit</td>
<td>Deinitializes the RNG peripheral registers to their default reset values.</td>
</tr>
</tbody>
</table>

**Parameters**

- None.

**Return values**

- None.

**Notes**

- None.

#### 20.2.5.2 RNG_Cmd

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void RNG_Cmd</td>
<td>Enables or disables the RNG peripheral.</td>
</tr>
</tbody>
</table>

**Parameters**

- **NewState**: new state of the RNG peripheral. This parameter can be: ENABLE or DISABLE.

**Return values**

- None.

**Notes**

- None.
20.2.6  Get 32 bit Random number function

20.2.6.1  RNG_GetRandomNumber

Function Name  uint32_t RNG_GetRandomNumber ( void )
Function Description  Returns a 32-bit random number.
Parameters  •  None.
Return values  •  32-bit random number.
Notes  •  Before to call this function you have to wait till DRDY (data ready) flag is set, using RNG_GetFlagStatus(RNG_FLAG_DRDY) function.
•  Each time the the Random number data is read (using RNG_GetRandomNumber() function), the RNG_FLAG_DRDY flag is automatically cleared.
•  In the case of a seed error, the generation of random numbers is interrupted for as long as the SECS bit is ‘1’. If a number is available in the RNG_DR register, it must not be used because it may not have enough entropy. In this case, it is recommended to clear the SEIS bit(using RNG_ClearFlag(RNG_FLAG_SECS) function), then disable and enable the RNG peripheral (using RNG_Cmd() function) to reinitialize and restart the RNG.
•  In the case of a clock error, the RNG is no more able to generate random numbers because the PLL48CLK clock is not correct. User have to check that the clock controller is correctly configured to provide the RNG clock and clear the CEIS bit (using RNG_ClearFlag(RNG_FLAG_CECS) function) . The clock error has no impact on the previously generated random numbers, and the RNG_DR register contents can be used.

20.2.7  Interrupt and flag management functions

20.2.7.1  RNG_ITConfig

Function Name  void RNG_ITConfig ( FunctionalState NewState)
Function Description  Enables or disables the RNG interrupt.
Parameters  •  NewState : new state of the RNG interrupt. This parameter can be: ENABLE or DISABLE.
Return values
- None.

Notes
- The RNG provides 3 interrupt sources, Computed data is ready event (DRDY), and Seed error Interrupt (SEI) and Clock error Interrupt (CEI), all these interrupts sources are enabled by setting the IE bit in CR register. However, each interrupt have its specific status bit (see RNG_GetITStatus() function) and clear bit except the DRDY event (see RNG_ClearITPendingBit() function).

### 20.2.7.2 RNG_GetFlagStatus

<table>
<thead>
<tr>
<th>Function Name</th>
<th>FlagStatus RNG_GetFlagStatus ( uint8_t RNG_FLAG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Checks whether the specified RNG flag is set or not.</td>
</tr>
</tbody>
</table>
| Parameters | - **RNG_FLAG** : specifies the RNG flag to check. This parameter can be one of the following values:  
  - **RNG_FLAG_DRDY** : Data Ready flag.  
  - **RNG_FLAG_CECS** : Clock Error Current flag.  
  - **RNG_FLAG_SECS** : Seed Error Current flag. |
| Return values | - The new state of RNG_FLAG (SET or RESET). |
| Notes | - None. |

### 20.2.7.3 RNG_ClearFlag

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void RNG_ClearFlag ( uint8_t RNG_FLAG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Clears the RNG flags.</td>
</tr>
</tbody>
</table>
| Parameters | - **RNG_FLAG** : specifies the flag to clear. This parameter can be any combination of the following values:  
  - **RNG_FLAG_CECS** : Clock Error Current flag.  
  - **RNG_FLAG_SECS** : Seed Error Current flag. |
| Return values | - None. |
| Notes | - **RNG_FLAG_DRDY** can not be cleared by RNG_ClearFlag() function. This flag is cleared only by reading the Random number data (using RNG_GetRandomNumber() function). |
20.2.7.4 RNG_GetITStatus

**Function Name**

ITStatus RNG_GetITStatus ( uint8_t RNG_IT)

**Function Description**

Checks whether the specified RNG interrupt has occurred or not.

**Parameters**

- **RNG_IT**: specifies the RNG interrupt source to check. This parameter can be one of the following values:
  - **RNG_IT_CEI**: Clock Error Interrupt.
  - **RNG_IT_SEI**: Seed Error Interrupt.

**Return values**

- The new state of RNG_IT (SET or RESET).

**Notes**

- None.

20.2.7.5 RNG_ClearITPendingBit

**Function Name**

void RNG_ClearITPendingBit ( uint8_t RNG_IT)

**Function Description**

Clears the RNG interrupt pending bit(s).

**Parameters**

- **RNG_IT**: specifies the RNG interrupt pending bit(s) to clear. This parameter can be any combination of the following values:
  - **RNG_IT_CEI**: Clock Error Interrupt.
  - **RNG_IT_SEI**: Seed Error Interrupt.

**Return values**

- None.

**Notes**

- None.

20.3 RNG Firmware driver defines

20.3.1 RNG Firmware driver defines

RNG

*RNG_flags_definition*

- #define: `RNG_FLAG_DRDY((uint8_t)0x0001)`

  Data ready
Random number generator (RNG)

- #define: `RNG_FLAG_CECS((uint8_t)0x0002)`
  Clock error current status

- #define: `RNG_FLAG_SECS((uint8_t)0x0004)`
  Seed error current status

**RNG_interrupts_definition**

- #define: `RNG_IT_CEI((uint8_t)0x20)`
  Clock error interrupt

- #define: `RNG_IT_SEI((uint8_t)0x40)`
  Seed error interrupt

## 20.4 RNG Programming Example

The example below explains how to generate random number using the RNG processor. For more examples about RNG configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx\StdPeriph_Examples\RNG\.

```c
__IO uint32_t random32bit = 0;

/* Enable RNG clock source */
RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_RNG, ENABLE);

/* RNG Peripheral enable */
RNG_Cmd(ENABLE);

/* Wait until one random number is ready */
while(RNG_GetFlagStatus(RNG_FLAG_DRDY)== RESET) {
}

/* Get the random number */
random32bit = RNG_GetRandomNumber();
```
21 Real-time clock (RTC)

21.1 RTC Firmware driver registers structures

21.1.1 RTC_TypeDef

`RTC_TypeDef` is defined in the `stm32f2xx.h` file and contains the RTC registers definition.

**Data Fields**

- `__IO uint32_t TR`
- `__IO uint32_t DR`
- `__IO uint32_t CR`
- `__IO uint32_t ISR`
- `__IO uint32_t PRER`
- `__IO uint32_t WUTR`
- `__IO uint32_t CALIBR`
- `__IO uint32_t ALRMAR`
- `__IO uint32_t ALRMBR`
- `__IO uint32_t WPR`
- `uint32_t RESERVED1`
- `uint32_t RESERVED2`
- `__IO uint32_t TSTR`
- `__IO uint32_t TSDR`
- `uint32_t RESERVED3`
- `uint32_t RESERVED4`
- `__IO uint32_t TAFCR`
- `uint32_t RESERVED5`
- `uint32_t RESERVED6`
- `uint32_t RESERVED7`
- `__IO uint32_t BKP0R`
- `__IO uint32_t BKP1R`
- `__IO uint32_t BKP2R`
- `__IO uint32_t BKP3R`
- `__IO uint32_t BKP4R`
- `__IO uint32_t BKP5R`
- `__IO uint32_t BKP6R`
- `__IO uint32_t BKP7R`
- `__IO uint32_t BKP8R`
- `__IO uint32_t BKP9R`
- `__IO uint32_t BKP10R`
- `__IO uint32_t BKP11R`
- `__IO uint32_t BKP12R`
- `__IO uint32_t BKP13R`
- `__IO uint32_t BKP14R`
- `__IO uint32_t BKP15R`
- `__IO uint32_t BKP16R`
- `__IO uint32_t BKP17R`
- `__IO uint32_t BKP18R`
- `__IO uint32_t BKP19R`
Field Documentation

- __IO uint32_t RTC_TypeDef::TR
  - RTC time register, Address offset: 0x00
- __IO uint32_t RTC_TypeDef::DR
  - RTC date register, Address offset: 0x04
- __IO uint32_t RTC_TypeDef::CR
  - RTC control register, Address offset: 0x08
- __IO uint32_t RTC_TypeDef::ISR
  - RTC initialization and status register, Address offset: 0x0C
- __IO uint32_t RTC_TypeDef::PRER
  - RTC prescaler register, Address offset: 0x10
- __IO uint32_t RTC_TypeDef::WUTR
  - RTC wakeup timer register, Address offset: 0x14
- __IO uint32_t RTC_TypeDef::CALIBR
  - RTC calibration register, Address offset: 0x18
- __IO uint32_t RTC_TypeDef::ALRMAR
  - RTC alarm A register, Address offset: 0x1C
- __IO uint32_t RTC_TypeDef::ALRMBR
  - RTC alarm B register, Address offset: 0x20
- __IO uint32_t RTC_TypeDef::WPR
  - RTC write protection register, Address offset: 0x24
- __IO uint32_t RTC_TypeDef::RESERVED1
  - Reserved, 0x28
- __IO uint32_t RTC_TypeDef::RESERVED2
  - Reserved, 0x2C
- __IO uint32_t RTC_TypeDef::TSTR
  - RTC time stamp time register, Address offset: 0x30
- __IO uint32_t RTC_TypeDef::TSDR
  - RTC time stamp date register, Address offset: 0x34
- __IO uint32_t RTC_TypeDef::RESERVED3
  - Reserved, 0x38
- __IO uint32_t RTC_TypeDef::RESERVED4
  - Reserved, 0x3C
- __IO uint32_t RTC_TypeDef::TAFCR
  - RTC tamper and alternate function configuration register, Address offset: 0x40
- __IO uint32_t RTC_TypeDef::RESERVED5
  - Reserved, 0x44
- __IO uint32_t RTC_TypeDef::RESERVED6
  - Reserved, 0x48
- __IO uint32_t RTC_TypeDef::RESERVED7
  - Reserved, 0x4C
- __IO uint32_t RTC_TypeDef::BKP0R
  - RTC backup register 1, Address offset: 0x50
- __IO uint32_t RTC_TypeDef::BKP1R
  - RTC backup register 1, Address offset: 0x54
- __IO uint32_t RTC_TypeDef::BKP2R
  - RTC backup register 2, Address offset: 0x58
- __IO uint32_t RTC_TypeDef::BKP3R
  - RTC backup register 3, Address offset: 0x5C
• __IO uint32_t RTC_TypeDef::BKP4R
  – RTC backup register 4, Address offset: 0x60
• __IO uint32_t RTC_TypeDef::BKP5R
  – RTC backup register 5, Address offset: 0x64
• __IO uint32_t RTC_TypeDef::BKP6R
  – RTC backup register 6, Address offset: 0x68
• __IO uint32_t RTC_TypeDef::BKP7R
  – RTC backup register 7, Address offset: 0x6C
• __IO uint32_t RTC_TypeDef::BKP8R
  – RTC backup register 8, Address offset: 0x70
• __IO uint32_t RTC_TypeDef::BKP9R
  – RTC backup register 9, Address offset: 0x74
• __IO uint32_t RTC_TypeDef::BKP10R
  – RTC backup register 10, Address offset: 0x78
• __IO uint32_t RTC_TypeDef::BKP11R
  – RTC backup register 11, Address offset: 0x7C
• __IO uint32_t RTC_TypeDef::BKP12R
  – RTC backup register 12, Address offset: 0x80
• __IO uint32_t RTC_TypeDef::BKP13R
  – RTC backup register 13, Address offset: 0x84
• __IO uint32_t RTC_TypeDef::BKP14R
  – RTC backup register 14, Address offset: 0x88
• __IO uint32_t RTC_TypeDef::BKP15R
  – RTC backup register 15, Address offset: 0x8C
• __IO uint32_t RTC_TypeDef::BKP16R
  – RTC backup register 16, Address offset: 0x90
• __IO uint32_t RTC_TypeDef::BKP17R
  – RTC backup register 17, Address offset: 0x94
• __IO uint32_t RTC_TypeDef::BKP18R
  – RTC backup register 18, Address offset: 0x98
• __IO uint32_t RTC_TypeDef::BKP19R
  – RTC backup register 19, Address offset: 0x9C

21.1.2 RTC_InitTypeDef

RTC_InitTypeDef is defined in the stm32f2xx_rtc.h file and contains the RTC common initialization parameters.

Data Fields

• uint32_t RTC_HourFormat
• uint32_t RTC_AsynchPrediv
• uint32_t RTC_SynchPrediv

Field Documentation

• uint32_t RTC_InitTypeDef::RTC_HourFormat
  – Specifies the RTC Hour Format. This parameter can be a value of RTC_Hour_Formats
• uint32_t RTC_InitTypeDef::RTC_AsynchPrediv
21.1.3 RTC_TimeTypeDef

**RTC_TimeTypeDef** is defined in the stm32f2xx_rtc.h file and contains the time configuration parameters.

*Data Fields*

- `uint8_t RTC_Hours`
- `uint8_t RTC_Minutes`
- `uint8_t RTC_Seconds`
- `uint8_t RTC_H12`

*Field Documentation*

- `uint8_t RTC_TimeTypeDef::RTC_Hours`
  - Specifies the RTC Time Hour. This parameter must be set to a value in the 0-12 range if the RTC_HourFormat_12 is selected or 0-23 range if the RTC_HourFormat_24 is selected.
- `uint8_t RTC_TimeTypeDef::RTC_Minutes`
  - Specifies the RTC Time Minutes. This parameter must be set to a value in the 0-59 range.
- `uint8_t RTC_TimeTypeDef::RTC_Seconds`
  - Specifies the RTC Time Seconds. This parameter must be set to a value in the 0-59 range.
- `uint8_t RTC_TimeTypeDef::RTC_H12`
  - Specifies the RTC AM/PM Time. This parameter can be a value of **RTC_AM_PM_Definitions**

21.1.4 RTC_DateTypeDef

**RTC_DateTypeDef** is defined in the stm32f2xx_rtc.h file and contains the date configuration parameters.

*Data Fields*

- `uint32_t RTC_WeekDay`
- `uint32_t RTC_Month`
- `uint8_t RTC_Date`
- `uint8_t RTC_Year`

*Field Documentation*
- `uint32_t RTC_DateTypeDef::RTC_WeekDay`: Specifies the RTC Date WeekDay. This parameter can be a value of `RTC_WeekDay_Definitions`.
- `uint32_t RTC_DateTypeDef::RTC_Month`: Specifies the RTC Date Month. This parameter can be a value of `RTC_Month_Date_Definitions`.
- `uint8_t RTC_DateTypeDef::RTC_Date`: Specifies the RTC Date. This parameter must be set to a value in the 1-31 range.
- `uint8_t RTC_DateTypeDef::RTC_Year`: Specifies the RTC Date Year. This parameter must be set to a value in the 0-99 range.

### 21.1.5 RTC_AlarmTypeDef

`RTC_AlarmTypeDef` is defined in the `stm32f2xx_rtc.h` file and contains the alarm configuration parameters.

#### Data Fields

- `RTC_TimeTypeDef RTC_AlarmTypeDef::RTC_AlarmTime`
- `uint32_t RTC_AlarmTypeDef::RTC_AlarmMask`
- `uint32_t RTC_AlarmTypeDef::RTC_AlarmDateWeekDaySel`
- `uint8_t RTC_AlarmTypeDef::RTC_AlarmDateWeekDay`

#### Field Documentation

- `RTC_TimeTypeDef RTC_AlarmTypeDef::RTC_AlarmTime`: Specifies the RTC Alarm Time members.
- `uint32_t RTC_AlarmTypeDef::RTC_AlarmMask`: Specifies the RTC Alarm Masks. This parameter can be a value of `RTC_AlarmMask_Definitions`.
- `uint32_t RTC_AlarmTypeDef::RTC_AlarmDateWeekDaySel`: Specifies the RTC Alarm is on Date or WeekDay. This parameter can be a value of `RTC_AlarmDateWeekDay_Definitions`.
- `uint8_t RTC_AlarmTypeDef::RTC_AlarmDateWeekDay`: Specifies the RTC Alarm Date/WeekDay. If the Alarm Date is selected, this parameter must be set to a value in the 1-31 range. If the Alarm WeekDay is selected, this parameter can be a value of `RTC_WeekDay_Definitions`.

### 21.2 RTC Firmware driver API description

The following section lists the various functions of the RTC library.

#### 21.2.1 Backup Domain operating conditions
The real-time clock (RTC), the RTC backup registers, and the backup SRAM (BKP SRAM) can be powered from the VBAT voltage when the main VDD supply is powered off.

To retain the content of the RTC backup registers, backup SRAM, and supply the RTC when VDD is turned off, VBAT pin can be connected to an optional standby voltage supplied by a battery or by another source.

To allow the RTC to operate even when the main digital supply (VDD) is turned off, the VBAT pin powers the following blocks:

- The RTC
- The LSE oscillator
- The backup SRAM when the low power backup regulator is enabled
- PC13 to PC15 I/Os, plus PI8 I/O (when available)

When the backup domain is supplied by VDD (analog switch connected to VDD), the following functions are available:

- PC14 and PC15 can be used as either GPIO or LSE pins
- PC13 can be used as a GPIO or as the RTC_AF1 pin
- PI8 can be used as a GPIO or as the RTC_AF2 pin

When the backup domain is supplied by VBAT (analog switch connected to VBAT because VDD is not present), the following functions are available:

- PC14 and PC15 can be used as LSE pins only
- PC13 can be used as the RTC_AF1 pin
- PI8 can be used as the RTC_AF2 pin

**Backup domain reset**

The backup domain reset sets all RTC registers and the RCC_BDCR register to their reset values.

The BKPSRAM is not affected by this reset. The only way of resetting the BKPSRAM is through the Flash interface by requesting a protection level change from 1 to 0. A backup domain reset is generated when one of the following events occurs:

- Software reset, triggered by setting the BDRST bit in the RCC Backup domain control register (RCC_BDCR). You can use the RCC_BackupResetCmd().
- VDD or VBAT power on, if both supplies have previously been powered off.

**Backup Domain Access**

After reset, the backup domain (RTC registers, RTC backup data registers and backup SRAM) is protected against possible unwanted write accesses. To enable access to the RTC Domain and RTC registers, proceed as follows:

1. Enable the Power Controller (PWR) APB1 interface clock using the RCC_APB1PeriphClockCmd() function.
2. Enable access to RTC domain using the PWR_BackupAccessCmd() function.
3. Select the RTC clock source using the RCC_RTCCLKConfig() function
4. Enable RTC Clock using the RCC_RTCCLKCmd() function.

**21.2.2 How to use the RTC driver**

The following steps are required before using the RTC:

1. Enable the RTC domain access (see description in the section above)
2. Configure the RTC Prescaler (Asynchronous and Synchronous) and RTC hour format using the RTC_Init() function.

### 21.2.3 RTC configuration

#### Time and Date configuration
1. To configure the RTC Calendar (Time and Date) use the RTC_SetTime() and RTC_SetDate() functions.
2. To read the RTC Calendar, use the RTC_GetTime() and RTC_GetDate() functions.
3. Use the RTC_DayLightSavingConfig() function to add or sub one hour to the RTC Calendar.

#### Alarm configuration
1. To configure the RTC Alarm use the RTC_SetAlarm() function.
2. Enable the selected RTC Alarm using the RTC_AlarmCmd() function.
3. To read the RTC Alarm, use the RTC_GetAlarm() function.

#### RTC Wakeup configuration
1. Configure the RTC Wakeup Clock source using the RTC_WakeUpClockConfig() function.
2. Configure the RTC WakeUp Counter using the RTC_SetWakeUpCounter() function.
3. Enable the RTC WakeUp using the RTC_WakeUpCmd() function.
4. To read the RTC WakeUp Counter register, use the RTC_GetWakeUpCounter() function.

#### Outputs configuration
The RTC has 2 different outputs:
- **AFO_ALARM**: this output is used to manage the RTC Alarm A, Alarm B and WakeUp signals. To output the selected RTC signal on RTC_AF1 pin, use the RTC_OutputConfig() function.
- **AFO_CALIB**: this output is used to manage the RTC Clock divided by 64 (512Hz) signal. To output the RTC Clock on RTC_AF1 pin, use the RTC_CalibOutputCmd() function.

#### Coarse Calibration configuration
1. Configure the RTC Coarse Calibration Value and the corresponding sign using the RTC_CoarseCalibConfig() function.
2. Enable the RTC Coarse Calibration using the RTC_CoarseCalibCmd() function.

#### TimeStamp configuration
- Configure the RTC_AF1 trigger and enables the RTC TimeStamp using the RTC_TimeStampCmd() function.
- To read the RTC TimeStamp Time and Date register, use the RTC_GetTimeStamp() function.

The TAMPER1 alternate function can be mapped either to RTC_AF1(PC13) or RTC_AF2 (PI8) depending on the value of TAMP1INSEL bit in RTC_TAFCR register. You can use the RTC_TamperPinSelection() function to select the corresponding pin.

#### Tamper configuration
1. Configure the RTC Tamper trigger using the RTC_TamperConfig() function.
2. Enable the RTC Tamper using the RTC_TamperCmd() function.
The TIMESTAMP alternate function can be mapped to either RTC_AF1 or RTC_AF2 depending on the value of the TSINSEL bit in the RTC_TAFCR register. You can use the RTC_TimeStampPinSelection() function to select the corresponding pin.

**Backup Data Registers configuration**

- To write to the RTC Backup Data registers, use the RTC_WriteBackupRegister() function.
- To read the RTC Backup Data registers, use the RTC_ReadBackupRegister() function.

**Selection of RTC_AF1 alternate functions**

The RTC_AF1 pin (PC13) can be used for the following purposes:

- AFO_ALARM output
- AFO_CALIB output
- AFI_TAMPER - AFI_TIMESTAMP

<table>
<thead>
<tr>
<th>Pin configuration and function</th>
<th>AFO_ALARM enable</th>
<th>AFO_CALIB enabled</th>
<th>Tamper enabled</th>
<th>Timestamp enabled</th>
<th>TAMP1INSEL_TIMESTAMP</th>
<th>TIME_STAMP TAMPER</th>
<th>ALARMOUTYPE AFO_ALARM configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alarm output OD</td>
<td>1</td>
<td>Don't care</td>
<td>Don't care</td>
<td>Don't care</td>
<td>Don't care</td>
<td>Don't care</td>
<td>0</td>
</tr>
<tr>
<td>Alarm output PP</td>
<td>1</td>
<td>Don't care</td>
<td>Don't care</td>
<td>Don't care</td>
<td>Don't care</td>
<td>Don't care</td>
<td>1</td>
</tr>
<tr>
<td>Calibration out PP</td>
<td>0</td>
<td>1</td>
<td>Don't care</td>
<td>Don't care</td>
<td>Don't care</td>
<td>Don't care</td>
<td>Don't care</td>
</tr>
<tr>
<td>TAMPER input floating</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Don't care</td>
<td>Don't care</td>
</tr>
<tr>
<td>TIMESTAMP and TAMPER1 input floating</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Don't care</td>
</tr>
<tr>
<td>TIMESTAMP input floating</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Don't care</td>
<td>0</td>
<td>Don't care</td>
</tr>
<tr>
<td>Standard GPIO</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Don't care</td>
<td>Don't care</td>
<td>Don't care</td>
</tr>
</tbody>
</table>

**Selection of RTC_AF2 alternate functions**

The RTC_AF2 pin (PI8) can be used for the following purposes:

- AFI_TAMPER
- AFI_TIMESTAMP
### RTC Initialization and Configuration functions

This section provides functions allowing to initialize and configure the RTC Prescaler (Synchronous and Asynchronous), RTC Hour format, disable RTC registers Write protection, enter and exit the RTC initialization mode, RTC registers synchronization check and reference clock detection enable.

1. The RTC Prescaler is programmed to generate the RTC 1 Hz time base. It is split into 2 programmable prescalers to minimize power consumption. When both prescalers are used, it is recommended to configure the asynchronous prescaler to a high value to minimize consumption.
   - A 7-bit asynchronous prescaler
   - A 13-bit synchronous prescaler.

2. All RTC registers are Write protected. Writing to the RTC registers is enabled by writing a key into the Write Protection register, RTC_WPR.

3. To Configure the RTC Calendar, user application should enter initialization mode. In this mode, the calendar counter is stopped and its value can be updated. When the initialization sequence is complete, the calendar restarts counting after 4 RTCCLK cycles.

4. To read the calendar through the shadow registers after Calendar initialization, calendar update or after wakeup from low power modes the software must first clear the RSF flag. The software must then wait until it is set again before reading the calendar, which means that the calendar registers have been correctly copied into the RTC_TR and RTC_DR shadow registers. The RTC_WaitForSyncro() function implements the above software sequence (RSF clear and RSF check).

The following functions can be used to initialize and configure the RTC:

- `RTC_DeInit()`
- `RTC_Init()`
- `RTC_StructInit()`
- `RTC_WriteProtectionCmd()`
- `RTC_EnterInitMode()`
- `RTC_ExitInitMode()`
- `RTC_WaitForSyncro()`
- `RTC_RefClockCmd()`

### 21.2.4 Backup Data registers configuration
21.2.5 RTC and low power modes

The MCU can be woken up from a low power mode by an RTC alternate function. The RTC alternate functions are the RTC alarms (Alarm A and Alarm B), RTC wakeup, RTC tamper event detection and RTC time stamp event detection. These RTC alternate functions can wake up the system from the Stop and Standby lowpower modes. The system can also wake up from low power modes without depending on an external interrupt (Auto-wakeup mode), by using the RTC alarm or the RTC wakeup events. The RTC provides a programmable time base for waking up from the Stop or Standby mode at regular intervals. Wakeup from STOP and Standby modes is possible only when the RTC clock source is LSE or LSI.

21.2.6 RTC Tamper and TimeStamp pin selection and Output Type Config configuration

21.2.7 Interrupt and flag management

All RTC interrupts are connected to the EXTI controller.

To enable the RTC Alarm interrupt, the following sequence is required:
1. Configure and enable the EXTI Line 17 in interrupt mode and select the rising edge sensitivity using the EXTI_Init() function.
2. Configure and enable the RTC_Alarm IRQ channel in the NVIC using the NVIC_Init() function.
3. Configure the RTC to generate RTC alarms (Alarm A and/or Alarm B) using the RTC_SetAlarm() and RTC_AlarmCmd() functions.

To enable the RTC Wakeup interrupt, the following sequence is required:
1. Configure and enable the EXTI Line 22 in interrupt mode and select the rising edge sensitivity using the EXTI_Init() function.
2. Configure and enable the RTC_WKUP IRQ channel in the NVIC using the NVIC_Init() function.
3. Configure the RTC to generate the RTC wakeup timer event using the RTC_WakeUpClockConfig(), RTC_SetWakeUpCounter() and RTC_WakeUpCmd() functions.

To enable the RTC Tamper interrupt, the following sequence is required:
1. Configure and enable the EXTI Line 21 in interrupt mode and select the rising edge sensitivity using the EXTI_Init() function.
2. Configure and enable the TAMP_STAMP IRQ channel in the NVIC using the NVIC_Init() function.
3. Configure the RTC to detect the RTC tamper event using the RTC_TamperTriggerConfig() and RTC_TamperCmd() functions.

To enable the RTC TimeStamp interrupt, the following sequence is required:
1. Configure and enable the EXTI Line 21 in interrupt mode and select the rising edge sensitivity using the EXTI_Init() function.
2. Configure and enable the TAMPS_AMP IRQ channel in the NVIC using the NVIC_Init() function.
3. Configure the RTC to detect the RTC time-stamp event using the RTC_TimeStampCmd() functions.

The following functions that can be used to configure the RTC interrupts and flags:

- RTC_ITConfig()
- RTC_GetFlagStatus()
- RTC_ClearFlag()
- RTC_GetITStatus()
- RTC_ClearITPendingBit()

Time and Date configuration functions

This section provide functions allowing to program and read the RTC Calendar (Time and Date).

- RTC_SetTime()
- RTC_TimeStructInit()
- RTC_GetTime()
- RTC_SetDate()
- RTC_DateStructInit()
- RTC_GetDate()

Alarms (Alarm A and Alarm B) configuration functions

This section provide functions allowing to program and read the RTC Alarms.

- RTC_SetAlarm()
- RTC_AlarmStructInit()
- RTC_GetAlarm()
- RTC_AlarmCmd()

WakeUp Timer configuration functions

This section provide functions allowing to program and read the RTC WakeUp.

- RTC_WakeUpClockConfig()
- RTC_SetWakeUpCounter()
- RTC_GetWakeUpCounter()
- RTC_WakeUpCmd()

Daylight Saving configuration functions

This section provide functions allowing to configure the RTC DayLight Saving.

- RTC_DaylightSavingConfig()
- RTC_GetStoreOperation()

Output pin Configuration function

This section provide functions allowing to configure the RTC Output source.

- RTC_OutputConfig()
Coarse Calibration configuration functions

- RTC_CoarseCalibConfig()
- RTC_CoarseCalibCmd()
- RTC_CalibOutputCmd()

TimeStamp configuration functions

- RTC_TimeStampCmd()
- RTC_GetTimeStamp()

Tampers configuration functions

- RTC_TamperTriggerConfig()
- RTC_TamperCmd()

21.2.8 Initialization and configuration functions

21.2.8.1 RTC_DeInit

Function Name: ErrorStatus RTC_DeInit ( void )

Function Description: Deinitializes the RTC registers to their default reset values.

Parameters:
- None.

Return values:
- An ErrorStatus enumeration value:
  - SUCCESS: RTC registers are deinitialized
  - ERROR: RTC registers are not deinitialized

Notes:
- This function doesn't reset the RTC Clock source and RTC Backup Data registers.

21.2.8.2 RTC_Init

Function Name: ErrorStatus RTC_Init ( RTC_InitTypeDef * RTC_InitStruct)

Function Description: Initializes the RTC registers according to the specified parameters in RTC_InitStruct.

Parameters:
- RTC_InitStruct: pointer to a RTC_InitTypeDef structure that contains the configuration information for the RTC peripheral.

Return values:
- An ErrorStatus enumeration value:
  - SUCCESS: RTC registers are initialized
  - ERROR: RTC registers are not initialized

Notes:
- The RTC Prescaler register is write protected and can be written in initialization mode only.
# RTC_StructInit

**Function Name**

void RTC_StructInit (RTC_InitTypeDef * RTC_InitStruct)

**Function Description**

Fills each RTC_InitStruct member with its default value.

**Parameters**

- `RTC_InitStruct` : pointer to a RTC_InitTypeDef structure which will be initialized.

**Return values**

- None.

**Notes**

- None.

# RTC_WriteProtectionCmd

**Function Name**

void RTC_WriteProtectionCmd (FunctionalState NewState)

**Function Description**

Enables or disables the RTC registers write protection.

**Parameters**

- `NewState` : new state of the write protection. This parameter can be: ENABLE or DISABLE.

**Return values**

- None.

**Notes**

- All the RTC registers are write protected except for RTC_ISR[13:8], RTC_TAFCR and RTC_BKPxR.
- Writing a wrong key reactivates the write protection.
- The protection mechanism is not affected by system reset.

# RTC_EnterInitMode

**Function Name**

ErrorStatus RTC_EnterInitMode (void)

**Function Description**

Enters the RTC Initialization mode.

**Parameters**

- None.

**Return values**

- An ErrorStatus enumeration value:
21.2.8.6  RTC_ExitInitMode

Function Name: void RTC_ExitInitMode ( void )
Function Description: Exits the RTC Initialization mode.
Parameters: None.
Return values: None.
Notes: When the initialization sequence is complete, the calendar restarts counting after 4 RTCCLK cycles.
• The RTC Initialization mode is write protected, use the RTC_WriteProtectionCmd(DISABLE) before calling this function.

21.2.8.7  RTC_WaitForSynchro

Function Name: ErrorStatus RTC_WaitForSynchro ( void )
Function Description: Waits until the RTC Time and Date registers (RTC_TR and RTC_DR) are synchronized with RTC APB clock.
Parameters: None.
Return values: An ErrorStatus enumeration value:
  - SUCCESS: RTC registers are synchronised
  - ERROR: RTC registers are not synchronised
Notes: The RTC Resynchronization mode is write protected, use the RTC_WriteProtectionCmd(DISABLE) before calling this function.
• To read the calendar through the shadow registers after Calendar initialization, calendar update or after wakeup from low power modes the software must first clear the RSF flag. The software must then wait until it is set again before reading the calendar, which means that the calendar registers have been correctly copied into the RTC_TR and RTC_DR shadow
21.2.8.8 RTC_RefClockCmd

Function Name: ErrorStatus RTC_RefClockCmd ( FunctionalState NewState)
Function Description: Enables or disables the RTC reference clock detection.
Parameters:
- **NewState**: new state of the RTC reference clock. This parameter can be: ENABLE or DISABLE.

Return values:
- An ErrorStatus enumeration value:
  - **SUCCESS**: RTC reference clock detection is enabled
  - **ERROR**: RTC reference clock detection is disabled

Notes:
- None.

21.2.9 Backup Data registers configuration functions

21.2.9.1 RTC_WriteBackupRegister

Function Name: void RTC_WriteBackupRegister ( uint32_t RTC_BKP_DR, uint32_t Data)
Function Description: Writes a data in a specified RTC Backup data register.
Parameters:
- **RTC_BKP_DR**: RTC Backup data Register number. This parameter can be: RTC_BKP_DRx where x can be from 0 to 19 to specify the register.
- **Data**: Data to be written in the specified RTC Backup data register.

Return values:
- None.

Notes:
- None.
Function Name | uint32_t RTC_ReadBackupRegister ( uint32_t RTC_BKP_DR)  
Function Description | Reads data from the specified RTC Backup data Register.  
Parameters |  
RTC_BKP_DR : RTC Backup data Register number. This parameter can be: RTC_BKP_DRx where x can be from 0 to 19 to specify the register.

Return values | None.

Notes | None.

21.2.10  RTC Tamper and TimeStamp pin selection and Output Type Config configuration functions

21.2.10.1  RTC_TamperPinSelection

Function Name | void RTC_TamperPinSelection ( uint32_t RTC_TamperPin)  
Function Description | Selects the RTC Tamper Pin.  
Parameters |  
RTC_TamperPin : specifies the RTC Tamper Pin. This parameter can be one of the following values:  
- RTC_TamperPin_PC13 : PC13 is selected as RTC Tamper Pin.  
- RTC_TamperPin_PI8 : PI8 is selected as RTC Tamper Pin.

Return values | None.

Notes | None.

21.2.10.2  RTC_TimeStampPinSelection

Function Name | void RTC_TimeStampPinSelection ( uint32_t RTC_TimeStampPin)  
Function Description | Selects the RTC TimeStamp Pin.  
Parameters |  
RTC_TimeStampPin : specifies the RTC TimeStamp Pin. This parameter can be one of the following values:  
- RTC_TimeStampPin_PC13 : PC13 is selected as RTC TimeStamp Pin.  
- RTC_TimeStampPin_PI8 : PI8 is selected as RTC TimeStamp Pin.


21.2.10.3 RTC_OutputTypeConfig

Function Name: void RTC_OutputTypeConfig ( uint32_t RTC_OutputType)
Function Description: Configures the RTC Output Pin mode.
Parameters:
- **RTC_OutputType**: specifies the RTC Output (PC13) pin mode. This parameter can be one of the following values:
  - **RTC_OutputType_OpenDrain**: RTC Output (PC13) is configured in Open Drain mode.
  - **RTC_OutputType_PushPull**: RTC Output (PC13) is configured in Push Pull mode.

Return values: None.
Notes: None.

21.2.11 Interrupt and flag management functions

21.2.11.1 RTC_ITConfig

Function Name: void RTC_ITConfig ( uint32_t RTC_IT, FunctionalState NewState)
Function Description: Enables or disables the specified RTC interrupts.
Parameters:
- **RTC_IT**: specifies the RTC interrupt sources to be enabled or disabled. This parameter can be any combination of the following values:
  - **RTC_IT_TS**: Time Stamp interrupt mask
  - **RTC_IT_WUT**: WakeUp Timer interrupt mask
  - **RTC_IT_ALRB**: Alarm B interrupt mask
  - **RTC_IT_ALRA**: Alarm A interrupt mask
  - **RTC_IT_TAMP**: Tamper event interrupt mask
- **NewState**: new state of the specified RTC interrupts. This parameter can be: ENABLE or DISABLE.

Return values: None.
Notes: None.
## 21.2.11.2 RTC_GetFlagStatus

<table>
<thead>
<tr>
<th>Function Name</th>
<th>FlagStatus RTC_GetFlagStatus ( uint32_t RTC_FLAG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Checks whether the specified RTC flag is set or not.</td>
</tr>
<tr>
<td>Parameters</td>
<td><strong>RTC_FLAG</strong> : specifies the flag to check. This parameter can be one of the following values:</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_TAMP1F</strong> : Tamper 1 event flag</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_TSOVF</strong> : Time Stamp Overflow flag</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_TSF</strong> : Time Stamp event flag</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_WUTF</strong> : WakeUp Timer flag</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_ALRBWF</strong> : Alarm B flag</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_ALRAWF</strong> : Alarm A Write flag</td>
</tr>
<tr>
<td>Return values</td>
<td><strong>The new state of RTC_FLAG (SET or RESET).</strong></td>
</tr>
<tr>
<td>Notes</td>
<td><strong>None.</strong></td>
</tr>
</tbody>
</table>

## 21.2.11.3 RTC_ClearFlag

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void RTC_ClearFlag ( uint32_t RTC_FLAG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Clears the RTC's pending flags.</td>
</tr>
<tr>
<td>Parameters</td>
<td><strong>RTC_FLAG</strong> : specifies the RTC flag to clear. This parameter can be any combination of the following values:</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_TAMP1F</strong> : Tamper 1 event flag</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_TSOVF</strong> : Time Stamp Overflow flag</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_TSF</strong> : Time Stamp event flag</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_WUTF</strong> : WakeUp Timer flag</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_ALRBWF</strong> : Alarm B flag</td>
</tr>
<tr>
<td></td>
<td>- <strong>RTC_FLAG_ALRAWF</strong> : Alarm A Write flag</td>
</tr>
<tr>
<td>Return values</td>
<td><strong>None.</strong></td>
</tr>
</tbody>
</table>


### 21.2.11.4 RTC_GetITStatus

**Function Name**

ITStatus RTC_GetITStatus ( uint32_t RTC_IT)

**Function Description**

Checks whether the specified RTC interrupt has occurred or not.

**Parameters**

- **RTC_IT**: specifies the RTC interrupt source to check. This parameter can be one of the following values:
  - **RTC_IT_TS**: Time Stamp interrupt
  - **RTC_IT_WUT**: WakeUp Timer interrupt
  - **RTC_IT_ALRB**: Alarm B interrupt
  - **RTC_IT_ALRA**: Alarm A interrupt
  - **RTC_IT_TAMP1**: Tamper 1 event interrupt

**Return values**

- The new state of RTC_IT (SET or RESET).

**Notes**

- None.

### 21.2.11.5 RTC_ClearITPendingBit

**Function Name**

void RTC_ClearITPendingBit ( uint32_t RTC_IT)

**Function Description**

Clears the RTC's interrupt pending bits.

**Parameters**

- **RTC_IT**: specifies the RTC interrupt pending bit to clear. This parameter can be any combination of the following values:
  - **RTC_IT_TS**: Time Stamp interrupt
  - **RTC_IT_WUT**: WakeUp Timer interrupt
  - **RTC_IT_ALRB**: Alarm B interrupt
  - **RTC_IT_ALRA**: Alarm A interrupt
  - **RTC_IT_TAMP1**: Tamper 1 event interrupt

**Return values**

- None.

**Notes**

- None.
21.2.12 Time and Date configuration functions

21.2.12.1 RTC_SetTime

Function Name: ErrorStatus RTC_SetTime (uint32_t RTC_Format, RTC_TimeTypeDef *RTC_TimeStruct)

Function Description: Set the RTC current time.

Parameters:
- RTC_Format: specifies the format of the entered parameters. This parameter can be one of the following values:
  - RTC_Format_BIN: Binary data format
  - RTC_Format_BCD: BCD data format
- RTC_TimeStruct: pointer to a RTC_TimeTypeDef structure that contains the time configuration information for the RTC.

Return values:
- An ErrorStatus enumeration value:
  - SUCCESS: RTC Time register is configured
  - ERROR: RTC Time register is not configured

Notes:
- None.

21.2.12.2 RTC_TimeStructInit

Function Name: void RTC_TimeStructInit (RTC_TimeTypeDef *RTC_TimeStruct)

Function Description: Fills each RTC_TimeStruct member with its default value (Time = 00h:00min:00sec).

Parameters:
- RTC_TimeStruct: pointer to a RTC_TimeTypeDef structure which will be initialized.

Return values:
- None.

Notes:
- None.

21.2.12.3 RTC_GetTime

Function Name: void RTC_GetTime (uint32_t RTC_Format, RTC_TimeTypeDef

---

DocID 18540 Rev 1 429/634
**21.2.12.4 RTC_SetDate**

**Function Name**  
`ErrorStatus RTC_SetDate ( uint32_t RTC_Format, RTC_DateTypeDef * RTC_DateStruct)`

**Function Description**  
Set the RTC current date.

**Parameters**  
- **RTC_Format**: specifies the format of the entered parameters. This parameter can be one of the following values:  
  - `RTC_Format_BIN`: Binary data format  
  - `RTC_Format_BCD`: BCD data format  
- **RTC_DateStruct**: pointer to a RTC_DateTypeDef structure that contains the date configuration information for the RTC.

**Return values**  
- An `ErrorStatus` enumeration value:  
  - `SUCCESS`: RTC Date register is configured  
  - `ERROR`: RTC Date register is not configured

**Notes**  
- None.

**21.2.12.5 RTC_DateStructInit**

**Function Name**  
`void RTC_DateStructInit ( RTC_DateTypeDef * RTC_DateStruct)`

**Function Description**  
Fills each RTC_DateStruct member with its default value (Monday, January 01 xx00).
21.2.12.6  RTC_GetDate

Function Name  void RTC_GetDate ( uint32_t RTC_Format, RTC_DateTypeDef * RTC_DateStruct)

Function Description  Get the RTC current date.

Parameters  
- **RTC_Format** : specifies the format of the returned parameters. This parameter can be one of the following values:
  - **RTC_Format_BIN** : Binary data format
  - **RTC_Format_BCD** : BCD data format
- **RTC_DateStruct** : pointer to a RTC_DateTypeDef structure that will contain the returned current date configuration.

Return values  
- None.

Notes  
- None.

21.2.13  Alarm configuration functions

21.2.13.1  RTC_SetAlarm

Function Name  void RTC_SetAlarm ( uint32_t RTC_Format, uint32_t RTC_Alarm, RTC_AlarmTypeDef * RTC_AlarmStruct)

Function Description  Set the specified RTC Alarm.

Parameters  
- **RTC_Format** : specifies the format of the returned parameters. This parameter can be one of the following values:
  - **RTC_Format_BIN** : Binary data format
  - **RTC_Format_BCD** : BCD data format
- **RTC_Alarm** : specifies the alarm to be configured. This parameter can be one of the following values:
  - **RTC_Alarm_A** : to select Alarm A
  - **RTC_Alarm_B** : to select Alarm B
• **RTC_AlarmStruct**: pointer to a RTC_AlarmTypeDef structure that contains the alarm configuration parameters.

**Return values**

• None.

**Notes**

• The Alarm register can only be written when the corresponding Alarm is disabled (Use the RTC_AlarmCmd(DISABLE)).

### 21.2.13.2 RTC_AlarmStructInit

**Function Name**

`void RTC_AlarmStructInit ( RTC_AlarmTypeDef *RTC_AlarmStruct)`

**Function Description**

Fills each RTC_AlarmStruct member with its default value (Time = 00h:00mn:00sec / Date = 1st day of the month/Mask = all fields are masked).

**Parameters**

• **RTC_AlarmStruct**: pointer to a RTC_AlarmTypeDef structure which will be initialized.

**Return values**

• None.

**Notes**

• None.

### 21.2.13.3 RTC_GetAlarm

**Function Name**

`void RTC_GetAlarm ( uint32_t RTC_Format, uint32_t RTC_Alarm, RTC_AlarmTypeDef *RTC_AlarmStruct)`

**Function Description**

Get the RTC Alarm value and masks.

**Parameters**

• **RTC_Format**: specifies the format of the output parameters. This parameter can be one of the following values:
  - **RTC_Format_BIN**: Binary data format
  - **RTC_Format_BCD**: BCD data format

• **RTC_Alarm**: specifies the alarm to be read. This parameter can be one of the following values:
  - **RTC_Alarm_A**: to select Alarm A
  - **RTC_Alarm_B**: to select Alarm B

• **RTC_AlarmStruct**: pointer to a RTC_AlarmTypeDef structure that will contains the output alarm configuration values.
21.2.13.4 RTC_AlarmCmd

**Function Name**
ErrorStatus RTC_AlarmCmd ( uint32_t RTC_Alarm, FunctionalState NewState)

**Function Description**
Enables or disables the specified RTC Alarm.

**Parameters**
- **RTC_Alarm**: specifies the alarm to be configured. This parameter can be any combination of the following values:
  - **RTC_Alarm_A**: to select Alarm A
  - **RTC_Alarm_B**: to select Alarm B
- **NewState**: new state of the specified alarm. This parameter can be: ENABLE or DISABLE.

**Return values**
- An ErrorStatus enumeration value:
  - **SUCCESS**: RTC Alarm is enabled/disabled
  - **ERROR**: RTC Alarm is not enabled/disabled

**Notes**
- None.

21.2.14 WakeUp Timer configuration functions

21.2.14.1 RTC_WakeUpClockConfig

**Function Name**
void RTC_WakeUpClockConfig ( uint32_t RTC_WakeUpClock)

**Function Description**
Configures the RTC Wakeup clock source.

**Parameters**
- **RTC_WakeUpClock**: Wakeup Clock source. This parameter can be one of the following values:
  - **RTC_WakeUpClock_RTCCLK_Div16**: RTC Wakeup Counter Clock = RTCCLK/16
  - **RTC_WakeUpClock_RTCCLK_Div8**: RTC Wakeup Counter Clock = RTCCLK/8
  - **RTC_WakeUpClock_RTCCLK_Div4**: RTC Wakeup Counter Clock = RTCCLK/4
  - **RTC_WakeUpClock_RTCCLK_Div2**: RTC Wakeup Counter Clock = RTCCLK/2
  - **RTC_WakeUpClock_CK_SPRE_16bits**: RTC Wakeup Counter Clock = CK_SPRE
21.2.14.2 RTC_SetWakeUpCounter

Function Name: void RTC_SetWakeUpCounter ( uint32_t RTC_WakeUpCounter )

Function Description: Configures the RTC Wakeup counter.

Parameters:
- RTC_WakeUpCounter: specifies the WakeUp counter. This parameter can be a value from 0x0000 to 0xFFFF.

Return values:
- None.

Notes:
- The RTC WakeUp counter can only be written when the RTC WakeUp is disabled (Use the RTC_WakeUpCmd(DISABLE)).

21.2.14.3 RTC_GetWakeUpCounter

Function Name: uint32_t RTC_GetWakeUpCounter ( void )

Function Description: Returns the RTC WakeUp timer counter value.

Parameters:
- None.

Return values:
- The RTC WakeUp Counter value.

Notes:
- None.

21.2.14.4 RTC_WakeUpCmd
Function Name: ErrorStatus RTC_WakeUpCmd ( FunctionalState NewState)
Function Description: Enables or Disables the RTC WakeUp timer.
Parameters:
- NewState: new state of the WakeUp timer. This parameter can be: ENABLE or DISABLE.
Return values:
- None.
Notes:
- None.

21.2.15 Daylight Saving configuration functions
21.2.15.1 RTC_DayLightSavingConfig

Function Name: void RTC_DayLightSavingConfig ( uint32_t RTC_DayLightSaving, uint32_t RTC_StoreOperation)
Function Description: Adds or substract one hour from the current time.
Parameters:
- RTC_DayLightSaveOperation: the value of hour adjustment. This parameter can be one of the following values:
  - RTC_DayLightSaving_SUB1H: Substract one hour (winter time)
  - RTC_DayLightSaving_ADD1H: Add one hour (summer time)
- RTC_StoreOperation: Specifies the value to be written in the BCK bit in CR register to store the operation. This parameter can be one of the following values:
  - RTC_StoreOperation_Reset: BCK Bit Reset
  - RTC_StoreOperation_Set: BCK Bit Set
Return values:
- None.
Notes:
- None.

21.2.15.2 RTC_GetStoreOperation

Function Name: uint32_t RTC_GetStoreOperation ( void )
Function Description: Returns the RTC Day Light Saving stored operation.
Parameters:
- None.
Return values

- RTC Day Light Saving stored operation.
  - `RTC_StoreOperation_Reset`
  - `RTC_StoreOperation_Set`

Notes

- None.

### 21.2.16 Output pin Configuration function

#### 21.2.16.1 RTC_OutputConfig

**Function Name**

```c
void RTC_OutputConfig ( uint32_t RTC_Output, uint32_t RTC_OutputPolarity)
```

**Function Description**

Configures the RTC output source (AFO_ALARM).

**Parameters**

- **RTC_Output** : Specifies which signal will be routed to the RTC output. This parameter can be one of the following values:
  - `RTC_Output_Disable` : No output selected
  - `RTC_Output_AlarmA` : signal of AlarmA mapped to output
  - `RTC_Output_AlarmB` : signal of AlarmB mapped to output
  - `RTC_Output_WakeUp` : signal of WakeUp mapped to output

- **RTC_OutputPolarity** : Specifies the polarity of the output signal. This parameter can be one of the following:
  - `RTC_OutputPolarity_High` : The output pin is high when the ALRAF/ALRBF/WUTF is high (depending on OSEL)
  - `RTC_OutputPolarity_Low` : The output pin is low when the ALRAF/ALRBF/WUTF is high (depending on OSEL)

**Return values**

- None.

**Notes**

- None.

### 21.2.17 Coarse Calibration configuration functions

#### 21.2.17.1 RTC_CoarseCalibConfig

**Function Name**

```c
ErrorStatus RTC_CoarseCalibConfig ( uint32_t
```

```c
```
RTC_CalibSign, uint32_t Value)

Function Description
Configures the Coarse calibration parameters.

Parameters
- **RTC_CalibSign**: specifies the sign of the coarse calibration value. This parameter can be one of the following values:
  - **RTC_CalibSign_Positive**: The value sign is positive
  - **RTC_CalibSign_Negative**: The value sign is negative
- **Value**: value of coarse calibration expressed in ppm (coded on 5 bits).

Return values
- An ErrorStatus enumeration value:
  - **SUCCESS**: RTC Coarse calibration are initialized
  - **ERROR**: RTC Coarse calibration are not initialized

Notes
- This Calibration value should be between 0 and 63 when using negative sign with a 2-ppm step.
- This Calibration value should be between 0 and 126 when using positive sign with a 4-ppm step.

---

21.2.17.2 RTC_CoarseCalibCmd

Function Name
ErrorStatus RTC_CoarseCalibCmd ( FunctionalState NewState)

Function Description
Enables or disables the Coarse calibration process.

Parameters
- **NewState**: new state of the Coarse calibration. This parameter can be: ENABLE or DISABLE.

Return values
- An ErrorStatus enumeration value:
  - **SUCCESS**: RTC Coarse calibration are enabled/disabled
  - **ERROR**: RTC Coarse calibration are not enabled/disabled

Notes
- None.

---

21.2.17.3 RTC_CalibOutputCmd

Function Name
void RTC_CalibOutputCmd ( FunctionalState NewState)

Function Description
Enables or disables the RTC clock to be output through the relative pin.
### Parameters
- **NewState**: new state of the digital calibration output. This parameter can be: ENABLE or DISABLE.

### Return values
- None.

### Notes
- None.

## 21.2.18 TimeStamp configuration functions

### 21.2.18.1 RTC_TimeStampCmd

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void RTC_TimeStampCmd ( uint32_t RTC_TimeStampEdge, FunctionalState NewState)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Enables or Disables the RTC TimeStamp functionality with the specified time stamp pin stimulating edge.</td>
</tr>
</tbody>
</table>
| Parameters | • **RTC_TimeStampEdge**: Specifies the pin edge on which the TimeStamp is activated. This parameter can be one of the following:  
  - **RTC_TimeStampEdge_Rising**: the Time stamp event occurs on the rising edge of the related pin.  
  - **RTC_TimeStampEdge_Falling**: the Time stamp event occurs on the falling edge of the related pin.  
  
  • **NewState**: new state of the TimeStamp. This parameter can be: ENABLE or DISABLE. |
| Return values | • None. |
| Notes | • None. |

### 21.2.18.2 RTC_GetTimeStamp

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void RTC_GetTimeStamp ( uint32_t RTC_Format, RTC_TimeTypeDef * RTC_StampTimeStruct, RTC_DateTypeDef * RTC_StampDateStruct)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Get the RTC TimeStamp value and masks.</td>
</tr>
</tbody>
</table>
| Parameters | • **RTC_Format**: specifies the format of the output parameters. This parameter can be one of the following values:  
  - **RTC_Format_BIN**: Binary data format  
  - **RTC_Format_BCD**: BCD data format  
  
  • **RTC_StampTimeStruct**: pointer to a RTC_TimeTypeDef |
| Notes |  |
structure that will contains the TimeStamp time values.
- **RTC_StampDateStruct**: pointer to a RTC_DateTypeDef structure that will contains the TimeStamp date values.

**Return values**
- None.

**Notes**
- None.

### 21.2.19 Tampers configuration functions

### 21.2.19.1 RTC_TamperTriggerConfig

**Function Name**  
`void RTC_TamperTriggerConfig ( uint32_t RTC_Tamper, uint32_t RTC_TamperTrigger)`  

**Function Description**  
Configures the select Tamper pin edge.

**Parameters**
- **RTC_Tamper**: Selected tamper pin. This parameter can be RTC_Tamper_1.
- **RTC_TamperTrigger**: Specifies the trigger on the tamper pin that stimulates tamper event. This parameter can be one of the following values:
  - **RTC_TamperTrigger_RisingEdge**: Rising Edge of the tamper pin causes tamper event.
  - **RTC_TamperTrigger_FallingEdge**: Falling Edge of the tamper pin causes tamper event.

**Return values**
- None.

**Notes**
- None.

### 21.2.19.2 RTC_TamperCmd

**Function Name**  
`void RTC_TamperCmd ( uint32_t RTC_Tamper, FunctionalState NewState)`  

**Function Description**  
Enables or Disables the Tamper detection.

**Parameters**
- **RTC_Tamper**: Selected tamper pin. This parameter can be RTC_Tamper_1.
- **NewState**: new state of the tamper pin. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.
21.3 RTC Firmware driver defines

21.3.1 RTC Firmware driver defines

RTC

RTC_AlarmDateWeekDay_Definitions

- #define: `RTC_AlarmDateWeekDaySel_Date((uint32_t)0x00000000)`

- #define: `RTC_AlarmDateWeekDaySel_WeekDay((uint32_t)0x40000000)`

RTC_AlarmMask_Definitions

- #define: `RTC_AlarmMask_None((uint32_t)0x00000000)`

- #define: `RTC_AlarmMask_DateWeekDay((uint32_t)0x80000000)`

- #define: `RTC_AlarmMask_Hours((uint32_t)0x00800000)`

- #define: `RTC_AlarmMask_Minutes((uint32_t)0x00008000)`

- #define: `RTC_AlarmMask_Seconds((uint32_t)0x00000080)`

- #define: `RTC_AlarmMask_All((uint32_t)0x80808080)`

RTC_Alarms_Definitions

- #define: `RTC_Alarm_A((uint32_t)0x00000100)`
• #define: \texttt{RTC\_Alarm\_B((uint32\_t)0x00000200)}

\textit{RTC\_AM\_PM\_Definitions}

• #define: \texttt{RTC\_H12\_AM((uint8\_t)0x00)}

• #define: \texttt{RTC\_H12\_PM((uint8\_t)0x40)}

\textit{RTC\_Backup\_Registers\_Definitions}

• #define: \texttt{RTC\_BKP\_DR0((uint32\_t)0x00000000)}

• #define: \texttt{RTC\_BKP\_DR1((uint32\_t)0x00000001)}

• #define: \texttt{RTC\_BKP\_DR2((uint32\_t)0x00000002)}

• #define: \texttt{RTC\_BKP\_DR3((uint32\_t)0x00000003)}

• #define: \texttt{RTC\_BKP\_DR4((uint32\_t)0x00000004)}

• #define: \texttt{RTC\_BKP\_DR5((uint32\_t)0x00000005)}

• #define: \texttt{RTC\_BKP\_DR6((uint32\_t)0x00000006)}

• #define: \texttt{RTC\_BKP\_DR7((uint32\_t)0x00000007)}
• #define: `RTC_BKP_DR8((uint32_t)0x00000008)`

• #define: `RTC_BKP_DR9((uint32_t)0x00000009)`

• #define: `RTC_BKP_DR10((uint32_t)0x0000000A)`

• #define: `RTC_BKP_DR11((uint32_t)0x0000000B)`

• #define: `RTC_BKP_DR12((uint32_t)0x0000000C)`

• #define: `RTC_BKP_DR13((uint32_t)0x0000000D)`

• #define: `RTC_BKP_DR14((uint32_t)0x0000000E)`

• #define: `RTC_BKP_DR15((uint32_t)0x0000000F)`

• #define: `RTC_BKP_DR16((uint32_t)0x00000010)`

• #define: `RTC_BKP_DR17((uint32_t)0x00000011)`

• #define: `RTC_BKP_DR18((uint32_t)0x00000012)`

• #define: `RTC_BKP_DR19((uint32_t)0x00000013)`
RTCP\_DayLightSaving\_Definitions

- \#define: \texttt{RTC\_DayLightSaving\_SUB1H((uint32\_t)0x00020000)}

- \#define: \texttt{RTC\_DayLightSaving\_ADD1H((uint32\_t)0x00010000)}

- \#define: \texttt{RTC\_StoreOperation\_Reset((uint32\_t)0x00000000)}

- \#define: \texttt{RTC\_StoreOperation\_Set((uint32\_t)0x00040000)}

RTCP\_Digital\_Calibration\_Definitions

- \#define: \texttt{RTC\_CalibSign\_Positive((uint32\_t)0x00000000)}

- \#define: \texttt{RTC\_CalibSign\_Negative((uint32\_t)0x00000080)}

RTCP\_Flags\_Definitions

- \#define: \texttt{RTC\_FLAG\_TAMP1F((uint32\_t)0x00002000)}

- \#define: \texttt{RTC\_FLAG\_TSOVF((uint32\_t)0x00001000)}

- \#define: \texttt{RTC\_FLAG\_TSF((uint32\_t)0x00000800)}

- \#define: \texttt{RTC\_FLAG\_WUTF((uint32\_t)0x00000400)}

- \#define: \texttt{RTC\_FLAG\_ALRBF((uint32\_t)0x00000200)}
- #define: `RTC_FLAG_ALRAF((uint32_t)0x00000100)`

- #define: `RTC_FLAG_INITF((uint32_t)0x00000040)`

- #define: `RTC_FLAG_RSF((uint32_t)0x00000020)`

- #define: `RTC_FLAG_INITS((uint32_t)0x00000010)`

- #define: `RTC_FLAG_WUTWF((uint32_t)0x00000004)`

- #define: `RTC_FLAG_ALRBWF((uint32_t)0x00000002)`

- #define: `RTC_FLAG_ALRAWF((uint32_t)0x00000001)`

**RTC_Hour_Formats**

- #define: `RTC_HourFormat_24((uint32_t)0x00000000)`

- #define: `RTC_HourFormat_12((uint32_t)0x00000040)`

**RTC_Input_parameter_format_definitions**

- #define: `RTC_Format_BIN((uint32_t)0x00000000)`

- #define: `RTC_Format_BCD((uint32_t)0x00000001)`
RTC Interrupts Definitions

- #define: `RTC_IT_TS((uint32_t)0x00008000)`

- #define: `RTC_IT_WUT((uint32_t)0x00004000)`

- #define: `RTC_IT_ALRB((uint32_t)0x00002000)`

- #define: `RTC_IT_ALRA((uint32_t)0x00001000)`

- #define: `RTC_IT_TAMP((uint32_t)0x00000004)`

- #define: `RTC_IT_TAMP1((uint32_t)0x00020000)`

RTC Legacy

- #define: `RTC_DigitalCalibConfigRTC_CoarseCalibConfig`

- #define: `RTC_DigitalCalibCmdRTC_CoarseCalibCmd`

RTC Month Date Definitions

- #define: `RTC_Month_January((uint32_t)0x00000001)`

- #define: `RTC_Month_February((uint32_t)0x00000002)`

- #define: `RTC_Month_March((uint32_t)0x00000003)`
Real-time clock (RTC)

- #define: `RTC_Month_April((uint32_t)0x00000004)`

- #define: `RTC_Month_May((uint32_t)0x00000005)`

- #define: `RTC_Month_June((uint32_t)0x00000006)`

- #define: `RTC_Month_July((uint32_t)0x00000007)`

- #define: `RTC_Month_August((uint32_t)0x00000008)`

- #define: `RTC_Month_September((uint32_t)0x00000009)`

- #define: `RTC_Month_October((uint32_t)0x00000010)`

- #define: `RTC_Month_November((uint32_t)0x00000011)`

- #define: `RTC_Month_December((uint32_t)0x00000012)`

*RTC_Output_Polarity_Definitions*

- #define: `RTC_OutputPolarity_High((uint32_t)0x00000000)`

- #define: `RTC_OutputPolarity_Low((uint32_t)0x00100000)`

*RTC_Output_selection_Definitions*

- #define: `RTC_Output_Disable((uint32_t)0x00000000)`
• #define: `RTC_Output_AlarmA((uint32_t)0x00200000)`

• #define: `RTC_Output_AlarmB((uint32_t)0x00400000)`

• #define: `RTC_Output_WakeUp((uint32_t)0x00600000)`

RTC_Output_Type_ALARM_OUT
- #define: `RTC_OutputType_OpenDrain((uint32_t)0x00000000)`

- #define: `RTC_OutputType_PushPull((uint32_t)0x00040000)`

RTC_Tamper_Pins_Definitions
- #define: `RTC_Tamper_1RTC_TAFCR_TAMP1E`

RTC_Tamper_Pin_Selection
- #define: `RTC_TamperPin_PC13((uint32_t)0x00000000)`

- #define: `RTC_TamperPin_PI8((uint32_t)0x00010000)`

RTC_Tamper_Trigger_Definitions
- #define: `RTC_TamperTrigger_RisingEdge((uint32_t)0x00000000)`

- #define: `RTC_TamperTrigger_FallingEdge((uint32_t)0x00000001)`
RTC_TimeStamp_Pin_Selection
• #define: RTC_TimeStampPin_PC13((uint32_t)0x00000000)

• #define: RTC_TimeStampPin_PI8((uint32_t)0x00020000)

RTC_Time_Stamp_Edges_definitions
• #define: RTC_TimeStampEdge_Rising((uint32_t)0x00000000)

• #define: RTC_TimeStampEdge_Falling((uint32_t)0x00000008)

RTC_Wakeup_Timer_Definitions
• #define: RTC_WakeUpClock_RTCCLK_Div16((uint32_t)0x00000000)

• #define: RTC_WakeUpClock_RTCCLK_Div8((uint32_t)0x00000001)

• #define: RTC_WakeUpClock_RTCCLK_Div4((uint32_t)0x00000002)

• #define: RTC_WakeUpClock_RTCCLK_Div2((uint32_t)0x00000003)

• #define: RTC_WakeUpClock_CK_SPRE_16bits((uint32_t)0x00000004)

• #define: RTC_WakeUpClock_CK_SPRE_17bits((uint32_t)0x00000006)

RTC_WeekDay_Definitions
• #define: RTC_Weekday_Monday((uint32_t)0x00000001)
### 21.4 RTC Programming Example

The example below explains how to configure the RTC clock source, calendar, Time and Date. For more examples about RTC configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\RTC\.

```c
RTC_InitTypeDef   RTC_InitStructure;
RTC_TimeTypeDef   RTC_TimeStructure;
RTC_DateTypeDef   RTC_DateStructure;

/* Enable write access to the RTC *************************************/
/* Enable the PWR clock */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR, ENABLE);

/* Allow access to RTC */
PWR_BackupAccessCmd(ENABLE);

/* Configure the RTC clock source *************************************/
/* Enable the LSE OSC */
RCC_LSEConfig(RCC_LSE_ON);

/* Wait till LSE is ready */
while(RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)
{
```

- `#define: RTC_Weekday_Tuesday((uint32_t)0x00000002)`
- `#define: RTC_Weekday_Wednesday((uint32_t)0x00000003)`
- `#define: RTC_Weekday_Thursday((uint32_t)0x00000004)`
- `#define: RTC_Weekday_Friday((uint32_t)0x00000005)`
- `#define: RTC_Weekday_Saturday((uint32_t)0x00000006)`
- `#define: RTC_Weekday_Sunday((uint32_t)0x00000007)`
/* Select the RTC Clock Source */
RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);

/* Enable the RTC Clock */
RCC_RTCCLKCmd(ENABLE);

/* Wait for RTC APB registers synchronisation */
RTC_WaitForSynchro();

/* Configure the RTC calendar, Time and Date */

/* RTC time base = LSE / ((AsynchPrediv+1) * (SynchPrediv+1)) */
RTC_InitStructure.RTC_AsynchPrediv = 127;
RTC_InitStructure.RTC_SynchPrediv = 255;
RTC_InitStructure.RTC_HourFormat = RTC_HourFormat_24;
RTC_Init(&RTC_InitStructure);

/* Set the Time */
RTC_TimeStructure.RTC_Hours = 0x08;
RTC_TimeStructure.RTC_Minutes = 0x00;
RTC_TimeStructure.RTC_Seconds = 0x00;
RTC_SetTime(RTC_Format_BCD, &RTC_TimeStructure);

/* Set the Date */
RTC_DateStructure.RTC_Month = RTC_Month_May;
RTC_DateStructure.RTC_WeekDay = RTC_Weekday_Monday;
RTC_DateStructure.RTC_Date = 0x30;
RTC_DateStructure.RTC_Year = 0x11;
RTC_SetDate(RTC_Format_BCD, &RTC_DateStructure);
Secure digital input/output interface (SDIO)

22.1 SDIO Firmware driver registers structures

22.1.1 SDIO_TypeDef

SDIO_TypeDef is defined in the stm32f2x2.h file and contains the SDIO registers definition.

Data Fields

- __IO uint32_t POWER
- __IO uint32_t CLKCR
- __IO uint32_t ARG
- __IO uint32_t CMD
- __I uint32_t RESPCMD
- __I uint32_t RESP1
- __I uint32_t RESP2
- __I uint32_t RESP3
- __I uint32_t RESP4
- __IO uint32_t DTIMER
- __IO uint32_t DLEN
- __IO uint32_t DCTRL
- __I uint32_t DCOUNT
- __I uint32_t STA
- __IO uint32_t ICR
- __IO uint32_t MASK
- uint32_t RESERVED0
- __I uint32_t FIFOCNT
- uint32_t RESERVED1
- __IO uint32_t FIFO

Field Documentation

- __IO uint32_t SDIO_TypeDef::POWER
  - SDIO power control register, Address offset: 0x00
- __IO uint32_t SDIO_TypeDef::CLKCR
  - SDI clock control register, Address offset: 0x04
- __IO uint32_t SDIO_TypeDef::ARG
  - SDIO argument register, Address offset: 0x08
- __IO uint32_t SDIO_TypeDef::CMD
  - SDIO command register, Address offset: 0x0C
- __I uint32_t SDIO_TypeDef::RESPCMD
  - SDIO command response register, Address offset: 0x10
- __I uint32_t SDIO_TypeDef::RESP1
  - SDIO response 1 register, Address offset: 0x14
- __I uint32_t SDIO_TypeDef::RESP2
  - SDIO response 2 register, Address offset: 0x18
- __I uint32_t SDIO_TypeDef::RESP3
Secure digital input/output interface (SDIO)

- SDIO response 3 register, Address offset: 0x1C
- __I uint32_t SDIO_TypeDef::RESP4
- SDIO response 4 register, Address offset: 0x20
- __IO uint32_t SDIO_TypeDef::DTIMER
- SDIO data timer register, Address offset: 0x24
- __IO uint32_t SDIO_TypeDef::DLEN
- SDIO data length register, Address offset: 0x28
- __IO uint32_t SDIO_TypeDef::DCTRL
- SDIO data control register, Address offset: 0x2C
- __I uint32_t SDIO_TypeDef::DCOUNT
- SDIO data counter register, Address offset: 0x30
- __I uint32_t SDIO_TypeDef::STA
- SDIO status register, Address offset: 0x34
- __IO uint32_t SDIO_TypeDef::ICR
- SDIO interrupt clear register, Address offset: 0x38
- __IO uint32_t SDIO_TypeDef::MASK
- SDIO mask register, Address offset: 0x3C
- uint32_t SDIO_TypeDef::RESERVED0[2]
- Reserved, 0x40-0x44
- __I uint32_t SDIO_TypeDef::FIFOCNT
- SDIO FIFO counter register, Address offset: 0x48
- uint32_t SDIO_TypeDef::RESERVED1[13]
- Reserved, 0x4C-0x7C
- __IO uint32_t SDIO_TypeDef::FIFO
- SDIO data FIFO register, Address offset: 0x80

22.1.2 SDIO_InitTypeDef

SDIO_InitTypeDef is defined in the stm32f2xx_sdio.h file and contains the SDIO initialization parameters.

Data Fields

- uint32_t SDIO_ClockEdge
- uint32_t SDIO_ClockBypass
- uint32_t SDIO_ClockPowerSave
- uint32_t SDIO_BusWide
- uint32_t SDIO_HardwareFlowControl
- uint8_t SDIO_ClockDiv

Field Documentation

- uint32_t SDIO_InitTypeDef::SDIO_ClockEdge
  - Specifies the clock transition on which the bit capture is made. This parameter can be a value of SDIO_Clock_Edge
- uint32_t SDIO_InitTypeDef::SDIO_ClockBypass
  - Specifies whether the SDIO Clock divider bypass is enabled or disabled. This parameter can be a value of SDIO_Clock_Bypass
- uint32_t SDIO_InitTypeDef::SDIO_ClockPowerSave
22.1.3 SDIO_CmdInitTypeDef

SDIO_CmdInitTypeDef is defined in the stm32f2xx_sdio.h file and contains the SDIO command parameters.

Data Fields

- uint32_t SDIO_Argument
- uint32_t SDIO_CmdIndex
- uint32_t SDIO_Response
- uint32_t SDIO_Wait
- uint32_t SDIO_CPSM

Field Documentation

- uint32_t SDIO_CmdInitTypeDef::SDIO_Argument
  - Specifies the SDIO command argument which is sent to a card as part of a command message. If a command contains an argument, it must be loaded into this register before writing the command to the command register.

- uint32_t SDIO_CmdInitTypeDef::SDIO_CmdIndex
  - Specifies the SDIO command index. It must be lower than 0x40.

- uint32_t SDIO_CmdInitTypeDef::SDIO_Response
  - Specifies the SDIO response type. This parameter can be a value of SDIO_Response_Type.

- uint32_t SDIO_CmdInitTypeDef::SDIO_Wait
  - Specifies whether SDIO wait-for-interrupt request is enabled or disabled. This parameter can be a value of SDIO.Wait Interrupt State.

- uint32_t SDIO_CmdInitTypeDef::SDIO_CPSM
  - Specifies whether SDIO Command path state machine (CPSM) is enabled or disabled. This parameter can be a value of SDIO.CPSM State.

22.1.4 SDIO_DataInitTypeDef

SDIO_DataInitTypeDef is defined in the stm32f2xx_sdio.h file and contains the SDIO data parameters.

Data Fields
Secure digital input/output interface (SDIO)

Field Documentation

- `uint32_t SDIO_DataTimeOut`
  - Specifies the data timeout period in card bus clock periods.
- `uint32_t SDIO_DataLength`
  - Specifies the number of data bytes to be transferred.
- `uint32_t SDIO_DataBlockSize`
  - Specifies the data block size for block transfer. This parameter can be a value of `SDIO_Data_Block_Size`.
- `uint32_t SDIO_TransferDir`
  - Specifies the data transfer direction, whether the transfer is a read or write. This parameter can be a value of `SDIO_Transfer_Direction`.
- `uint32_t SDIO_TransferMode`
  - Specifies whether data transfer is in stream or block mode. This parameter can be a value of `SDIO_Transfer_Type`.
- `uint32_t SDIO_DPSM`
  - Specifies whether SDIO Data path state machine (DPSM) is enabled or disabled. This parameter can be a value of `SDIO_DPSM_State`.

22.2 SDIO Firmware driver API description

The following section lists the various functions of the SDIO library.

22.2.1 How to use this driver

1. The SDIO clock (SDIOCLK = 48 MHz) is coming from a specific output of PLL (PLL48CLK). Before starting to work with SDIO peripheral make sure that the PLL is well configured. The SDIO peripheral uses two clock signals: SDIO adapter clock (SDIOCLK = 48 MHz) APB2 bus clock (PCLK2). PCLK2 and SDIO_CK clock frequencies must respect the following condition: `Freq(PCLK2) >= (3 / 8 x Frequency(SDIO_CK))`.
2. Enable peripheral clock using `RCC_APB2PeriphClockCmd(RCC_APB2Periph_SDIO, ENABLE)`.
3. According to the SDIO mode, enable the GPIO clocks using `RCC_AHB1PeriphClockCmd()` function. The I/O can be one of the following configurations:
   - 1-bit data length: SDIO_CMD, SDIO_CK and D0.
   - 4-bit 8-bit data length: SDIO_CMD, SDIO_CK and D[7:0].
   - 8-bit data length: SDIO_CMD, SDIO_CK and D[7:0].
4. Peripheral's alternate function:
Connect the pin to the desired peripherals' Alternate Function (AF) using GPIO_PinAFConfig() function

Configure the desired pin in alternate function by: GPIO_InitStruct->GPIO_Mode = GPIO_Mode_AF

Select the type, pull-up/pull-down and output speed via GPIO_Speed, GPIO_OType and GPIO_PuPd members

Call GPIO_Init() function

5. Program the Clock Edge, Clock Bypass, Clock Power Save, Bus Wide, hardware, flow control and the Clock Divider using the SDIO_Init() function.

6. Enable the Power ON State using the SDIO_SetPowerState(SDIO_PowerState_ON) function.

7. Enable the clock using the SDIO_ClockCmd() function.

8. Enable the NVIC and the corresponding interrupt using the function SDIO_ITConfig() if you need to use interrupt mode.

9. When using the DMA mode
   - Configure the DMA using DMA_Init() function
   - Activate the needed channel Request using SDIO_DMAChCmd() function

10. Enable the DMA using the DMA_Cmd() function, when using DMA mode.

11. To control the CPSM (Command Path State Machine) and send commands to the card use the SDIO_SendCommand(), SDIO_GetCommandResponse() and SDIO_GetResponse() functions. First, user has to fill the command structure (pointer to SDIO_CmdInitTypeDef) according to the selected command to be sent. The parameters that should be filled are: Command Argument Command Index Command Response type Command Wait CPSM Status (Enable or Disable)

To check if the command is well received, read the SDIO_CMDRESP register using the SDIO_GetCommandResponse(). The SDIO responses registers (SDIO_RESP1 to SDIO_RESP2), use the SDIO_GetResponse() function.

12. To control the DPSM (Data Path State Machine) and send/receive data to/from the card use the SDIO_DataConfig(), SDIO_GetDataCounter(), SDIO_ReadData(), SDIO_WriteData() and SDIO_GetFIFOCount() functions.

Read Operations

1. First, user has to fill the data structure (pointer to SDIO_DataInitTypeDef) according to the selected data type to be received. The parameters that should be filled are:
   - Data TimeOut
   - Data Length
   - Data Block size
   - Data Transfer direction: should be from card (To SDIO)
   - Data Transfer mode
   - DPSM Status (Enable or Disable)

2. Configure the SDIO resources to receive the data from the card according to selected transfer mode (Refer to Step 8, 9 and 10).

3. Send the selected Read command (refer to step 11).

4. Use the SDIO flags/interrupts to check the transfer status.

Write Operations

1. First, user has to fill the data structure (pointer to SDIO_DataInitTypeDef) according to the selected data type to be received. The parameters that should be filled are:
   - Data TimeOut
   - Data Length
   - Data Block size
   - Data Transfer direction: should be to card (To CARD)
   - Data Transfer mode
22.2.2 Initialization and configuration

- Section 22.2.5.1: "SDIO_DeInit"
- Section 22.2.5.2: "SDIO_Init"
- Section 22.2.5.3: "SDIO_StructInit"
- Section 22.2.5.4: "SDIO_ClockCmd"
- Section 22.2.5.6: "SDIO_GetPowerState"
- Section 22.2.5.5: "SDIO_SetPowerState"

22.2.3 Command path state machine (CPSM) management

This section provides functions allowing to program and read the Command path state machine (CPSM).

- SDIO_SendCommand()
- SDIO_CmdStructInit()
- SDIO_GetCommandResponse()
- SDIO_GetResponse()

Data path state machine (DPSM) management functions

This section provides functions allowing to program and read the Data path state machine (DPSM).

- SDIO_DataConfig()
- SDIO_DataStructInit()
- SDIO_GetDataCounter()
- SDIO_ReadData()
- SDIO_WriteData()
- SDIO_GetFIFOCount()

SDIO IO Cards mode management functions

This section provides functions allowing to program and read the SDIO IO Cards.

- SDIO_StartSDIOReadWait()
- SDIO_StopSDIOReadWait()
- SDIO_SetSDIOReadWaitMode()
- SDIO_SetSDIOOperation()
- SDIO_SendSDIOSuspendCmd()

CE-ATA mode management functions

This section provides functions allowing to program and read the CE-ATA card.

- SDIO_CommandCompletionCmd()
- SDIO_CEATAITCmd()
- SDIO_SendCEATAITCmd()

DMA transfers management functions

This section provides functions allowing to program SDIO DMA transfer.
Secure digital input/output interface (SDIO)

- SDIO_DMACmd()

### 22.2.4 Interrupt and flag management

- SDIO_ITConfig()
- SDIO_GetFlagStatus()
- SDIO_ClearFlag()
- SDIO_GetITStatus()
- SDIO_ClearITPendingBit()

### 22.2.5 Initialization and configuration functions

#### 22.2.5.1 SDIO_DeInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void SDIO_DeInit ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Deinitializes the SDIO peripheral registers to their default reset values.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• None.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>

#### 22.2.5.2 SDIO_Init

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void SDIO_Init ( SDIO_InitTypeDef * SDIO_InitStruct)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Initializes the SDIO peripheral according to the specified parameters in the SDIO_InitStruct.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• SDIO_InitStruct : : pointer to a SDIO_InitTypeDef structure that contains the configuration information for the SDIO peripheral.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>

#### 22.2.5.3 SDIO_StructInit
Function Name: `void SDIO_StructInit ( SDIO_InitTypeDef * SDIO_InitStruct)`

Function Description: Fills each SDIO_InitStruct member with its default value.

Parameters:
- `SDIO_InitStruct`: pointer to an SDIO_InitTypeDef structure which will be initialized.

Return values:
- None.

Notes:
- None.

### 22.2.5.4 SDIO_ClockCmd

Function Name: `void SDIO_ClockCmd ( FunctionalState NewState)`

Function Description: Enables or disables the SDIO Clock.

Parameters:
- `NewState`: new state of the SDIO Clock. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.

### 22.2.5.5 SDIO_SetPowerState

Function Name: `void SDIO_SetPowerState ( uint32_t SDIO_PowerState)`

Function Description: Sets the power status of the controller.

Parameters:
- `SDIO_PowerState`: new state of the Power state. This parameter can be one of the following values:
  - `SDIO_PowerState_OFF`: SDIO Power OFF
  - `SDIO_PowerState_ON`: SDIO Power ON

Return values:
- None.

Notes:
- None.
22.2.5.6 SDIO_GetPowerState

Function Name: uint32_t SDIO_GetPowerState ( void )

Function Description: Gets the power status of the controller.

Parameters:
- None.

Return values:
- Power status of the controller. The returned value can be one of the following values:
  - 0x00: Power OFF
  - 0x02: Power UP
  - 0x03: Power ON

Notes:
- None.

22.2.6 Command path state machine (CPSM) management functions

22.2.6.1 SDIO_SendCommand

Function Name: void SDIO_SendCommand ( SDIO_CmdInitTypeDef * SDIO_CmdInitStruct)

Function Description: Initializes the SDIO Command according to the specified parameters in the SDIO_CmdInitStruct and sends the command.

Parameters:
- SDIO_CmdInitStruct: pointer to a SDIO_CmdInitTypeDef structure that contains the configuration information for the SDIO command.

Return values:
- None.

Notes:
- None.

22.2.6.2 SDIO_CmdStructInit

Function Name: void SDIO_CmdStructInit ( SDIO_CmdInitTypeDef * SDIO_CmdInitStruct)

Function Description: Fills each SDIO_CmdInitStruct member with its default value.

Parameters:
- SDIO_CmdInitStruct: pointer to an SDIO_CmdInitTypeDef
22.2.6.3  SDIO_GetCommandResponse

Function Name  
uint8_t SDIO_GetCommandResponse ( void )

Function Description  
Returns command index of last command for which response received.

Parameters  
- None.

Return values  
- Returns the command index of the last command response received.

Notes  
- None.

22.2.6.4  SDIO_GetResponse

Function Name  
uint32_t SDIO_GetResponse ( uint32_t SDIO_RESP)

Function Description  
Returns response received from the card for the last command.

Parameters  
- SDIO_RESP : Specifies the SDIO response register. This parameter can be one of the following values:
  - SDIO_RESP1 : Response Register 1
  - SDIO_RESP2 : Response Register 2
  - SDIO_RESP3 : Response Register 3
  - SDIO_RESP4 : Response Register 4

Return values  
- The Corresponding response register value.

Notes  
- None.
22.2.7 Data path state machine (DPSM) management functions

22.2.7.1 SDIO_DataConfig

Function Name: void SDIO_DataConfig (SDIO_DataInitTypeDef *SDIO_DataInitStruct)

Function Description: Initializes the SDIO data path according to the specified parameters in the SDIO_DataInitStruct.

Parameters:
- SDIO_DataInitStruct: pointer to a SDIO_DataInitTypeDef structure that contains the configuration information for the SDIO command.

Return values:
- None.

Notes:
- None.

22.2.7.2 SDIO_DataStructInit

Function Name: void SDIO_DataStructInit (SDIO_DataInitTypeDef *SDIO_DataInitStruct)

Function Description: Fills each SDIO_DataInitStruct member with its default value.

Parameters:
- SDIO_DataInitStruct: pointer to an SDIO_DataInitTypeDef structure which will be initialized.

Return values:
- None.

Notes:
- None.

22.2.7.3 SDIO_GetDataCounter

Function Name: uint32_t SDIO_GetDataCounter (void)

Function Description: Returns number of remaining data bytes to be transferred.

Parameters:
- None.

Return values:
- Number of remaining data bytes to be transferred

Notes:
- None.
22.2.7.4  SDIO_ReadData

Function Name          uint32_t SDIO_ReadData ( void )
Function Description   Read one data word from Rx FIFO.
Parameters            •  None.
Return values         •  Data received
Notes                 •  None.

22.2.7.5  SDIO_WriteData

Function Name          void SDIO_WriteData ( uint32_t Data )
Function Description   Write one data word to Tx FIFO.
Parameters            •  Data : 32-bit data word to write.
Return values         •  None.
Notes                 •  None.

22.2.7.6  SDIO_GetFIFOCount

Function Name          uint32_t SDIO_GetFIFOCount ( void )
Function Description   Returns the number of words left to be written to or read from FIFO.
Parameters            •  None.
Return values         •  Remaining number of words.
Notes                 •  None.
22.2.8 SDIO IO Cards mode management functions

22.2.8.1 SDIO_StartSDIOReadWait

Function Name: void SDIO_StartSDIOReadWait ( FunctionalState NewState)
Function Description: Starts the SD I/O Read Wait operation.
Parameters:
- **NewState**: new state of the Start SDIO Read Wait operation. This parameter can be: ENABLE or DISABLE.
Return values:
- None.
Notes:
- None.

22.2.8.2 SDIO_StopSDIOReadWait

Function Name: void SDIO_StopSDIOReadWait ( FunctionalState NewState)
Function Description: Stops the SD I/O Read Wait operation.
Parameters:
- **NewState**: new state of the Stop SDIO Read Wait operation. This parameter can be: ENABLE or DISABLE.
Return values:
- None.
Notes:
- None.

22.2.8.3 SDIO_SetSDIOReadWaitMode

Function Name: void SDIO_SetSDIOReadWaitMode ( uint32_t SDIO_ReadWaitMode)
Function Description: Sets one of the two options of inserting read wait interval.
Parameters:
- **SDIO_ReadWaitMode**: SD I/O Read Wait operation mode. This parameter can be:
  - **SDIO_ReadWaitMode_CLK**: Read Wait control by stopping SDIOCLK
  - **SDIO_ReadWaitMode_DATA2**: Read Wait control
22.2.8.4 SDIO_SetSDIOOperation

Function Name  void SDIO_SetSDIOOperation ( FunctionalState NewState)
Function Description  Enables or disables the SD I/O Mode Operation.
Parameters  • NewState : new state of SDIO specific operation. This parameter can be: ENABLE or DISABLE.
Return values  • None.
Notes  • None.

22.2.8.5 SDIO_SendSDIOSuspendCmd

Function Name  void SDIO_SendSDIOSuspendCmd ( FunctionalState NewState)
Function Description  Enables or disables the SD I/O Mode suspend command sending.
Parameters  • NewState : new state of the SD I/O Mode suspend command. This parameter can be: ENABLE or DISABLE.
Return values  • None.
Notes  • None.

22.2.9 CE-ATA mode management functions

22.2.9.1 SDIO_CommandCompletionCmd

Function Name  void SDIO_CommandCompletionCmd ( FunctionalState
Function Name: SDIO_CEATAITCmd
Function Description: Enables or disables the CE-ATA interrupt.
Parameters:
- NewState: new state of CE-ATA interrupt. This parameter can be: ENABLE or DISABLE.
Return values:
- None.
Notes:
- None.

Function Name: SDIO_SendCEATACmd
Function Description: Sends CE-ATA command (CMD61).
Parameters:
- NewState: new state of CE-ATA command. This parameter can be: ENABLE or DISABLE.
Return values:
- None.
Notes:
- None.
Function Name: void SDIO_DMACmd (FunctionalState NewState)

Function Description: Enables or disables the SDIO DMA request.

Parameters:
- **NewState**: new state of the selected SDIO DMA request. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

22.2.11 Interrupt and flag management functions

22.2.11.1 SDIO_ITConfig

Function Name: void SDIO_ITConfig (uint32_t SDIO_IT, FunctionalState NewState)

Function Description: Enables or disables the SDIO interrupts.

Parameters:
- **SDIO_IT**: specifies the SDIO interrupt sources to be enabled or disabled. This parameter can be one or a combination of the following values:
  - **SDIO_IT_CCRCFAIL**: Command response received (CRC check failed) interrupt
  - **SDIO_IT_DCRCFAIL**: Data block sent/received (CRC check failed) interrupt
  - **SDIO_IT_CTIMEOUT**: Command response timeout interrupt
  - **SDIO_IT_DTIMEOUT**: Data timeout interrupt
  - **SDIO_IT_TXUNDERR**: Transmit FIFO underrun error interrupt
  - **SDIO_IT_RXOVERR**: Received FIFO overrun error interrupt
  - **SDIO_IT_CMDREND**: Command response received (CRC check passed) interrupt
  - **SDIO_IT_CMDSENT**: Command sent (no response required) interrupt
  - **SDIO_IT_DATAEND**: Data end (data counter, SDIDCOUNT, is zero) interrupt
  - **SDIO_IT_STBITERR**: Start bit not detected on all data signals in wide bus mode interrupt
  - **SDIO_IT_DBCKEND**: Data block sent/received (CRC check passed) interrupt
  - **SDIO_IT_CMDACT**: Command transfer in progress interrupt
  - **SDIO_IT_TXACT**: Data transmit in progress interrupt
  - **SDIO_IT_RXACT**: Data receive in progress interrupt
  - **SDIO_IT_TXFIFOHE**: Transmit FIFO Half Empty
interrupt

- **SDIO_IT_RXFIFOHF**: Receive FIFO Half Full interrupt
- **SDIO_IT_TXFIFOF**: Transmit FIFO full interrupt
- **SDIO_IT_RXFIFOF**: Receive FIFO full interrupt
- **SDIO_IT_TXFIFOE**: Transmit FIFO empty interrupt
- **SDIO_IT_RXFIFOE**: Receive FIFO empty interrupt
- **SDIO_IT_TXDAVL**: Data available in transmit FIFO interrupt
- **SDIO_IT_RXDAVL**: Data available in receive FIFO interrupt
- **SDIO_IT_SDIOIT**: SD I/O interrupt received interrupt
- **SDIO_IT_CEATAEND**: CE-ATA command completion signal received for CMD61 interrupt

- **NewState**: new state of the specified SDIO interrupts. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.

**Notes**
- None.

### 22.2.11.2 SDIO_GetFlagStatus

<table>
<thead>
<tr>
<th>Function Name</th>
<th>FlagStatus SDIO_GetFlagStatus ( uint32_t SDIO_FLAG)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function Description</strong></td>
<td>Checks whether the specified SDIO flag is set or not.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Parameters</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SDIO_FLAG</strong> : specifies the flag to check. This parameter can be one of the following values:</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_CCRCFAIL</strong>: Command response received (CRC check failed)</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_DCRCEFAIL</strong>: Data block sent/received (CRC check failed)</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_CTIMEOUT</strong>: Command response timeout</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_DTIMEOUT</strong>: Data timeout</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_TXUNDERR</strong>: Transmit FIFO underrun error</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_RXOVERR</strong>: Received FIFO overrun error</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_CMDREND</strong>: Command response received (CRC check passed)</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_CMDSNT</strong>: Command sent (no response required)</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_DATAEND</strong>: Data end (data counter, SDIDCOUNT, is zero)</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_STBITERR</strong>: Start bit not detected on all data signals in wide bus mode.</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_DBCKEND</strong>: Data block sent/received (CRC check passed)</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_CMDACT</strong>: Command transfer in progress</td>
</tr>
<tr>
<td>- <strong>SDIO_FLAG_TXACT</strong>: Data transmit in progress</td>
</tr>
</tbody>
</table>
Secure digital input/output interface (SDIO)

- **SDIO_FLAG_RXACT**: Data receive in progress
- **SDIO_FLAG_TXFIFOHE**: Transmit FIFO Half Empty
- **SDIO_FLAG_RXFIFOHF**: Receive FIFO Half Full
- **SDIO_FLAG_TXFIFOF**: Transmit FIFO full
- **SDIO_FLAG_RXFIFOE**: Receive FIFO empty
- **SDIO_FLAG_TXFIFOE**: Transmit FIFO empty
- **SDIO_FLAG_TXDAVL**: Data available in transmit FIFO
- **SDIO_FLAG_RXDAVL**: Data available in receive FIFO
- **SDIO_FLAG_SDIOIT**: SD I/O interrupt received
- **SDIO_FLAG_CEATAEND**: CE-ATA command completion signal received for CMD61

Return values
- The new state of SDIO_FLAG (SET or RESET).

Notes
- None.

### 22.2.11.3 SDIO_ClearFlag

**Function Name**
void SDIO_ClearFlag ( uint32_t SDIO_FLAG)

**Function Description**
Clears the SDIO's pending flags.

**Parameters**
- **SDIO_FLAG**: specifies the flag to clear. This parameter can be one or a combination of the following values:
  - **SDIO_FLAG_CCRCFAIL**: Command response received (CRC check failed)
  - **SDIO_FLAG_DCRFAIL**: Data block sent/received (CRC check failed)
  - **SDIO_FLAG_CTIMEOUT**: Command response timeout
  - **SDIO_FLAG_DTIMEOUT**: Data timeout
  - **SDIO_FLAG_TXUNDERR**: Transmit FIFO underrun error
  - **SDIO_FLAG_RXOVERR**: Received FIFO overrun error
  - **SDIO_FLAG_CMDREND**: Command response received (CRC check passed)
  - **SDIO_FLAG_CMDSENT**: Command sent (no response required)
  - **SDIO_FLAG_DATAEND**: Data end (data counter, SDIDCOUNT, is zero)
  - **SDIO_FLAG_STBITERR**: Start bit not detected on all data signals in wide bus mode
  - **SDIO_FLAG_DBCKEND**: Data block sent/received (CRC check passed)
  - **SDIO_FLAG_SDIOIT**: SD I/O interrupt received
  - **SDIO_FLAG_CEATAEND**: CE-ATA command completion signal received for CMD61

**Return values**
- None.
22.2.11.4 SDIO_GetITStatus

**Function Name**

ITStatus SDIO_GetITStatus ( uint32_t SDIO_IT)

**Function Description**

Checks whether the specified SDIO interrupt has occurred or not.

**Parameters**

- **SDIO_IT** : specifies the SDIO interrupt source to check. This parameter can be one of the following values:
  - **SDIO_IT_CCRCFAIL** : Command response received (CRC check failed) interrupt
  - **SDIO_IT_DCRCFAIL** : Data block sent/received (CRC check failed) interrupt
  - **SDIO_IT_CTIMEOUT** : Command response timeout interrupt
  - **SDIO_IT_DTIMEOUT** : Data timeout interrupt
  - **SDIO_IT_TXUNDERR** : Transmit FIFO underrun error interrupt
  - **SDIO_IT_RXOVERR** : Received FIFO overrun error interrupt
  - **SDIO_IT_CMDREND** : Command response received (CRC check passed) interrupt
  - **SDIO_IT_CMDSENT** : Command sent (no response required) interrupt
  - **SDIO_IT_DATAEND** : Data end (data counter, SDIDCOUNT, is zero) interrupt
  - **SDIO_IT_STBITERR** : Start bit not detected on all data signals in wide bus mode interrupt
  - **SDIO_IT_DBCKEND** : Data block sent/received (CRC check passed) interrupt
  - **SDIO_IT_CMDACT** : Command transfer in progress interrupt
  - **SDIO_IT_TXACT** : Data transmit in progress interrupt
  - **SDIO_IT_RXACT** : Data receive in progress interrupt
  - **SDIO_IT_TXFIFOHE** : Transmit FIFO Half Empty interrupt
  - **SDIO_IT_RXFIFOHE** : Receive FIFO Half Full interrupt
  - **SDIO_IT_TXFIFOF** : Transmit FIFO full interrupt
  - **SDIO_IT_RXFIFOF** : Receive FIFO full interrupt
  - **SDIO_IT_TXFIFOE** : Transmit FIFO empty interrupt
  - **SDIO_IT_RXFIFOE** : Receive FIFO empty interrupt
  - **SDIO_IT_TXDAVL** : Data available in transmit FIFO interrupt
  - **SDIO_IT_RXDAVL** : Data available in receive FIFO interrupt
  - **SDIO_IT_SDIOIT** : SD I/O interrupt received interrupt
  - **SDIO_IT_CEATAEND** : CE-ATA command completion
## 22.2.11.5 SDIO_ClearITPendingBit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void SDIO_ClearITPendingBit ( uint32_t SDIO_IT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Clears the SDIO's interrupt pending bits.</td>
</tr>
</tbody>
</table>
| Parameters | - **SDIO_IT** : specifies the interrupt pending bit to clear. This parameter can be one or a combination of the following values:  
  - **SDIO_IT_CCRCFAIL** : Command response received (CRC check failed) interrupt  
  - **SDIO_IT_DCRCFAIL** : Data block sent/received (CRC check failed) interrupt  
  - **SDIO_IT_CTIMEOUT** : Command response timeout interrupt  
  - **SDIO_IT_DTIMEOUT** : Data timeout interrupt  
  - **SDIO_IT_TXUNDERR** : Transmit FIFO underrun error interrupt  
  - **SDIO_IT_RXOVERR** : Received FIFO overrun error interrupt  
  - **SDIO_IT_CMDREND** : Command response received (CRC check passed) interrupt  
  - **SDIO_IT_CMDSENT** : Command sent (no response required) interrupt  
  - **SDIO_IT_DATAEND** : Data end (data counter, SDIO_DCOUNT, is zero) interrupt  
  - **SDIO_IT_STBITERR** : Start bit not detected on all data signals in wide bus mode interrupt  
  - **SDIO_IT_SDIOT** : SD I/O interrupt received interrupt  
  - **SDIO_IT_CEATAEND** : CE-ATA command completion signal received for CMD61 |
| Return values | - None. |
| Notes | - None. |
22.3 SDIO Firmware driver defines

22.3.1 SDIO Firmware driver defines

SDIO

SDIO_Bus_Wide

- #define: `SDIO_BusWide_1b((uint32_t)0x00000000)`

- #define: `SDIO_BusWide_4b((uint32_t)0x00000800)`

- #define: `SDIO_BusWide_8b((uint32_t)0x00001000)`

SDIO_Clock_Bypass

- #define: `SDIO_ClockBypass_Disable((uint32_t)0x00000000)`

- #define: `SDIO_ClockBypass_Enable((uint32_t)0x00000400)`

SDIO_Clock_Edge

- #define: `SDIO_ClockEdge_Rising((uint32_t)0x00000000)`

- #define: `SDIO_ClockEdge_Falling((uint32_t)0x00002000)`

SDIO_Clock_Power_Save

- #define: `SDIO_ClockPowerSave_Disable((uint32_t)0x00000000)`

- #define: `SDIO_ClockPowerSave_Enable((uint32_t)0x00000200)`

SDIO_CPSM_State

- #define: `SDIO_CPSM_Disable((uint32_t)0x00000000)`
- #define: `SDIO_CPSM_Enable((uint32_t)0x00000400)`

**SDIO_Data_Block_Size**

- #define: `SDIO_DataBlockSize_1b((uint32_t)0x00000000)`
- #define: `SDIO_DataBlockSize_2b((uint32_t)0x00000010)`
- #define: `SDIO_DataBlockSize_4b((uint32_t)0x00000020)`
- #define: `SDIO_DataBlockSize_8b((uint32_t)0x00000030)`
- #define: `SDIO_DataBlockSize_16b((uint32_t)0x00000040)`
- #define: `SDIO_DataBlockSize_32b((uint32_t)0x00000050)`
- #define: `SDIO_DataBlockSize_64b((uint32_t)0x00000060)`
- #define: `SDIO_DataBlockSize_128b((uint32_t)0x00000070)`
- #define: `SDIO_DataBlockSize_256b((uint32_t)0x00000080)`
- #define: `SDIO_DataBlockSize_512b((uint32_t)0x00000090)`
- #define: `SDIO_DataBlockSize_1024b((uint32_t)0x000000A0)`

- #define: `SDIO_DataBlockSize_2048b((uint32_t)0x000000B0)`

- #define: `SDIO_DataBlockSize_4096b((uint32_t)0x000000C0)`

- #define: `SDIO_DataBlockSize_8192b((uint32_t)0x000000D0)`

- #define: `SDIO_DataBlockSize_16384b((uint32_t)0x000000E0)`

**SDIO_DPSM_State**

- #define: `SDIO_DPSM_Disable((uint32_t)0x00000000)`

- #define: `SDIO_DPSM_Enable((uint32_t)0x00000001)`

**SDIO_Flags**

- #define: `SDIO_FLAG_CCRCFAIL((uint32_t)0x00000001)`

- #define: `SDIO_FLAG_DCRCFAIL((uint32_t)0x00000002)`

- #define: `SDIO_FLAG_CTIMEOUT((uint32_t)0x00000004)`

- #define: `SDIO_FLAG_DTIMEOUT((uint32_t)0x00000008)`

- #define: `SDIO_FLAG_TXUNDERR((uint32_t)0x00000010)`
• #define: `SDIO_FLAG_RXOVERR(uint32_t)0x00000020)

• #define: `SDIO_FLAG_CMDREND((uint32_t)0x00000040)

• #define: `SDIO_FLAG_CMDSENT((uint32_t)0x00000080)

• #define: `SDIO_FLAG_DATAEND((uint32_t)0x00000100)

• #define: `SDIO_FLAG_STBITERR((uint32_t)0x00000200)

• #define: `SDIO_FLAG_DBCKEND((uint32_t)0x00000400)

• #define: `SDIO_FLAG_CMDACT((uint32_t)0x00000800)

• #define: `SDIO_FLAG_TXACT((uint32_t)0x00001000)

• #define: `SDIO_FLAG_RXACT((uint32_t)0x00002000)

• #define: `SDIO_FLAG_TXFIFOHE((uint32_t)0x00004000)

• #define: `SDIO_FLAG_RXFIFOHF((uint32_t)0x00008000)

• #define: `SDIO_FLAG_TXFIFO((uint32_t)0x00010000)
- #define: `SDIO_FLAG_RXFIFO((uint32_t)0x00020000)`

- #define: `SDIO_FLAG_TXFIFOE((uint32_t)0x00040000)`

- #define: `SDIO_FLAG_RXFIFOE((uint32_t)0x00080000)`

- #define: `SDIO_FLAG_TXDAVL((uint32_t)0x00100000)`

- #define: `SDIO_FLAG_RXDAVL((uint32_t)0x00200000)`

- #define: `SDIO_FLAG_SDIOIT((uint32_t)0x00400000)`

- #define: `SDIO_FLAG_CEATAEND((uint32_t)0x00800000)`

**SDIO_Hardware_Flow_Control**
- #define: `SDIO_HardwareFlowControl_Disable((uint32_t)0x00000000)`

- #define: `SDIO_HardwareFlowControl_Enable((uint32_t)0x00004000)`

**SDIO_Interrupt_sources**
- #define: `SDIO_IT_CCRCFAIL((uint32_t)0x00000001)`

- #define: `SDIO_IT_DCRFAIL((uint32_t)0x00000002)`
- #define: `SDIO_IT_CTIMEOUT((uint32_t)0x00000004)`

- #define: `SDIO_IT_DTIMEOUT((uint32_t)0x00000008)`

- #define: `SDIO_IT_TXUNDERR((uint32_t)0x00000010)`

- #define: `SDIO_IT_RXOVERR((uint32_t)0x00000020)`

- #define: `SDIO_IT_CMDREND((uint32_t)0x00000040)`

- #define: `SDIO_IT_CMDSENT((uint32_t)0x00000080)`

- #define: `SDIO_IT_DATAEND((uint32_t)0x00000100)`

- #define: `SDIO_IT_STBITERR((uint32_t)0x00000200)`

- #define: `SDIO_IT_DBCKEND((uint32_t)0x00000400)`

- #define: `SDIO_IT_CMDACT((uint32_t)0x00000800)`

- #define: `SDIO_IT_TXACT((uint32_t)0x00001000)`

- #define: `SDIO_IT_RXACT((uint32_t)0x00002000)`
• #define: `SDIO_IT_TXFIFOHE((uint32_t)0x00004000)`

• #define: `SDIO_IT_RXFIFOHF((uint32_t)0x00008000)`

• #define: `SDIO_IT_TXFIFOF((uint32_t)0x00010000)`

• #define: `SDIO_IT_RXFIFOE((uint32_t)0x00020000)`

• #define: `SDIO_IT_TXFIFOE((uint32_t)0x00040000)`

• #define: `SDIO_IT_RXFIFOE((uint32_t)0x00080000)`

• #define: `SDIO_IT_TXDAVL((uint32_t)0x00100000)`

• #define: `SDIO_IT_RXDAVL((uint32_t)0x00200000)`

• #define: `SDIO_IT_SDIOIT((uint32_t)0x00400000)`

• #define: `SDIO_IT_CEATAEND((uint32_t)0x00800000)`

SDIO_Power_State
• #define: `SDIO_PowerState_OFF((uint32_t)0x00000000)`

• #define: `SDIO_PowerState_ON((uint32_t)0x00000003)`
**SDIO_Read_Wait_Mode**

- #define: `SDIO_ReadWaitMode_CLK((uint32_t)0x00000000)`

- #define: `SDIO_ReadWaitMode_DATA2((uint32_t)0x00000001)`

**SDIO_Response_Registers**

- #define: `SDIO_RESP1((uint32_t)0x00000000)`

- #define: `SDIO_RESP2((uint32_t)0x00000004)`

- #define: `SDIO_RESP3((uint32_t)0x00000008)`

- #define: `SDIO_RESP4((uint32_t)0x0000000C)`

**SDIO_Response_Type**

- #define: `SDIO_Response_No((uint32_t)0x00000000)`

- #define: `SDIO_Response_Short((uint32_t)0x00000040)`

- #define: `SDIO_Response_Long((uint32_t)0x000000C0)`

**SDIO_Transfer_Direction**

- #define: `SDIO_TransferDir_ToCard((uint32_t)0x00000000)`
**Secure digital input/output interface (SDIO)**

- **#define** `SDIO_TransferDir_ToSDIO((uint32_t)0x00000002)`

- **SDIO_Transfer_Type**
  - **#define** `SDIO_TransferMode_Block((uint32_t)0x00000000)`
  - **#define** `SDIO_TransferMode_Stream((uint32_t)0x00000004)`

- **SDIO_Wait_Interrupt_State**
  - **#define** `SDIO_Wait_No((uint32_t)0x00000000)`
    - SDIO No Wait, TimeOut is enabled
  - **#define** `SDIO_Wait_IT((uint32_t)0x00000100)`
    - SDIO Wait Interrupt Request
  - **#define** `SDIO_Wait_Pend((uint32_t)0x00000200)`
    - SDIO Wait End of transfer

### 22.4 SDIO Programming Example

The example below provides a typical configuration of the SDIO peripheral. For more examples about SDIO configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\SDIO.

```c
#define SDIO_TRANSFER_CLK_DIV ((uint8_t)0x00)

SDIO_InitTypeDef SDIO_InitStructure;
/* Enable SDIO Clock */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SDIO, ENABLE);

/* Configure the SDIO peripheral
   SDIO_CLK = SDIOCLK /(2 + SDIO_TRANSFER_CLK_DIV)
   On STM32F2xx devices, SDIOCLK is fixed to 48MHz */
SDIO_InitStructure.SDIO_ClockDiv = SDIO_TRANSFER_CLK_DIV;
SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;
SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;
SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable;
```

```c
#define SDIO_TRANSFER_CLK_DIV ((uint8_t)0x00)

SDIO_InitTypeDef SDIO_InitStructure;
/* Enable SDIO Clock */
RCC_APB2PeriphClockCmd(RCC_APB2Periph_SDIO, ENABLE);

/* Configure the SDIO peripheral
   SDIO_CLK = SDIOCLK /(2 + SDIO_TRANSFER_CLK_DIV)
   On STM32F2xx devices, SDIOCLK is fixed to 48MHz */
SDIO_InitStructure.SDIO_ClockDiv = SDIO_TRANSFER_CLK_DIV;
SDIO_InitStructure.SDIO_ClockEdge = SDIO_ClockEdge_Rising;
SDIO_InitStructure.SDIO_ClockBypass = SDIO_ClockBypass_Disable;
SDIO_InitStructure.SDIO_ClockPowerSave = SDIO_ClockPowerSave_Disable;
```
Secure digital input/output interface (SDIO)

```c
SDIO_InitStructure.SDIO_BusWide = SDIO_BusWide_1b;
SDIO_InitStructure.SDIO_HardwareFlowControl =
SDIO_HardwareFlowControl_Disable;
SDIO_Init(&SDIO_InitStructure);
```
23 Serial peripheral interface (SPI)

23.1 SPI Firmware driver registers structures

23.1.1 SPI_TypeDef

`SPI_TypeDef` is defined in the stm32f2xx.h file and contains the SPI/I2S registers definition.

Data Fields

- `__IO uint16_t CR1`
- `uint16_t RESERVED0`
- `__IO uint16_t CR2`
- `uint16_t RESERVED1`
- `__IO uint16_t SR`
- `uint16_t RESERVED2`
- `__IO uint16_t DR`
- `uint16_t RESERVED3`
- `__IO uint16_t CRCPR`
- `uint16_t RESERVED4`
- `__IO uint16_t RXCRCR`
- `uint16_t RESERVED5`
- `__IO uint16_t TXCRCR`
- `uint16_t RESERVED6`
- `__IO uint16_t I2SCFGR`
- `uint16_t RESERVED7`
- `__IO uint16_t I2SPR`
- `uint16_t RESERVED8`

Field Documentation

- `__IO uint16_t SPI_TypeDef::CR1`
  - SPI control register 1 (not used in I2S mode), Address offset: 0x00
- `uint16_t SPI_TypeDef::RESERVED0`
  - Reserved, 0x02
- `__IO uint16_t SPI_TypeDef::CR2`
  - SPI control register 2, Address offset: 0x04
- `uint16_t SPI_TypeDef::RESERVED1`
  - Reserved, 0x06
- `__IO uint16_t SPI_TypeDef::SR`
  - SPI status register, Address offset: 0x08
- `uint16_t SPI_TypeDef::RESERVED2`
  - Reserved, 0x0A
- `__IO uint16_t SPI_TypeDef::DR`
  - SPI data register, Address offset: 0x0C
- `uint16_t SPI_TypeDef::RESERVED3`
  - Reserved, 0x0E
- `__IO uint16_t SPI_TypeDef::CRCPR`
23.1.2 SPI_InitTypeDef

`SPI_InitTypeDef` is defined in the `stm32f2xx_spi.h` file and contains the SPI initialization parameters.

**Data Fields**

- `uint16_t SPI_Direction`
- `uint16_t SPI_Mode`
- `uint16_t SPI_DataSize`
- `uint16_t SPI_CPOL`
- `uint16_t SPI_CPHA`
- `uint16_t SPI_NSS`
- `uint16_t SPI_BaudRatePrescaler`
- `uint16_t SPI_FirstBit`
- `uint16_t SPI_CRCPolynomial`

**Field Documentation**

- `uint16_t SPI_InitTypeDef::SPI_Direction`
  - Specifies the SPI unidirectional or bidirectional data mode. This parameter can be a value of `SPI_data_direction`
- `uint16_t SPI_InitTypeDef::SPI_Mode`
  - Specifies the SPI operating mode. This parameter can be a value of `SPI_mode`
- `uint16_t SPI_InitTypeDef::SPI_DataSize`
  - Specifies the SPI data size. This parameter can be a value of `SPI_data_size`
- `uint16_t SPI_InitTypeDef::SPI_CPOL`
  - Specifies the serial clock steady state. This parameter can be a value of `SPI_Clock_Polarity`
- `uint16_t SPI_InitTypeDef::SPI_CPHA`
• Specifies the clock active edge for the bit capture. This parameter can be a value of
  SPI_Clock_Phase

• `uint16_t SPI_InitTypeDef::SPI_NSS`
  - Specifies whether the NSS signal is managed by hardware (NSS pin) or by
    software using the SSI bit. This parameter can be a value of
    SPI_Slave_Select_management

• `uint16_t SPI_InitTypeDef::SPI_BaudRatePrescaler`
  - Specifies the Baud Rate prescaler value which will be used to configure the
    transmit and receive SCK clock. This parameter can be a value of
    SPI_BaudRate_Prescaler

• `uint16_t SPI_InitTypeDef::SPI_FirstBit`
  - Specifies whether data transfers start from MSB or LSB bit. This parameter can
    be a value of SPI_MSB_LSB_transmission

• `uint16_t SPI_InitTypeDef::SPI_CRC Polynomial`
  - Specifies the polynomial used for the CRC calculation.

### 23.1.3 I2S_InitTypeDef

`I2S_InitTypeDef` is defined in the stm32f2xx_spi.h file and contains the I2S initialization
parameters.

**Data Fields**

- `uint16_t I2S_Mode`
- `uint16_t I2S_Standard`
- `uint16_t I2S_DataFormat`
- `uint16_t I2S_MCLKOutput`
- `uint32_t I2S_AudioFreq`
- `uint16_t I2S_CPOL`

**Field Documentation**

- `uint16_t I2S_InitTypeDef::I2S_Mode`
  - Specifies the I2S operating mode. This parameter can be a value of I2S_Mode
- `uint16_t I2S_InitTypeDef::I2S_Standard`
  - Specifies the standard used for the I2S communication. This parameter can be a
    value of I2S_Mode
- `uint16_t I2S_InitTypeDef::I2S_DataFormat`
  - Specifies the data format for the I2S communication. This parameter can be a
    value of I2S_Data_Format
- `uint16_t I2S_InitTypeDef::I2S_MCLKOutput`
  - Specifies whether the I2S MCLK output is enabled or not. This parameter can be a
    value of I2S_MCLK_Output
- `uint32_t I2S_InitTypeDef::I2S_AudioFreq`
  - Specifies the frequency selected for the I2S communication. This parameter can be a
    value of I2S_Audio_Frequency
- `uint16_t I2S_InitTypeDef::I2S_CPOL`
  - Specifies the idle state of the I2S clock. This parameter can be a value of
    I2S_Clock_Polarity
23.2 SPI Firmware driver API description

The following section lists the various functions of the SPI library.

23.2.1 How to use this driver

1. Enable peripheral clock using the following functions:
   - RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE) for SPI1
   - RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE) for SPI2
   - RCC_APB1PeriphResetCmd(RCC_APB1Periph_SPI3, ENABLE) for SPI3.

2. Enable SCK, MOSI, MISO and NSS GPIO clocks using:
   - RCC_AHB1PeriphClockCmd() function. In I2S mode, if an external clock source is used then the I2S CKIN pin GPIO clock should also be enabled.

3. Peripherals alternate function:
   - Connect the pin to the desired peripherals' Alternate Function (AF) using:
     - GPIO_PinAFConfig() function
     - Configure the desired pin in alternate function by:
       - GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF
     - Select the type, pull-up/pull-down and output speed via GPIO_PuPd, GPIO_OType and GPIO_Speed members
     - Call GPIO_Init() function. In I2S mode, if an external clock source is used then the I2S CKIN pin should be also configured in Alternate function Push-pull up mode.

4. Program the Polarity, Phase, First Data, Baud Rate Prescaler, Slave Management, Peripheral Mode and CRC Polynomial values using the SPI_Init() function. In I2S mode, program the Mode, Standard, Data Format, MCLK Output, Audio frequency and Polarity using I2S_Init() function. For I2S mode, make sure that either: I2S PLL is configured using the functions:
   - RCC_I2SCLKConfig(RCC_I2S2CLKSource_PLLI2S),
   - RCC_PLLI2SCmd(ENABLE) and RCC_GetFlagStatus(RCC_FLAG_PLLI2SRDY). Or that external clock source is configured using the function:
   - RCC_I2SCLKConfig(RCC_I2S2CLKSource_Ext) and after setting correctly the define constant I2S_EXTERNAL_CLOCK_VAL in the stm32f2xx_conf.h file.

5. Enable the NVIC and the corresponding interrupt using the function SPI_ITConfig() if you need to use interrupt mode.

6. When using the DMA mode:
   - Configure the DMA using DMA_Init() function
   - Activate the needed channel Request using SPI_I2S_DMACmd() function

7. Enable the SPI using the SPI_Cmd() function or enable the I2S using I2S_Cmd().

8. Enable the DMA using the DMA_Cmd() function when using DMA mode.

9. Optionally, you can enable/configure the following parameters without re-initialization (i.e there is no need to call again SPI_Init() function):
   - When bidirectional mode (SPI_Direction_1Line_Rx or SPI_Direction_1Line_Tx) is programmed as Data direction parameter using the SPI_Init() function it can be possible to switch between SPI_Direction_Tx or SPI_Direction_Rx using the SPI_BiDirectionalLineConfig() function.
   - When SPI_NSS_Soft is selected as Slave Select Management parameter using the SPI_Init() function it can be possible to manage the NSS internal signal using the SPI_NSSInternalSoftwareConfig() function.
   - Reconfigure the data size using the SPI_DataSizeConfig() function
   - Enable or disable the SS output using the SPI_SSOutputCmd() function

10. To use the CRC Hardware calculation feature refer to the Peripheral CRC hardware Calculation subsection.
This driver supports only the I2S clock scheme available in Silicon RevisionB and RevisionY.

In I2S mode: if an external clock is used as source clock for the I2S, then the define I2S_EXTERNAL_CLOCK_VAL in file stm32f2xx_conf.h should be enabled and set to the value of the source clock frequency (in Hz).

In SPI mode: To use the SPI TI mode, call the function SPI_TIModeCmd() just after calling the function SPI_Init().

23.2.2 Initialization and configuration

This section provides a set of functions allowing to initialize the SPI Direction, SPI Mode, SPI Data Size, SPI Polarity, SPI Phase, SPI NSS Management, SPI Baud Rate Prescaler, SPI First Bit and SPI CRC Polynomial.

The SPI_Init() function follows the SPI configuration procedures for Master mode and Slave mode (details for these procedures are available in reference manual (RM0033)).

- SPI_I2S_DelInit()
- SPI_Init()
- I2S_Init()
- SPI_StructInit()
- I2S_StructInit()
- SPI_Cmd()
- I2S_Cmd()
- SPI_DataSizeConfig()
- SPI_BiDirectionalLineConfig()
- SPI_NSSInternalSoftwareConfig()
- SPI_SSOutputCmd()
- SPI_TIModeCmd()

23.2.3 Data transfers

This section provides a set of functions allowing to manage the SPI data transfers. In reception, data are received and then stored into an internal Rx buffer while in transmission, data are first stored into an internal Tx buffer before being transmitted.

The read access of the SPI_DR register can be done using the SPI_I2S_ReceiveData() function and returns the Rx buffered value. Whereas a write access to the SPI_DR can be done using SPI_I2S_SendData() function and stores the written data into Tx buffer.

- SPI_I2S_ReceiveData()
- SPI_I2S_SendData()

23.2.4 Hardware CRC calculation
This section provides a set of functions allowing to manage the SPI CRC hardware calculation.

SPI communication using CRC is possible through the following procedure:

1. Program the Data direction, Polarity, Phase, First Data, Baud Rate Prescaler, Slave Management, Peripheral Mode and CRC Polynomial values using the SPI_Init() function.
2. Enable the CRC calculation using the SPI_CalculateCRC() function.
3. Enable the SPI using the SPI_Cmd() function.
4. Before writing the last data to the TX buffer, set the CRCNext bit using the SPI_TransmitCRC() function to indicate that after transmission of the last data, the CRC should be transmitted.
5. After transmitting the last data, the SPI transmits the CRC. The SPI_CR1_CRCNEXT bit is reset. The CRC is also received and compared against the SPI_RXCRCR value. If the value does not match, the SPI_FLAG_CRCERR flag is set and an interrupt can be generated when the SPI_I2S_IT_ERR interrupt is enabled.

It is recommended not to read the calculated CRC values during the communication.

When the SPI is in slave mode, be careful to enable CRC calculation only when the clock is stable, that is, when the clock is in the steady state. If not, a wrong CRC calculation may be done. In fact, the CRC is sensitive to the SCK slave input clock as soon as CRCEN is set, and this, whatever the value of the SPE bit.

With high bitrate frequencies, be careful when transmitting the CRC. As the number of used CPU cycles has to be as low as possible in the CRC transfer phase, it is forbidden to call software functions in the CRC transmission sequence to avoid errors in the last data and CRC reception. In fact, CRCNEXT bit has to be written before the end of the transmission/reception of the last data.

For high bit rate frequencies, it is advised to use the DMA mode to avoid the degradation of the SPI speed performance due to CPU accesses impacting the SPI bandwidth.

When the STM32F2xx is configured as slave and the NSS hardware mode is used, the NSS pin needs to be kept low between the data phase and the CRC phase.

When the SPI is configured in slave mode with the CRC feature enabled, CRC calculation takes place even if a high level is applied on the NSS pin. This may happen for example in case of a multi-slave environment where the communication master addresses slaves alternately.
Between a slave de-selection (high level on NSS) and a new slave selection (low level on NSS), the CRC value should be cleared on both master and slave sides in order to resynchronize the master and slave for their respective CRC calculation.

To clear the CRC, follow the procedure below:

1. Disable SPI using the SPI_Cmd() function
2. Disable the CRC calculation using the SPI_CalculateCRC() function.
3. Enable the CRC calculation using the SPI_CalculateCRC() function.
4. Enable SPI using the SPI_Cmd() function

The hardware CRC calculation functions are:

- SPI_CalculateCRC()
- SPI_TransmitCRC()
- SPI_GetCRC()
- SPI_GetCRC Polynomial()

### 23.2.5 DMA transfers management

- SPI_I2S_DMACmd()

### 23.2.6 Interrupt and flag management

This section provides a set of functions allowing to configure the SPI Interrupts sources and check or clear the flags or pending bits status.

The user should identify which mode will be used in his application to manage the communication: Polling mode, Interrupt mode or DMA mode.

**Polling Mode**

In Polling Mode, the SPI/I2S communication can be managed by 9 flags:

- SPI_I2S_FLAG_TXE : to indicate the status of the transmit buffer register
- SPI_I2S_FLAG_RXNE : to indicate the status of the receive buffer register
- SPI_I2S_FLAG_BSY : to indicate the state of the communication layer of the SPI.
- SPI_FLAG_CRCERR : to indicate if a CRC Calculation error occur
- SPI_FLAG_MODF : to indicate if a Mode Fault error occur
- SPI_I2S_FLAG_OVR : to indicate if an Overrun error occur
- I2S_FLAG_TIFRFE: to indicate a Frame Format error occurs.
- I2S_FLAG_UDR: to indicate an Underrun error occurs.
- I2S_FLAG_CHSIDE: to indicate Channel Side.

Do not use the BSY flag to handle each data transmission or reception. It is better to use the TXE and RXNE flags instead.

In this Mode it is advised to use the following functions:

- FlagStatus SPI_I2S_GetFlagStatus(SPI_TypeDef SPIx, uint16_t SPI_I2S_FLAG);
- void SPI_I2S_ClearFlag(SPI_TypeDef SPIx, uint16_t SPI_I2S_FLAG);
Interrupt Mode

In Interrupt Mode, the SPI communication can be managed by 3 interrupt sources and 7 pending bits:

- **Pending Bits:**
  - SPI_I2S_IT_TXE : to indicate the status of the transmit buffer register
  - SPI_I2S_IT_RXNE : to indicate the status of the receive buffer register
  - SPI_IT_CRCERR : to indicate if a CRC Calculation error occur (available in SPI mode only)
  - SPI_IT_MODF : to indicate if a Mode Fault error occur (available in SPI mode only)
  - SPI_I2S_IT_OVR : to indicate if an Overrun error occur
  - I2S_IT_UDR : to indicate an Underrun Error occurs (available in I2S mode only).
  - I2S_FLAG_TIFRFE : to indicate a Frame Format error occurs (available in I2S mode only).

- **Interrupt Source:**
  - SPI_I2S_IT_TXE: specifies the interrupt source for the Tx buffer empty interrupt.
  - SPI_I2S_IT_RXNE : specifies the interrupt source for the Rx buffer not empty interrupt.
  - SPI_I2S_IT_ERR : specifies the interrupt source for the errors interrupt. In this Mode it is advised to use the following functions:
    - void SPI_I2S_ITConfig(SPI_TypeDef SPIx, uint8_t SPI_I2S_IT, FunctionalState NewState);
    - ITStatus SPI_I2S_GetITStatus(SPI_TypeDef SPIx, uint8_t SPI_I2S_IT);
    - void SPI_I2S_ClearITPendingBit(SPI_TypeDef SPIx, uint8_t SPI_I2S_IT);

- **DMA Mode**
  - SPI_I2S_DMAReq_Tx: specifies the Tx buffer DMA transfer request
  - SPI_I2S_DMAReq_Rx: specifies the Rx buffer DMA transfer request
  - void SPI_I2S_DMACmd(SPI_TypeDef SPIx, uint16_t SPI_I2S_DMAReq, FunctionalState NewState);

The following functions can be used to manage the SPI flags and interrupts:

- **SPI_I2S_ITConfig()**
- **SPI_I2S_GetFlagStatus()**
- **SPI_I2S_ClearFlag()**
- **SPI_I2S_GetITStatus()**
- **SPI_I2S_ClearITPendingBit()**

23.2.7 Initialization and configuration functions

23.2.7.1 SPI_I2S_DeInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void SPI_I2S_DeInit (SPI_TypeDef * SPIx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Deinitialize the SPIx peripheral registers to their default reset values.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• SPIx : To select the SPIx/I2Sx peripheral, where x can be: 1, 2 or 3 in SPI mode or 2 or 3 in I2S mode.</td>
</tr>
</tbody>
</table>
### 23.2.7.2 SPI_Init

**Function Name**

```c
void SPI_Init ( SPI_TypeDef * SPIx, SPI_InitTypeDef * SPI_InitStruct)
```

**Function Description**

Initializes the SPIx peripheral according to the specified parameters in the SPI_InitStruct.

**Parameters**

- `SPIx`: where x can be 1, 2 or 3 to select the SPI peripheral.
- `SPI_InitStruct`: pointer to a SPI_InitTypeDef structure that contains the configuration information for the specified SPI peripheral.

**Return values**

- None.

**Notes**

- None.

### 23.2.7.3 I2S_Init

**Function Name**

```c
void I2S_Init ( SPI_TypeDef * SPIx, I2S_InitTypeDef * I2S_InitStruct)
```

**Function Description**

Initializes the SPIx peripheral according to the specified parameters in the I2S_InitStruct.

**Parameters**

- `SPIx`: where x can be 2 or 3 to select the SPI peripheral configured in I2S mode.
- `I2S_InitStruct`: pointer to an I2S_InitTypeDef structure that contains the configuration information for the specified SPI peripheral configured in I2S mode.

**Return values**

- None.

**Notes**

- The function calculates the optimal prescaler needed to obtain the most accurate audio frequency (depending on the I2S clock source, the PLL values and the product configuration). But in case the prescaler value is greater than 511, the default value (0x02) will be configured instead.
- if an external clock is used as source clock for the I2S, then the define I2S_EXTERNAL_CLOCK_VAL in file stm32f2xx_conf.h should be enabled and set to the value of
23.2.7.4   SPI_StructInit

Function Name  void SPI_StructInit (SPI_InitTypeDef * SPI_InitStruct)
Function Description  Fills each SPI_InitStruct member with its default value.
Parameters  • SPI_InitStruct : pointer to a SPI_InitTypeDef structure which will be initialized.
Return values  • None.
Notes  • None.

23.2.7.5   I2S_StructInit

Function Name  void I2S_StructInit (I2S_InitTypeDef * I2S_InitStruct)
Function Description  Fills each I2S_InitStruct member with its default value.
Parameters  • I2S_InitStruct : pointer to a I2S_InitTypeDef structure which will be initialized.
Return values  • None.
Notes  • None.

23.2.7.6   SPI_Cmd

Function Name  void SPI_Cmd (SPI_TypeDef * SPIx, FunctionalState NewState)
Function Description  Enables or disables the specified SPI peripheral.
Parameters  • SPIx : where x can be 1, 2 or 3 to select the SPI peripheral.
• NewState : new state of the SPIx peripheral. This parameter can be: ENABLE or DISABLE.
23.2.7.7  I2S_Cmd

Function Name: void I2S_Cmd ( SPI_TypeDef * SPIx, FunctionalState NewState)

Function Description: Enables or disables the specified SPI peripheral (in I2S mode).

Parameters:
- SPIx : where x can be 2 or 3 to select the SPI peripheral.
- NewState : new state of the SPIx peripheral. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

23.2.7.8  SPI_DataSizeConfig

Function Name: void SPI_DataSizeConfig ( SPI_TypeDef * SPIx, uint16_t SPI_DataSize)

Function Description: Configures the data size for the selected SPI.

Parameters:
- SPIx : where x can be 1, 2 or 3 to select the SPI peripheral.
- SPI_DataSize : specifies the SPI data size. This parameter can be one of the following values:
  - SPI_DataSize_16b : Set data frame format to 16bit
  - SPI_DataSize_8b : Set data frame format to 8bit

Return values: None.

Notes: None.

23.2.7.9  SPI_BiDirectionalLineConfig
### SPI_BiDirectionalLineConfig

**Function Name**

void SPI_BiDirectionalLineConfig (SPI_TypeDef *SPIx, uint16_t SPI_Direction)

**Function Description**

Selects the data transfer direction in bidirectional mode for the specified SPI.

**Parameters**

- **SPIx**: where x can be 1, 2 or 3 to select the SPI peripheral.
- **SPI_Direction**: specifies the data transfer direction in bidirectional mode. This parameter can be one of the following values:
  - **SPI_Direction_Tx**: Selects Tx transmission direction
  - **SPI_Direction_Rx**: Selects Rx receive direction

**Return values**

- None.

**Notes**

- None.

---

### SPI_NSSInternalSoftwareConfig

**Function Name**

void SPI_NSSInternalSoftwareConfig (SPI_TypeDef *SPIx, uint16_t SPI_NSSInternalSoft)

**Function Description**

Configures internally by software the NSS pin for the selected SPI.

**Parameters**

- **SPIx**: where x can be 1, 2 or 3 to select the SPI peripheral.
- **SPI_NSSInternalSoft**: specifies the SPI NSS internal state. This parameter can be one of the following values:
  - **SPI_NSSInternalSoft_Set**: Set NSS pin internally
  - **SPI_NSSInternalSoft_Reset**: Reset NSS pin internally

**Return values**

- None.

**Notes**

- None.

---

### SPI_SSOutputCmd

**Function Name**

void SPI_SSOutputCmd (SPI_TypeDef *SPIx, FunctionalState NewState)

**Function Description**

Enables or disables the SS output for the selected SPI.

**Parameters**

- **SPIx**: where x can be 1, 2 or 3 to select the SPI peripheral.
- **NewState**: new state of the SPIx SS output. This parameter can be: ENABLE or DISABLE.
23.2.7.12 SPI_TIModeCmd

Function Name: void SPI_TIModeCmd (SPI_TypeDef * SPIx, FunctionalState NewState)

Function Description: Enables or disables the SPIx/I2Sx DMA interface.

Parameters:
- SPIx: where x can be 1, 2 or 3
- NewState: new state of the selected SPI TI communication mode. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes:
- This function can be called only after the SPI_Init() function has been called.
- When TI mode is selected, the control bits SSM, SSI, CPOL and CPHA are not taken into consideration and are configured by hardware respectively to the TI mode requirements.

23.2.8 Data transfers functions

23.2.8.1 SPI_I2S_ReceiveData

Function Name: uint16_t SPI_I2S_ReceiveData (SPI_TypeDef * SPIx)

Function Description: Returns the most recent received data by the SPIx/I2Sx peripheral.

Parameters:
- SPIx: To select the SPIx/I2Sx peripheral, where x can be: 1, 2 or 3 in SPI mode or 2 or 3 in I2S mode.

Return values:
- The value of the received data.

Notes:
- None.
23.2.8.2 SPI_I2S_SendData

Function Name: void SPI_I2S_SendData (SPI_TypeDef * SPIx, uint16_t Data)
Function Description: Transmits a Data through the SPIx/I2Sx peripheral.
Parameters:
- SPIx: To select the SPIx/I2Sx peripheral, where x can be: 1, 2 or 3 in SPI mode or 2 or 3 in I2S mode.
- Data: Data to be transmitted.
Return values:
- None.
Notes:
- None.

23.2.9 Hardware CRC calculation functions

23.2.9.1 SPI_CalculateCRC

Function Name: void SPI_CalculateCRC (SPI_TypeDef * SPIx, FunctionalState NewState)
Function Description: Enables or disables the CRC value calculation of the transferred bytes.
Parameters:
- SPIx: where x can be 1, 2 or 3 to select the SPI peripheral.
- NewState: new state of the SPIx CRC value calculation. This parameter can be: ENABLE or DISABLE.
Return values:
- None.
Notes:
- None.

23.2.9.2 SPI_TransmitCRC

Function Name: void SPI_TransmitCRC (SPI_TypeDef * SPIx)
Function Description: Transmit the SPIx CRC value.
Parameters:
- SPIx: where x can be 1, 2 or 3 to select the SPI peripheral.
Return values:
- None.
Notes:
- None.
### 23.2.9.3 SPI_GetCRC

**Function Name**
```
uint16_t SPI_GetCRC ( SPI_TypeDef * SPIx, uint8_t SPI_CRC)
```

**Function Description**
Returns the transmit or the receive CRC register value for the specified SPI.

**Parameters**
- **SPIx**: where x can be 1, 2 or 3 to select the SPI peripheral.
- **SPI_CRC**: specifies the CRC register to be read. This parameter can be one of the following values:
  - **SPI_CRC_Tx**: Selects Tx CRC register
  - **SPI_CRC_Rx**: Selects Rx CRC register

**Return values**
- The selected CRC register value.

**Notes**
- None.

### 23.2.9.4 SPI_GetCRCPolynomial

**Function Name**
```
uint16_t SPI_GetCRCPolynomial ( SPI_TypeDef * SPIx)
```

**Function Description**
Returns the CRC Polynomial register value for the specified SPI.

**Parameters**
- **SPIx**: where x can be 1, 2 or 3 to select the SPI peripheral.

**Return values**
- The CRC Polynomial register value.

**Notes**
- None.

### 23.2.10 DMA transfers management function

#### 23.2.10.1 SPI_I2S_DMACmd

**Function Name**
```
void SPI_I2S_DMACmd ( SPI_TypeDef * SPIx, uint16_t SPI_I2S_DMAReq, FunctionalState NewState)
```

**Function Description**
Enables or disables the SPIx/I2Sx DMA interface.
23.2.11 Interrupt and flag management functions

23.2.11.1 SPI_I2S_ITConfig

Function Name: void SPI_I2S_ITConfig (SPI_TypeDef * SPIx, uint8_t SPI_I2S_IT, FunctionalState NewState)

Function Description: Enables or disables the specified SPI/I2S interrupts.

Parameters:
- **SPIx**: To select the SPIx/I2Sx peripheral, where x can be: 1, 2 or 3 in SPI mode or 2 or 3 in I2S mode.
- **SPI_I2S_IT**: specifies the SPI interrupt source to be enabled or disabled. This parameter can be one of the following values:
  - **SPI_I2S_IT_TXE**: Tx buffer empty interrupt mask
  - **SPI_I2S_IT_RXNE**: Rx buffer not empty interrupt mask
  - **SPI_I2S_IT_ERR**: Error interrupt mask
- **NewState**: new state of the specified SPI interrupt. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

23.2.11.2 SPI_I2S_GetFlagStatus

Function Name: FlagStatus SPI_I2S_GetFlagStatus (SPI_TypeDef * SPIx, uint16_t SPI_I2S_FLAG)


Function Description
Checks whether the specified SPIx/I2Sx flag is set or not.

Parameters
- **SPIx**: To select the SPIx/I2Sx peripheral, where x can be: 1, 2 or 3 in SPI mode or 2 or 3 in I2S mode.
- **SPI_I2S_FLAG**: specifies the SPI flag to check. This parameter can be one of the following values:
  - **SPI_I2S_FLAG_TXE**: Transmit buffer empty flag.
  - **SPI_I2S_FLAG_RXNE**: Receive buffer not empty flag.
  - **SPI_I2S_FLAG_BSY**: Busy flag.
  - **SPI_I2S_FLAG_OVR**: Overrun flag.
  - **SPI_FLAG_MODF**: Mode Fault flag.
  - **SPI_FLAG_CRCERR**: CRC Error flag.
  - **SPI_I2S_FLAG_TIFRFE**: Format Error.
  - **I2S_FLAG_UDR**: Underrun Error flag.
  - **I2S_FLAG_CHSIDE**: Channel Side flag.

Return values
- The new state of SPI_I2S_FLAG (SET or RESET).

Notes
- None.

### 23.2.11.3 SPI_I2S_ClearFlag

**Function Name**
void SPI_I2S_ClearFlag ( SPI_TypeDef * SPIx, uint16_t SPI_I2S_FLAG)

**Function Description**
Clears the SPIx CRC Error (CRCERR) flag.

**Parameters**
- **SPIx**: To select the SPIx/I2Sx peripheral, where x can be: 1, 2 or 3 in SPI mode or 2 or 3 in I2S mode.
- **SPI_I2S_FLAG**: specifies the SPI flag to clear. This function clears only CRCERR flag.
  - **SPI_FLAG_CRCERR**: CRC Error flag.

**Return values**
- None.

**Notes**
- OVR (OverRun error) flag is cleared by software sequence: a read operation to SPI_DR register (SPI_I2S_ReceiveData()) followed by a read operation to SPI_SR register (SPI_I2S_GetFlagStatus()).
- UDR (UnderRun error) flag is cleared by a read operation to SPI_SR register (SPI_I2S_GetFlagStatus()).
- MODF (Mode Fault) flag is cleared by software sequence: a read/write operation to SPI_SR register (SPI_I2S_GetFlagStatus()) followed by a write operation to SPI_CR1 register (SPI_Cmd() to enable the SPI).
Function Name | SPI_I2S_GetITStatus (SPI_TypeDef * SPIx, uint8_t SPI_I2S_IT)  
Function Description | Checks whether the specified SPIx/I2Sx interrupt has occurred or not.  
Parameters |  
* SPIx : To select the SPIx/I2Sx peripheral, where x can be: 1, 2 or 3 in SPI mode or 2 or 3 in I2S mode.  
* SPI_I2S_IT : specifies the SPI interrupt source to check. This parameter can be one of the following values:  
  - SPI_I2S_IT_TXE : Transmit buffer empty interrupt.  
  - SPI_I2S_IT_RXNE : Receive buffer not empty interrupt.  
  - SPI_I2S_IT_OVR : Overrun interrupt.  
  - SPI_IT_MODF : Mode Fault interrupt.  
  - SPI_IT_CRCERR : CRC Error interrupt.  
  - I2S_IT_UDR : Underrun interrupt.  
  - SPI_I2S_IT_TIFRFE : Format Error interrupt.  
Return values | The new state of SPI_I2S_IT (SET or RESET).  
Notes | None.

Function Name | SPI_I2S_ClearITPendingBit (SPI_TypeDef * SPIx, uint8_t SPI_I2S_IT)  
Function Description | Clears the SPIx CRC Error (CRCERR) interrupt pending bit.  
Parameters |  
* SPIx : To select the SPIx/I2Sx peripheral, where x can be: 1, 2 or 3 in SPI mode or 2 or 3 in I2S mode.  
* SPI_I2S_IT : specifies the SPI interrupt pending bit to clear. This function clears only CRCERR interrupt pending bit.  
  - SPI_IT_CRCERR : CRC Error interrupt.  
Return values | None.  
Notes |  
* OVR (OverRun Error) interrupt pending bit is cleared by software sequence: a read operation to SPI_DR register (SPI_I2S_ReceiveData()) followed by a read operation to SPI_SR register (SPI_I2S_GetITStatus()).  
* UDR (UnderRun Error) interrupt pending bit is cleared by a read operation to SPI_SR register (SPI_I2S_GetITStatus()).  
* MODF (Mode Fault) interrupt pending bit is cleared by software sequence: a read/write operation to SPI_SR register (SPI_I2S_GetITStatus()) followed by a write operation to
23.3 SPI Firmware driver defines

23.3.1 SPI Firmware driver defines

SPI

SPI_BaudRate_Prescaler

- #define: `SPI_BaudRatePrescaler_2((uint16_t)0x0000)`

- #define: `SPI_BaudRatePrescaler_4((uint16_t)0x0008)`

- #define: `SPI_BaudRatePrescaler_8((uint16_t)0x0010)`

- #define: `SPI_BaudRatePrescaler_16((uint16_t)0x0018)`

- #define: `SPI_BaudRatePrescaler_32((uint16_t)0x0020)`

- #define: `SPI_BaudRatePrescaler_64((uint16_t)0x0028)`

- #define: `SPI_BaudRatePrescaler_128((uint16_t)0x0030)`

- #define: `SPI_BaudRatePrescaler_256((uint16_t)0x0038)`

SPI_Clock_Phase

- #define: `SPI_CPHA_1Edge((uint16_t)0x0000)`
- #define: `SPI_CPHA_2Edge((uint16_t)0x0001)`

**SPI Clock Polarity**
- #define: `SPI_CPOL_Low((uint16_t)0x0000)`
- #define: `SPI_CPOL_High((uint16_t)0x0002)`

**SPI CRC Transmit/Receive**
- #define: `SPI_CRC_Tx((uint8_t)0x00)`
- #define: `SPI_CRC_Rx((uint8_t)0x01)`

**SPI Data Direction**
- #define: `SPI_Direction_2Lines_FullDuplex((uint16_t)0x0000)`
- #define: `SPI_Direction_2Lines_RxOnly((uint16_t)0x0400)`
- #define: `SPI_Direction_1Line_Rx((uint16_t)0x8000)`
- #define: `SPI_Direction_1Line_Tx((uint16_t)0xC000)`

**SPI Data Size**
- #define: `SPI_DataSize_16b((uint16_t)0x0800)`
- #define: `SPI_DataSize_8b((uint16_t)0x0000)`
SPI_direction_transmit_receive
- #define: SPI_Direction_Rx((uint16_t)0xBFFF)
- #define: SPI_Direction_Tx((uint16_t)0x4000)

SPI_I2S_Audio_Frequency
- #define: I2S_AudioFreq_192k((uint32_t)192000)
- #define: I2S_AudioFreq_96k((uint32_t)96000)
- #define: I2S_AudioFreq_48k((uint32_t)48000)
- #define: I2S_AudioFreq_44k((uint32_t)44100)
- #define: I2S_AudioFreq_32k((uint32_t)32000)
- #define: I2S_AudioFreq_22k((uint32_t)22050)
- #define: I2S_AudioFreq_16k((uint32_t)16000)
- #define: I2S_AudioFreq_11k((uint32_t)11025)
- #define: I2S_AudioFreq_8k((uint32_t)8000)
• #define: I2S_AudioFreq_Default((uint32_t)2)

SPI_I2S_Clock_Polarity
• #define: I2S_CPOL_Low((uint16_t)0x0000)

• #define: I2S_CPOL_High((uint16_t)0x0008)

SPI_I2S_Data_Format
• #define: I2S_DataFormat_16b((uint16_t)0x0000)

• #define: I2S_DataFormat_16bextended((uint16_t)0x0001)

• #define: I2S_DataFormat_24b((uint16_t)0x0003)

• #define: I2S_DataFormat_32b((uint16_t)0x0005)

SPI_I2S_DMA_transfer_requests
• #define: SPI_I2S_DMAReq_Tx((uint16_t)0x0002)

• #define: SPI_I2S_DMAReq_Rx((uint16_t)0x0001)

SPI_I2S_flags_definition
• #define: SPI_I2S_FLAG_RXNE((uint16_t)0x0001)

• #define: SPI_I2S_FLAG_TXE((uint16_t)0x0002)
- #define: \texttt{I2S\_FLAG\_CHSIDE((uint16\_t)0x0004)}

- #define: \texttt{I2S\_FLAG\_UDR((uint16\_t)0x0008)}

- #define: \texttt{SPI\_FLAG\_CRCERR((uint16\_t)0x0010)}

- #define: \texttt{SPI\_FLAG\_MODF((uint16\_t)0x0020)}

- #define: \texttt{SPI\_I2S\_FLAG\_OVR((uint16\_t)0x0040)}

- #define: \texttt{SPI\_I2S\_FLAG\_BSY((uint16\_t)0x0080)}

- #define: \texttt{SPI\_I2S\_FLAG\_TIIFRFE((uint16\_t)0x0100)}

\textit{SPI\_I2S\_interrupts\_definition}

- #define: \texttt{SPI\_I2S\_IT\_TXE((uint8\_t)0x71)}

- #define: \texttt{SPI\_I2S\_IT\_RXNE((uint8\_t)0x60)}

- #define: \texttt{SPI\_I2S\_IT\_ERR((uint8\_t)0x50)}

- #define: \texttt{I2S\_IT\_UDR((uint8\_t)0x53)}
- #define: `SPI_I2S_IT_TIFRFE((uint8_t)0x58)`

- #define: `SPI_I2S_IT_OVR((uint8_t)0x56)`

- #define: `SPI_IT_MODF((uint8_t)0x55)`

- #define: `SPI_IT_CRCERR((uint8_t)0x54)`

**SPI_I2S_Legacy**

- #define: `SPI_DMAReq_TxSPI_I2S_DMAReq_Tx`

- #define: `SPI_DMAReq_RxSPI_I2S_DMAReq_Rx`

- #define: `SPI_IT_TXE` `SPI_I2S_IT_TXE`

- #define: `SPI_IT_RXNE` `SPI_I2S_IT_RXNE`

- #define: `SPI_IT_ERR` `SPI_I2S_IT_ERR`

- #define: `SPI_IT_OVR` `SPI_I2S_IT_OVR`

- #define: `SPI_FLAG_RXNE` `SPI_I2S_FLAG_RXNE`

- #define: `SPI_FLAG_TXE` `SPI_I2S_FLAG_TXE`
- #define: SPI_FLAG_OVR SPI_I2S_FLAG_OVR

- #define: SPI_FLAG_BSY SPI_I2S_FLAG_BSY

- #define: SPI_DeInit SPI_I2S_DeInit

- #define: SPI_ITConfig SPI_I2S_ITConfig

- #define: SPI_DMACmd SPI_I2S_DMACmd

- #define: SPI_SendData SPI_I2S_SendData

- #define: SPI_ReceiveData SPI_I2S_ReceiveData

- #define: SPI_GetFlagStatus SPI_I2S_GetFlagStatus

- #define: SPI_ClearFlag SPI_I2S_ClearFlag

- #define: SPI_GetITStatus SPI_I2S_GetITStatus

- #define: SPI_ClearITPendingBit SPI_I2S_ClearITPendingBit

SPI_I2S_MCLK_Output
- #define: I2S_MCLKOutput_Enable((uint16_t)0x0200)
• #define: `I2S_MCLKOutput_Disable((uint16_t)0x0000)`

**SPI_I2S_Mode**
• #define: `I2S_Mode_SlaveTx((uint16_t)0x0000)`

• #define: `I2S_Mode_SlaveRx((uint16_t)0x0100)`

• #define: `I2S_Mode_MasterTx((uint16_t)0x0200)`

• #define: `I2S_Mode_MasterRx((uint16_t)0x0300)`

**SPI_I2S_Standard**
• #define: `I2S_Standard_Phillips((uint16_t)0x0000)`

• #define: `I2S_Standard_MSB((uint16_t)0x0010)`

• #define: `I2S_Standard_LSB((uint16_t)0x0020)`

• #define: `I2S_Standard_PCMShort((uint16_t)0x0030)`

• #define: `I2S_Standard_PCMLong((uint16_t)0x00B0)`

**SPI_mode**
• #define: `SPI_Mode_Master((uint16_t)0x0104)`
23.4 **SPI Programming Example**

The example below explains how to initialize the SPI, and associated resources, in Master mode with NSS managed by software and send continuously 8-bit data. For more examples about SPI configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\SPI\.

```c
/* Includes -----------------------------------------------*/
#include "stm32f2xx.h"

/* Private function prototypes -------------------------------*/
static void SPI_Config(void);

/* Private functions -----------------------------------------------*/
```
```c
/**
 * @brief  Main program
 * @param  None
 * @retval None
 */
int main(void)
{
    /* SPI1 configuration ---------------------------------------*/
    SPI_Config();

    while (1)
    {
        /* Wait till Transmit buffer is empty */
        while(SPI_I2S_GetFlagStatus(SPI1, SPI_I2S_FLAG_TXE) == RESET)
        {
            /* Send dummy data */
            SPI_I2S_SendData(SPI1, 0xA5);
        }
    }
}

/**
 * @brief  Configures the SPI1 Peripheral.
 * @param  None
 * @retval None
 */
static void SPI_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    SPI_InitTypeDef  SPI_InitStructure;

    /* Enable the SPI clock */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SPI1, ENABLE);

    /* Enable the GPIOA clock */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);

    /* Connect PA5 to SPI1_SCK */
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource5, GPIO_AF_SPI1);
    /* Connect PA6 to SPI1_MISO */
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource6, GPIO_AF_SPI1);
    /* Connect PA7 to SPI1_MOSI */
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource7, GPIO_AF_SPI1);

    /* Configure SPI1 pins as alternate function (No need to configure PA4 since NSS will be managed by software) */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
    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);

    /* SPI configuration **************************************/
```
SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;
SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_256;
SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
SPI_InitStructure.SPI_CRC Polynomial = 7;
SPI_Init(SPI1, &SPI_InitStructure);
SPI_Cmd(SPI1, ENABLE);

23.4.1 I2S Programming Example

The example below provides a typical configuration of the I2S peripheral. For more examples about I2S configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripherals Library package under Project\STM32F2xx_StdPeriph_Examples\I2S\.

I2S_InitTypeDef I2S_InitStructure;
/* Enable the SPI2/I2S2 peripheral clock */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);
/* I2S2 peripheral configuration */
I2S_InitStructure.I2S_AudioFreq = I2S_AudioFreq_48k;
I2S_InitStructure.I2S_Standard = I2S_Standard_Phillips;
I2S_InitStructure.I2S_DataFormat = I2S_DataFormat_16b;
I2S_InitStructure.I2S_CPOL = I2S_CPOL_Low;
I2S_InitStructure.I2S_Mode = I2S_Mode_MasterTx;
I2S_InitStructure.I2S_MCLKOutput = I2S_MCLKOutput_Enable;
/* Initialize the I2S2 peripheral with the structure above */
I2S_Init(SPI2, &I2S_InitStructure);
24 System configuration controller (SYSCFG)

24.1 SYSCFG Firmware driver registers structures

24.1.1 SYSCFG_TypeDef

SYSCFG_TypeDef is defined in the stm32f2xx.h file and contains the SYSCFG registers definition.

Data Fields

- __IO uint32_t MEMRMP
- __IO uint32_t PMC
- __IO uint32_t EXTICR
- uint32_t RESERVED
- __IO uint32_t CMPCR

Field Documentation

- __IO uint32_t SYSCFG_TypeDef::MEMRMP
  - SYSCFG memory remap register, Address offset: 0x00
- __IO uint32_t SYSCFG_TypeDef::PMC
  - SYSCFG peripheral mode configuration register, Address offset: 0x04
- __IO uint32_t SYSCFG_TypeDef::EXTICR[4]
  - SYSCFG external interrupt configuration registers, Address offset: 0x08-0x14
- uint32_t SYSCFG_TypeDef::RESERVED[2]
  - Reserved, 0x18-0x1C
- __IO uint32_t SYSCFG_TypeDef::CMPCR
  - SYSCFG Compensation cell control register, Address offset: 0x20

24.2 SYSCFG Firmware driver API description

The following section lists the various functions of the SYSCFG library.

How to use this driver

This driver provides functions for:

- Remapping the memory accessible in the code area using SYSCFG_MemoryRemapConfig()
- Manage the EXTI lines connection to the GPIOs using SYSCFG_EXTILineConfig()
- Select the ETHERNET media interface (RMII/RII) using SYSCFG_ETH_MediaInterfaceConfig()

SYSCFG APB clock must be enabled to get write access to SYSCFG registers, using RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

Functions
24.2.1 Functions

24.2.1.1 SYSCFG_DeInit

Function Name: void SYSCFG_DeInit ( void )
Function Description: Deinitializes the Alternate Functions (remap and EXTI configuration) registers to their default reset values.
Parameters:
- None.
Return values:
- None.
Notes:
- None.

24.2.1.2 SYSCFG_MemoryRemapConfig

Function Name: void SYSCFG_MemoryRemapConfig ( uint8_t SYSCFG_MemoryRemap)
Function Description: Changes the mapping of the specified pin.
Parameters:
- SYSCFG_Memory: selects the memory remapping. This parameter can be one of the following values:
  - SYSCFG_MemoryRemap_Flash: Main Flash memory mapped at 0x00000000
  - SYSCFG_MemoryRemap_SystemFlash: System Flash memory mapped at 0x00000000
  - SYSCFG_MemoryRemap_FSMC: FSMC (Bank1 (NOR/PSRAM 1 and 2) mapped at 0x00000000
  - SYSCFG_MemoryRemap_SRAM: Embedded SRAM (112kB) mapped at 0x00000000
Return values:
- None.
Notes:
- In remap mode, the FSMC addressing is fixed to the remap address area only (Bank1 NOR/PSRAM 1 and 2) and FSMC control registers are not accessible. The FSMC remap function must be disabled to allow addressing other memory devices through the FSMC and/or to access FSMC control registers.
24.2.1.3 SYSCFG_EXTILineConfig

Function Name: void SYSCFG_EXTILineConfig ( uint8_t EXTI_PortSourceGPIOx, uint8_t EXTI_PinSourceExtI )

Function Description: Selects the GPIO pin used as EXTI Line.

Parameters:
- EXTI_PortSourceGPIOx: selects the GPIO port to be used as source for EXTI lines where x can be (A..I).
- EXTI_PinSourceExtI: specifies the EXTI line to be configured. This parameter can be EXTI_PinSourceEx where x can be (0..15, except for EXTI_PortSourceGPIOI x can be (0..11).

Return values:
- None.

Notes:
- None.

24.2.1.4 SYSCFG_ETH_MediaInterfaceConfig

Function Name: void SYSCFG_ETH_MediaInterfaceConfig ( uint32_t SYSCFG_ETH_MediaInterface )

Function Description: Selects the ETHERNET media interface.

Parameters:
- SYSCFG_ETH_MediaInterface: specifies the Media Interface mode. This parameter can be one of the following values:
  - SYSCFG_ETH_MediaInterface_MII: MII mode selected
  - SYSCFG_ETH_MediaInterface_RMII: RMII mode selected

Return values:
- None.

Notes:
- None.
Function Name | void SYSCFG_CompensationCellCmd ( FunctionalState NewState)
---|---
Function Description | Enables or disables the I/O Compensation Cell.
Parameters | - NewState : new state of the I/O Compensation Cell. This parameter can be one of the following values:
  - ENABLE: I/O compensation cell enabled
  - DISABLE: I/O compensation cell power-down mode
Return values | - None.
Notes | - The I/O compensation cell can be used only when the device supply voltage ranges from 2.4 to 3.6 V.

**24.2.1.6 SYSCFG_GetCompensationCellStatus**

Function Name | FlagStatus SYSCFG_GetCompensationCellStatus ( void )
---|---
Function Description | Checks whether the I/O Compensation Cell ready flag is set or not.
Parameters | - None.
Return values | - The new state of the I/O Compensation Cell ready flag (SET or RESET)
Notes | - None.

**24.3 SYSCFG Firmware driver defines**

SYSCFG

*SYSCFG\_ETHERNET\_Media\_Interface*

- #define: `SYSCFG\_ETH\_MediaInterface\_MII((uint32\_t)0x00000000)`

- #define: `SYSCFG\_ETH\_MediaInterface\_RMII((uint32\_t)0x00000001)`

*SYSCFG\_EXTI\_Pin\_Sources*

- #define: `EXTI\_PinSource0((uint8\_t)0x00)`
• #define: `EXTI_PinSource1((uint8_t)0x01)`

• #define: `EXTI_PinSource2((uint8_t)0x02)`

• #define: `EXTI_PinSource3((uint8_t)0x03)`

• #define: `EXTI_PinSource4((uint8_t)0x04)`

• #define: `EXTI_PinSource5((uint8_t)0x05)`

• #define: `EXTI_PinSource6((uint8_t)0x06)`

• #define: `EXTI_PinSource7((uint8_t)0x07)`

• #define: `EXTI_PinSource8((uint8_t)0x08)`

• #define: `EXTI_PinSource9((uint8_t)0x09)`

• #define: `EXTI_PinSource10((uint8_t)0xA)`

• #define: `EXTI_PinSource11((uint8_t)0xB)`

• #define: `EXTI_PinSource12((uint8_t)0xC)`
- #define: \texttt{EXTI\_PinSource13}\((\text{uint8\_t})0x0D)\)

- #define: \texttt{EXTI\_PinSource14}\((\text{uint8\_t})0x0E)\)

- #define: \texttt{EXTI\_PinSource15}\((\text{uint8\_t})0x0F)\)

\texttt{SYSCFG\_EXTI\_Port\_Sources}

- #define: \texttt{EXTI\_PortSourceGPIOA}\((\text{uint8\_t})0x00)\)

- #define: \texttt{EXTI\_PortSourceGPIOB}\((\text{uint8\_t})0x01)\)

- #define: \texttt{EXTI\_PortSourceGPIOC}\((\text{uint8\_t})0x02)\)

- #define: \texttt{EXTI\_PortSourceGPIOD}\((\text{uint8\_t})0x03)\)

- #define: \texttt{EXTI\_PortSourceGPIOE}\((\text{uint8\_t})0x04)\)

- #define: \texttt{EXTI\_PortSourceGPIOF}\((\text{uint8\_t})0x05)\)

- #define: \texttt{EXTI\_PortSourceGPIOG}\((\text{uint8\_t})0x06)\)

- #define: \texttt{EXTI\_PortSourceGPIOH}\((\text{uint8\_t})0x07)\)

- #define: \texttt{EXTI\_PortSourceGPIOI}\((\text{uint8\_t})0x08)\)
SYSCFG_Memory_Remap_Config

- #define: SYSCFG_MemoryRemap_Flash((uint8_t)0x00)

- #define: SYSCFG_MemoryRemap_SystemFlash((uint8_t)0x01)

- #define: SYSCFG_MemoryRemap_FSMC((uint8_t)0x02)

- #define: SYSCFG_MemoryRemap_SRAM((uint8_t)0x03)
25 General-purpose timers (TIM)

25.1 TIM Firmware driver registers structures

25.1.1 TIM_TypeDef

TIM_TypeDef is defined in the stm32f2xx.h file and contains the TIM registers definition.

Data Fields

- __IO uint16_t CR1
- uint16_t RESERVED0
- __IO uint16_t CR2
- uint16_t RESERVED1
- __IO uint16_t SMCR
- uint16_t RESERVED2
- __IO uint16_t DIER
- uint16_t RESERVED3
- __IO uint16_t SR
- uint16_t RESERVED4
- __IO uint16_t EGR
- uint16_t RESERVED5
- __IO uint16_t CCMR1
- uint16_t RESERVED6
- __IO uint16_t CCMR2
- uint16_t RESERVED7
- __IO uint16_t CCER
- uint16_t RESERVED8
- __IO uint16_t CCR1
- __IO uint16_t CCR2
- __IO uint16_t CCR3
- __IO uint16_t CCR4
- __IO uint16_t BDTR
- __IO uint16_t CNT
- __IO uint16_t PSC
- __IO uint16_t ARR
- __IO uint16_t RCR
- __IO uint16_t RESERVED9
- __IO uint32_t CCR1
- __IO uint32_t CCR2
- __IO uint32_t CCR3
- __IO uint32_t CCR4
- __IO uint32_t BDTR
- __IO uint16_t RESERVED10
- __IO uint16_t DCR
- __IO uint16_t RESERVED11
- __IO uint16_t DMAR
- __IO uint16_t RESERVED12
- __IO uint16_t RESERVED13
- __IO uint16_t OR
- uint16_t RESERVED14

Field Documentation
- **__IO uint16_t TIM_TypeDef::CR1**
  - TIM control register 1, Address offset: 0x00
- **uint16_t TIM_TypeDef::RESERVED0**
  - Reserved, 0x02
- **__IO uint16_t TIM_TypeDef::CR2**
  - TIM control register 2, Address offset: 0x04
- **uint16_t TIM_TypeDef::RESERVED1**
  - Reserved, 0x06
- **__IO uint16_t TIM_TypeDef::SMCR**
  - TIM slave mode control register, Address offset: 0x08
- **uint16_t TIM_TypeDef::RESERVED2**
  - Reserved, 0x0A
- **__IO uint16_t TIM_TypeDef::DIER**
  - TIM DMA/interrupt enable register, Address offset: 0x0C
- **uint16_t TIM_TypeDef::RESERVED3**
  - Reserved, 0x0E
- **__IO uint16_t TIM_TypeDef::SR**
  - TIM status register, Address offset: 0x10
- **uint16_t TIM_TypeDef::RESERVED4**
  - Reserved, 0x12
- **__IO uint16_t TIM_TypeDef::EGR**
  - TIM event generation register, Address offset: 0x14
- **uint16_t TIM_TypeDef::RESERVED5**
  - Reserved, 0x16
- **__IO uint16_t TIM_TypeDef::CCMR1**
  - TIM capture/compare mode register 1, Address offset: 0x18
- **uint16_t TIM_TypeDef::RESERVED6**
  - Reserved, 0x1A
- **__IO uint16_t TIM_TypeDef::CCMR2**
  - TIM capture/compare mode register 2, Address offset: 0x1C
- **uint16_t TIM_TypeDef::RESERVED7**
  - Reserved, 0x1E
- **__IO uint16_t TIM_TypeDef::CCER**
  - TIM capture/compare enable register, Address offset: 0x20
- **uint16_t TIM_TypeDef::RESERVED8**
  - Reserved, 0x22
- **__IO uint32_t TIM_TypeDef::CNT**
  - TIM counter register, Address offset: 0x24
- **__IO uint16_t TIM_TypeDef::PSC**
  - TIM prescaler, Address offset: 0x28
- **uint16_t TIM_TypeDef::RESERVED9**
  - Reserved, 0x2A
- **__IO uint32_t TIM_TypeDef::ARR**
  - TIM auto-reload register, Address offset: 0x2C
- **__IO uint16_t TIM_TypeDef::RCR**
  - TIM repetition counter register, Address offset: 0x30
- **uint16_t TIM_TypeDef::RESERVED10**
  - Reserved, 0x32
- **__IO uint32_t TIM_TypeDef::CCR1**
  - TIM capture/compare register 1, Address offset: 0x34
- **__IO uint32_t TIM_TypeDef::CCR2**
  - TIM capture/compare register 2, Address offset: 0x38
General-purpose timers (TIM)

- __IO uint32_t TIM_TypeDef::CCR3
  - TIM capture/compare register 3, Address offset: 0x3C
- __IO uint32_t TIM_TypeDef::CCR4
  - TIM capture/compare register 4, Address offset: 0x40
- __IO uint16_t TIM_TypeDef::BDTR
  - TIM break and dead-time register, Address offset: 0x44
- uint16_t TIM_TypeDef::RESERVED11
  - Reserved, 0x46
- __IO uint16_t TIM_TypeDef::DCR
  - TIM DMA control register, Address offset: 0x48
- uint16_t TIM_TypeDef::RESERVED12
  - Reserved, 0x4A
- __IO uint16_t TIM_TypeDef::DMAR
  - TIM DMA address for full transfer, Address offset: 0x4C
- uint16_t TIM_TypeDef::RESERVED13
  - Reserved, 0x50
- __IO uint16_t TIM_TypeDef::OR
  - TIM option register, Address offset: 0x50
- uint16_t TIM_TypeDef::RESERVED14
  - Reserved, 0x52

25.1.2 TIM_TimeBaseInitTypeDef

TIM_TimeBaseInitTypeDef is defined in the stm32f2xx_tim.h file and contains the TIM time base initialization parameters.

Data Fields

- uint16_t TIM_Prescaler
- uint16_t TIM_CounterMode
- uint32_t TIM_Period
- uint16_t TIM_ClockDivision
- uint8_t TIM_RepetitionCounter

Field Documentation

- uint16_t TIM_TimeBaseInitTypeDef::TIM_Prescaler
  - Specifies the prescaler value used to divide the TIM clock. This parameter can be a number between 0x0000 and 0xFFFF
- uint16_t TIM_TimeBaseInitTypeDef::TIM_CounterMode
  - Specifies the counter mode. This parameter can be a value of TIM_Counter_Mode
- uint32_t TIM_TimeBaseInitTypeDef::TIM_Period
  - Specifies the period value to be loaded into the active Auto-Reload Register at the next update event. This parameter must be a number between 0x0000 and 0xFFFF.
- uint16_t TIM_TimeBaseInitTypeDef::TIM_ClockDivision
  - Specifies the clock division. This parameter can be a value of TIM_Clock_Division_CKD
- uint8_t TIM_TimeBaseInitTypeDef::TIM_RepetitionCounter
General-purpose timers (TIM)

Specifies the repetition counter value. Each time the RCR downcounter reaches zero, an update event is generated and counting restarts from the RCR value (N). This means in PWM mode that (N+1) corresponds to: the number of PWM periods in edge-aligned mode the number of half PWM period in center-aligned mode This parameter must be a number between 0x00 and 0xFF. This parameter is valid only for TIM1 and TIM8.

25.1.3 TIM_OCInitTypeDef

TIM_OCInitTypeDef is defined in the stm32f2xx_tim.h file and contains the TIM Output Compare initialization parameters.

Data Fields

- `uint16_t TIM_OCMode`
- `uint16_t TIM_OutputState`
- `uint16_t TIM_OutputNState`
- `uint32_t TIM_Pulse`
- `uint16_t TIM_OCPolarity`
- `uint16_t TIM_OCNPolarity`
- `uint16_t TIM_OCIdleState`
- `uint16_t TIM_OCNIdleState`

Field Documentation

- `uint16_t TIM_OCInitTypeDef::TIM_OCMode`:
  - Specifies the TIM mode. This parameter can be a value of `TIM_Output_Compare_and_PWM_modes`
- `uint16_t TIM_OCInitTypeDef::TIM_OutputState`:
  - Specifies the TIM Output Compare state. This parameter can be a value of `TIM_Output_Compare_State`
- `uint16_t TIM_OCInitTypeDef::TIM_OutputNState`:
  - Specifies the TIM complementary Output Compare state. This parameter can be a value of `TIM_Output_Compare_N_State`
- `uint32_t TIM_OCInitTypeDef::TIM_Pulse`:
  - Specifies the pulse value to be loaded into the Capture Compare Register. This parameter can be a number between 0x0000 and 0xFFFF
- `uint16_t TIM_OCInitTypeDef::TIM_OCPolarity`:
  - Specifies the output polarity. This parameter can be a value of `TIM_Output_Compare_Polarity`
- `uint16_t TIM_OCInitTypeDef::TIM_OCNPolarity`:
  - Specifies the complementary output polarity. This parameter can be a value of `TIM_Output_Compare_N_Polarity`
- `uint16_t TIM_OCInitTypeDef::TIM_OCIdleState`:
  - Specifies the TIM Output Compare pin state during Idle state. This parameter can be a value of `TIM_Output_Compare_Idle_State`
- `uint16_t TIM_OCInitTypeDef::TIM_OCNIdleState`:
  - Specifies the TIM Output Compare pin state during Idle state. This parameter can be a value of `TIM_Output_Compare_N_Idle_State`
25.1.4 TIM_ICInitTypeDef

TIM_ICInitTypeDef is defined in the stm32f2xx_tim.h file and contains the TIM Input Compare initialization parameters.

Data Fields

- `uint16_t TIM_Channel`
- `uint16_t TIM_ICPolarity`
- `uint16_t TIM_ICSelection`
- `uint16_t TIM_ICPrescaler`
- `uint16_t TIM_ICFilter`

Field Documentation

- `uint16_t TIM_ICInitTypeDef::TIM_Channel`  
  - Specifies the TIM channel. This parameter can be a value of TIM_Channel
- `uint16_t TIM_ICInitTypeDef::TIM_ICPolarity`  
  - Specifies the active edge of the input signal. This parameter can be a value of TIM_Input_Capture_Polarity
- `uint16_t TIM_ICInitTypeDef::TIM_ICSelection`  
  - Specifies the input. This parameter can be a value of TIM_Input_Capture_Selection
- `uint16_t TIM_ICInitTypeDef::TIM_ICPrescaler`  
  - Specifies the Input Capture Prescaler. This parameter can be a value of TIM_Input_Capture_Prescaler
- `uint16_t TIM_ICInitTypeDef::TIM_ICFilter`  
  - Specifies the input capture filter. This parameter can be a number between 0x0 and 0xF

25.1.5 TIM_BDTRInitTypeDef

TIM_BDTRInitTypeDef is defined in the stm32f2xx_tim.h file and contains the TIM Break feature initialization parameters.

Data Fields

- `uint16_t TIM_OSSRState`
- `uint16_t TIM_OSSIState`
- `uint16_t TIM_LOCKLevel`
- `uint16_t TIM_DeadTime`
- `uint16_t TIM_Break`
- `uint16_t TIM_BreakPolarity`
- `uint16_t TIM_AutomaticOutput`

Field Documentation

- `uint16_t TIM_BDTRInitTypeDef::TIM_OSSRState`
General-purpose timers (TIM)  

- Specifies the Off-State selection used in Run mode. This parameter can be a value of `TIM_OSSR_Off_State_Selection_for_Run_mode_state`
- `uint16_t TIM_BDTRInitTypeDef::TIM_OSSIState`
  - Specifies the Off-State used in Idle state. This parameter can be a value of `TIM_OSSI_Off_State_Selection_for_Idle_mode_state`
- `uint16_t TIM_BDTRInitTypeDef::TIM_LOCKLevel`
  - Specifies the LOCK level parameters. This parameter can be a value of `TIM_Lock_level`
- `uint16_t TIM_BDTRInitTypeDef::TIM_DeadTime`
  - Specifies the delay time between the switching-off and the switching-on of the outputs. This parameter can be a number between 0x00 and 0xFF
- `uint16_t TIM_BDTRInitTypeDef::TIM_Break`
  - Specifies whether the TIM Break input is enabled or not. This parameter can be a value of `TIM_Break_Input_enable_disable`
- `uint16_t TIM_BDTRInitTypeDef::TIM_BreakPolarity`
  - Specifies the TIM Break Input pin polarity. This parameter can be a value of `TIM_Break_Polarity`
- `uint16_t TIM_BDTRInitTypeDef::TIM_AutomaticOutput`
  - Specifies whether the TIM Automatic Output feature is enabled or not. This parameter can be a value of `TIM_AOE_Bit_Set_Reset`

25.2 TIM Firmware driver API description

The following section lists the various functions of the TIM library.

25.2.1 How to use this driver

This driver provides functions to configure and program the TIM of all STM32F2xx devices. These functions are split in 9 groups:

1. TIM TimeBase management: this group includes all needed functions to configure the TM Timebase unit:
   - Set/Get Prescaler - Set/Get Autoreload
   - Counter modes configuration
   - Set Clock division
   - Select the One Pulse mode
   - Update Request Configuration
   - Update Disable Configuration
   - Auto-Preload Configuration
   - Enable/Disable the counter
2. TIM Output Compare management: this group includes all needed functions to configure the Capture/Compare unit used in Output compare mode:
   - Configure each channel, independently, in Output Compare mode
   - Select the output compare modes
   - Select the Polarities of each channel
   - Set/Get the Capture/Compare register values
   - Select the Output Compare Fast mode
   - Select the Output Compare Forced mode
   - Output Compare-Preload Configuration
   - Clear Output Compare Reference
   - Select the OCREF Clear signal
   - Enable/Disable the Capture/Compare Channels
3. TIM Input Capture management: this group includes all needed functions to configure the Capture/Compare unit used in Input Capture mode:
   - Configure each channel in input capture mode
   - Configure Channel1/2 in PWM Input mode
   - Set the Input Capture Prescaler
   - Get the Capture/Compare values
4. Advanced-control timers (TIM1 and TIM8) specific features
   - Configures the Break input, dead time, Lock level, the OSSI, the OSSR State and the AOE (automatic output enable)
   - Enable/Disable the TIM peripheral Main Outputs
   - Select the Commutation event
   - Set/Reset the Capture Compare Preload Control bit
5. TIM interrupts, DMA and flags management
   - Enable/Disable interrupt sources
   - Get flags status
   - Clear flags/ Pending bits
   - Enable/Disable DMA requests
   - Configure DMA burst mode
   - Select Capture Compare DMA request
6. TIM clocks management: this group includes all needed functions to configure the clock controller unit:
   - Select internal/External clock
   - Select the external clock mode: ETR (Mode1/Mode2), TIx or ITRx
7. TIM synchronization management: this group includes all needed functions to configure the Synchronization unit:
   - Select Input Trigger
   - Select Output Trigger
   - Select Master Slave Mode
   - ETR Configuration when used as external trigger
8. TIM specific interface management, this group includes all needed functions to use the specific TIM interface:
   - Encoder Interface Configuration
   - Select Hall Sensor
9. TIM specific remapping management includes the Remapping configuration of specific timers

25.2.2 Output Compare management

This section explains how to use the TIM Driver in Output Compare Mode.

To use the Timer in Output Compare mode, the following steps are mandatory:

1. Enable TIM clock using RCC_APBxPeriphClockCmd(RCC_APBxPeriph_TIMx, ENABLE) function
2. Configure the TIM pins by configuring the corresponding GPIO pins. Configure the Time base unit as described in the first part of this driver, if needed, else the Timer will run with the default configuration:
   - Autoreload value = 0xFFFF
   - Prescaler value = 0x0000
   - Counter mode = Up counting
   - Clock Division = TIM_CKD_DIV1
3. Fill the TIM_OCInitStruct with the desired parameters including:
   - The TIM Output Compare mode: TIM_OCMode
   - TIM Output State: TIM_OutputState
   - TIM Pulse value: TIM_Pulse
   - TIM Output Compare Polarity : TIM_OCPolarity
4. Call TIM_OCxInit(TIMx, &TIM_OCInitStruct) to configure the desired channel with the corresponding configuration.

5. Call the TIM_Cmd(ENABLE) function to enable the TIM counter.

All other functions can be used separately to modify, if needed, a specific feature of the Timer.

In case of PWM mode, this function is mandatory: TIM_OCxPreloadConfig(TIMx, TIM_OCPreload_ENABLE);

If the corresponding interrupt or DMA request are needed, the user should:

1. Enable the NVIC (or the DMA) to use the TIM interrupts (or DMA requests).
2. Enable the corresponding interrupt (or DMA request) using the function
   TIM_ITConfig(TIMx, TIM_IT_CCx) (or TIM_DMA_Cmd(TIMx, TIM_DMA_CCx))

- TIM_OC1Init()
- TIM_OC2Init()
- TIM_OC3Init()
- TIM_OC4Init()
- TIM_OCStructInit()
- TIM_SelectOCxM()
- TIM_SetCompare1()
- TIM_SetCompare2()
- TIM_SetCompare3()
- TIM_SetCompare4()
- TIM_ForcedOC1Config()
- TIM_ForcedOC2Config()
- TIM_ForcedOC3Config()
- TIM_ForcedOC4Config()
- TIM_OC1PreloadConfig()
- TIM_OC2PreloadConfig()
- TIM_OC3PreloadConfig()
- TIM_OC4PreloadConfig()
- TIM_OC1FastConfig()
- TIM_OC2FastConfig()
- TIM_OC3FastConfig()
- TIM_OC4FastConfig()
- TIM_ClearOC1Ref()
- TIM_ClearOC2Ref()
- TIM_ClearOC3Ref()
- TIM_ClearOC4Ref()
- TIM_OC1PolarityConfig()
- TIM_OC1NPolarityConfig()
- TIM_OC2PolarityConfig()
- TIM_OC2NPolarityConfig()
- TIM_OC3PolarityConfig()
25.2.3 Input Capture management

To use the Timer in Input Capture mode, the following steps are mandatory:

1. Enable TIM clock using RCC_APBxPeriphClockCmd(RCC_APBxPeriph_TIMx, ENABLE) function
2. Configure the TIM pins by configuring the corresponding GPIO pins
3. Configure the Time base unit as described in the first part of this driver, if needed, else the Timer will run with the default configuration:
   - Autoreload value = 0xFFFF
   - Prescaler value = 0x0000 - Counter mode = Up counting
   - Clock Division = TIM_CKD_DIV1
4. Fill the TIM_ICInitStruct with the desired parameters including:
   - TIM Channel: TIM_Channel
   - TIM Input Capture polarity: TIM_ICPolarity
   - TIM Input Capture selection: TIM_ICSelection
   - TIM Input Capture Prescaler: TIM_ICPrescaler
   - TIM Input Capture filter value: TIM_ICFilter
5. Call TIM_ICInit(TIMx, &TIM_ICInitStruct) to configure the desired channel with the corresponding configuration and to measure only frequency or duty cycle of the input signal, or, call TIM_PWMIConfig(TIMx, &TIM_ICInitStruct) to configure the desired channels with the corresponding configuration and to measure the frequency and the duty cycle of the input signal.
6. Enable the NVIC or the DMA to read the measured frequency.
7. Enable the corresponding interrupt (or DMA request) to read the Captured value, using the function TIM_ITConfig(TIMx, TIM_IT_CCx) (or TIM_DMA_Cmd(TIMx, TIM_DMA_CCx))
8. Call the TIM_Cmd(ENABLE) function to enable the TIM counter.
9. Use TIM_GetCapturex(TIMx); to read the captured value.

All other functions can be used separately to modify, if needed, a specific feature of the Timer.

The Input Capture functions are the following:

- `TIM_ICInit()`
- `TIM_ICStructInit()`
- `TIM_PWMIConfig()`
- `TIM_GetCapture1()`
- `TIM_GetCapture2()`
- `TIM_GetCapture3()`
- `TIM_GetCapture4()`
- `TIM_SetIC1Prescaler()`
- `TIM_SetIC2Prescaler()`
- `TIM_SetIC3Prescaler()`
- `TIM_SetIC4Prescaler()`
25.2.4 Advanced-control timers (TIM1 and TIM8) specific features

**TIM Driver: how to use the Break feature**

After configuring the Timer channel(s) in the appropriate Output Compare mode:

1. Fill the TIM_BDTRInitStruct with the desired parameters for the Timer Break Polarity, dead time, Lock level, the OSSI/OSSR State and the AOE(automatic output enable).
2. Call TIM_BDTRConfig(TIMx, &TIM_BDTRInitStruct) to configure the Timer
3. Enable the Main Output using TIM_CtrlPWMOutputs(TIM1, ENABLE)
4. Once the break even occurs, the Timer's output signals are put in reset state or in a known state (according to the configuration made in TIM_BDTRConfig() function).

The following functions can be used to configure the advanced control timers:

- TIM_BDTRConfig()
- TIM_BDTRStructInit()
- TIM_CtrlPWMOutputs()
- TIM_SelectCOM()
- TIM_CCPreloadControl()

25.2.5 Interrupts DMA and flags management

- TIM_ITConfig()
- TIM_GenerateEvent()
- TIM_GetFlagStatus()
- TIM_ClearFlag()
- TIM_GetITStatus()
- TIM_ClearITPendingBit()
- TIM_DMAConfig()
- TIM_DMACmd()
- TIM_SelectCCDMA()

25.2.6 Clocks management

- TIM_InternalClockConfig()
- TIM_ITRxExternalClockConfig()
- TIM_TIxExternalClockConfig()
- TIM_ETRClockMode1Config()
- TIM_ETRClockMode2Config()

25.2.7 Synchronization management

Case of two/several Timers

1. Configure the Master Timers using the following functions:
   - void TIM_SelectOutputTrigger(TIM_TypeDef TIMx, uint16_t TIM_TRGOSource);
   - void TIM_SelectMasterSlaveMode(TIM_TypeDef TIMx, uint16_t TIM_MasterSlaveMode);
2. Configure the Slave Timers using the following functions:
void TIM_SelectInputTrigger(TIM_TypeDef TIMx, uint16_t TIM/InputTriggerSource);
void TIM_SelectSlaveMode(TIM_TypeDef TIMx, uint16_t TIM/SlaveMode);

Case of Timers and external trigger(ETR pin)
1. Configure the External trigger using the function: void TIM_ETRConfig(TIM_TypeDef TIMx, uint16_t TIM/ExtTRGPrescaler, uint16_t TIM/ExtTRGPolarity, uint16_t ExtTRGFilter);
2. Configure the Slave Timers using the following functions:
   - void TIM_SelectInputTrigger(TIM_TypeDef TIMx, uint16_t TIM/InputTriggerSource);
   - void TIM_SelectSlaveMode(TIM_TypeDef TIMx, uint16_t TIM/SlaveMode);

The following functions can be used to in synchronous mode:
- TIM/SelectInputTrigger()
- TIM/SelectOutputTrigger()
- TIM/SelectSlaveMode()
- TIM/SelectMasterSlaveMode()
- TIM/ETRConfig()

25.2.8 Specific functions

Specific interface management functions
- TIM/EncoderInterfaceConfig()
- TIM/SelectHallSensor()

Specific remapping management function
- TIM/RemapConfig()

25.2.9 TimeBase management functions

25.2.9.1 TIM_DeInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void TIM_DeInit ( TIM_TypeDef * TIMx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Deinitializes the TIMx peripheral registers to their default reset values.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• TIMx : where x can be 1 to 14 to select the TIM peripheral.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>

25.2.9.2 TIM_TimeBaseInit
Function Name: void TIM_TimeBaseInit (TIM_TypeDef * TIMx, TIM_TimeBaseInitTypeDef * TIM_TimeBaseInitStruct)

Function Description: Initializes the TIMx Time Base Unit peripheral according to the specified parameters in the TIM_TimeBaseInitStruct.

Parameters:
- TIMx: where x can be 1 to 14 to select the TIM peripheral.
- TIM_TimeBaseInitStruct: pointer to a TIM_TimeBaseInitTypeDef structure that contains the configuration information for the specified TIM peripheral.

Return values:
- None.

Notes:
- None.

25.2.9.3 TIM_TimeBaseStructInit

Function Name: void TIM_TimeBaseStructInit (TIM_TimeBaseInitStruct * TIM_TimeBaseInitStruct)

Function Description: Fills each TIM_TimeBaseInitStruct member with its default value.

Parameters:
- TIM_TimeBaseInitStruct: pointer to a TIM_TimeBaseInitStruct structure which will be initialized.

Return values:
- None.

Notes:
- None.

25.2.9.4 TIM_PrescalerConfig

Function Name: void TIM_PrescalerConfig (TIM_TypeDef * TIMx, uint16_t Prescaler, uint16_t TIM_PSCReloadMode)

Function Description: Configures the TIMx Prescaler.

Parameters:
- TIMx: where x can be 1 to 14 to select the TIM peripheral.
- Prescaler: specifies the Prescaler Register value
- TIM_PSCReloadMode: specifies the TIM Prescaler Reload mode This parameter can be one of the following values:
  - TIM_PSCReloadMode_Update: The Prescaler is loaded at the update event.
  - TIM_PSCReloadMode_Immediate: The Prescaler is loaded immediately.
25.2.9.5  **TIM.CounterModeConfig**

**Function Name**  
void TIM.CounterModeConfig ( TIM_TypeDef * TIMx, uint16_t TIM.CounterMode)

**Function Description**  
Specifies the TIMx Counter Mode to be used.

**Parameters**  
- **TIMx**: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- **TIM.CounterMode**: specifies the Counter Mode to be used
  This parameter can be one of the following values:
  - TIM.CounterMode.Up: TIM Up Counting Mode
  - TIM.CounterMode.Down: TIM Down Counting Mode
  - TIM.CounterMode.CenterAligned1: TIM Center Aligned Mode1
  - TIM.CounterMode.CenterAligned2: TIM Center Aligned Mode2
  - TIM.CounterMode.CenterAligned3: TIM Center Aligned Mode3

**Return values**  
- None.

**Notes**  
- None.

25.2.9.6  **TIM.SetCounter**

**Function Name**  
void TIM.SetCounter ( TIM_TypeDef * TIMx, uint32_t Counter)

**Function Description**  
Sets the TIMx Counter Register value.

**Parameters**  
- **TIMx**: where x can be 1 to 14 to select the TIM peripheral.
- **Counter**: specifies the Counter register new value.

**Return values**  
- None.

**Notes**  
- None.
### General-purpose timers (TIM)

#### 25.2.9.7  TIM_SetAutoreload

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void TIM_SetAutoreload (TIM_TypeDef * TIMx, uint32_t Autoreload)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Sets the TIMx Autoreload Register value.</td>
</tr>
</tbody>
</table>
| Parameters | • TIMx : where x can be 1 to 14 to select the TIM peripheral.  
• Autoreload : specifies the Autoreload register new value. |
| Return values | • None. |
| Notes | • None. |

#### 25.2.9.8  TIM_GetCounter

<table>
<thead>
<tr>
<th>Function Name</th>
<th>uint32_t TIM_GetCounter (TIM_TypeDef * TIMx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Gets the TIMx Counter value.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• TIMx : where x can be 1 to 14 to select the TIM peripheral.</td>
</tr>
<tr>
<td>Return values</td>
<td>• Counter Register value</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>

#### 25.2.9.9  TIM_GetPrescaler

<table>
<thead>
<tr>
<th>Function Name</th>
<th>uint16_t TIM_GetPrescaler (TIM_TypeDef * TIMx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Gets the TIMx Prescaler value.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• TIMx : where x can be 1 to 14 to select the TIM peripheral.</td>
</tr>
<tr>
<td>Return values</td>
<td>• Prescaler Register value.</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>
25.2.9.10 TIM_UpdateDisableConfig

Function Name
void TIM_UpdateDisableConfig (TIM_TypeDef * TIMx, FunctionalState NewState)

Function Description
Enables or Disables the TIMx Update event.

Parameters
- **TIMx**: where x can be 1 to 14 to select the TIM peripheral.
- **NewState**: new state of the TIMx UDIS bit. This parameter can be: ENABLE or DISABLE.

Return values
- None.

Notes
- None.

25.2.9.11 TIM_UpdateRequestConfig

Function Name
void TIM_UpdateRequestConfig (TIM_TypeDef * TIMx, uint16_t TIM_UpdateSource)

Function Description
Configures the TIMx Update Request Interrupt source.

Parameters
- **TIMx**: where x can be 1 to 14 to select the TIM peripheral.
- **TIM_UpdateSource**: specifies the Update source. This parameter can be one of the following values:
  - **TIM_UpdateSource_REGULAR**: Source of update is the counter overflow/underflow or the setting of UG bit, or an update generation through the slave mode controller.
  - **TIM_UpdateSource_GLOBAL**: Source of update is counter overflow/underflow.

Return values
- None.

Notes
- None.

25.2.9.12 TIM_ARRPreloadConfig

Function Name
void TIM_ARRPreloadConfig (TIM_TypeDef * TIMx, FunctionalState NewState)

Function Description
Enables or disables TIMx peripheral Preload register on ARR.
### 25.2.9.13 TIM_SelectOnePulseMode

**Function Name**

```c
void TIM_SelectOnePulseMode ( TIM_TypeDef * TIMx, uint16_t TIM_OPMode)
```

**Function Description**

Selects the TIMx's One Pulse Mode.

**Parameters**

- **TIMx** : where x can be 1 to 14 to select the TIM peripheral.
- **TIM_OPMode** : specifies the OPM Mode to be used. This parameter can be one of the following values:
  - `TIM_OPMode_Single`
  - `TIM_OPMode_Repetitive`

**Return values**

- None.

**Notes**

- None.

---

### 25.2.9.14 TIM_SetClockDivision

**Function Name**

```c
void TIM_SetClockDivision ( TIM_TypeDef * TIMx, uint16_t TIM_CKD)
```

**Function Description**

Sets the TIMx Clock Division value.

**Parameters**

- **TIMx** : where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.
- **TIM_CKD** : specifies the clock division value. This parameter can be one of the following value:
  - `TIM_CKD_DIV1` : TDTS = Tck_tim
  - `TIM_CKD_DIV2` : TDTS = 2*Tck_tim
  - `TIM_CKD_DIV4` : TDTS = 4*Tck_tim

**Return values**

- None.

**Notes**

- None.
25.2.9.15 TIM_Cmd

Function Name: void TIM_Cmd (TIM_TypeDef * TIMx, FunctionalState NewState)

Function Description: Enables or disables the specified TIM peripheral.

Parameters:
- TIMx: where x can be 1 to 14 to select the TIMx peripheral.
- NewState: new state of the TIMx peripheral. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

25.2.10 Output Compare management functions

25.2.10.1 TIM_OC1Init

Function Name: void TIM_OC1Init (TIM_TypeDef * TIMx, TIM_OCInitTypeDef * TIM_OCInitStruct)

Function Description: Initializes the TIMx Channel1 according to the specified parameters in the TIM_OCInitStruct.

Parameters:
- TIMx: where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.
- TIM_OCInitStruct: pointer to a TIM_OCInitStruct structure that contains the configuration information for the specified TIM peripheral.

Return values: None.

Notes: None.

25.2.10.2 TIM_OC2Init

Function Name: void TIM_OC2Init (TIM_TypeDef * TIMx, TIM_OCInitTypeDef *
25.2.10.3 TIM_OC3Init

Function Name void TIM_OC3Init (TIM_TypeDef * TIMx, TIM_OCInitTypeDef * TIM_OCInitStruct)

Function Description Initializes the TIMx Channel3 according to the specified parameters in the TIM_OCInitStruct.

Parameters
- TIMx : where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_OCInitStruct : pointer to a TIM_OCInitTypeDef structure that contains the configuration information for the specified TIM peripheral.

Return values
- None.

Notes
- None.

25.2.10.4 TIM_OC4Init

Function Name void TIM_OC4Init (TIM_TypeDef * TIMx, TIM_OCInitTypeDef * TIM_OCInitStruct)

Function Description Initializes the TIMx Channel4 according to the specified parameters in the TIM_OCInitStruct.

Parameters
- TIMx : where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_OCInitStruct : pointer to a TIM_OCInitTypeDef structure that contains the configuration information for the specified TIM peripheral.

Return values
- None.

Notes
- None.
25.2.10.5  TIM_OCStructInit

Function Name  
void TIM_OCStructInit (TIM_OCInitTypeDef *TIM_OCInitStruct)

Function Description  
Fills each TIM_OCInitStruct member with its default value.

Parameters  
- TIM_OCInitStruct : pointer to a TIM_OCInitTypeDef structure which will be initialized.

Return values  
- None.

Notes  
- None.

25.2.10.6  TIM_SelectOCxM

Function Name  
void TIM_SelectOCxM (TIM_TypeDef *TIMx, uint16_t TIM_Channel, uint16_t TIM_OCMode)

Function Description  
Selects the TIM Output Compare Mode.

Parameters  
- TIMx : where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.
- TIM_Channel : specifies the TIM Channel This parameter can be one of the following values:
  - TIM_Channel_1 : TIM Channel 1
  - TIM_Channel_2 : TIM Channel 2
  - TIM_Channel_3 : TIM Channel 3
  - TIM_Channel_4 : TIM Channel 4
- TIM_OCMode : specifies the TIM Output Compare Mode. This parameter can be one of the following values:
  - TIM_OCMode_Timing :
  - TIM_OCMode_Active :
  - TIM_OCMode_Toggle :
  - TIM_OCMode_PWM1 :
  - TIM_OCMode_PWM2 :
  - TIM_ForcedAction_Active :
  - TIM_ForcedAction_InActive :
### 25.2.10.7 TIM_SetCompare1

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void TIM_SetCompare1 (TIM_TypeDef *, uint32_t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Sets the TIMx Capture Compare1 Register value.</td>
</tr>
</tbody>
</table>
| Parameters             | • TIMx : where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.  
                        | • Compare1 : specifies the Capture Compare1 register new value. |
| Return values          | None.                                         |
| Notes                  | None.                                         |

### 25.2.10.8 TIM_SetCompare2

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void TIM_SetCompare2 (TIM_TypeDef *, uint32_t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Sets the TIMx Capture Compare2 Register value.</td>
</tr>
</tbody>
</table>
| Parameters             | • TIMx : where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.  
                        | • Compare2 : specifies the Capture Compare2 register new value. |
| Return values          | None.                                         |
| Notes                  | None.                                         |
25.2.10.9 TIM_SetCompare3

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void TIM_SetCompare3 (TIM_TypeDef * TIMx, uint32_t Compare3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Sets the TIMx Capture Compare3 Register value.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• TIMx : where x can be 1, 2, 3, 4, 5 or 8 to select the TIM</td>
</tr>
<tr>
<td></td>
<td>peripheral.</td>
</tr>
<tr>
<td></td>
<td>• Compare3 : specifies the Capture Compare3 register new</td>
</tr>
<tr>
<td></td>
<td>value.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>

25.2.10.10 TIM_SetCompare4

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void TIM_SetCompare4 (TIM_TypeDef * TIMx, uint32_t Compare4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Sets the TIMx Capture Compare4 Register value.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• TIMx : where x can be 1, 2, 3, 4, 5 or 8 to select the TIM</td>
</tr>
<tr>
<td></td>
<td>peripheral.</td>
</tr>
<tr>
<td></td>
<td>• Compare4 : specifies the Capture Compare4 register new</td>
</tr>
<tr>
<td></td>
<td>value.</td>
</tr>
<tr>
<td>Return values</td>
<td>• None.</td>
</tr>
<tr>
<td>Notes</td>
<td>• None.</td>
</tr>
</tbody>
</table>

25.2.10.11 TIM_ForcedOC1Config

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void TIM_ForcedOC1Config (TIM_TypeDef * TIMx, uint16_t TIM_ForcedAction)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Forces the TIMx output 1 waveform to active or inactive level.</td>
</tr>
<tr>
<td>Parameters</td>
<td>• TIMx : where x can be 1 to 14 except 6 and 7, to select the TIM</td>
</tr>
<tr>
<td></td>
<td>peripheral.</td>
</tr>
<tr>
<td></td>
<td>• TIM_ForcedAction : specifies the forced Action to be set to the</td>
</tr>
<tr>
<td></td>
<td>output waveform. This parameter can be one of the</td>
</tr>
</tbody>
</table>
25.2.10.12 TIM_ForcedOC2Config

Function Name: void TIM_ForcedOC2Config (TIM_TypeDef * TIMx, uint16_t TIM_ForcedAction)

Function Description: Forces the TIMx output 2 waveform to active or inactive level.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
- TIM_ForcedAction: specifies the forced Action to be set to the output waveform. This parameter can be one of the following values:
  - TIM_ForcedAction_Active: Force active level on OC2REF
  - TIM_ForcedAction_InActive: Force inactive level on OC2REF.

Return values: None.
Notes: None.

25.2.10.13 TIM_ForcedOC3Config

Function Name: void TIM_ForcedOC3Config (TIM_TypeDef * TIMx, uint16_t TIM_ForcedAction)

Function Description: Forces the TIMx output 3 waveform to active or inactive level.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_ForcedAction: specifies the forced Action to be set to the output waveform. This parameter can be one of the following values:
  - TIM_ForcedAction_Active: Force active level on OC2REF
  - TIM_ForcedAction_InActive: Force inactive level on OC2REF.

Return values: None.
Notes: None.
25.2.10.14 TIM_ForcedOC4Config

Function Name: void TIM_ForcedOC4Config (TIM_TypeDef * TIMx, uint16_t TIM_ForcedAction)

Function Description: Forces the TIMx output 4 waveform to active or inactive level.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_ForcedAction: specifies the forced Action to be set to the output waveform. This parameter can be one of the following values:
  - TIM_ForcedAction_Active: Force active level on OC4REF
  - TIM_ForcedAction_InActive: Force inactive level on OC4REF.

Return values:
- None.

Notes:
- None.

25.2.10.15 TIM_OC1PreloadConfig

Function Name: void TIM_OC1PreloadConfig (TIM_TypeDef * TIMx, uint16_t TIM_OCPreload)

Function Description: Enables or disables the TIMx peripheral Preload register on CCR1.

Parameters:
- TIMx: where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.
- TIM_OCPreload: new state of the TIMx peripheral Preload register. This parameter can be one of the following values:
  - TIM_OCPreload_Enable:
  - TIM_OCPreload_Disable:
25.2.10.16 TIM_OC2PreloadConfig

Function Name
void TIM_OC2PreloadConfig ( TIM_TypeDef * TIMx, uint16_t TIM_OCPreload)

Function Description
Enables or disables the TIMx peripheral Preload register on CCR2.

Parameters
- TIMx : where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
- TIM_OCPreload : new state of the TIMx peripheral Preload register This parameter can be one of the following values:
  - TIM_OCPreload_Enable :
  - TIM_OCPreload_Disable :

Return values
- None.

Notes
- None.

25.2.10.17 TIM_OC3PreloadConfig

Function Name
void TIM_OC3PreloadConfig ( TIM_TypeDef * TIMx, uint16_t TIM_OCPreload)

Function Description
Enables or disables the TIMx peripheral Preload register on CCR3.

Parameters
- TIMx : where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_OCPreload : new state of the TIMx peripheral Preload register This parameter can be one of the following values:
  - TIM_OCPreload_Enable :
  - TIM_OCPreload_Disable :

Return values
- None.

Notes
- None.
25.2.10.18 TIM_OC4PreloadConfig

Function Name: void TIM_OC4PreloadConfig (TIM_TypeDef * TIMx, uint16_t TIM_OCPreload)

Function Description: Enables or disables the TIMx peripheral Preload register on CCR4.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_OCPreload: new state of the TIMx peripheral Preload register. This parameter can be one of the following values:
  - TIM_OCPreload_Enable
  - TIM_OCPreload_Disable

Return values:
- None.

Notes:
- None.

25.2.10.19 TIM_OC1FastConfig

Function Name: void TIM_OC1FastConfig (TIM_TypeDef * TIMx, uint16_t TIM_OCFast)

Function Description: Configures the TIMx Output Compare 1 Fast feature.

Parameters:
- TIMx: where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.
- TIM_OCFast: new state of the Output Compare Fast Enable Bit. This parameter can be one of the following values:
  - TIM_OCFast_Enable: TIM output compare fast enable
  - TIM_OCFast_Disable: TIM output compare fast disable

Return values:
- None.

Notes:
- None.

25.2.10.20 TIM_OC2FastConfig
Function Name: void TIM_OC2FastConfig (TIM_TypeDef * TIMx, uint16_t TIM_OCFast)

Function Description: Configures the TIMx Output Compare 2 Fast feature.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
- TIM_OCFast: new state of the Output Compare Fast Enable Bit. This parameter can be one of the following values:
  - TIM_OCFast_Enable: TIM output compare fast enable
  - TIM_OCFast_Disable: TIM output compare fast disable

Return values: None.

Notes: None.

25.2.10.21 TIM_OC3FastConfig

Function Name: void TIM_OC3FastConfig (TIM_TypeDef * TIMx, uint16_t TIM_OCFast)

Function Description: Configures the TIMx Output Compare 3 Fast feature.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_OCFast: new state of the Output Compare Fast Enable Bit. This parameter can be one of the following values:
  - TIM_OCFast_Enable: TIM output compare fast enable
  - TIM_OCFast_Disable: TIM output compare fast disable

Return values: None.

Notes: None.

25.2.10.22 TIM_OC4FastConfig

Function Name: void TIM_OC4FastConfig (TIM_TypeDef * TIMx, uint16_t TIM_OCFast)

Function Description: Configures the TIMx Output Compare 4 Fast feature.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
25.2.10.23 TIM_ClearOC1Ref

Function Name: void TIM_ClearOC1Ref (TIM_TypeDef * TIMx, uint16_t TIM_OCClear)
Function Description: Clears or safeguards the OCREF1 signal on an external event.
Parameters:
- TIMx: where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.
- TIM_OCClear: new state of the Output Compare Clear Enable Bit. This parameter can be one of the following values:
  - TIM_OCClear_Enable: TIM Output clear enable
  - TIM_OCClear_Disable: TIM Output clear disable

Return values
- None.

Notes
- None.

25.2.10.24 TIM_ClearOC2Ref

Function Name: void TIM_ClearOC2Ref (TIM_TypeDef * TIMx, uint16_t TIM_OCClear)
Function Description: Clears or safeguards the OCREF2 signal on an external event.
Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
- TIM_OCClear: new state of the Output Compare Clear Enable Bit. This parameter can be one of the following values:
  - TIM_OCClear_Enable: TIM Output clear enable
  - TIM_OCClear_Disable: TIM Output clear disable

Return values
- None.

Notes
- None.
25.2.10.25 TIM_ClearOC3Ref

Function Name | void TIM_ClearOC3Ref (TIM_TypeDef * TIMx, uint16_t TIM_OCClear)
---|---
Function Description | Clears or safeguards the OREF3 signal on an external event.
Parameters | - TIMx : where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_OCClear : new state of the Output Compare Clear Enable Bit. This parameter can be one of the following values:
  - TIM_OCClear_Enable : TIM Output clear enable
  - TIM_OCClear_Disable : TIM Output clear disable
Return values | None.
Notes | None.

25.2.10.26 TIM_ClearOC4Ref

Function Name | void TIM_ClearOC4Ref (TIM_TypeDef * TIMx, uint16_t TIM_OCClear)
---|---
Function Description | Clears or safeguards the OREF4 signal on an external event.
Parameters | - TIMx : where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_OCClear : new state of the Output Compare Clear Enable Bit. This parameter can be one of the following values:
  - TIM_OCClear_Enable : TIM Output clear enable
  - TIM_OCClear_Disable : TIM Output clear disable
Return values | None.
Notes | None.

25.2.10.27 TIM_OC1PolarityConfig
Function Name: void TIM_OC1PolarityConfig (TIM_TypeDef * TIMx, uint16_t TIM_OCPolarity)

Function Description: Configures the TIMx channel 1 polarity.

Parameters:
- TIMx: where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.
- TIM_OCPolarity: specifies the OC1 Polarity This parameter can be one of the following values:
  - TIM_OCPolarity_High: Output Compare active high
  - TIM_OCPolarity_Low: Output Compare active low

Return values: None.

Notes: None.

25.2.10.28 TIM_OC1NPolarityConfig

Function Name: void TIM_OC1NPolarityConfig (TIM_TypeDef * TIMx, uint16_t TIM_OCNPolarity)

Function Description: Configures the TIMx Channel 1N polarity.

Parameters:
- TIMx: where x can be 1 or 8 to select the TIM peripheral.
- TIM_OCNPolarity: specifies the OC1N Polarity This parameter can be one of the following values:
  - TIM_OCNPolarity_High: Output Compare active high
  - TIM_OCNPolarity_Low: Output Compare active low

Return values: None.

Notes: None.

25.2.10.29 TIM_OC2PolarityConfig

Function Name: void TIM_OC2PolarityConfig (TIM_TypeDef * TIMx, uint16_t TIM_OCPolarity)

Function Description: Configures the TIMx channel 2 polarity.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
- TIM_OCPolarity: specifies the OC2 Polarity This parameter can be one of the following values:
General-purpose timers (TIM)

- **TIM_OCPolarity_High**: Output Compare active high
- **TIM_OCPolarity_Low**: Output Compare active low

**Return values**
- None.

**Notes**
- None.

### 25.2.10.30 TIM_OC2NPolarityConfig

**Function Name**

```c
void TIM_OC2NPolarityConfig ( TIM_TypeDef * TIMx, uint16_t TIM_OCNPolarity)
```

**Function Description**

Configures the TIMx Channel 2N polarity.

**Parameters**

- **TIMx**: where x can be 1 or 8 to select the TIM peripheral.
- **TIM_OCNPolarity**: specifies the OC2N Polarity. This parameter can be one of the following values:
  - **TIM_OCNPolarity_High**: Output Compare active high
  - **TIM_OCNPolarity_Low**: Output Compare active low

**Return values**
- None.

**Notes**
- None.

### 25.2.10.31 TIM_OC3PolarityConfig

**Function Name**

```c
void TIM_OC3PolarityConfig ( TIM_TypeDef * TIMx, uint16_t TIM_OCPolarity)
```

**Function Description**

Configures the TIMx channel 3 polarity.

**Parameters**

- **TIMx**: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- **TIM_OCPolarity**: specifies the OC3 Polarity. This parameter can be one of the following values:
  - **TIM_OCPolarity_High**: Output Compare active high
  - **TIM_OCPolarity_Low**: Output Compare active low

**Return values**
- None.

**Notes**
- None.
**25.2.10.32 TIM_OC3NPolarityConfig**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void TIM_OC3NPolarityConfig (TIM_TypeDef * TIMx, uint16_t TIM_OCNPolarity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Configures the TIMx Channel 3N polarity.</td>
</tr>
</tbody>
</table>
| Parameters | • TIMx : where x can be 1 or 8 to select the TIM peripheral.  
• TIM_OCNPolarity : specifies the OC3N Polarity This parameter can be one of the following values:
  – TIM_OCNPolarity_High : Output Compare active high  
  – TIM_OCNPolarity_Low : Output Compare active low |
| Return values | • None. |
| Notes | • None. |

**25.2.10.33 TIM_OC4PolarityConfig**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void TIM_OC4PolarityConfig (TIM_TypeDef * TIMx, uint16_t TIM_OCPolarity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Configures the TIMx channel 4 polarity.</td>
</tr>
</tbody>
</table>
| Parameters | • TIMx : where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.  
• TIM_OCPolarity : specifies the OC4 Polarity This parameter can be one of the following values:
  – TIM_OCPolarity_High : Output Compare active high  
  – TIM_OCPolarity_Low : Output Compare active low |
| Return values | • None. |
| Notes | • None. |

**25.2.10.34 TIM_CCxCmd**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void TIM_CCxCmd (TIM_TypeDef * TIMx, uint16_t TIM_Channel, uint16_t TIM_CCx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Enables or disables the TIM Capture Compare Channel x.</td>
</tr>
</tbody>
</table>
Parameters

- **TIMx**: where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.
- **TIM_Channel**: specifies the TIM Channel. This parameter can be one of the following values:
  - **TIM_Channel_1**: TIM Channel 1
  - **TIM_Channel_2**: TIM Channel 2
  - **TIM_Channel_3**: TIM Channel 3
  - **TIM_Channel_4**: TIM Channel 4
- **TIM_CCx**: specifies the TIM Channel CCxE bit new state. This parameter can be: **TIM_CCx_Enable** or **TIM_CCx_Disable**.

Return values

- None.

Notes

- None.

---

**25.2.10.35 TIM_CCxNCmd**

**Function Name**

`void TIM_CCxNCmd (TIM_TypeDef *TIMx, uint16_t TIM_Channel, uint16_t TIM_CCxN)`

**Function Description**

Enables or disables the TIM Capture Compare Channel xN.

**Parameters**

- **TIMx**: where x can be 1 or 8 to select the TIM peripheral.
- **TIM_Channel**: specifies the TIM Channel. This parameter can be one of the following values:
  - **TIM_Channel_1**: TIM Channel 1
  - **TIM_Channel_2**: TIM Channel 2
  - **TIM_Channel_3**: TIM Channel 3
- **TIM_CCxN**: specifies the TIM Channel CCxE bit new state. This parameter can be: **TIM_CCxN_Enable** or **TIM_CCxN_Disable**.

Return values

- None.

Notes

- None.

---

**25.2.11 Input Capture management functions**

**25.2.11.1 TIM_ICInit**

**Function Name**

`void TIM_ICInit (TIM_TypeDef *TIMx, TIM_ICInitTypeDef *TIM_ICInitStruct)`
Function Description | Initializes the TIM peripheral according to the specified parameters in the TIM_ICInitStruct.
--- | ---
Parameters | - **TIMx**: where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.
- **TIM_ICInitStruct**: pointer to a TIM_ICInitTypeDef structure that contains the configuration information for the specified TIM peripheral.
Return values | - None.
Notes | - None.

### 25.2.11.2 TIM_ICStructInit

| Function Name | void TIM_ICStructInit (TIM_ICInitTypeDef * TIM_ICInitStruct) |
| Function Description | Fills each TIM_ICInitStruct member with its default value. |
| Parameters | - **TIM_ICInitStruct**: pointer to a TIM_ICInitTypeDef structure which will be initialized. |
| Return values | - None. |
| Notes | - None. |

### 25.2.11.3 TIM_PWMIConfig

| Function Name | void TIM_PWMIConfig (TIM_TypeDef * TIMx, TIM_ICInitTypeDef * TIM_ICInitStruct) |
| Function Description | Configures the TIM peripheral according to the specified parameters in the TIM_ICInitStruct to measure an external PWM signal. |
| Parameters | - **TIMx**: where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
- **TIM_ICInitStruct**: pointer to a TIM_ICInitTypeDef structure that contains the configuration information for the specified TIM peripheral. |
| Return values | - None. |
| Notes | - None. |
25.2.11.4 TIM_GetCapture1

Function Name: uint32_t TIM_GetCapture1 (TIM_TypeDef * TIMx)
Function Description: Gets the TIMx Input Capture 1 value.
Parameters:
- TIMx: where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.
Return values:
- Capture Compare 1 Register value.
Notes:
- None.

25.2.11.5 TIM_GetCapture2

Function Name: uint32_t TIM_GetCapture2 (TIM_TypeDef * TIMx)
Function Description: Gets the TIMx Input Capture 2 value.
Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
Return values:
- Capture Compare 2 Register value.
Notes:
- None.

25.2.11.6 TIM_GetCapture3

Function Name: uint32_t TIM_GetCapture3 (TIM_TypeDef * TIMx)
Function Description: Gets the TIMx Input Capture 3 value.
Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
Return values:
- Capture Compare 3 Register value.
Notes:
- None.
25.2.11.7 TIM_GetCapture4

Function Name: uint32_t TIM_GetCapture4 (TIM_TypeDef * TIMx)
Function Description: Gets the TIMx Input Capture 4 value.
Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
Return values:
- Capture Compare 4 Register value.
Notes:
- None.

25.2.11.8 TIM_SetIC1Prescaler

Function Name: void TIM_SetIC1Prescaler (TIM_TypeDef * TIMx, uint16_t TIM_ICPSC)
Function Description: Sets the TIMx Input Capture 1 prescaler.
Parameters:
- TIMx: where x can be 1 to 14 except 6 and 7, to select the TIM peripheral.
- TIM_ICPSC: specifies the Input Capture1 prescaler new value. This parameter can be one of the following values:
  - TIM_ICPSC_DIV1: no prescaler
  - TIM_ICPSC_DIV2: capture is done once every 2 events
  - TIM_ICPSC_DIV4: capture is done once every 4 events
  - TIM_ICPSC_DIV8: capture is done once every 8 events
Return values:
- None.
Notes:
- None.

25.2.11.9 TIM_SetIC2Prescaler
Function Name | void TIM_SetIC2Prescaler (TIM_TypeDef * TIMx, uint16_t TIM_ICPSC)
--- | ---
Function Description | Sets the TIMx Input Capture 2 prescaler.
Parameters | • TIMx : where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
• TIM_ICPSC : specifies the Input Capture2 prescaler new value. This parameter can be one of the following values:
  – TIM_ICPSC_DIV1 : no prescaler
  – TIM_ICPSC_DIV2 : capture is done once every 2 events
  – TIM_ICPSC_DIV4 : capture is done once every 4 events
  – TIM_ICPSC_DIV8 : capture is done once every 8 events
Return values | • None.
Notes | • None.

25.2.11.10 TIM_SetIC3Prescaler

Function Name | void TIM_SetIC3Prescaler (TIM_TypeDef * TIMx, uint16_t TIM_ICPSC)
--- | ---
Function Description | Sets the TIMx Input Capture 3 prescaler.
Parameters | • TIMx : where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
• TIM_ICPSC : specifies the Input Capture3 prescaler new value. This parameter can be one of the following values:
  – TIM_ICPSC_DIV1 : no prescaler
  – TIM_ICPSC_DIV2 : capture is done once every 2 events
  – TIM_ICPSC_DIV4 : capture is done once every 4 events
  – TIM_ICPSC_DIV8 : capture is done once every 8 events
Return values | • None.
Notes | • None.

25.2.11.11 TIM_SetIC4Prescaler
Function Name: void TIM_SetIC4Prescaler (TIM_TypeDef * TIMx, uint16_t TIM_ICPSC)

Function Description: Sets the TIMx Input Capture 4 prescaler.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_ICPSC: specifies the Input Capture4 prescaler new value. This parameter can be one of the following values:
  - TIM_ICPSC_DIV1: no prescaler
  - TIM_ICPSC_DIV2: capture is done once every 2 events
  - TIM_ICPSC_DIV4: capture is done once every 4 events
  - TIM_ICPSC_DIV8: capture is done once every 8 events

Return values:
- None.

Notes:
- None.

25.2.12 Advanced-control timers (TIM1 and TIM8) specific features

25.2.12.1 TIM_BDTRConfig

Function Name: void TIM_BDTRConfig (TIM_TypeDef * TIMx, TIM_BDTRInitTypeDef * TIM_BDTRInitStruct)

Function Description: Configures the Break feature, dead time, Lock level, OSSI/OSSR State and the AOE (automatic output enable).

Parameters:
- TIMx: where x can be 1 or 8 to select the TIM
- TIM_BDTRInitStruct: pointer to a TIM_BDTRInitStruct structure that contains the BDTR Register configuration information for the TIM peripheral.

Return values:
- None.

Notes:
- None.

25.2.12.2 TIM_BDTRStructInit
### 25.2.12.3 TIM_CtrlPWMOutputs

**Function Name**  
`void TIM_CtrlPWMOutputs (TIM_TypeDef * TIMx, FunctionalState NewState)`  

**Function Description**  
Enables or disables the TIM peripheral Main Outputs.

**Parameters**  
- `TIMx` : where x can be 1 or 8 to select the TIMx peripheral.  
- `NewState` : new state of the TIM peripheral Main Outputs. This parameter can be: ENABLE or DISABLE.

**Return values**  
- None.

**Notes**  
- None.

### 25.2.12.4 TIM_SelectCOM

**Function Name**  
`void TIM_SelectCOM (TIM_TypeDef * TIMx, FunctionalState NewState)`  

**Function Description**  
Selects the TIM peripheral Commutation event.

**Parameters**  
- `TIMx` : where x can be 1 or 8 to select the TIMx peripheral  
- `NewState` : new state of the Commutation event. This parameter can be: ENABLE or DISABLE.

**Return values**  
- None.

**Notes**  
- None.
Function Name: void TIM_CCPreloadControl (TIM_TypeDef * TIMx,
FunctionalState NewState)

Function Description: Sets or Resets the TIM peripheral Capture Compare Preload
Control bit.

Parameters:
- TIMx: where x can be 1 or 8 to select the TIMx peripheral
- NewState: new state of the Capture Compare Preload
  Control bit. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.

25.2.13 Interrupts DMA and flags management functions

25.2.13.1 TIM_ITConfig

Function Name: void TIM_ITConfig (TIM_TypeDef * TIMx, uint16_t TIM_IT,
FunctionalState NewState)

Function Description: Enables or disables the specified TIM interrupts.

Parameters:
- TIMx: where x can be 1 to 14 to select the TIMx peripheral.
- TIM_IT: specifies the TIM interrupts sources to be enabled
  or disabled. This parameter can be any combination of the
  following values:
  - TIM_IT_Update: TIM update Interrupt source
  - TIM_IT_CC1: TIM Capture Compare 1 Interrupt source
  - TIM_IT_CC2: TIM Capture Compare 2 Interrupt source
  - TIM_IT_CC3: TIM Capture Compare 3 Interrupt source
  - TIM_IT_CC4: TIM Capture Compare 4 Interrupt source
  - TIM_IT_COM: TIM Commutation Interrupt source
  - TIM_IT_Trigger: TIM Trigger Interrupt source
  - TIM_IT_Break: TIM Break Interrupt source
- NewState: new state of the TIM interrupts. This parameter
  can be: ENABLE or DISABLE.

Parameters:
- NewState: new state of the TIM interrupts. This parameter
  can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- For TIM6 and TIM7 only the parameter TIM_IT_Update can
  be used
- For TIM9 and TIM12 only one of the following parameters can
  be used: TIM_IT_Update, TIM_IT_CC1, TIM_IT_CC2 or
  TIM_IT_Trigger.
- For TIM10, TIM11, TIM13 and TIM14 only one of the following
25.2.13.2 TIM_GenerateEvent

Function Name

void TIM_GenerateEvent (TIM_TypeDef * TIMx, uint16_t TIM_EventSource)

Function Description

Configures the TIMx event to be generate by software.

Parameters

- **TIMx**: where x can be 1 to 14 to select the TIM peripheral.
- **TIM_EventSource**: specifies the event source. This parameter can be one or more of the following values:
  - **TIM_EventSource_Update**: Timer update Event source
  - **TIM_EventSource_CC1**: Timer Capture Compare 1 Event source
  - **TIM_EventSource_CC2**: Timer Capture Compare 2 Event source
  - **TIM_EventSource_CC3**: Timer Capture Compare 3 Event source
  - **TIM_EventSource_CC4**: Timer Capture Compare 4 Event source
  - **TIM_EventSource_COM**: Timer COM event source
  - **TIM_EventSource_Trigger**: Timer Trigger Event source
  - **TIM_EventSource_Break**: Timer Break event source

Return values

- None.

Notes

- TIM6 and TIM7 can only generate an update event.
- TIM_EventSource_COM and TIM_EventSource_Break are used only with TIM1 and TIM8.

25.2.13.3 TIM_GetFlagStatus

Function Name

FlagStatus TIM_GetFlagStatus (TIM_TypeDef * TIMx, uint16_t TIM_FLAG)

Function Description

Checks whether the specified TIM flag is set or not.

Parameters

- **TIMx**: where x can be 1 to 14 to select the TIM peripheral.
- **TIM_FLAG**: specifies the flag to check. This parameter can be one of the following values:
  - **TIM_FLAG_Update**: TIM update Flag
  - **TIM_FLAG_CC1**: TIM Capture Compare 1 Flag
  - **TIM_FLAG_CC2**: TIM Capture Compare 2 Flag
  - **TIM_FLAG_CC3**: TIM Capture Compare 3 Flag
  - **TIM_FLAG_CC4**: TIM Capture Compare 4 Flag
  - **TIM_FLAG_COM**: TIM Commutation Flag
  - **TIM_FLAG_Trigger**: TIM Trigger Flag
  - **TIM_FLAG_Break**: TIM Break Flag
  - **TIM_FLAG_CC1OF**: TIM Capture Compare 1 over capture Flag
  - **TIM_FLAG_CC2OF**: TIM Capture Compare 2 over capture Flag
  - **TIM_FLAG_CC3OF**: TIM Capture Compare 3 over capture Flag
  - **TIM_FLAG_CC4OF**: TIM Capture Compare 4 over capture Flag

Return values
- The new state of TIM_FLAG (SET or RESET).

Notes
- TIM6 and TIM7 can have only one update flag.
- TIM_FLAG_COM and TIM_FLAG_Break are used only with TIM1 and TIM8.

### 25.2.13.4 TIM_ClearFlag

**Function Name**

```c
void TIM_ClearFlag ( TIM_TypeDef * TIMx, uint16_t TIM_FLAG)
```

**Function Description**

Clears the TIMx's pending flags.

**Parameters**

- **TIMx**: where x can be 1 to 14 to select the TIM peripheral.
- **TIM_FLAG**: specifies the flag bit to clear. This parameter can be any combination of the following values:
  - **TIM_FLAG_Update**: TIM update Flag
  - **TIM_FLAG_CC1**: TIM Capture Compare 1 Flag
  - **TIM_FLAG_CC2**: TIM Capture Compare 2 Flag
  - **TIM_FLAG_CC3**: TIM Capture Compare 3 Flag
  - **TIM_FLAG_CC4**: TIM Capture Compare 4 Flag
  - **TIM_FLAG_COM**: TIM Commutation Flag
  - **TIM_FLAG_Trigger**: TIM Trigger Flag
  - **TIM_FLAG_Break**: TIM Break Flag
  - **TIM_FLAG_CC1OF**: TIM Capture Compare 1 over capture Flag
  - **TIM_FLAG_CC2OF**: TIM Capture Compare 2 over capture Flag
  - **TIM_FLAG_CC3OF**: TIM Capture Compare 3 over capture Flag
  - **TIM_FLAG_CC4OF**: TIM Capture Compare 4 over capture Flag
25.2.13.5 TIM_GetITStatus

Function Name

ITStatus TIM_GetITStatus (TIM_TypeDef * TIMx, uint16_t TIM_IT)

Function Description

Checks whether the TIM interrupt has occurred or not.

Parameters

- **TIMx**: where x can be 1 to 14 to select the TIM peripheral.
- **TIM_IT**: specifies the TIM interrupt source to check. This parameter can be one of the following values:
  - **TIM_IT_Update**: TIM update Interrupt source
  - **TIM_IT_CC1**: TIM Capture Compare 1 Interrupt source
  - **TIM_IT_CC2**: TIM Capture Compare 2 Interrupt source
  - **TIM_IT_CC3**: TIM Capture Compare 3 Interrupt source
  - **TIM_IT_CC4**: TIM Capture Compare 4 Interrupt source
  - **TIM_IT_COM**: TIM Commutation Interrupt source
  - **TIM_IT_Trigger**: TIM Trigger Interrupt source
  - **TIM_IT_Break**: TIM Break Interrupt source

Return values

- The new state of the TIM_IT(SET or RESET).

Notes

- TIM6 and TIM7 can generate only an update interrupt.
- TIM_IT_COM and TIM_IT_Break are used only with TIM1 and TIM8.

25.2.13.6 TIM_ClearITPendingBit

Function Name

void TIM_ClearITPendingBit (TIM_TypeDef * TIMx, uint16_t TIM_IT)

Function Description

Clears the TIMx's interrupt pending bits.

Parameters

- **TIMx**: where x can be 1 to 14 to select the TIM peripheral.
- **TIM_IT**: specifies the pending bit to clear. This parameter can be any combination of the following values:
General-purpose timers (TIM)

- **TIM_IT_Update**: TIM1 update Interrupt source
- **TIM_IT_CC1**: TIM Capture Compare 1 Interrupt source
- **TIM_IT_CC2**: TIM Capture Compare 2 Interrupt source
- **TIM_IT_CC3**: TIM Capture Compare 3 Interrupt source
- **TIM_IT_CC4**: TIM Capture Compare 4 Interrupt source
- **TIM_IT_COM**: TIM Commutation Interrupt source
- **TIM_IT_Trigger**: TIM Trigger Interrupt source
- **TIM_IT_Break**: TIM Break Interrupt source

**Return values**
- None.

**Notes**
- TIM6 and TIM7 can generate only an update interrupt.
- TIM_IT_COM and TIM_IT_Break are used only with TIM1 and TIM8.

### 25.2.13.7 TIM_DMAConfig

**Function Name**
void TIM_DMAConfig (TIM_TypeDef *TIMx, uint16_t TIM_DMABase, uint16_t TIM_DMABurstLength)

**Function Description**
Configures the TIMx's DMA interface.

**Parameters**
- **TIMx**: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- **TIM_DMABase**: DMA Base address. This parameter can be one of the following values:
  - **TIM_DMABase_CR1**:
  - **TIM_DMABase_CR2**:
  - **TIM_DMABase_SMCR**:
  - **TIM_DMABase_DIER**:
  - TIM1_DMABase_SR
  - **TIM_DMABase_EGR**:
  - **TIM_DMABase_CCMR1**:
  - **TIM_DMABase_CCMR2**:
  - **TIM_DMABase_CCRER**:
  - **TIM_DMABase_CNT**:
  - **TIM_DMABase_PSC**:
  - **TIM_DMABase_ARR**:
  - **TIM_DMABase_RCR**:
  - **TIM_DMABase_CCR1**:
  - **TIM_DMABase_CCR2**:
  - **TIM_DMABase_CCR3**:
  - **TIM_DMABase_CCR4**:
  - **TIM_DMABase_BDTR**:
  - **TIM_DMABase_DCR**:
- **TIM_DMABurstLength**: DMA Burst length. This parameter can be one value between: TIM_DMABurstLength_1Transfer and TIM_DMABurstLength_18Transfers.
25.2.13.8 TIM_DMACmd

Function Name: `void TIM_DMACmd (TIM_TypeDef * TIMx, uint16_t TIM_DMASource, FunctionalState NewState)`

Function Description: Enables or disables the TIMx's DMA Requests.

Parameters:
- `TIMx`: where x can be 1, 2, 3, 4, 5, 6, 7 or 8 to select the TIM peripheral.
- `TIM_DMASource`: specifies the DMA Request sources. This parameter can be any combination of the following values:
  - `TIM_DMA_Update`: TIM update Interrupt source
  - `TIM_DMA_CC1`: TIM Capture Compare 1 DMA source
  - `TIM_DMA_CC2`: TIM Capture Compare 2 DMA source
  - `TIM_DMA_CC3`: TIM Capture Compare 3 DMA source
  - `TIM_DMA_CC4`: TIM Capture Compare 4 DMA source
  - `TIM_DMA_COM`: TIM Commutation DMA source
  - `TIM_DMA_Trigger`: TIM Trigger DMA source
- `NewState`: new state of the DMA Request sources. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

25.2.13.9 TIM_SelectCCDMA

Function Name: `void TIM_SelectCCDMA (TIM_TypeDef * TIMx, FunctionalState NewState)`

Function Description: Selects the TIMx peripheral Capture Compare DMA source.

Parameters:
- `TIMx`: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- `NewState`: new state of the Capture Compare DMA source. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.
25.2.14 Clocks management functions

25.2.14.1 TIM_InternalClockConfig

Function Name: void TIM_InternalClockConfig (TIM_TypeDef * TIMx)
Function Description: Configures the TIMx internal Clock.
Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.

Return values: None.
Notes: None.

25.2.14.2 TIM_ITRxExternalClockConfig

Function Name: void TIM_ITRxExternalClockConfig (TIM_TypeDef * TIMx, uint16_t TIM_InputTriggerSource)
Function Description: Configures the TIMx Internal Trigger as External Clock.
Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
- TIM_InputTriggerSource: Trigger source. This parameter can be one of the following values:
  - TIM_TS_ITR0: Internal Trigger 0
  - TIM_TS_ITR1: Internal Trigger 1
  - TIM_TS_ITR2: Internal Trigger 2
  - TIM_TS_ITR3: Internal Trigger 3

Return values: None.
Notes: None.

25.2.14.3 TIM_TIxExternalClockConfig
Function Name: void TIM_TIxExternalClockConfig (TIM_TypeDef * TIMx, uint16_t TIM_TIxExternalCLKSource, uint16_t TIM_ICPolarity, uint16_t ICFilter)

Function Description: Configures the TIMx Trigger as External Clock.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 13 or 14 to select the TIM peripheral.
- TIM_TIxExternalCLKSource: Trigger source. This parameter can be one of the following values:
  - TIM_TIxExternalCLKSource_TI1ED: TI1 Edge Detector
  - TIM_TIxExternalCLKSource_TI1: Filtered Timer Input 1
  - TIM_TIxExternalCLKSource_TI2: Filtered Timer Input 2
- TIM_ICPolarity: specifies the TIx Polarity. This parameter can be one of the following values:
  - TIM_ICPolarity_Rising
  - TIM_ICPolarity_Falling
- ICFilter: specifies the filter value. This parameter must be a value between 0x0 and 0xF.

Return values: None.

Notes: None.

25.2.14.4 TIM_ETRClockMode1Config

Function Name: void TIM_ETRClockMode1Config (TIM_TypeDef * TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter)

Function Description: Configures the External clock Mode1.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_ExtTRGPrescaler: The external Trigger Prescaler. This parameter can be one of the following values:
  - TIM_ExtTRGPSC_OFF: ETRP Prescaler OFF.
  - TIM_ExtTRGPSC_DIV2: ETRP frequency divided by 2.
  - TIM_ExtTRGPSC_DIV4: ETRP frequency divided by 4.
  - TIM_ExtTRGPSC_DIV8: ETRP frequency divided by 8.
- TIM_ExtTRGPolarity: The external Trigger Polarity. This parameter can be one of the following values:
  - TIM_ExtTRGPolarity_Inverted: active low or falling edge active.
  - TIM_ExtTRGPolarity_NonInverted: active high or rising edge active.
- ExtTRGFilter: External Trigger Filter. This parameter must
be a value between 0x00 and 0x0F

Return values
- None.

Notes
- None.

25.2.14.5 TIM_ETRClockMode2Config

Function Name
void TIM_ETRClockMode2Config ( TIM_TypeDef * TIMx, uint16_t TIM_ExtTRGPrescaler, uint16_t TIM_ExtTRGPolarity, uint16_t ExtTRGFilter)

Function Description
Configures the External clock Mode2.

Parameters
- TIMx : where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- TIM_ExtTRGPrescaler : The external Trigger Prescaler. This parameter can be one of the following values:
  - TIM_ExtTRGPSC_OFF : ETRP Prescaler OFF.
  - TIM_ExtTRGPSC_DIV2 : ETRP frequency divided by 2.
  - TIM_ExtTRGPSC_DIV4 : ETRP frequency divided by 4.
  - TIM_ExtTRGPSC_DIV8 : ETRP frequency divided by 8.
- TIM_ExtTRGPolarity : The external Trigger Polarity. This parameter can be one of the following values:
  - TIM_ExtTRGPolarity_Inverted : active low or falling edge active.
  - TIM_ExtTRGPolarity_NonInverted : active high or rising edge active.
- ExtTRGFilter : External Trigger Filter. This parameter must be a value between 0x00 and 0x0F

Return values
- None.

Notes
- None.

25.2.15 Synchronization management functions

25.2.15.1 TIM_SelectInputTrigger

Function Name
void TIM_SelectInputTrigger ( TIM_TypeDef * TIMx, uint16_t TIM_InputTriggerSource)

Function Description
Selects the Input Trigger source.
Parameters
- **TIMx**: where x can be 1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 13 or 14 to select the TIM peripheral.
- **TIM_InputTriggerSource**: The Input Trigger source. This parameter can be one of the following values:
  - `TIM_TS_ITR0`: Internal Trigger 0
  - `TIM_TS_ITR1`: Internal Trigger 1
  - `TIM_TS_ITR2`: Internal Trigger 2
  - `TIM_TS_ITR3`: Internal Trigger 3
  - `TIM_TS_TI1F_ED`: TI1 Edge Detector
  - `TIM_TS_TI1FP1`: Filtered Timer Input 1
  - `TIM_TS_TI2FP2`: Filtered Timer Input 2
  - `TIM_TS_ETRF`: External Trigger input

Return values
- **None.**

Notes
- **None.**

---

### 25.2.15.2 TIM_SelectOutputTrigger

**Function Name**
void TIM_SelectOutputTrigger (TIM_TypeDef *TIMx, uint16_t TIM_TRGOSource)

**Function Description**
Selects the TIMx Trigger Output Mode.

**Parameters**
- **TIMx**: where x can be 1, 2, 3, 4, 5, 6, 7 or 8 to select the TIM peripheral.
- **TIM_TRGOSource**: specifies the Trigger Output source. This parameter can be one of the following values:

**Notes**
- **None.**

---

### 25.2.15.3 TIM_SelectSlaveMode

**Function Name**
void TIM_SelectSlaveMode (TIM_TypeDef *TIMx, uint16_t TIM_SlaveMode)

**Function Description**
Selects the TIMx Slave Mode.

**Parameters**
- **TIMx**: where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
- **TIM_SlaveMode**: specifies the Timer Slave Mode. This parameter can be one of the following values:
  - `TIM_SlaveMode_Reset`: Rising edge of the selected trigger signal (TRGI) reinitialize the counter and triggers
an update of the registers

- **TIM_SlaveMode_Gated**: The counter clock is enabled when the trigger signal (TRGI) is high
- **TIM_SlaveMode_Trigger**: The counter starts at a rising edge of the trigger TRGI
- **TIM_SlaveMode_External1**: Rising edges of the selected trigger (TRGI) clock the counter

**Return values**
- None.

**Notes**
- None.

### 25.2.15.4 TIM_SelectMasterSlaveMode

**Function Name**

```c
void TIM_SelectMasterSlaveMode ( TIM_TypeDef *TIMx,
                               uint16_t TIM_MasterSlaveMode)
```

**Function Description**
Sets or Resets the TIMx Master/Slave Mode.

**Parameters**

- **TIMx**: where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
- **TIM_MasterSlaveMode**: specifies the Timer Master Slave Mode. This parameter can be one of the following values:
  - **TIM_MasterSlaveMode_Enable**: synchronization between the current timer and its slaves (through TRGO)
  - **TIM_MasterSlaveMode_Disable**: No action

**Return values**
- None.

**Notes**
- None.

### 25.2.15.5 TIM_ETRConfig

**Function Name**

```c
void TIM_ETRConfig ( TIM_TypeDef *TIMx,
                     uint16_t TIM_ExtTRGPrescaler,
                     uint16_t TIM_ExtTRGPolarity,
                     uint16_t ExtTRGFilter)
```

**Function Description**
Configures the TIMx External Trigger (ETR).

**Parameters**

- **TIMx**: where x can be 1, 2, 3, 4, 5 or 8 to select the TIM peripheral.
- **TIM_ExtTRGPrescaler**: The external Trigger Prescaler. This parameter can be one of the following values:
  - **TIM_ExtTRGPrescaler_OFF**: ETRP Prescaler OFF.
25.2.16 Specific interface management functions

25.2.16.1 TIM_EncoderInterfaceConfig

Function Name

```c
void TIM_EncoderInterfaceConfig ( TIM_TypeDef *TIMx, uint16_t TIM_EncoderMode, uint16_t TIM_IC1Polarity, uint16_t TIM_IC2Polarity)
```

Function Description

Configures the TIMx Encoder Interface.

Parameters

- **TIMx** : where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
- **TIM_EncoderMode** : specifies the TIMx Encoder Mode. This parameter can be one of the following values:
  - **TIM_EncoderMode_TI1** : Counter counts on TI1FP1 edge depending on TI2FP2 level.
  - **TIM_EncoderMode_TI2** : Counter counts on TI2FP2 edge depending on TI1FP1 level.
  - **TIM_EncoderMode_TI12** : Counter counts on both TI1FP1 and TI2FP2 edges depending on the level of the other input.
- **TIM_IC1Polarity** : specifies the IC1 Polarity. This parameter can be one of the following values:
  - **TIM_ICPolarity_Falling** : IC Falling edge.
  - **TIM_ICPolarity_Rising** : IC Rising edge.
- **TIM_IC2Polarity** : specifies the IC2 Polarity. This parameter can be one of the following values:
  - **TIM_ICPolarity_Falling** : IC Falling edge.
  - **TIM_ICPolarity_Rising** : IC Rising edge.

Return values

- None.

Notes

- None.
25.2.16.2 TIM_SelectHallSensor

Function Name: void TIM_SelectHallSensor (TIM_TypeDef * TIMx, FunctionalState NewState)

Function Description: Enables or disables the TIMx's Hall sensor interface.

Parameters:
- TIMx: where x can be 1, 2, 3, 4, 5, 8, 9 or 12 to select the TIM peripheral.
- NewState: new state of the TIMx Hall sensor interface. This parameter can be: ENABLE or DISABLE.

Return values:
- None.

Notes:
- None.

25.2.17 Specific remapping management function

25.2.17.1 TIM_RemapConfig

Function Name: void TIM_RemapConfig (TIM_TypeDef * TIMx, uint16_t TIM_Remap)

Function Description: Configures the TIM2, TIM5 and TIM11 Remapping input capabilities.

Parameters:
- TIMx: where x can be 2, 5 or 11 to select the TIM peripheral.
- TIM_Remap: specifies the TIM input remapping source. This parameter can be one of the following values:
  - TIM2_TIM8_TRGO: TIM2 ITR1 input is connected to TIM8 Trigger output (default)
  - TIM2_ETH_PTP: TIM2 ITR1 input is connected to ETH PTP trigger output.
  - TIM2_USBFS_SOF: TIM2 ITR1 input is connected to USB FS SOF.
  - TIM2_USBHS_SOF: TIM2 ITR1 input is connected to USB HS SOF.
  - TIM5_GPIO: TIM5 CH4 input is connected to dedicated Timer pin (default)
  - TIM5_LSI: TIM5 CH4 input is connected to LSI clock.
  - TIM5_LSE: TIM5 CH4 input is connected to LSE clock.
  - TIM5_RTC: TIM5 CH4 input is connected to RTC Output event.
General-purpose timers (TIM)

\[ \text{TIM11_GPIO} : \] TIM11 CH4 input is connected to dedicated Timer pin (default)

\[ \text{TIM11_HSE} : \] TIM11 CH4 input is connected to HSE_RTC clock (HSE divided by a programmable prescaler)

**Return values**
- None.

**Notes**
- None.

### 25.3 TIM Firmware driver defines

#### 25.3.1 TIM Firmware driver defines

**TIM**

*TIM_AOE_Bit_Set_Reset*
- \#define: `TIM_AutomaticOutput_Enable((uint16_t)0x4000)`

- \#define: `TIM_AutomaticOutput_Disable((uint16_t)0x0000)`

*TIM_Break_Input_enable_disable*
- \#define: `TIM_Break_Enable((uint16_t)0x1000)`

- \#define: `TIM_Break_Disable((uint16_t)0x0000)`

**TIM_Break_Polarity**
- \#define: `TIM_BreakPolarity_Low((uint16_t)0x0000)`

- \#define: `TIM_BreakPolarity_High((uint16_t)0x2000)`

*TIM_Capture_Compare_N_State*
- \#define: `TIM_CCxN_Enable((uint16_t)0x0004)`
• #define: `TIM_CCxN_Disable((uint16_t)0x0000)`

`TIM_Capture_Compare_State`

• #define: `TIM_CCx_Enable((uint16_t)0x0001)`

• #define: `TIM_CCx_Disable((uint16_t)0x0000)`

`TIM_Channel`

• #define: `TIM_Channel_1((uint16_t)0x0000)`

• #define: `TIM_Channel_2((uint16_t)0x0004)`

• #define: `TIM_Channel_3((uint16_t)0x0008)`

• #define: `TIM_Channel_4((uint16_t)0x000C)`

`TIM_Clock_Division_CKD`

• #define: `TIM_CKD_DIV1((uint16_t)0x0000)`

• #define: `TIM_CKD_DIV2((uint16_t)0x0100)`

• #define: `TIM_CKD_DIV4((uint16_t)0x0200)`

`TIM.Counter_Mode`

• #define: `TIM.CounterMode_Up((uint16_t)0x0000)`
General-purpose timers (TIM)

- #define: `TIM_CounterMode_Down((uint16_t)0x0010)`

- #define: `TIM_CounterMode_CenterAligned1((uint16_t)0x0020)`

- #define: `TIM_CounterMode_CenterAligned2((uint16_t)0x0040)`

- #define: `TIM_CounterMode_CenterAligned3((uint16_t)0x0060)`

`TIM_DMA_Base_address`

- #define: `TIM_DMABase_CR1((uint16_t)0x0000)`

- #define: `TIM_DMABase_CR2((uint16_t)0x0001)`

- #define: `TIM_DMABase_SMCR((uint16_t)0x0002)`

- #define: `TIM_DMABase_DIER((uint16_t)0x0003)`

- #define: `TIM_DMABase_SR((uint16_t)0x0004)`

- #define: `TIM_DMABase_EGR((uint16_t)0x0005)`

- #define: `TIM_DMABase_CCMR1((uint16_t)0x0006)`
- #define: `TIM_DMABase_CCMR2((uint16_t)0x0007)`

- #define: `TIM_DMABase_CCER((uint16_t)0x0008)`

- #define: `TIM_DMABase_CNT((uint16_t)0x0009)`

- #define: `TIM_DMABase_PSC((uint16_t)0x000A)`

- #define: `TIM_DMABase_ARR((uint16_t)0x000B)`

- #define: `TIM_DMABase_RCR((uint16_t)0x000C)`

- #define: `TIM_DMABase_CCR1((uint16_t)0x000D)`

- #define: `TIM_DMABase_CCR2((uint16_t)0x000E)`

- #define: `TIM_DMABase_CCR3((uint16_t)0x000F)`

- #define: `TIM_DMABase_CCR4((uint16_t)0x0010)`

- #define: `TIM_DMABase_BDTR((uint16_t)0x0011)`

- #define: `TIM_DMABase_DCR((uint16_t)0x0012)`
General-purpose timers (TIM)

- `#define: TIM_DMABase_OR((uint16_t)0x0013)`

`TIM_DMA_Burst_Length`

- `#define: TIM_DMA_BurstLength_1Transfer((uint16_t)0x0000)`

- `#define: TIM_DMA_BurstLength_2Transfers((uint16_t)0x0010)`

- `#define: TIM_DMA_BurstLength_3Transfers((uint16_t)0x0020)`

- `#define: TIM_DMA_BurstLength_4Transfers((uint16_t)0x0030)`

- `#define: TIM_DMA_BurstLength_5Transfers((uint16_t)0x0040)`

- `#define: TIM_DMA_BurstLength_6Transfers((uint16_t)0x0050)`

- `#define: TIM_DMA_BurstLength_7Transfers((uint16_t)0x0060)`

- `#define: TIM_DMA_BurstLength_8Transfers((uint16_t)0x0070)`

- `#define: TIM_DMA_BurstLength_9Transfers((uint16_t)0x0080)`

- `#define: TIM_DMA_BurstLength_10Transfers((uint16_t)0x0090)`

- `#define: TIM_DMA_BurstLength_11Transfers((uint16_t)0x00A0)`
• #define: `TIM_DMABurstLength_12Transfers((uint16_t)0x0B00)`

• #define: `TIM_DMABurstLength_13Transfers((uint16_t)0x0C00)`

• #define: `TIM_DMABurstLength_14Transfers((uint16_t)0x0D00)`

• #define: `TIM_DMABurstLength_15Transfers((uint16_t)0x0E00)`

• #define: `TIM_DMABurstLength_16Transfers((uint16_t)0x0F00)`

• #define: `TIM_DMABurstLength_17Transfers((uint16_t)0x1000)`

• #define: `TIM_DMABurstLength_18Transfers((uint16_t)0x1100)`

**TIM_DMA_sources**

• #define: `TIM_DMA_Update((uint16_t)0x0100)`

• #define: `TIM_DMA_CC1((uint16_t)0x0200)`

• #define: `TIM_DMA_CC2((uint16_t)0x0400)`

• #define: `TIM_DMA_CC3((uint16_t)0x0800)`

• #define: `TIM_DMA_CC4((uint16_t)0x1000)`
• #define: \texttt{TIM\_DMA\_COM\((uint16\_t)0x2000\)}

• #define: \texttt{TIM\_DMA\_Trigger\((uint16\_t)0x4000\)}

\textit{TIM\_Encoder\_Mode}
• #define: \texttt{TIM\_EncoderMode\_TI1\((uint16\_t)0x0001\)}

• #define: \texttt{TIM\_EncoderMode\_TI2\((uint16\_t)0x0002\)}

• #define: \texttt{TIM\_EncoderMode\_TI12\((uint16\_t)0x0003\)}

\textit{TIM\_Event\_Source}
• #define: \texttt{TIM\_EventSource\_Update\((uint16\_t)0x0001\)}

• #define: \texttt{TIM\_EventSource\_CC1\((uint16\_t)0x0002\)}

• #define: \texttt{TIM\_EventSource\_CC2\((uint16\_t)0x0004\)}

• #define: \texttt{TIM\_EventSource\_CC3\((uint16\_t)0x0008\)}

• #define: \texttt{TIM\_EventSource\_CC4\((uint16\_t)0x0010\)}

• #define: \texttt{TIM\_EventSource\_COM\((uint16\_t)0x0020\)}
• #define: `TIM_EventSource_Trigger((uint16_t)0x0040)`

• #define: `TIM_EventSource_Break((uint16_t)0x0080)`

`TIM_External_Trigger_Polarity`

• #define: `TIM_ExtTRGPolarity_Inverted((uint16_t)0x8000)`

• #define: `TIM_ExtTRGPolarity_NonInverted((uint16_t)0x0000)`

`TIM_External_Trigger_Prescaler`

• #define: `TIM_ExtTRGPSC_OFF((uint16_t)0x0000)`

• #define: `TIM_ExtTRGPSC_DIV2((uint16_t)0x1000)`

• #define: `TIM_ExtTRGPSC_DIV4((uint16_t)0x2000)`

• #define: `TIM_ExtTRGPSC_DIV8((uint16_t)0x3000)`

`TIM_Flags`

• #define: `TIM_FLAG_Update((uint16_t)0x0001)`

• #define: `TIM_FLAG_CC1((uint16_t)0x0002)`

• #define: `TIM_FLAG_CC2((uint16_t)0x0004)`
- `#define TIM_FLAG_CC3((uint16_t)0x0008)`

- `#define TIM_FLAG_CC4((uint16_t)0x0010)`

- `#define TIM_FLAG_COM((uint16_t)0x0020)`

- `#define TIM_FLAG_Trigger((uint16_t)0x0040)`

- `#define TIM_FLAG_Break((uint16_t)0x0080)`

- `#define TIM_FLAG_CC1OF((uint16_t)0x0200)`

- `#define TIM_FLAG_CC2OF((uint16_t)0x0400)`

- `#define TIM_FLAG_CC3OF((uint16_t)0x0800)`

- `#define TIM_FLAG_CC4OF((uint16_t)0x1000)`

- `#define TIM_Forced_Action_Active((uint16_t)0x0050)`

- `#define TIM_ForcedAction_InActive((uint16_t)0x0040)`

- `TIM_Input_Capture_Polarity`
• #define: `TIM_ICPolarity_Rising((uint16_t)0x0000)`

• #define: `TIM_ICPolarity_Falling((uint16_t)0x0002)`

• #define: `TIM_ICPolarity_BothEdge((uint16_t)0x000A)`

**TIM_Input_Capture_Prescaler**

• #define: `TIM_ICPSC_DIV1((uint16_t)0x0000)`
Capture performed each time an edge is detected on the capture input.

• #define: `TIM_ICPSC_DIV2((uint16_t)0x0004)`
Capture performed once every 2 events.

• #define: `TIM_ICPSC_DIV4((uint16_t)0x0008)`
Capture performed once every 4 events.

• #define: `TIM_ICPSC_DIV8((uint16_t)0x000C)`
Capture performed once every 8 events.

**TIM_Input_Capture_Selection**

• #define: `TIM_ICSelection_DirectTI((uint16_t)0x0001)`
TIM Input 1, 2, 3 or 4 is selected to be connected to IC1, IC2, IC3 or IC4, respectively.

• #define: `TIM_ICSelection_IndirectTI((uint16_t)0x0002)`
TIM Input 1, 2, 3 or 4 is selected to be connected to IC2, IC1, IC4 or IC3, respectively.

• #define: `TIM_ICSelection_TRC((uint16_t)0x0003)`
TIM Input 1, 2, 3 or 4 is selected to be connected to TRC.

**TIM_Internal_Trigger_Selection**

• #define: `TIM_TS_ITR0((uint16_t)0x0000)`
General-purpose timers (TIM)

- \#define: `TIM_TS_ITR1((uint16_t)0x0010)`
- \#define: `TIM_TS_ITR2((uint16_t)0x0020)`
- \#define: `TIM_TS_ITR3((uint16_t)0x0030)`
- \#define: `TIM_TS_TI1F_ED((uint16_t)0x0040)`
- \#define: `TIM_TS_TI1FP1((uint16_t)0x0050)`
- \#define: `TIM_TS_TI2FP2((uint16_t)0x0060)`
- \#define: `TIM_TS_ETRF((uint16_t)0x0070)`

**TIM_interrupt_sources**
- \#define: `TIM_IT_Update((uint16_t)0x0001)`
- \#define: `TIM_IT_CC1((uint16_t)0x0002)`
- \#define: `TIM_IT_CC2((uint16_t)0x0004)`
- \#define: `TIM_IT_CC3((uint16_t)0x0008)`
- \#define: `TIM_IT_CC4((uint16_t)0x0010)`
- \#define: `TIM_IT_COM((uint16_t)0x0020)`

- \#define: `TIM_IT_Trigger((uint16_t)0x0040)`

- \#define: `TIM_IT_Break((uint16_t)0x0080)`

**TIM_Legacy**

- \#define: `TIM_DMABurstLength_1Byte`\`TIM_DMABurstLength_1Transfer`

- \#define: `TIM_DMABurstLength_2Bytes`\`TIM_DMABurstLength_2Transfers`

- \#define: `TIM_DMABurstLength_3Bytes`\`TIM_DMABurstLength_3Transfers`

- \#define: `TIM_DMABurstLength_4Bytes`\`TIM_DMABurstLength_4Transfers`

- \#define: `TIM_DMABurstLength_5Bytes`\`TIM_DMABurstLength_5Transfers`

- \#define: `TIM_DMABurstLength_6Bytes`\`TIM_DMABurstLength_6Transfers`

- \#define: `TIM_DMABurstLength_7Bytes`\`TIM_DMABurstLength_7Transfers`

- \#define: `TIM_DMABurstLength_8Bytes`\`TIM_DMABurstLength_8Transfers`

- \#define: `TIM_DMABurstLength_9Bytes`\`TIM_DMABurstLength_9Transfers`
- #define: `TIM_DMABurstLength_10Bytes` `TIM_DMABurstLength_10Transfers`

- #define: `TIM_DMABurstLength_11Bytes` `TIM_DMABurstLength_11Transfers`

- #define: `TIM_DMABurstLength_12Bytes` `TIM_DMABurstLength_12Transfers`

- #define: `TIM_DMABurstLength_13Bytes` `TIM_DMABurstLength_13Transfers`

- #define: `TIM_DMABurstLength_14Bytes` `TIM_DMABurstLength_14Transfers`

- #define: `TIM_DMABurstLength_15Bytes` `TIM_DMABurstLength_15Transfers`

- #define: `TIM_DMABurstLength_16Bytes` `TIM_DMABurstLength_16Transfers`

- #define: `TIM_DMABurstLength_17Bytes` `TIM_DMABurstLength_17Transfers`

- #define: `TIM_DMABurstLength_18Bytes` `TIM_DMABurstLength_18Transfers`

`TIM_Lock_level`

- #define: `TIM_LOCKLevel_OFF` `((uint16_t)0x0000)`

- #define: `TIM_LOCKLevel_1` `((uint16_t)0x0100)`
#define TIM_LOCKLevel_2((uint16_t)0x0200)

#define TIM_LOCKLevel_3((uint16_t)0x0300)

**TIM_Master_Slave_Mode**

- #define TIM_MasterSlaveMode_Enable((uint16_t)0x0080)

- #define TIM_MasterSlaveMode_Disable((uint16_t)0x0000)

**TIM_One_Pulse_Mode**

- #define TIM_OPMode_Single((uint16_t)0x0008)

- #define TIM_OPMode_Repetitive((uint16_t)0x0000)

**TIM_OSSI_Off_State_Selection_for_Idle_mode_state**

- #define TIM_OSSIState_Enable((uint16_t)0x0400)

- #define TIM_OSSIState_Disable((uint16_t)0x0000)

**TIM_OSSR_Off_State_Selection_for_Run_mode_state**

- #define TIM_OSSRState_Enable((uint16_t)0x0800)

- #define TIM_OSSRState_Disable((uint16_t)0x0000)

**TIM_Output_Compare_and_PWM_modes**

- #define TIM_OCMode_Timing((uint16_t)0x0000)
- #define: `TIM_OCMode_Active((uint16_t)0x0010)`

- #define: `TIM_OCMode_Inactive((uint16_t)0x0020)`

- #define: `TIM_OCMode_Toggle((uint16_t)0x0030)`

- #define: `TIM_OCMode_PWM1((uint16_t)0x0060)`

- #define: `TIM_OCMode_PWM2((uint16_t)0x0070)`

**TIM_Output_Compare_Clear_State**

- #define: `TIM_OCClear_Enable((uint16_t)0x0080)`

- #define: `TIM_OCClear_Disable((uint16_t)0x0000)`

**TIM_Output_Compare_Fast_State**

- #define: `TIM_OCFast_Enable((uint16_t)0x0004)`

- #define: `TIM_OCFast_Disable((uint16_t)0x0000)`

**TIM_Output_Compare_Idle_State**

- #define: `TIM_OCIdleState_Set((uint16_t)0x0100)`

- #define: `TIM_OCIdleState_Reset((uint16_t)0x0000)`
**TIM\_Output\_Compare\_N\_Idle\_State**

- #define: `TIM\_OCNIdleState\_Set((uint16\_t)0x0200)`

- #define: `TIM\_OCNIdleState\_Reset((uint16\_t)0x0000)`

**TIM\_Output\_Compare\_N\_Polarity**

- #define: `TIM\_OCNPolarity\_High((uint16\_t)0x0000)`

- #define: `TIM\_OCNPolarity\_Low((uint16\_t)0x0008)`

**TIM\_Output\_Compare\_N\_State**

- #define: `TIM\_OutputNState\_Disable((uint16\_t)0x0000)`

- #define: `TIM\_OutputNState\_Enable((uint16\_t)0x0004)`

**TIM\_Output\_Compare\_Polarity**

- #define: `TIM\_OCPolarity\_High((uint16\_t)0x0000)`

- #define: `TIM\_OCPolarity\_Low((uint16\_t)0x0002)`

**TIM\_Output\_Compare\_Preload\_State**

- #define: `TIM\_OCPreload\_Enable((uint16\_t)0x0008)`

- #define: `TIM\_OCPreload\_Disable((uint16\_t)0x0000)`
**TIM_Output_Compare_State**

- #define: `TIM_OutputState_Disable((uint16_t)0x0000)`

- #define: `TIM_OutputState_Enable((uint16_t)0x0001)`

**TIM_Prescaler_Reload_Mode**

- #define: `TIM_PSCReloadMode_Update((uint16_t)0x0000)`

- #define: `TIM_PSCReloadMode_Immediate((uint16_t)0x0001)`

**TIM_Remap**

- #define: `TIM2_TIM8_TRGO((uint16_t)0x0000)`

- #define: `TIM2_ETH_PTP((uint16_t)0x0400)`

- #define: `TIM2_USBFS_SOF((uint16_t)0x0800)`

- #define: `TIM2_USBHS_SOF((uint16_t)0x0C00)`

- #define: `TIM5_GPIO((uint16_t)0x0000)`

- #define: `TIM5_LSI((uint16_t)0x0040)`

- #define: `TIM5_LSE((uint16_t)0x0080)`
- `#define: TIM5_RTC((uint16_t)0x00C0)`

- `#define: TIM11_GPIO((uint16_t)0x0000)`

- `#define: TIM11_HSE((uint16_t)0x0002)`

### TIM_Slave_Mode
- `#define: TIM_SlaveMode_Reset((uint16_t)0x0004)`

- `#define: TIM_SlaveMode_Gated((uint16_t)0x0005)`

- `#define: TIM_SlaveMode_Trigger((uint16_t)0x0006)`

- `#define: TIM_SlaveMode_External1((uint16_t)0x0007)`

### TIM_TIx_External_Clock_Source
- `#define: TIM_TIxExternalCLK1Source_TI1((uint16_t)0x0050)`

- `#define: TIM_TIxExternalCLK1Source_TI2((uint16_t)0x0060)`

- `#define: TIM_TIxExternalCLK1Source_TI1ED((uint16_t)0x0040)`

### TIM_Trigger_Output_Source
- `#define: TIM_TRGOSource_Reset((uint16_t)0x0000)`
- \#define: `TIM_TRGOSource_Enable((uint16_t)0x0010)`

- \#define: `TIM_TRGOSource_Update((uint16_t)0x0020)`

- \#define: `TIM_TRGOSource_OC1((uint16_t)0x0030)`

- \#define: `TIM_TRGOSource_OC1Ref((uint16_t)0x0040)`

- \#define: `TIM_TRGOSource_OC2Ref((uint16_t)0x0050)`

- \#define: `TIM_TRGOSource_OC3Ref((uint16_t)0x0060)`

- \#define: `TIM_TRGOSource_OC4Ref((uint16_t)0x0070)`

**TIM_Update_Source**

- \#define: `TIM_UpdateSource_Global((uint16_t)0x0000)`

  Source of update is the counter overflow/underflow or the setting of UG bit, or an update generation through the slave mode controller.

- \#define: `TIM_UpdateSource_Regular((uint16_t)0x0001)`

  Source of update is counter overflow/underflow.

### 25.4 TIM Programming Example

The example below explains how to configure the TIM3 to generate a PWM signal on CH1 with a frequency of 30 KHz and 50% duty cycle. For more examples about TIM configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\TIM\.
/* Includes --------------------------------------------*/
#include "stm32f2xx.h"

/* Private function prototypes */
void TIM3_PWMConfig(void);

/* Private functions */

/**
 * @brief Main program
 * @param None
 * @retval None
 */
int main(void)
{
    /* Configure TIM3 to generate a PWM signal on CH1 with
     * a frequency of 30 KHz and 50% duty cycle */
    TIM3_PWMConfig();

    while (1)
    {
    }
}

/**
 * @brief Configure the TIM3 in PWM mode.
 * @param None
 * @retval None
 */
void TIM3_PWMConfig(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_OCInitTypeDef  TIM_OCInitStructure;

    /* TIM3 IO configuration */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource6, GPIO_AF_TIM3);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_Init(GPIOC, &GPIO_InitStructure);

    /* TIM3 configuration */

    TIM3 is configured to generate PWM signal on CH1 with a
    frequency of 30 KHz and 50% duty cycle.

    TIM3 input clock (TIM3CLK) is equal to:
- PCLK1 if PCLK1 prescaler is 1
- 2 x PCLK1, otherwise

This example assumes that HCLK = 120 MHz and PCLK1 = 30 MHz
=> TIM3CLK = 2 x PCLK1 = 60 MHz

TIM3 signal frequency = TIM6CLK /((Prescaler + 1) * Period) = 30 KHz

Setting the Prescaler to 0, the Period = TIM6CLK / 30 KHz = 2000

TIM6 signal duty cycle = (CCR1 / ARR) * 100 = 50%
= (Pulse / Period) * 100 = 50%
=> with a Period of 2000, the Pulse = 1000

/* Enable TIM3 clock */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

/* Time base configuration */
TIM_TimeBaseStructure.TIM_Period = 2000;
TIM_TimeBaseStructure.TIM_Prescaler = 0;
TIM_TimeBaseStructure.TIM_ClockDivision = 0;
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);

/* Configure CH1 in PWM1 Mode */
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
TIM_OCInitStructure.TIM_Pulse = 1000;
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;

TIM_OC1Init(TIM3, &TIM_OCInitStructure);
TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable);
TIM_ARRPreloadConfig(TIM3, ENABLE);

/* Enable TIM3 counter */
TIM_Cmd(TIM3, ENABLE);
26 Universal synchronous asynchronous receiver transmitter (USART)

26.1 USART Firmware driver registers structures

26.1.1 USART_TypeDef

**USART_TypeDef** is defined in the stm32f2xx.h file and contains the USART registers definition.

**Data Fields**

- `__IO uint16_t SR`
- `uint16_t RESERVED0`
- `__IO uint16_t DR`
- `uint16_t RESERVED1`
- `__IO uint16_t BRR`
- `uint16_t RESERVED2`
- `__IO uint16_t CR1`
- `uint16_t RESERVED3`
- `__IO uint16_t CR2`
- `uint16_t RESERVED4`
- `__IO uint16_t CR3`
- `uint16_t RESERVED5`
- `__IO uint16_t GTPR`
- `uint16_t RESERVED6`

**Field Documentation**

- `__IO uint16_t USART_TypeDef::SR`  
  - USART Status register, Address offset: 0x00
- `uint16_t USART_TypeDef::RESERVED0`  
  - Reserved, 0x02
- `__IO uint16_t USART_TypeDef::DR`  
  - USART Data register, Address offset: 0x04
- `uint16_t USART_TypeDef::RESERVED1`  
  - Reserved, 0x06
- `__IO uint16_t USART_TypeDef::BRR`  
  - USART Baud rate register, Address offset: 0x08
- `uint16_t USART_TypeDef::RESERVED2`  
  - Reserved, 0x0A
- `__IO uint16_t USART_TypeDef::CR1`  
  - USART Control register 1, Address offset: 0x0C
- `uint16_t USART_TypeDef::RESERVED3`  
  - Reserved, 0x0E
- `__IO uint16_t USART_TypeDef::CR2`  
  - USART Control register 2, Address offset: 0x10
26.1.2 USART_InitTypeDef

`USART_InitTypeDef` is defined in the stm32f2xx_usart.h file and contains the USART initialization parameters.

**Data Fields**

- `__IO uint16_t USART_TypeDef::RESERVED4`
  - Reserved, 0x12
- `__IO uint16_t USART_TypeDef::CR3`
  - USART Control register 3, Address offset: 0x14
- `uint16_t USART_TypeDef::RESERVED5`
  - Reserved, 0x16
- `__IO uint16_t USART_TypeDef::GTPR`
  - USART Guard time and prescaler register, Address offset: 0x18
- `uint16_t USART_TypeDef::RESERVED6`
  - Reserved, 0x1A

**Field Documentation**

- `uint32_t USART_InitTypeDef::USART_BaudRate`
  - This member configures the USART communication baud rate. The baud rate is computed using the following formula:
    
    \[
    \text{IntegerDivider} = \left(\frac{\text{PCLKx}}{8 \times (\text{OVR8} + 1) \times \text{USART_InitStruct.USART_BaudRate}}\right)
    \]
    
    \[
    \text{FractionalDivider} = \left(\frac{\text{IntegerDivider}}{(\text{u32}) \text{IntegerDivider}}\right) \times 8 \times (\text{OVR8} + 1) + 0.5
    \]
    
    Where OVR8 is the "oversampling by 8 mode" configuration bit in the CR1 register.

- `uint16_t USART_InitTypeDef::USART_WordLength`
  - Specifies the number of data bits transmitted or received in a frame. This parameter can be a value of `USART_Word_LENGTH`.

- `uint16_t USART_InitTypeDef::USART_StopBits`
  - Specifies the number of stop bits transmitted. This parameter can be a value of `USART_Stop_Bits`.

- `uint16_t USART_InitTypeDef::USART_Parity`
  - Specifies the parity mode. This parameter can be a value of `USART_Parity`.

- `uint16_t USART_InitTypeDef::USART_Mode`
  - Specifies whether the Receive or Transmit mode is enabled or disabled. This parameter can be a value of `USART_MODE`.

- `uint16_t USART_InitTypeDef::USART_HardwareFlowControl`
  - Specifies whether the hardware flow control mode is enabled or disabled. This parameter can be a value of `USART_Hardware_Flow_Control`. 
26.1.3 USART_ClockInitTypeDef

**USART_ClockInitTypeDef** is defined in the stm32f2xx_usart.h file and contains the USART Synchronous mode initialization parameters.

**Data Fields**

- `uint16_t USART_Clock`
- `uint16_t USART_CPOL`
- `uint16_t USART_CPHA`
- `uint16_t USART_LastBit`

**Field Documentation**

- `uint16_t USART_ClockInitTypeDef::USART_Clock`
  - Specifies whether the USART clock is enabled or disabled. This parameter can be a value of **USART_Clock**
- `uint16_t USART_ClockInitTypeDef::USART_CPOL`
  - Specifies the steady state of the serial clock. This parameter can be a value of **USART_Clock_Polarity**
- `uint16_t USART_ClockInitTypeDef::USART_CPHA`
  - Specifies the clock transition on which the bit capture is made. This parameter can be a value of **USART_Clock_Phase**
- `uint16_t USART_ClockInitTypeDef::USART_LastBit`
  - Specifies whether the clock pulse corresponding to the last transmitted data bit (MSB) has to be output on the SCLK pin in synchronous mode. This parameter can be a value of **USART_Last_Bit**

26.2 USART Firmware driver API description

The following section lists the various functions of the USART library.

26.2.1 How to use this driver

1. Enable peripheral clock using the following functions
   - RCC_APB2PeriphClockCmd(RCC_APB2Periph_USARTx, ENABLE) for USART1 and
     USART6
   - RCC_APB1PeriphClockCmd(RCC_APB1Periph_USARTx, ENABLE) for
     USART2, USART3, UART4 or UART5.
2. According to the USART mode, enable the GPIO clocks using
   - RCC_AHB1PeriphClockCmd() function. (The I/O can be TX, RX, CTS, or/and SCLK).
3. Peripheral's alternate function:
   - Connect the pin to the desired peripherals Alternate Function (AF) using
     GPIO_PinAFConfig() function
   - Configure the desired pin in alternate function by:
     - GPIO_InitStruct->GPIO_Mode = GPIO_Mode_AF
   - Select the type, pull-up/pull-down and output speed via GPIO_PuPd,
     GPIO_OType and GPIO_Speed members
   - Call GPIO_Init() function
4. Program the Baud Rate, Word Length, Stop Bit, Parity, Hardware flow control and Mode (Receiver/Transmitter) using the USART_Init() function.

5. For synchronous mode, enable the clock and program the polarity, phase and last bit using the USART_ClockInit() function.

6. Enable the NVIC and the corresponding interrupt using the function USART_ITConfig() if you need to use interrupt mode.

7. When using the DMA mode
   – Configure the DMA using DMA_Init() function
   – Active the needed channel Request using USART_DMACmd() function

8. Enable the USART using the USART_Cmd() function.

9. Enable the DMA using the DMA_Cmd() function, when using DMA mode.

Refer to Multi-Processor, LIN, half-duplex, Smartcard, IrDA sub-sections for more details.

To reach higher communication baudrates, it is possible to enable the oversampling by 8 mode using the function USART_OverSampling8Cmd(). This function should be called after enabling the USART clock (RCC_APBxPeriphClockCmd()) and before calling the function USART_Init().

26.2.2 Initialization and configuration

Initialization and Configuration functions

This subsection provides a set of functions allowing to initialize the USART in asynchronous and in synchronous modes.

For the asynchronous mode only these parameters can be configured:

- Baud Rate
- Word Length
- Stop Bit
- Parity: If the parity is enabled, then the MSB bit of the data written in the data register is transmitted but is changed by the parity bit. Depending on the frame length defined by the M bit (8-bits or 9-bits), the possible USART frame formats are as listed in the following table: M bit PCE bit USART frame
  - 0 0 SB 8-bit data STB
  - 0 1 SB 7-bit data PB STB
  - 1 0 SB 8-bit data STB
  - 1 1 SB 8-bit data PB STB
- Hardware flow control
- Receiver/transmitter modes

The USART_Init() function follows the USART asynchronous configuration procedure (details for the procedure are available in reference manual (RM0033)).

For the synchronous mode in addition to the asynchronous mode parameters these parameters should be also configured:

- USART Clock Enabled
- USART polarity
- USART phase
- USART LastBit

These parameters can be configured using the USART_ClockInit() function.

- USART_DeInit()
- USART_Init()
- USART_StructInit()
- USART_ClockInit()
- USART_ClockStructInit()
- USART_Cmd()
- USART_SetPrescaler()
26.2.3 Data transfers

This subsection provides a set of functions allowing to manage the USART data transfers. During an USART reception, data shifts in least significant bit first through the RX pin. In this mode, the USART_DR register consists of a buffer (RDR) between the internal bus and the received shift register.

When a transmission is taking place, a write instruction to the USART_DR register stores the data in the TDR register and which is copied in the shift register at the end of the current transmission. The read access of the USART_DR register can be done using the USART_ReceiveData() function and returns the RDR buffered value. Whereas a write access to the USART_DR can be done using USART_SendData() function and stores the written data into TDR buffer.

- USART_SendDate()
- USART_ReceiveData()

26.2.4 Multiprocessor communication

This subsection provides a set of functions allowing to manage the USART multiprocessor communication.

For instance one of the USARTs can be the master, its TX output is connected to the RX input of the other USART. The others are slaves, their respective TX outputs are logically ANDed together and connected to the RX input of the master. USART multiprocessor communication is possible through the following procedure:

1. Program the Baud rate, Word length = 9 bits, Stop bits, Parity, Mode transmitter or Mode receiver and hardware flow control values using the USART_Init() function.
2. Configures the USART address using the USART_SetAddress() function.
3. Configures the wake up method (USART_WakeUp_IdleLine or USART_WakeUp_AddressMark) using USART_WakeUpConfig() function only for the slaves.
4. Enable the USART using the USART_Cmd() function.
5. Enter the USART slaves in mute mode using USART_ReceiverWakeUpCmd() function. The USART Slave exit from mute mode when receive the wake up condition.

- USART_SetAddress()
- USART_ReceiverWakeUpCmd()
- USART_WakeUpConfig()

26.2.5 LIN mode

This subsection provides a set of functions allowing to manage the USART LIN Mode communication.

In LIN mode, 8-bit data format with 1 stop bit is required in accordance with the LIN standard.

The USART IP supports only the LIN Master Synchronous Break send capability and LIN slave break detection capability.

- USART_OverSampling8Cmd()
- USART_OneBitMethodCmd()
13-bit break generation and 10/11 bit break detection USART LIN Master transmitter communication is possible through the following procedure:
   a. Program the Baud rate, Word length = 8bits, Stop bits = 1bit, Parity, Mode transmitter or Mode receiver and hardware flow control values using the USART_Init() function.
   b. Enable the USART using the USART_Cmd() function.
   c. Enable the LIN mode using the USART_LINCmd() function.
   d. Send the break character using USART_SendBreak() function.

USART LIN Master receiver communication is possible through the following procedure:
   a. Program the Baud rate, Word length = 8bits, Stop bits = 1bit, Parity, Mode transmitter or Mode receiver and hardware flow control values using the USART_Init() function.
   b. Enable the USART using the USART_Cmd() function.
   c. Configures the break detection length using the USART_LINBreakDetectLengthConfig() function.
   d. Enable the LIN mode using the USART_LINCmd() function.

In LIN mode, the following bits must be kept cleared:
   • CLKEN in the USART_CR2 register,
   • STOP[1:0], SCEN, HDSEL and IREN in the USART_CR3 register.

26.2.6 Halfduplex mode

This subsection provides a set of functions allowing to manage the USART Half-duplex communication.

The USART can be configured to follow a single-wire half-duplex protocol where the TX and RX lines are internally connected. USART Half duplex communication is possible through the following procedure:

1. Program the Baud rate, Word length, Stop bits, Parity, Mode transmitter or Mode receiver and hardware flow control values using the USART_Init() function.
2. Configures the USART address using the USART_SetAddress() function.
3. Enable the USART using the USART_Cmd() function.
4. Enable the half duplex mode using USART_HalfDuplexCmd() function.

The RX pin is no longer used

In Half-duplex mode the following bits must be kept cleared:
   • LINEN and CLKEN bits in the USART_CR2 register.
   • SCEN and IREN bits in the USART_CR3 register.
26.2.7 Smartcard mode

This subsection provides a set of functions allowing to manage the USART Smartcard communication.

The Smartcard interface is designed to support asynchronous protocol Smartcards as defined in the ISO 7816-3 standard.

The USART can provide a clock to the smartcard through the SCLK output. In smartcard mode, SCLK is not associated to the communication but is simply derived from the internal peripheral input clock through a 5-bit prescaler.

Smartcard communication is possible through the following procedure:

1. Configures the Smartcard Prescaler using the USART_SetPrescaler() function.
2. Configures the Smartcard Guard Time using the USART_SetGuardTime() function.
3. Program the USART clock using the USART_ClockInit() function as following:
   - USART Clock enabled
   - USART CPOL Low
   - USART CPHA on first edge
   - USART Last Bit Clock Enabled
4. Program the Smartcard interface using the USART_Init() function as following:
   - Word Length = 9 Bits
   - 1.5 Stop Bit
   - Even parity
   - BaudRate = 12096 baud
   - Hardware flow control disabled (RTS and CTS signals)
   - Tx and Rx enabled
5. Optionally you can enable the parity error interrupt using the USART_ITConfig() function
6. Enable the USART using the USART_Cmd() function.
7. Enable the Smartcard NACK using the USART_SmartCardNACKCmd() function.
8. Enable the Smartcard interface using the USART_SmartCardCmd() function.

Please refer to the ISO 7816-3 specification for more details.

It is also possible to choose 0.5 stop bit for receiving but it is recommended to use 1.5 stop bits for both transmitting and receiving to avoid switching between the two configurations.

In smartcard mode, the following bits must be kept cleared:

- LINEN bit in the USART_CR2 register.
- HDSEL and IREN bits in the USART_CR3 register.

Smartcard mode is available on USART peripherals only (not available on UART4 and UART5 peripherals).
26.2.8 IrDA mode

This subsection provides a set of functions allowing to manage the USART IrDA communication.

IrDA is a half duplex communication protocol. If the Transmitter is busy, any data on the IrDA receive line will be ignored by the IrDA decoder and if the Receiver is busy, data on the TX from the USART to IrDA will not be encoded by IrDA. While receiving data, transmission should be avoided as the data to be transmitted could be corrupted.

IrDA communication is possible through the following procedure:

1. Program the Baud rate, Word length = 8 bits, Stop bits, Parity, Transmitter/Receiver modes and hardware flow control values using the USART_Init() function.
2. Enable the USART using the USART_Cmd() function.
3. Configures the IrDA pulse width by configuring the prescaler using the USART_SetPrescaler() function.
4. Configures the IrDA USART_IrDAMode_LowPower or USART_IrDAMode_Normal mode using the USART_IrDAConfig() function.
5. Enable the IrDA using the USART_IrDACmd() function.

A pulse of width less than two and greater than one PSC period(s) may or may not be rejected.

The receiver set up time should be managed by software. The IrDA physical layer specification specifies a minimum of 10 ms delay between transmission and reception (IrDA is a half duplex protocol).

In IrDA mode, the following bits must be kept cleared:

- LINEN, STOP and CLKEN bits in the USART_CR2 register.
- SCEN and HDSEL bits in the USART_CR3 register.

26.2.9 DMA transfers management

- USART_DMACmd()

26.2.10 Interrupt and flag management

This subsection provides a set of functions allowing to configure the USART Interrupts sources, DMA channels requests and check or clear the flags or pending bits status.
The user should identify which mode will be used in his application to manage the communication: Polling mode, Interrupt mode or DMA mode.

**Polling Mode**

In Polling Mode, the SPI communication can be managed by 10 flags:

1. USART_FLAG_TXE : to indicate the status of the transmit buffer register
2. USART_FLAG_RXNE : to indicate the status of the receive buffer register
3. USART_FLAG_TC : to indicate the status of the transmit operation
4. USART_FLAG_IDLE : to indicate the status of the Idle Line
5. USART_FLAG_CTS : to indicate the status of the nCTS input
6. USART_FLAG_LBD : to indicate the status of the LIN break detection
7. USART_FLAG_NE : to indicate if a noise error occur
8. USART_FLAG_FE : to indicate if a frame error occur
9. USART_FLAG_PEE : to indicate if a parity error occur
10. USART_FLAG_ORE : to indicate if an Overrun error occur

In this Mode it is advised to use the following functions:

- FlagStatus USART_GetFlagStatus(USART_TypeDef USARTx, uint16_t USART_FLAG);
- void USART_ClearFlag(USART_TypeDef USARTx, uint16_t USART_FLAG);

**Interrupt Mode**

In Interrupt Mode, the USART communication can be managed by 10 pending bits and 8 interrupt sources:

- Pending Bits:
  a. USART_IT_TXE : to indicate the status of the transmit buffer register
  b. USART_IT_RXNE : to indicate the status of the receive buffer register
  c. USART_IT_TC : to indicate the status of the transmit operation
  d. USART_IT_IDLE : to indicate the status of the Idle Line
  e. USART_IT_CTS : to indicate the status of the nCTS input
  f. USART_IT_LBD : to indicate the status of the LIN break detection
  g. USART_IT_NE : to indicate if a noise error occur
  h. USART_IT_FE : to indicate if a frame error occur
  i. USART_IT_PEE : to indicate if a parity error occur
  j. USART_IT_ORE : to indicate if an Overrun error occur

- Interrupt Source: Some parameters are coded in order to use them as interrupt source or as pending bits. In this Mode it is advised to use the following functions: void USART_ITConfig(USART_TypeDef USARTx, uint16_t USART_IT, FunctionalState NewState); ITStatus USART_GetITStatus(USART_TypeDef USARTx, uint16_t USART_IT); void USART_ClearITPendingBit(USART_TypeDef USARTx, uint16_t USART_IT);
  a. USART_IT_TXE : specifies the interrupt source for the Tx buffer empty interrupt.
  b. USART_IT_RXNE : specifies the interrupt source for the Rx buffer not empty interrupt.
  c. USART_IT_TC : specifies the interrupt source for the Transmit complete interrupt.
  d. USART_IT_IDLE : specifies the interrupt source for the Idle Line interrupt.
  e. USART_IT_CTS : specifies the interrupt source for the CTS interrupt.
  f. USART_IT_LBD : specifies the interrupt source for the LIN break detection interrupt.
  g. USART_IT_PEE : specifies the interrupt source for the parity error interrupt.
  h. USART_IT_ERR : specifies the interrupt source for the errors interrupt.

**DMA Mode**

In DMA Mode, the USART communication can be managed by 2 DMA Channel requests:
In this Mode it is advised to use the function void
USART_DMACmd(USART_TypeDef USARTx, uint16_t USART_DMAReq,
FunctionState NewState);

- **USART_DMAReq_Tx**: specifies the Tx buffer DMA transfer request
- **USART_DMAReq_Rx**: specifies the Rx buffer DMA transfer request.

- **USART_ITConfig()**
- **USART_GetFlagStatus()**
- **USART_ClearFlag()**
- **USART_GetITStatus()**
- **USART_ClearITPendingBit()**

### 26.2.11 Initialization and configuration functions

#### 26.2.11.1 USART_DeInit

**Function Name**

```c
void USART_DeInit ( USART_TypeDef * USARTx)
```

**Function Description**

Deinitializes the USARTx peripheral registers to their default reset values.

**Parameters**

- **USARTx**: where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.

**Return values**

- None.

**Notes**

- None.

#### 26.2.11.2 USART_Init

**Function Name**

```c
void USART_Init ( USART_TypeDef * USARTx, USART_InitTypeDef * USART_InitStruct)
```

**Function Description**

Initializes the USARTx peripheral according to the specified parameters in the USART_InitStruct.

**Parameters**

- **USARTx**: where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- **USART_InitStruct**: pointer to a USART_InitTypeDef structure that contains the configuration information for the specified USART peripheral.

**Return values**

- None.

**Notes**

- None.
26.2.11.3 USART_StructInit

Function Name: void USART_StructInit (USART_InitTypeDef *USART_InitStruct)
Function Description: Fills each USART_InitStruct member with its default value.
Parameters:
- USART_InitStruct: pointer to a USART_InitTypeDef structure which will be initialized.
Return values:
- None.
Notes:
- None.

26.2.11.4 USART_ClockInit

Function Name: void USART_ClockInit (USART_TypeDef *USARTx, USART_ClockInitTypeDef *USART_ClockInitStruct)
Function Description: Initializes the USARTx peripheral Clock according to the specified parameters in the USART_ClockInitStruct.
Parameters:
- USARTx: where x can be 1, 2, 3 or 6 to select the USART peripheral.
- USART_ClockInitStruct: pointer to a USART_ClockInitTypeDef structure that contains the configuration information for the specified USART peripheral.
Return values:
- None.
Notes:
- The Smart Card and Synchronous modes are not available for UART4 and UART5.

26.2.11.5 USART_ClockStructInit

Function Name: void USART_ClockStructInit (USART_ClockInitTypeDef *USART_ClockInitStruct)
Function Description: Fills each USART_ClockInitStruct member with its default value.
### Parameters
- **USART_ClockInitStruct**: pointer to a `USART_ClockInitTypeDef` structure which will be initialized.

### Return values
- None.

### Notes
- None.

### 26.2.11.6 USART_Cmd

**Function Name**
```c
void USART_Cmd ( USART_TypeDef * USARTx, FunctionalState NewState)
```

**Function Description**
Enables or disables the specified USART peripheral.

**Parameters**
- **USARTx**: where `x` can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- **NewState**: new state of the USARTx peripheral. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.

**Notes**
- None.

### 26.2.11.7 USART_SetPrescaler

**Function Name**
```c
void USART_SetPrescaler ( USART_TypeDef * USARTx,
                        uint8_t USART_Prescaler)
```

**Function Description**
Sets the system clock prescaler.

**Parameters**
- **USARTx**: where `x` can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- **USART_Prescaler**: specifies the prescaler clock.

**Return values**
- None.

**Notes**
- The function is used for IrDA mode with UART4 and UART5.
### 26.2.11.8 USART_OverSampling8Cmd

Function Name: \texttt{void USART\_OverSampling8Cmd (USART\_TypeDef * USARTx, FunctionalState NewState)}

Function Description: Enables or disables the USART's 8x oversampling mode.

Parameters:
- \texttt{USARTx}: where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- \texttt{NewState}: new state of the USART 8x oversampling mode. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: This function has to be called before calling \texttt{USART\_Init()} function in order to have correct baudrate Divider value.

### 26.2.11.9 USART\_OneBitMethodCmd

Function Name: \texttt{void USART\_OneBitMethodCmd (USART\_TypeDef * USARTx, FunctionalState NewState)}

Function Description: Enables or disables the USART's one bit sampling method.

Parameters:
- \texttt{USARTx}: where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- \texttt{NewState}: new state of the USART one bit sampling method. This parameter can be: ENABLE or DISABLE.

Return values: None.

Notes: None.

### 26.2.12 Data transfers functions

#### 26.2.12.1 USART\_SendData

Function Name: \texttt{void USART\_SendData (USART\_TypeDef * USARTx, uint16_t Data)}

Function Description: Transmits single data through the USARTx peripheral.
26.2.12.2 USART_ReceiveData

**Function Name**

```c
uint16_t USART_ReceiveData ( USART_TypeDef * USARTx)
```

**Function Description**

Returns the most recent received data by the USARTx peripheral.

**Parameters**

- **USARTx** : where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.

**Return values**

- **None.**

**Notes**

- **None.**

26.2.13 MultiProcessor communication functions

26.2.13.1 USART_SetAddress

**Function Name**

```c
void USART_SetAddress ( USART_TypeDef * USARTx,
                        uint8_t USART_Address)
```

**Function Description**

Sets the address of the USART node.

**Parameters**

- **USARTx** : where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- **USART_Address** : Indicates the address of the USART node.

**Return values**

- **None.**

**Notes**

- **None.**
26.2.13.2 USART_ReceiverWakeUpCmd

Function Name  
void USART_ReceiverWakeUpCmd (USART_TypeDef * USARTx, FunctionalState NewState)

Function Description  
Determines if the USART is in mute mode or not.

Parameters  
- **USARTx**: where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- **NewState**: new state of the USART mute mode. This parameter can be: ENABLE or DISABLE.

Return values  
- None.

Notes  
- None.

26.2.13.3 USART_WakeUpConfig

Function Name  
void USART_WakeUpConfig (USART_TypeDef * USARTx, uint16_t USART_WakeUp)

Function Description  
Selects the USART WakeUp method.

Parameters  
- **USARTx**: where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- **USART_WakeUp**: specifies the USART wakeup method. This parameter can be one of the following values:
  - **USART_WakeUp_IdleLine**: WakeUp by an idle line detection
  - **USART_WakeUp_AddressMark**: WakeUp by an address mark

Return values  
- None.

Notes  
- None.

26.2.14 LIN mode functions

26.2.14.1 USART_LINBreakDetectLengthConfig
### 26.2.14.2 USART_LINCmd

**Function Name**

```c
void USART_LINCmd ( USART_TypeDef * USARTx, FunctionalState NewState)
```

**Function Description**

Enables or disables the USART's LIN mode.

**Parameters**

- `USARTx`: where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- `NewState`: new state of the USART LIN mode. This parameter can be: ENABLE or DISABLE.

**Return values**

- None.

**Notes**

- None.

---

### 26.2.14.3 USART_SendBreak

**Function Name**

```c
void USART_SendBreak ( USART_TypeDef * USARTx)
```

**Function Description**

Transmits break characters.

**Parameters**

- `USARTx`: where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
### 26.2.15 Halfduplex mode function

#### 26.2.15.1 USART_HalfDuplexCmd

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void USART_HalfDuplexCmd (USART_TypeDef * USARTx, FunctionalState NewState)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Enables or disables the USART’s Half Duplex communication.</td>
</tr>
</tbody>
</table>
| Parameters | • USARTx : where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.  
• NewState : new state of the USART Communication. This parameter can be: ENABLE or DISABLE. |
| Return values | • None. |
| Notes | • None. |

### 26.2.16 Smartcard mode functions

#### 26.2.16.1 USART_SetGuardTime

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void USART_SetGuardTime (USART_TypeDef * USARTx, uint8_t USART_GuardTime)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Sets the specified USART guard time.</td>
</tr>
</tbody>
</table>
| Parameters | • USARTx : where x can be 1, 2, 3 or 6 to select the USART or UART peripheral.  
• USART_GuardTime : specifies the guard time. |
| Return values | • None. |
| Notes | • None. |
26.2.16.2 USART_SmartCardCmd

Function Name  void USART_SmartCardCmd (USART_TypeDef * USARTx, FunctionalState NewState)
Function Description  Enables or disables the USART's Smart Card mode.
Parameters  
- **USARTx**: where x can be 1, 2, 3 or 6 to select the USART or UART peripheral.
- **NewState**: new state of the Smart Card mode. This parameter can be: ENABLE or DISABLE.
Return values  
- None.
Notes  
- None.

26.2.16.3 USART_SmartCardNACKCmd

Function Name  void USART_SmartCardNACKCmd (USART_TypeDef * USARTx, FunctionalState NewState)
Function Description  Enables or disables NACK transmission.
Parameters  
- **USARTx**: where x can be 1, 2, 3 or 6 to select the USART or UART peripheral.
- **NewState**: new state of the NACK transmission. This parameter can be: ENABLE or DISABLE.
Return values  
- None.
Notes  
- None.

26.2.17 IrDA mode functions

26.2.17.1 USART_IrDAConfig

Function Name  void USART_IrDAConfig (USART_TypeDef * USARTx, uint16_t USART_IrDAMode)
Function Description  Configures the USART's IrDA interface.
Parameters
- **USARTx** : where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- **USART_IrDAMode** : specifies the IrDA mode. This parameter can be one of the following values:
  - **USART_IrDAMode_LowPower** :
  - **USART_IrDAMode_Normal** :

Return values
- None.

Notes
- None.

### 26.2.17.2 USART_IrDACmd

**Function Name**

```c
void USART_IrDACmd ( USART_TypeDef * USARTx, FunctionalState NewState)
```

**Function Description**

Enables or disables the USART's IrDA interface.

**Parameters**
- **USARTx** : where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- **NewState** : new state of the IrDA mode. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.

**Notes**
- None.

### 26.2.18 DMA transfers management functions

#### 26.2.18.1 USART_DMACmd

**Function Name**

```c
void USART_DMACmd ( USART_TypeDef * USARTx, uint16_t USART_DMAReq, FunctionalState NewState)
```

**Function Description**

Enables or disables the USART's DMA interface.

**Parameters**
- **USARTx** : where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- **USART_DMAReq** : specifies the DMA request. This parameter can be any combination of the following values:
  - **USART_DMAReq_Tx** : USART DMA transmit request
  - **USART_DMAReq_Rx** : USART DMA receive request
- **NewState** : new state of the DMA Request sources. This
26.2.19 Interrupt and flag management functions

26.2.19.1 USART_ITConfig

**Function Name**
void USART_ITConfig (USART_TypeDef * USARTx, uint16_t USART_IT, FunctionalState NewState)

**Function Description**
Enables or disables the specified USART interrupts.

**Parameters**
- **USARTx**: where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- **USART_IT**: specifies the USART interrupt sources to be enabled or disabled. This parameter can be one of the following values:
  - **USART_IT_CTS**: CTS change interrupt
  - **USART_IT_LBD**: LIN Break detection interrupt
  - **USART_IT_TXE**: Transmit Data Register empty interrupt
  - **USART_IT_TC**: Transmission complete interrupt
  - **USART_IT_RXNE**: Receive Data register not empty interrupt
  - **USART_IT_IDLE**: Idle line detection interrupt
  - **USART_IT_PE**: Parity Error interrupt
  - **USART_IT_ERR**: Error interrupt(Frame error, noise error, overrun error)
- **NewState**: new state of the specified USARTx interrupts. This parameter can be: ENABLE or DISABLE.

**Return values**
- None.

**Notes**
- None.

26.2.19.2 USART_GetFlagStatus

**Function Name**
FlagStatus USART_GetFlagStatus (USART_TypeDef * USARTx, uint16_t USART_FLAG)
Function Description
Checks whether the specified USART flag is set or not.

Parameters
- USARTx : where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- USART_FLAG : specifies the flag to check. This parameter can be one of the following values:
  - USART_FLAG_CTS : CTS Change flag (not available for UART4 and UART5)
  - USART_FLAG_LBD : LIN Break detection flag
  - USART_FLAG_TXE : Transmit data register empty flag
  - USART_FLAG_TC : Transmission Complete flag
  - USART_FLAG_RXNE : Receive data register not empty flag
  - USART_FLAG_IDLE : Idle Line detection flag
  - USART_FLAG_ORE : OverRun Error flag
  - USART_FLAG_NE : Noise Error flag
  - USART_FLAG_FE : Framing Error flag
  - USART_FLAG_PE : Parity Error flag

Return values
- The new state of USART_FLAG (SET or RESET).

Notes
- None.

26.2.19.3 USART_ClearFlag

Function Name
void USART_ClearFlag ( USART_TypeDef * USARTx, uint16_t USART_FLAG)

Function Description
Clears the USARTx's pending flags.

Parameters
- USARTx : where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- USART_FLAG : specifies the flag to clear. This parameter can be any combination of the following values:
  - USART_FLAG_CTS : CTS Change flag (not available for UART4 and UART5).
  - USART_FLAG_LBD : LIN Break detection flag.
  - USART_FLAG_TC : Transmission Complete flag.
  - USART_FLAG_RXNE : Receive data register not empty flag.

Return values
- None.

Notes
- PE (Parity error), FE (Framing error), NE (Noise error), ORE (OverRun error) and IDLE (Idle line detected) flags are cleared by software sequence: a read operation to USART_SR register (USART_GetFlagStatus()) followed by a read operation to USART_DR register (USART_ReceiveData()).
26.2.19.4 USART_GetITStatus

Function Name

ITStatus USART_GetITStatus ( USART_TypeDef * USARTx, uint16_t USART_IT)

Function Description
Checks whether the specified USART interrupt has occurred or not.

Parameters

- **USARTx**: where x can be 1, 2, 3, 4, 5 or 6 to select the USART or UART peripheral.
- **USART_IT**: specifies the USART interrupt source to check. This parameter can be one of the following values:
  - **USART_IT_CTS**: CTS change interrupt (not available for UART4 and UART5)
  - **USART_IT_LBD**: LIN Break detection interrupt
  - **USART_IT_TXE**: Transmit Data Register empty interrupt
  - **USART_IT_TC**: Transmission complete interrupt
  - **USART_IT_RXNE**: Receive Data register not empty interrupt
  - **USART_IT_IDLE**: Idle line detection interrupt
  - **USART_IT_ORE**: OverRun Error interrupt
  - **USART_IT_NE**: Noise Error interrupt
  - **USART_IT_FE**: Framing Error interrupt
  - **USART_IT_PE**: Parity Error interrupt

Return values

- The new state of USART_IT (SET or RESET).

Notes

- None.

26.2.19.5 USART_ClearITPendingBit
26.3 USART Firmware driver defines

26.3.1 USART Firmware driver defines

USART

USART_Clock

- #define: **USART_Clock_Enable((uint16_t)0x0800)**

- #define: **USART_Clock_Disable((uint16_t)0x0000)**

USART_Clock_Phase
• #define: `USART_CPHA_1Edge((uint16_t)0x0000)`

• #define: `USART_CPHA_2Edge((uint16_t)0x0200)`

`USART_Clock_Polarity`
• #define: `USART_CPOL_Low((uint16_t)0x0000)`

• #define: `USART_CPOL_High((uint16_t)0x0400)`

`USART_DMA_Requests`
• #define: `USART_DMAReq_Tx((uint16_t)0x0080)`

• #define: `USART_DMAReq_Rx((uint16_t)0x0040)`

`USART_Flags`
• #define: `USART_FLAG_CTS((uint16_t)0x0200)`

• #define: `USART_FLAG_LBD((uint16_t)0x0100)`

• #define: `USART_FLAG_TXE((uint16_t)0x0080)`

• #define: `USART_FLAG_TC((uint16_t)0x0040)`

• #define: `USART_FLAG_RXNE((uint16_t)0x0020)`
- #define: `USART_FLAG_IDLE((uint16_t)0x0010)`

- #define: `USART_FLAG_ORE((uint16_t)0x0008)`

- #define: `USART_FLAG_NE((uint16_t)0x0004)`

- #define: `USART_FLAG_FE((uint16_t)0x0002)`

- #define: `USART_FLAG_PE((uint16_t)0x0001)`

**USART_Hardware_Flow_Control**

- #define: `USART_HardwareFlowControl_None((uint16_t)0x0000)`

- #define: `USART_HardwareFlowControl_RTS((uint16_t)0x0100)`

- #define: `USART_HardwareFlowControl_CTS((uint16_t)0x0200)`

- #define: `USART_HardwareFlowControl_RTS_CTS((uint16_t)0x0300)`

**USART_Interrupt_definition**

- #define: `USART_IT_PE((uint16_t)0x0028)`

- #define: `USART_IT_TXE((uint16_t)0x0727)`

- #define: `USART_IT_TC((uint16_t)0x0626)`
- #define: `USART_IT_RXNE((uint16_t)0x0525)`

- #define: `USART_IT_IDLE((uint16_t)0x0424)`

- #define: `USART_IT_LBD((uint16_t)0x0846)`

- #define: `USART_IT_CTS((uint16_t)0x096A)`

- #define: `USART_IT_ERR((uint16_t)0x0060)`

- #define: `USART_IT_ORE((uint16_t)0x0360)`

- #define: `USART_IT_NE((uint16_t)0x0260)`

- #define: `USART_IT_FE((uint16_t)0x0160)`

**USART_IrDA_Low_Power**

- #define: `USART_IrDAMode_LowPower((uint16_t)0x0004)`

- #define: `USART_IrDAMode_Normal((uint16_t)0x0000)`

**USART_Last_Bit**

- #define: `USART_LastBit_Disable((uint16_t)0x0000)`
- #define: `USART_LastBit_Enable(uint16_t)0x0100`

**USART_LIN_Break_Detection_Length**
- #define: `USART_LINBreakDetectLength_10b(uint16_t)0x0000`
- #define: `USART_LINBreakDetectLength_11b(uint16_t)0x0020`

**USART_Mode**
- #define: `USART_Mode_Rx(uint16_t)0x0004`
- #define: `USART_Mode_Tx(uint16_t)0x0008`

**USART_Parity**
- #define: `USART_Parity_No(uint16_t)0x0000`
- #define: `USART_Parity_Even(uint16_t)0x0400`
- #define: `USART_Parity_Odd(uint16_t)0x0600`

**USART_Stop_Bits**
- #define: `USART_StopBits_1(uint16_t)0x0000`
- #define: `USART_StopBits_0_5(uint16_t)0x1000`
- #define: `USART_StopBits_2(uint16_t)0x2000`
#define: **USART_StopBits_1_5((uint16_t)0x3000)**

**_USART_WakeUp_methods**
- #define: **USART_WakeUp_IdleLine((uint16_t)0x0000)**
- #define: **USART_WakeUp_AddressMark((uint16_t)0x0800)**

**_USART_Word_Length**
- #define: **USART_WordLength_8b((uint16_t)0x0000)**
- #define: **USART_WordLength_9b((uint16_t)0x1000)**

## 26.4 USART Programming Example

The example below explains how to initialize the USART, and associated resources, and send continuously 8-bit data. For more examples about SPI configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\SPI\.

```c
/* Includes -------------------------------------------------*/
#include "stm32f2xx.h"

/* Private function prototypes */
static void USART3_Config(void);

/* Private functions */

/**
 * @brief Main program
 * @param None
 * @retval None
 */
int main(void)
{
    /* USART3 configuration */
    USART3_Config();
```

```c
/* Private function prototypes */
static void USART3_Config(void);

/* Private functions */

/**
 * @brief Main program
 * @param None
 * @retval None
 */
int main(void)
{
    /* USART3 configuration */
    USART3_Config();
```
while (1) {
    /* Send dummy data */
    USART_SendData(USART3, 0xA5);

    /* Loop until the end of transmission */
    while (USART_GetFlagStatus(USART3, USART_FLAG_TC) == RESET) {} 
}

/**
 * @brief  Configures the USART3 Peripheral.
 * @param  None
 * @retval None
 */
static void USART3_Config(void) {
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef  USART_InitStructure;

    /* USART IOs configuration ******************************************/
    /* Enable GPIOC clock */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

    /* Connect PC10 to USART3_Tx */
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource10, GPIO_AF_USART3);

    /* Connect PC11 to USART3_Rx*/
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource11, GPIO_AF_USART3);

    /* Configure USART3_Tx and USART3_Rx as alternate function */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10 | GPIO_Pin_11;
    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(GPIOC, &GPIO_InitStructure);

    /* USART configuration ******************************************/
    /* USART3 configured as follow:
    - BaudRate = 115200 baud
    - Word Length = 8 Bits
    - One Stop Bit
    - No parity
    - Hardware flow control disabled (RTS and CTS signals)
    - Receive and transmit enabled
    */

    /* Enable USART3 clock */
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART3, ENABLE);
    USART_InitStructure.USART_BaudRate = 115200;

    USART_InitStructure.USART_BaudRate = 115200;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_No;
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;

USART_Init(USART3, &USART_InitStructure);
/* Enable USART */
USART_Cmd(USART3, ENABLE);
27 Window watchdog (WWDG)

27.1 WWDG Firmware driver registers structures

27.1.1 WWDG_TypeDef

WWDG_TypeDef is defined in the stm32f2xx.h file and contains the WWDG registers definition.

Data Fields

- __IO uint32_t CR
- __IO uint32_t CFR
- __IO uint32_t SR

Field Documentation

- __IO uint32_t WWDG_TypeDef::CR
  - WWDG Control register, Address offset: 0x00
- __IO uint32_t WWDG_TypeDef::CFR
  - WWDG Configuration register, Address offset: 0x04
- __IO uint32_t WWDG_TypeDef::SR
  - WWDG Status register, Address offset: 0x08

27.2 WWDG Firmware driver API description

The following section lists the various functions of the WWDG library.

WWDG features

Once enabled the WWDG generates a system reset on expiry of a programmed time period, unless the program refreshes the counter (downcounter) before to reach 0x3F value (i.e. a reset is generated when the counter value rolls over from 0x40 to 0x3F).

An MCU reset is also generated if the counter value is refreshed before the counter has reached the refresh window value. This implies that the counter must be refreshed in a limited window.

Once enabled the WWDG cannot be disabled except by a system reset.

WWDGRST flag in RCC_CSR register can be used to inform when a WWDG reset occurs.

The WWDG counter input clock is derived from the APB clock divided by a programmable prescaler.

WWDG counter clock = PCLK1 / Prescaler

WWDG timeout = (WWDG counter clock) (counter value)

Min-max timeout value @30 MHz(PCLK1): ~136.5 us / ~69.9 ms
How to use this driver

1. Enable WWDG clock using RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE) function
2. Configure the WWDG prescaler using WWDG_SetPrescaler() function
3. Configure the WWDG refresh window using WWDG_SetWindowValue() function
4. Set the WWDG counter value and start it using WWDG_Enable() function. When the WWDG is enabled the counter value should be configured to a value greater than 0x40 to prevent generating an immediate reset.
5. Optionally you can enable the Early wakeup interrupt which is generated when the counter reach 0x40. Once enabled this interrupt cannot be disabled except by a system reset.
6. Then the application program must refresh the WWDG counter at regular intervals during normal operation to prevent an MCU reset, using WWDG_SetCounter() function. This operation must occur only when the counter value is lower than the refresh window value, programmed using WWDG_SetWindowValue().

Prescaler, Refresh window and Counter configuration functions

- WWDG_DeInit()
- WWDG_SetPrescaler()
- WWDG_SetWindowValue()
- WWDG_EnableIT()
- WWDG_SetCounter()

WWDG activation functions

- WWDG_Enable()

Interrupts and flags management functions

- WWDG_GetFlagStatus()
- WWDG_ClearFlag()

27.2.1 Prescaler, Refresh window and Counter configuration functions

27.2.1.1 WWDG_DeInit

<table>
<thead>
<tr>
<th>Function Name</th>
<th>void WWDG_DeInit ( void )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Description</td>
<td>Deinitializes the WWDG peripheral registers to their default reset values.</td>
</tr>
<tr>
<td>Parameters</td>
<td>None.</td>
</tr>
<tr>
<td>Return values</td>
<td>None.</td>
</tr>
<tr>
<td>Notes</td>
<td>None.</td>
</tr>
</tbody>
</table>

27.2.1.2 WWDG_SetPrescaler
Window watchdog (WWDG)

Function Name: void WWDG_SetPrescaler ( uint32_t WWDG_Prescaler)
Function Description: Sets the WWDG Prescaler.
Parameters:
- **WWDG_Prescaler**: specifies the WWDG Prescaler. This parameter can be one of the following values:
  - **WWDG_Prescaler_1**: WWDG counter clock = (PCLK1/4096)/1
  - **WWDG_Prescaler_2**: WWDG counter clock = (PCLK1/4096)/2
  - **WWDG_Prescaler_4**: WWDG counter clock = (PCLK1/4096)/4
  - **WWDG_Prescaler_8**: WWDG counter clock = (PCLK1/4096)/8

Return values:
- None.

Notes:
- None.

27.2.1.3 WWDG_SetWindowValue

Function Name: void WWDG_SetWindowValue ( uint8_t WindowValue)
Function Description: Sets the WWDG window value.
Parameters:
- **WindowValue**: specifies the window value to be compared to the downcounter. This parameter value must be lower than 0x80.

Return values:
- None.

Notes:
- None.

27.2.1.4 WWDG_EnableIT

Function Name: void WWDG_EnableIT ( void )
Function Description: Enables the WWDG Early Wakeup interrupt (EWI).
Parameters:
- None.

Return values:
- None.

Notes:
- Once enabled this interrupt cannot be disabled except by a
27.2.1.5  WWDG_SetCounter

Function Name  void WWDG_SetCounter ( uint8_t Counter)
Function Description  Sets the WWDG counter value.
Parameters  •  Counter : specifies the watchdog counter value. This parameter must be a number between 0x40 and 0x7F (to prevent generating an immediate reset)
Return values  •  None.
Notes  •  None.

27.2.2  WWDG activation function

27.2.2.1  WWDG_Enable

Function Name  void WWDG_Enable ( uint8_t Counter)
Function Description  Enables WWDG and load the counter value.
Parameters  •  Counter : specifies the watchdog counter value. This parameter must be a number between 0x40 and 0x7F (to prevent generating an immediate reset)
Return values  •  None.
Notes  •  None.

27.2.3  Interrupt and flag management functions

27.2.3.1  WWDG_GetFlagStatus

Function Name  FlagStatus WWDG_GetFlagStatus ( void )
Function Description
Checks whether the Early Wakeup interrupt flag is set or not.
Parameters
• None.
Return values
• The new state of the Early Wakeup interrupt flag (SET or RESET)
Notes
• None.

27.2.3.2 WWDG_ClearFlag

Function Name
void WWDG_ClearFlag ( void )
Function Description
Clears Early Wakeup interrupt flag.
Parameters
• None.
Return values
• None.
Notes
• None.

27.3 WWDG Firmware driver defines

WWDG

WWDG_Prescaler

• #define: WWDG_Prescaler_1((uint32_t)0x00000000)
• #define: WWDG_Prescaler_2((uint32_t)0x00000080)
• #define: WWDG_Prescaler_4((uint32_t)0x00000100)
• #define: WWDG_Prescaler_8((uint32_t)0x00000180)
27.4 WWDG Programming Example

The example below explains how to configure the WWDG to have a timeout of ~69.9 ms with the refresh window set to 80. For more examples about WWDG configuration and usage, please refer to the SPI examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\WWDG\.

```c
/* Enable WWDG clock */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE);

/* WWDG clock counter = (PCLK1 (30MHz)/4096) / 8
 = 915 Hz (~1092 us) */
WWDG_SetPrescaler(WWDG_Prescaler_8);

/* Set Window value to 80; WWDG counter should be refreshed only when the counter is below 80 (and greater than 64) otherwise a reset will be generated */
WWDG_SetWindowValue(80);

/* Enable WWDG and set counter value to 127,
 WWDG timeout = ~1092 us * 64 = ~69.9 ms
 In this case the refresh window is:
 ~1092*(127-80)=~51.3 ms < refresh window < ~1092*64=~69.9ms */
WWDG_Enable(127);
```
28 Miscellaneous add-on to CMSIS functions (misc)

28.1 MISC Firmware driver registers structures

28.1.1 NVIC_InitTypeDef

`NVIC_InitTypeDef` is defined in the misc.h file and contains the NVIC initialization parameters.

Data Fields

- `uint8_t NVIC_IRQChannel`
- `uint8_t NVIC_IRQChannelPreemptionPriority`
- `uint8_t NVIC_IRQChannelSubPriority`
- `FunctionalState NVIC_IRQChannelCmd`

Field Documentation

- `uint8_t NVIC_InitTypeDef::NVIC_IRQChannel`
  - Specifies the IRQ channel to be enabled or disabled. This parameter can be an enumerator ofIRQn_Type enumeration (For the complete STM32 Devices IRQ Channels list, please refer to stm32f2xx.h file)
- `uint8_t NVIC_InitTypeDef::NVIC_IRQChannelPreemptionPriority`
  - Specifies the pre-emption priority for the IRQ channel specified in NVIC_IRQChannel. This parameter can be a value between 0 and 15 as described in the table MISC_NVIC_Priority_Table A lower priority value indicates a higher priority
- `uint8_t NVIC_InitTypeDef::NVIC_IRQChannelSubPriority`
  - Specifies the subpriority level for the IRQ channel specified in NVIC_IRQChannel. This parameter can be a value between 0 and 15 as described in the table MISC_NVIC_Priority_Table A lower priority value indicates a higher priority
- `FunctionalState NVIC_InitTypeDef::NVIC_IRQChannelCmd`
  - Specifies whether the IRQ channel defined in NVIC_IRQChannel will be enabled or disabled. This parameter can be set either to ENABLE or DISABLE

28.2 MISC Firmware driver API description

The following section lists the various functions of the MISC library.

28.2.1 How to configure Interrupts using driver

This section provide functions allowing to configure the NVIC interrupts (IRQ). The Cortex-M3 exceptions are managed by CMSIS functions.
1. Configure the NVIC Priority Grouping using NVIC_PriorityGroupConfig() function according to the following table. The table below gives the allowed values of the pre-emption priority and subpriority according to the Priority Grouping configuration performed by NVIC_PriorityGroupConfig function NVIC_PriorityGroup

<table>
<thead>
<tr>
<th>NVIC_PriorityGroup_Name</th>
<th>Preemption Priority</th>
<th>Subpriority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIC_PriorityGroup_0</td>
<td>0-15</td>
<td>0</td>
<td>4 bits for pre-emption priority 4 bits for subpriority</td>
</tr>
<tr>
<td>NVIC_PriorityGroup_1</td>
<td>0-1</td>
<td>0-7</td>
<td>1 bits for pre-emption priority 3 bits for subpriority</td>
</tr>
<tr>
<td>NVIC_PriorityGroup_2</td>
<td>0-3</td>
<td>0-3</td>
<td>2 bits for pre-emption priority 2 bits for subpriority</td>
</tr>
<tr>
<td>NVIC_PriorityGroup_3</td>
<td>0-7</td>
<td>0-1</td>
<td>3 bits for pre-emption priority 1 bits for subpriority</td>
</tr>
<tr>
<td>NVIC_PriorityGroup_4</td>
<td>0-15</td>
<td>0</td>
<td>4 bits for pre-emption priority 0 bits for subpriority</td>
</tr>
</tbody>
</table>

2. Enable and Configure the priority of the selected IRQ Channels using NVIC_Init()

When the NVIC_PriorityGroup_0 is selected, IRQ pre-emption is no more possible. The pending IRQ priority will be managed only by the subpriority. IRQ priority order (sorted by highest to lowest priority):

- Lowest pre-emption priority
- Lowest subpriority
- Lowest hardware priority (IRQ number)

Functions

- `NVIC_PriorityGroupConfig()`
- `NVIC_Init()`
- `NVIC_SetVectorTable()`
- `NVIC_SystemLPConfig()`
- `SysTick_CLKSourceConfig()`

28.2.2 Functions

28.2.2.1 NVIC_Init

Function Name: `void NVIC_Init (NVIC_InitTypeDef * NVIC_InitStruct)`

Function Description: Initializes the NVIC peripheral according to the specified parameters in the NVIC_InitStruct.

Parameters:

- `NVIC_InitStruct` : pointer to a NVIC_InitTypeDef structure that contains the configuration information for the specified NVIC peripheral.

Return values: None.

Notes:

- To configure interrupts priority correctly, the NVIC_PriorityGroupConfig() function should be called before.
### Function Name
void NVIC_PriorityGroupConfig (uint32_t NVIC_PriorityGroup)

### Function Description
Configures the priority grouping: pre-emption priority and subpriority.

### Parameters
- **NVIC_PriorityGroup**: specifies the priority grouping bits length. This parameter can be one of the following values:
  - **NVIC_PriorityGroup_0**: 0 bits for pre-emption priority 4 bits for subpriority
  - **NVIC_PriorityGroup_1**: 1 bits for pre-emption priority 3 bits for subpriority
  - **NVIC_PriorityGroup_2**: 2 bits for pre-emption priority 2 bits for subpriority
  - **NVIC_PriorityGroup_3**: 3 bits for pre-emption priority 1 bits for subpriority
  - **NVIC_PriorityGroup_4**: 4 bits for pre-emption priority 0 bits for subpriority

### Return values
- None.

### Notes
- When the NVIC_PriorityGroup_0 is selected, IRQ pre-emption is no more possible. The pending IRQ priority will be managed only by the subpriority.

### 28.2.2.3 NVIC_SetVectorTable

### Function Name
void NVIC_SetVectorTable (uint32_t NVIC_VectTab, uint32_t Offset)

### Function Description
Sets the vector table location and Offset.

### Parameters
- **NVIC_VectTab**: specifies if the vector table is in RAM or FLASH memory. This parameter can be one of the following values:
  - **NVIC_VectTab_RAM**: Vector Table in internal SRAM.
  - **NVIC_VectTab_FLASH**: Vector Table in internal FLASH.
- **Offset**: Vector Table base offset field. This value must be a multiple of 0x200.

### Return values
- None.

### Notes
- None.
28.2.2.4 NVIC_SystemLPConfig

Function Name
void NVIC_SystemLPConfig ( uint8_t LowPowerMode,
FunctionalState NewState)

Function Description
Selects the condition for the system to enter low power mode.

Parameters
- **LowPowerMode**: Specifies the new mode for the system to
  enter low power mode. This parameter can be one of the
  following values:
  - **NVIC_LP_SEVONPEND**: Low Power SEV on Pend.
  - **NVIC_LP_SLEEPDEEP**: Low Power DEEPSLEEP request.
  - **NVIC_LP_SLEEPONEXIT**: Low Power Sleep on Exit.
- **NewState**: new state of LP condition. This parameter can be:
  ENABLE or DISABLE.

Return values
- None.

Notes
- None.

28.2.2.5 SysTick_CLKSourceConfig

Function Name
void SysTick_CLKSourceConfig ( uint32_t
SysTick_CLKSource)

Function Description
Configures the SysTick clock source.

Parameters
- **SysTick_CLKSource**: specifies the SysTick clock source.
  This parameter can be one of the following values:
  - **SysTick_CLKSource_HCLK_Div8**: AHB clock divided by 8 selected as SysTick clock source.
  - **SysTick_CLKSource_HCLK**: AHB clock selected as SysTick clock source.

Return values
- None.

Notes
- None.

28.3 MISC Firmware driver defines

28.3.1 MISC Firmware driver defines
MISC
28.4 Interrupt Programming Example

The example below explains step by step how to configure and manage the peripheral interrupt; this program generates an interrupt each time a byte is received by the USART3. For more examples about misc.c driver usage, please refer to the NVIC and SysTick examples provided within the STM32F2xx Standard Peripheral Library package under Project\STM32F2xx_StdPeriph_Examples\.

How to proceed
Copy the code below to main.c file

```c
/* Includes -------------------------------*/
#include "stm32f2xx.h"

/* Private function prototypes ----------------*/
void USART3_Config(void);

/* Private functions ----------------------*/

/* @brief Main program */
/* @param None */
/* @retval None */
```
int main(void)
{
  NVIC_InitTypeDef NVIC_InitStructure;
  
  /****************************************************************
  -Step1-
  Configure the peripheral and enable the interrupt generation
  ******************************************/
  /* Configure USART3 with receive interrupt enabled (generated
   when the receive data register is not empty) */
  USART3_Config();

  /****************************************************************
  -Step2-
  Enable peripheral IRQ channel in the NVIC controller
  ******************************************/
  /* Enable USART3 IRQ channel in the NVIC controller.
   When the USART3 interrupt is generated (in this example when
   data is received) the USART3_IRQHandler will be served */
  NVIC_InitStructure.NVIC_IRQChannel = USART3_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);

  /****************************************************************
  -Step3-
  Implement the peripheral interrupt handler (PPP_IRQHandler)
  in stm32f2xx_it.c file.
  ******************************************/
  /* Refer to next section "stm32f2xx_it.c" */

  while (1)
  {
    
  }

  /**
   * @brief  Configures the USART3 Peripheral.
   * @param  None
   * @retval None
   */
  void USART3_Config(void)
  {
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;

    /****************************************************************
    /* USART IOs configuration ***********************************/
    /* Enable GPIOC clock */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

    /* Connect PC10 to USART3_Tx */
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource10, GPIO_AF_USART3);
    /* Connect PC11 to USART3_Rx*/

    /* Configure USART3 with receive interrupt enabled (generated
     when the receive data register is not empty) */
    USART3_Init(USART_InitStruct);

    /****************************************************************
    -Step2-
    Enable peripheral IRQ channel in the NVIC controller
    ******************************************/
    /* Enable USART3 IRQ channel in the NVIC controller.
      When the USART3 interrupt is generated (in this example when
      data is received) the USART3_IRQHandler will be served */
    NVIC_InitStructure.NVIC_IRQChannel = USART3_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    /****************************************************************
    -Step3-
    Implement the peripheral interrupt handler (PPP_IRQHandler)
    in stm32f2xx_it.c file.
    ******************************************/
    /* Refer to next section "stm32f2xx_it.c" */

    while (1)
    {
      
    }

  /**
   * @brief  Configures the USART3 Peripheral.
   * @param  None
   * @retval None
   */
  void USART3_Config(void)
  {
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;

    /****************************************************************
    /* USART IOs configuration ***********************************/
    /* Enable GPIOC clock */
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);

    /* Connect PC10 to USART3_Tx */
    GPIO_PinAFConfig(GPIOC, GPIO_PinSource10, GPIO_AF_USART3);
    /* Connect PC11 to USART3_Rx*/
GPIO_PinAFConfig(GPIOC, GPIO_PinSource11, GPIO_AF_USART3);

/* Configure USART3_Tx and USART3_Rx as alternate function */
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10 | GPIO_Pin_11;
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(GPIOC, &GPIO_InitStructure);

/* USART configuration ****************************************/
/* USART3 configured as follow:
- BaudRate = 115200 baud
- Word Length = 8 Bits
- One Stop Bit
- No parity
- Hardware flow control disabled (RTS and CTS signals)
- Receive and transmit enabled
*/

/* Enable USART3 clock */
RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART3, ENABLE);

USART_InitStructure.USART_BaudRate = 115200;
USART_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_InitStructure.USART_StopBits = USART_StopBits_1;
USART_InitStructure.USART_Parity = USART_Parity_No;
USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
USART_Init(USART3, &USART_InitStructure);

/* Enable USART3 */
USART_Cmd(USART3, ENABLE);

/* Enable USART3 Receive interrupt */
USART_ITConfig(USART3, USART_IT_RXNE, ENABLE);

} // End of ISR

/* UART Global Interrupt Handler */
void USART3_IRQHandler(void)
{
  if(USART_GetITStatus(USART3, USART_IT_RXNE) != RESET)
  {
    /* Read one byte from the receive data register */
    // Code to read data from receive data register
  }
}

Declare a global variable "__IO uint8_t RxData;" and copy the code below to
stm32f2xx_it.c file

/**
 * @brief This function handles USART3 global interrupt request.
 * @param None
 * @retval None
 */
void USART3_IRQHandler(void)
{
    if(USART_GetITStatus(USART3, USART_IT_RXNE) != RESET)
    {
        /* Read one byte from the receive data register */
    }
}
RxData = USART_ReceiveData(USART3);


### Revision History

**Table 14: Revision History**

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>05-Dec-2011</td>
<td>1</td>
<td>Initial release.</td>
</tr>
</tbody>
</table>
30 Disclaimer

Please Read Carefully

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at anytime, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY TWO AUTHORIZED ST REPRESENTATIVES ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER'S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2011 STMicroelectronics - All rights reserved
STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com

634/634 DocID 18540 Rev 1