RM0002
Reference manual

STR71xF microcontroller family

Introduction

This Reference Manual provides complete information for application developers on how to use the STR71x Microcontroller memory and peripherals.

The STR71xF is a family of microcontrollers with different memory sizes, packages and peripherals.

For Ordering Information, Mechanical and Electrical Device Characteristics please refer to the STR71x datasheet.

For information on programming, erasing and protection of the internal Flash memory please refer to the STR7 Flash Programming Reference Manual

For information on the ARM7TDMI® core please refer to the ARM7TDMI® Technical Reference Manual.

Related documents:

Available from www.arm.com:

ARM7TDMI® Technical Reference Manual

Available from http://www.st.com:

STR71x Datasheet

STR7 Flash Programming Reference Manual

AN1775 - STR71x Hardware Development Getting Started

The above is a selected list only, a full list STR71x application notes can be viewed at http://www.st.com.

Contents

1 Memory

1.1 Memory organization

The ARM7 native bus is used as the main system bus, connecting CPU, memories and system service blocks while the low-power APB rev. E is used as a peripheral bus.

The native bus system includes CPU, RAM, Flash, External Memory Interface (EMI) and the Power, Reset and Clock Control Unit (PRCCU).

The APB bridges (APB1 & APB2) interface two groups of peripherals. Wait states are inserted automatically on the CPU clock when accessing APB peripherals clocked slower than the ARM7 core.

Program memory, data memory, registers and I/O ports are organized within the same linear address space of 4 GBytes.

The bytes are treated in memory as being in Little Endian format. The lowest numbered byte in a word is considered the word's least significant byte and the highest numbered byte the most significant.

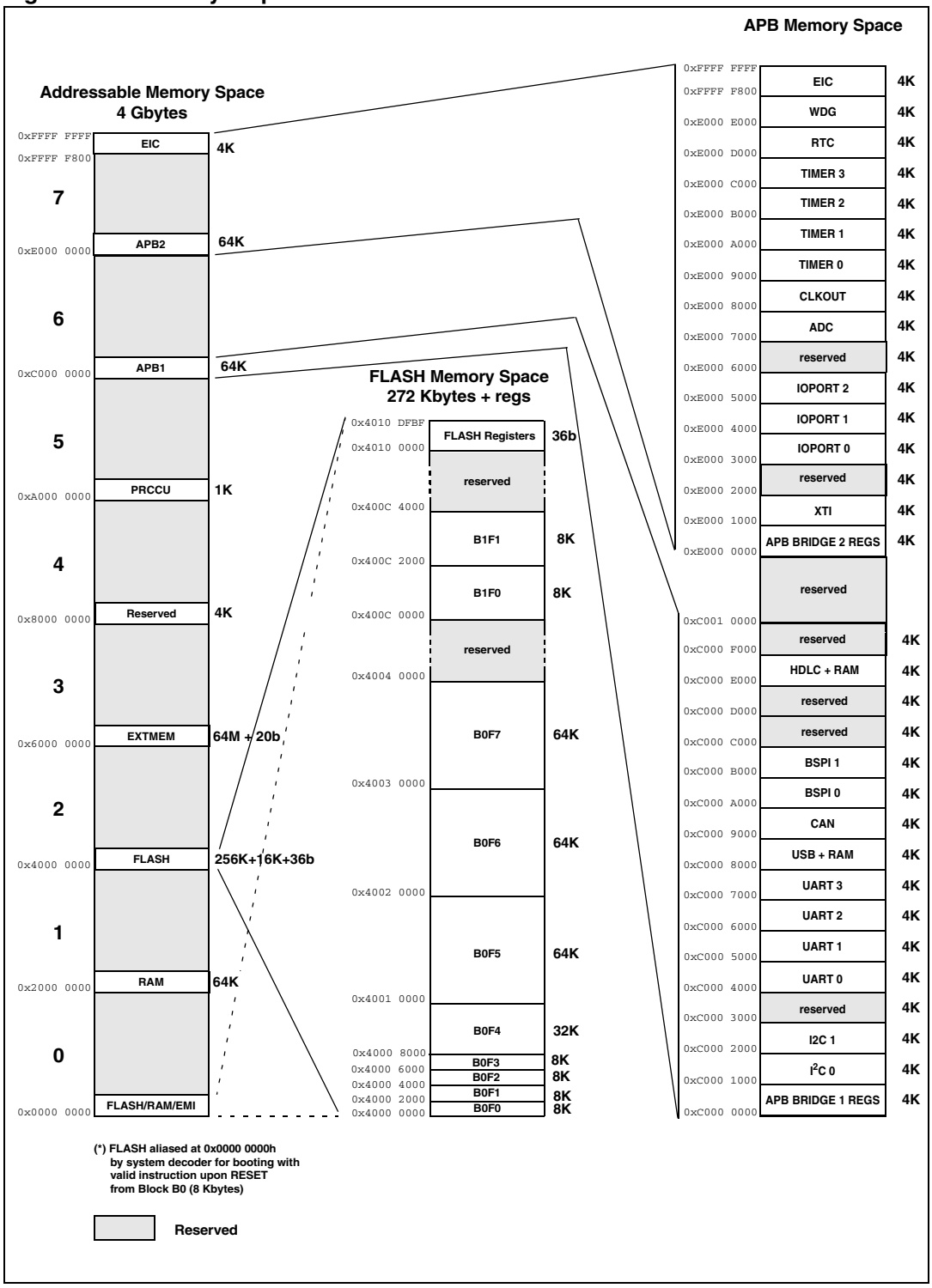
Figure 1 on page 12 shows the STR71x Memory Map. For the detailed mapping of peripheral registers, please refer to the related chapters.

The addressable memory space is divided into 8 main blocks, selected by the three most significant bits A[31:29] of the memory address bus A[31:0]

- 000 = Boot memory
- 001 = RAM Memory
- 010 = Flash Memory
- 011 = External Memory
- 100 = Reserved Memory
- 101 = PRCCU registers
- 110 = APB Bridge 1 - Serial Communication Peripherals
- 111 = APB Bridge 2 - System Peripherals and Interrupt Controller

1.1.1 Memory map

Figure 1. Memory map



1.1.2 Mapping of memory blocks

Table 1. Mapping of memory blocks

STR71x Memory Blocks	Block Base Address	Section
Boot Memory	0x0000 0000	*Section 1.1.5*
RAM	0x2000 0000	*Section 1.1.6*
Flash	0x4000 0000	Refer to STR7 Flash Programming Reference Manual
External Memory Interface (EMI)	0x6000 0000	*Section 1.4.2*
PRCCU	0xA000 0000	*Section 2.7*
APB1	0xC000 0000	see *Table 2*
APB2	0xE000 0000	See *Table 3*

1.1.3 APB1 memory map

APB1 base address = 0xC000 0000h.

Table 2. APB1 memory map

Pos	STR71x APB1 Peripheral	Address Offset	Peripheral Register Map
0	APB1 Bridge Configuration registers	0x0000	*Section 15.3*
1	I2C0	0x1000	*Section 9.6*
2	I2C1	0x2000	*Section 9.6*
3	Reserved		
4	UART0	0x4000	*Section 11.5*
5	UART1 + SMARTCARD Interface	0x5000	*Section 11.5* and *Section 12.6*
6	UART2	0x6000	*Section 11.5*
7	UART3	0x7000	*Section 11.5*
8	USB RAM	0x8000	*Section 13.7*
	USB Registers	0x8800	
9	CAN	0x9000	*Section 8.6*
10	BSPI0	0xA000	*Section 10.11*
11	BSPI1	0xB000	*Section 10.11*
12	Reserved		
13	Reserved		
14	HDLC Registers	0xE000	Refer to separate HDLC reference manual
	HDLC RAM	0xE800	
15	Reserved		

1.1.4 APB2 memory map

APB2 base address = 0xE000 0000h.

Table 3. APB2 memory map

Pos	STR71x APB2 Peripheral	Address Offset	Peripheral Register Map
0	APB2 Bridge Configuration registers	0x0000	*Section 15.3*
1	External Interrupts (XTI)	0x1000	*Section 4.4.5*
2	Reserved	0x2000	*Section 3.2.5*
3	IOPORT0	0x3000	*Section 3.2.5*
4	IOPORT1	0x4000	*Section 3.2.5*
5	IOPORT2	0x5000	*Section 3.2.5*
6	Reserved	0x6000	
7	ADC	0x7000	*Section 14.5*
8	CKOUT	n.a.	
9	TIMER0	0x9000	*Section 7.7*
10	TIMER1	0xA000	*Section 7.7*
11	TIMER2	0xB000	*Section 7.7*
12	TIMER3	0xC000	*Section 7.7*
13	RTC	0xD000	*Section 5.5*
14	WDG	0xE000	*Section 6.5*
15	EIC	0xF800	*Section 4.2*

Note: EIC is aliased in the memory map; it can be addressed with an offset of 0xFFFF F800. This is used to branch from the ARM7 Interrupt vector (0x0000 0018) to the EIC_IVR register, that points to the active Interrupt Routine (see Section 4 on page 63).

1.1.5 Boot memory

Three boot modes are selected by configuration pins on exit from Reset (*Section 1.2: Boot configuration on page 18*)

- **Flash boot mode**: In this mode, the Flash is mapped in both memory block 010 and memory block 000. The system boots from bank 0, sector 0 of the Flash
- **RAM boot mode**: In this mode, RAM is mapped in both memory block 001 and memory block 000, and the system boots from RAM Memory. This is useful for debug purposes, the RAM can be pre-loaded by an external JTAG controller or development system (emulator).
- **External Memory boot mode**: In this mode, External Memory is mapped in both memory block 011 and memory block 000, and the system boots from External Memory bank 0.

1.1.6 RAM

STR71x features 64 KBytes of fully static, synchronous RAM. It can be accessed as bytes, half-words (16 bits) or full words (32 bits). The RAM start address is 0x2000 0000.

In RAM boot mode (see *Section 1.1.5*) the RAM start address is mapped at both 0x0000 0000h and at 0x2000 0000h.

You can remap the RAM on-the-fly to RAM boot mode configuration, using the BOOT[1:0] bits in the PCU_BOOTCR register. This is particularly useful for managing interrupt vectors and routines, you can copy them to RAM, modify and access them even when Flash is not available (i.e. during Flash programming or erasing).

1.1.7 Flash

The Flash Module is organized in banks and sectors as shown in *Table 4*.

Table 4. Flash Module Organisation

Bank	Sector	Addresses	Size (bytes)
Bank 0 256 Kbytes Program Memory	Bank 0 Flash Sector 0 (B0F0)	0x00 0000 - 0x00 1FFF	8K
	Bank 0 Flash Sector 1 (B0F1)	0x00 2000 - 0x00 3FFF	8K
	Bank 0 Flash Sector 2 (B0F2)	0x00 4000 - 0x00 5FFF	8K
	Bank 0 Flash Sector 3 (B0F3)	0x00 6000 - 0x00 7FFF	8K
	Bank 0 Flash Sector 4 (B0F4)	0x00 8000 - 0x00 FFFF	32K
	Bank 0 Flash Sector 5 (B0F5)	0x01 0000 - 0x01 FFFF	64K
	Bank 0 Flash Sector 6 (B0F6)	0x02 0000 - 0x02 FFFF	64K[1)]
	Bank 0 Flash Sector7 (B0F7)	0x03 0000 - 0x03 FFFF	64K[1)]
Bank 1 16 Kbytes Data Memory	Bank 1 Flash Sector 0 (B1F0)	0x0C 0000 - 0x0C 1FFF	8K
	Bank 1 Flash Sector 1 (B1F1)	0x0C 2000 - 0x0C 3FFF	8K
Flash Control Registers	Flash Control/Data Registers	0x10 0000 - 0x0010 0017	24
	Flash Protection Registers	0x10 DFB0 - 0x0010 DFBC	12

[1)]Not available in 128K versions.

Bank 0 is intended for program memory. Sectors B0F0-B0F7 can be used as Boot sectors; they can be write protected against unwanted write operations.

Bank 1 contains 16 Kbytes of Data Memory: it is divided into 2 sectors (8 Kbytes each). You can program application data in this area.

You can Program Bank 0 and Bank 1 independently, i.e. you can read from one bank while writing to the other.

Flash memory can be protected against different types of unwanted access (read/write/erase).

You can program Flash memory using In-Circuit Programming and In-Application programming. Refer to the STR7 Flash Programming Reference Manual.

Flash Burst and Low Power modes

The STR71x Flash memory has two access modes: Burst mode and Low power mode.

You can enable or disable Burst mode using the FLASHLP bit in the PCU_PWRCR register.

In Burst mode, sequential accesses are performed with zero wait states at speeds of up to the maximum device frequency. Non sequential accesses are performed with 1 wait state.

When Burst Mode is disabled, the Flash operates in low power (LP) mode and all accesses are performed at speeds of up to 33 MHz with zero wait states.

Flash power down mode

Depending on your application requirements, you can optionally power down the Flash in Stop or Low power wait for interrupt modes (Refer to the STR7 Flash programming reference manual). Otherwise, in Stop mode, the Flash automatically reduces its power consumption and can be read immediately after wake-up.

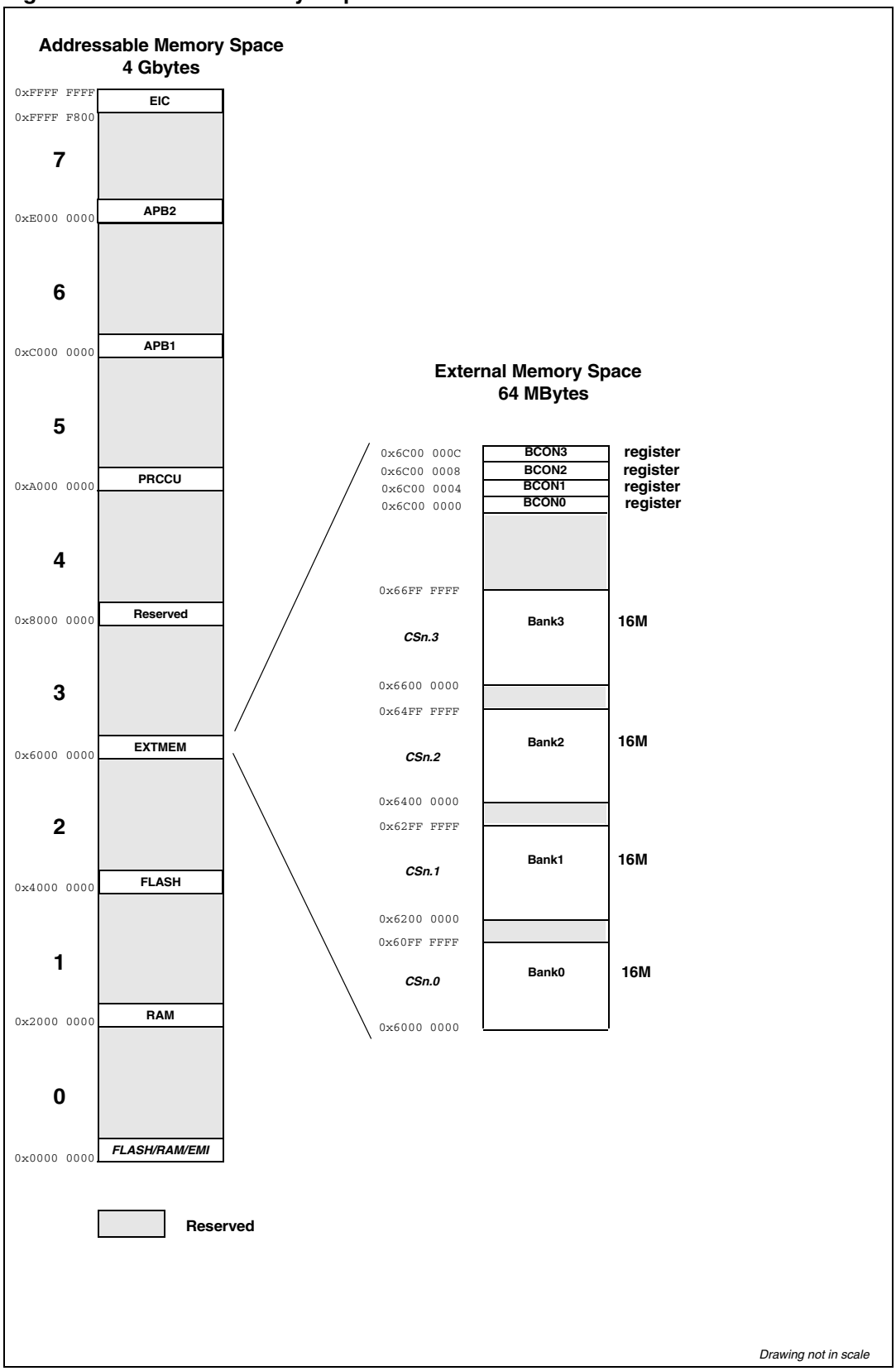
1.1.8 External memory

The External Memory Interface can be used to interface external memory components such as ROM, FLASH, SRAM or external peripherals.

The External Memory space is divided into 4 banks based on decoding of A[26:25]. Each Bank can address up to 16 MBytes of external memory. Addressing one of these banks will activate the related Chip Select output CSN[3:0]. EMI address bus A[23:0] is available externally, as well as the control signals WEN[1:0] (Byte Write Enable) and RDN (Read Enable).

The length of time required to properly transfer data from/to external memory banks is controlled by programming the control registers for each bank which determines the number of wait states used to access it. The access time also depends on the selected external bus width. Each register also contains a flag bit to indicate if a particular memory bank is accessible or not.

Figure 2. External Memory Map



1.2 Boot configuration

In the STR71x, three boot modes are available and can be enabled by means of three input pins: BOOTEN, BOOT0 and BOOT1.

BOOTEN is a dedicated pin. It must normally be tied to ground through a 10K resistor. When BOOTEN = 0, the device is in FLASH boot mode, the BOOT[1:0] pins are not used, and they may have any value.

When BOOTEN = 1, the BOOT[1:0] pins are latched at the second CK rising edge after the release of the external RSTINn pin. Their value is used to configure the device BOOT mode, as shown in *Table 5*.

Table 5. Boot modes

BOOTEN	BOOT1	BOOT0	Mode	Boot Memory Mapping	Note
0	any	any	FLASH	FLASH mapped at 0h	System executes code from Flash
1	0	0			
1	1	0	RAM	RAM mapped at 0h	System executes code from internal RAM For Lab development.
1	1	1	EXTMEM	EXTMEM mapped at 0h	System executes code from external memory

Note: When booting with the following configuration BOOTEN=1, BOOT1=0 and BOOT0=1 the system boots in a reserved ST boot mode.

In the following section, the abbreviation B[1:0] refers to the BOOT[1:0] pins, and BOOTEN is considered equal to 1 unless specified.

1.2.1 FLASH boot mode

This is the standard operating mode; to enter this mode it is not necessary to control BOOT[1:0], if BOOTEN = 0.

This mode is also entered by forcing the external pins B1=0 and B0=0. This status is latched when the external Reset is released.

It is up to the user to develop the in-application programming (IAP) procedure and determine the best usage of the different sectors. For example, B0F0 and B0F1 can be used as the user boot-loader. When this mode is used, at least block B0F0 must have been previously programmed, as the system boot is performed from Flash sector B0F0.

Note: To guarantee maximum security, it is recommended that the flash Erase and Programming routines are not stored in the Flash itself, but loaded into RAM from an external tool at the start of the ICP procedure.

1.2.2 RAM and EXTMEM boot modes

RAM mode is provided to ease application development. RAM mode is entered by forcing the external pins B1=1 and B0=0, while EXTMEM mode is entered by forcing the external pins B1=1 and B0=1. This status is latched on the second CK clock pulse after external Reset is released.

In EXTMEM mode the system boot is performed from the external memory, bank 0 (CSN0 is activated). The external memory is also mapped at address 0h (see PCU_BOOTCR register).

In RAM mode the system boot is performed from the internal RAM, which is also visible at address 0h (see PCU_BOOTCR register). You must pre-load your boot code in RAM, for example using a Development System (MultiICE™ or equivalent).

It is up to the user application to provide the proper signals on B0, B1, since there is no specific microcontroller I/O to control B0 and B1 during the reset phase.

Note: When DEBUG protection is activated for internal Flash memory, these Boot modes are no longer available.

1.3 External memory interface (EMI)

1.3.1 EMI bus interface signal description

EMI external bus signals are detailed in *Table 6*.

Table 6. EMI Bus Interface Signals

Name	I/O	Description
A[23:0]	O	External interface address bus
D[15:0]	I/O	External interface data bus
RDn	O	Active low read signal for external memory. It maps to the OE_N input of the external components
WEn.0	O	External write enable signal. When ‘0’, enables write operation to 8 LSBs (bits 7:0) of external memory
WEn.1	O	External write enable signal. When ‘0’, enables write operation to bits 15:8 of external memory
CSn.0	O	Active low chip select for bank 0.
CSn.1	O	Active low chip select for bank 1.
CSn.2	O	Active low chip select for bank 2.
CSn.3	O	Active low chip select for bank 3.

1.3.2 EMI memory map

The EMI memory map is shown in *Table 7*. Each bank makes use of all 24 bits of A[23:0], providing 16 MByte of addressable space. The base address of the External Memory space is EMI_BASE = 0x6000 0000.

Table 7. EMI Memory Map

Address Range	Description	Addressable Size (Bytes)	Bus Width (bit)
0x6000 0000 - 0x60FF FFFF	Bank 0 - BOOT (CSn.0)	16M	8/16 bit access only[1)]
0x6200 0000 - 0x62FF FFFF	Bank 1 (CSn.1)	16M	8/16 SW Selectable
0x6400 0000 - 0x64FF FFFF	Bank 2 (CSn.2)	16M	8/16 SW Selectable
0x6600 0000 - 0x66FF FFFF	Bank 3 (CSn.3)	16M	8/16 SW Selectable
0x6C00 0000 - 0x6C00 0010	Internal Registers	n/a	16 bit access only

1. If External memory is used for boot operation, it must be 16-bit as the CSN0 memory bank has been hard-wired for 16-bit memory interface only.

1.3.3 EMI programmable timings

Each memory bank of the EMI can have a programmable number of wait states (up to 15) added to any read or write cycle: this is software configurable via the C_LENGTH bitfield of the EMI_BCONx register (x=0,..,3).

Depending on the used external memory data bus width, two access types are possible: Single cycle access and Multiple cycle access:

Single cycle access:

For write cycles which translate in a single external bus operation (e.g. a 16-bit write to a bank configured 16-bit wide), the total length (measured in EMI internal clock units) that a single access will require (equal to the length over which the Chip Select will be active) will depend on the memory bank the cycle is executed on. It can be calculated using the formula:

CSn[3:0] length (Write) = C_LENGTH+2

This value is also the number of WAIT states generated. Hence for performance calculations each external access will last (Int_Access_length + CSn[3:0] length) MCLK cycles where Int_Access_length may be derived from ARM7TDMI Technical Reference Manual, depending on the type of access.

For read cycles which translate in a single external bus operation (e.g. a 16-bit read from a bank configured 16-bit wide), the total length (measured in EMI internal clock units) that a single access will require (equal to the length over which the Chip Select will be active) can be calculated using the formula:

CSn[3:0] length (Read) = C_LENGTH+2

Each external access will last (Int_Access_length + CSn[3:0] length) MCLK cycles as above.

Multiple cycle access:

For read or write cycles that translate in multiple (N) external bus cycles (e.g. a 32-bit write to a region configured 8-bit wide, for which N=4), the total external bus cycle duration can be obtained using the following formula

Total CS length (Read/Write Bank x) = (Length of the single cycle)*(N)

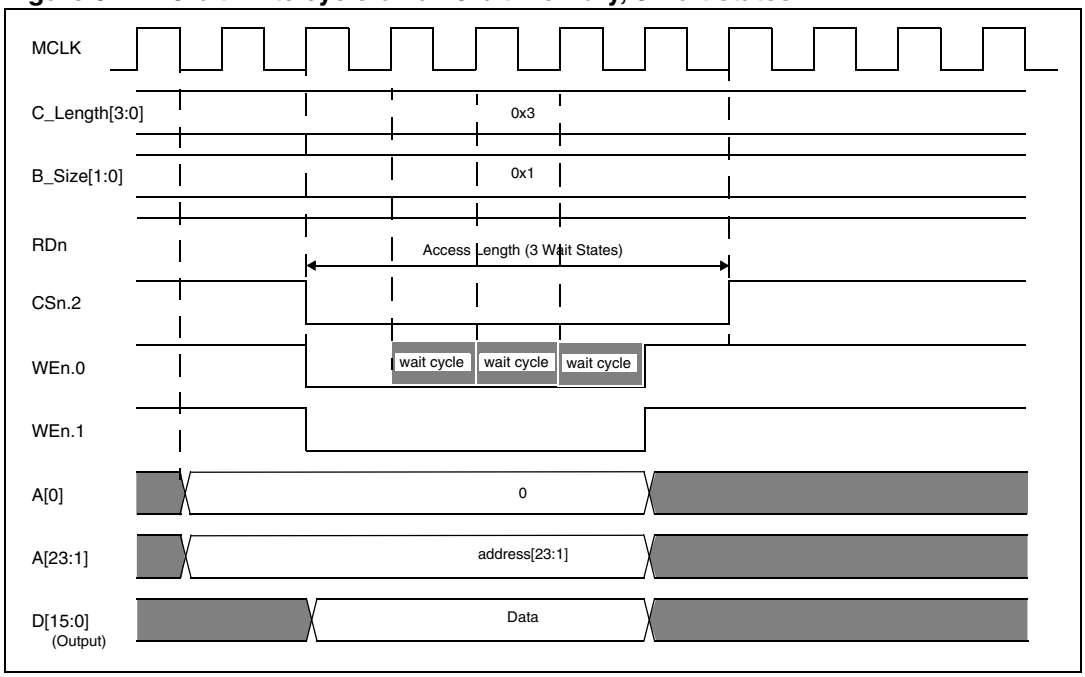
In this case the total duration of the access will be (Int_Access_length + Total CS length) MCLK cycles.

Note: If an access is attempted to EMI when the clock to the EMI is disabled, the CPU will hang until it is reset.

1.3.4 Write access examples

In the *Figure 3* shown below, a 16-bit write is being performed on a 16-bit external memory. As can be seen from the diagram, both external write strobes are asserted for the duration of the write cycle. This is a one-cycle write, hence the LSB of the external address is not modified, and takes 5 MCLK cycles to complete (C_LENGTH = 3).

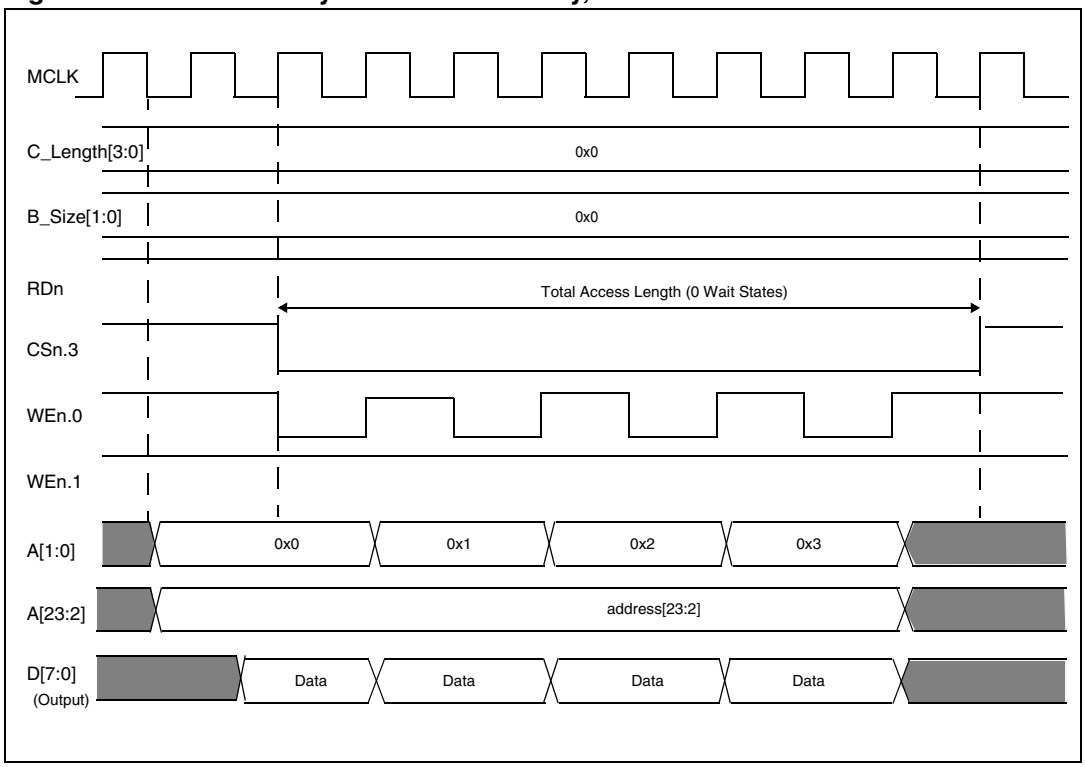
Figure 3. 16-bit write cycle on a 16-bit memory, 3 wait states.



In *Figure 4*, a 32-bit write is being performed. However in this case, the external bus width is only 8-bits (B_SIZE = 0). This has the effect that 4 write cycles are required to perform the full 32-bit write. The 2 LSBs of the external address are modified by the read-write controller for each write beginning at “00” for the 1st write and incrementing for each write cycle until it reaches “11”.

As only the 8 LSBs of the external data bus are used, only WEn.0 is asserted for any write.

Figure 4. 32-bit write cycle on 8-bit memory, no wait states

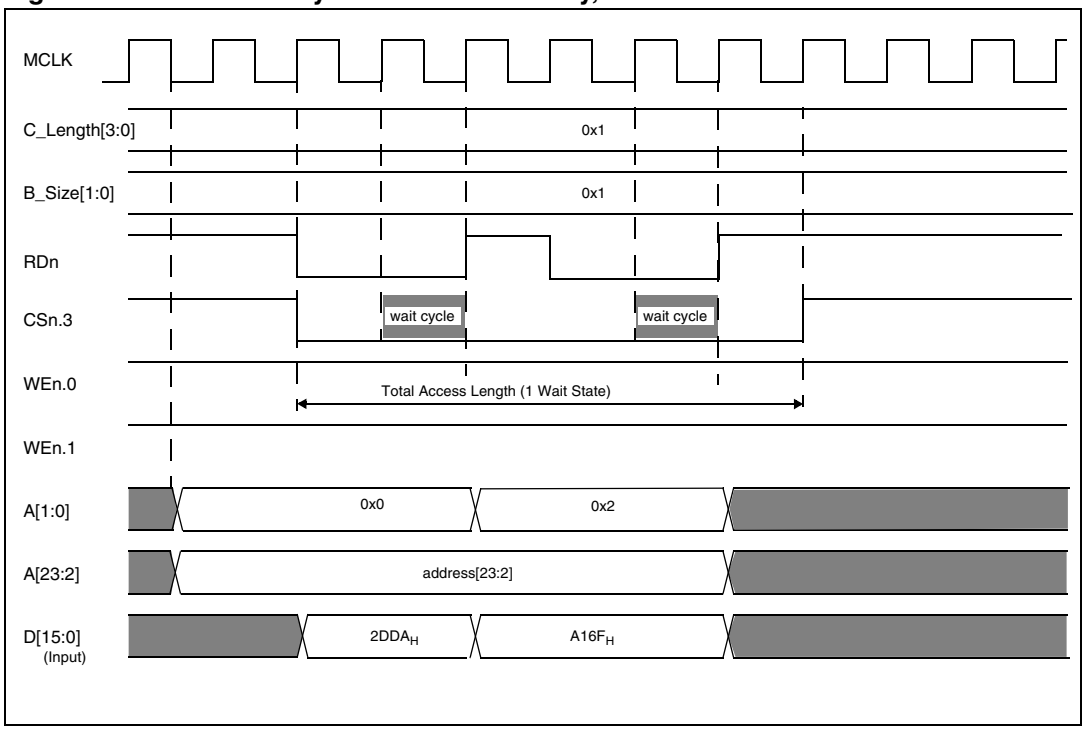


In this case CS3n is used; it should be noted that CS3n is maintained asserted until all write cycles have been completed whereas WEn.0 is de-asserted for 1 MCLK cycle between separate write operations.

1.3.5 Read access examples

The following diagram shows a basic READ operation. In the case of a READ only 1 external read strobe is required (RDn). In this example, a 32-bit read is being performed. The external bus size is 16 bits however (B_SIZE = 1) so the EMI peripheral has to perform 2 successive read operations. The results from the 1st read operation are latched internally in the EMI block (In this case - “2DDA” - i.e 16 LSBs of data) so that the correct 32-bit data (i.e “A16F2DDA”) is returned after the 2nd 16-bit read.

Figure 5. 32-bit read cycle on 16-bit memory, 1 wait state



For the 1st read operation, A[1:0] is assigned “00”, for the second this is incremented by 2 (i.e “10”).

1.4 Register description

1.4.1 Bank n configuration register (EMI_BCONn)

Base address: 0x6C00 0000

Address offset:

– BCON0: 0x00h – BCON1: 0x04h

– BCON2: 0x08h – BCON3: 0x0Ch

Reset values:

– BCON0: 0x803Dh – BCON1: 0x003Dh

– BCON2: 0x003Eh – BCON3: 0x003Ch

The Bank *n* Configuration Register (BCON*n*) is a 16-bit read/write control register used to configure the operation of Bank n. The BCONn control bits are described below.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BE	Reserved									C_LENGTH[3:0]				B_SIZE[1:0]	
rw	-									rw	rw	rw	rw	rw	rw

Bit 15	**BE**: *Bank n Enable*. 0: Bank disabled 1: Bank enabled
Bits 5:2	**C_LENGTH[3:0]**: *Cycle Length*. The C_LENGTH field selects the number of wait states to be inserted in any read/write cycle performed in Bank *n*. The total CS length of any read or write cycle will be equal to C_LENGTH+1 periods of the EMI internal clock. 0x0h = 0 wait states 0x1h = 1 wait state 0x2h = 2 wait states 0x3h = 3 wait states .... 0xFh =15 wait states
Bits 1:0	**B_SIZE[1:0]**: *Bus Size*. The B_SIZE field defines the effective external bus size for an access to Bank *n*. 0x00b = 8-bit 0x01b = 16-bit 0x10b = Reserved (do not use) 0x11b = Reserved (do not use)

1.4.2 EMI register map

Table 8. EMI register map

Addr. Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00h	BCON0	BE	Reserved									C_LENGTH[3:0]				B_SIZE [1:0]	
04h	BCON1	BE	Reserved									C_LENGTH[3:0]				B_SIZE [1:0]	
08h	BCON2	BE	Reserved									C_LENGTH[3:0]				B_SIZE [1:0]	
0Ch	BCON3	BE	Reserved									C_LENGTH[3:0]				B_SIZE [1:0]	

Base address = 0x6C00 0000

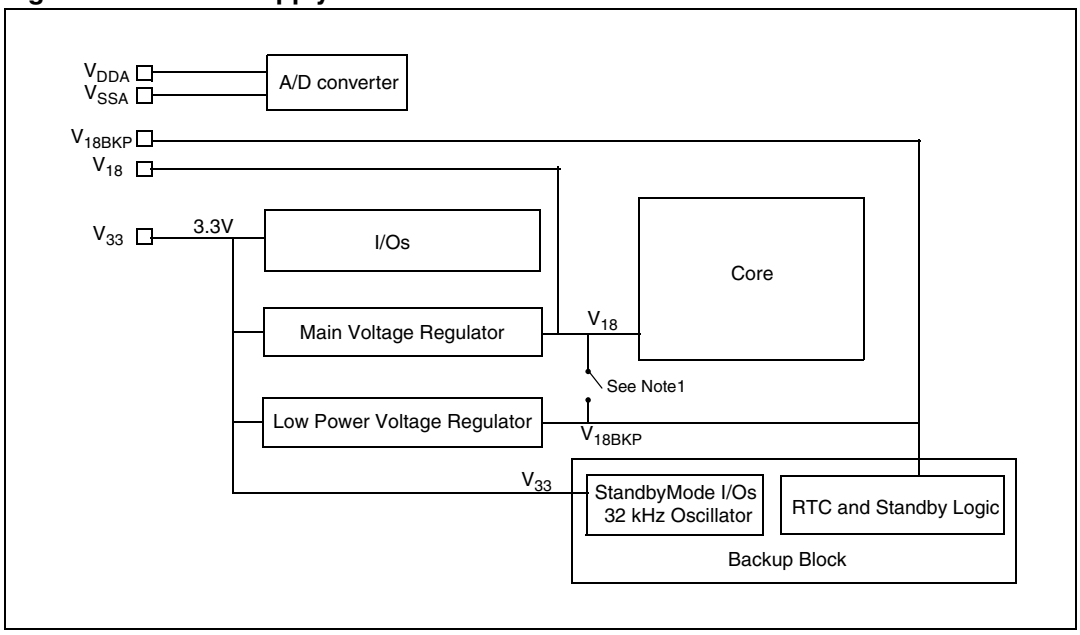
2 Power, reset and clock control unit

2.1 Power supply

The chip is powered by an external 3.3V supply. All I/Os are 3V-capable. Two internal voltage regulators : the main voltage Regulator and the low power voltage regulator generate the 1.8V supply voltage for core logic.

The V_{DDA} and V_{SSA} pins supply the reference voltages for the A/D Converter.

Figure 6. Power Supply Overview



Note: 1 In normal operating mode V_{18} is shorted with V_{18BKP} In STANDBY Mode the V_{18} domain is disconnected from the V_{18BKP} domain.

2 The two V_{18} pins must be connected to external stabilization capacitors. Connecting an external 1.8V supply to the v_{18} pins is not supported (refer to the str71x datasheet).

2.1.1 Optional use of external V_{18BKP} supply

In Standby mode, (see *Section 2.5.5 on page 37*) the Main Voltage Regulator is switched off, and the Low Power Voltage Regulator supplies power to the backup block. It is possible to bypass the Low Power Voltage Regulator under software control when an external 1.8V supply (V_{18BKP}) is available in addition to V_{33}, increasing the power efficiency of the system.

In this case V_{18BKP} must not be connected directly but through a diode to avoid any contention when the device is not in Standby mode.

Because it powers the I/Os, V_{33} cannot be switched off in Standby mode since the nSTDBY, nRSTIN and WAKEUP pins must remain functional.

2.2 Voltage regulators

2.2.1 Main voltage regulator

The Main Voltage Regulator (MVR) is able to generate sufficient current for the device to operate in any mode through ballast P-channel transistors located inside the I/O ring. It includes a bandgap reference for thermal compensation and it has a static power consumption of 100 µA (typical).

Note: 1 The MVR is automatically switched off in Standby mode.

2 The MVR can be configured (using the LPVRWFI bit in the PCU_PWRCR register) to automatically switch off when the device enters Stop mode or LPWFI mode, leaving the Low Power VR as the only power supply.

3 The MVR could be switched off using the VRBYP bit in the PCU_PWRCR register, In this configuration the device is only powered by the Low Power VR, the maximum allowed operation frequency is 1MHz and the PLL is disabled.

4 When the VROK bit in the PCU_PWRCR register is set by hardware it is guaranteed that the main regulator output voltage is stabilized at the specification value.

2.2.2 Low power voltage regulator

The separate Low Power Regulator should be used only when the device is in Standby, STOP or LPWFI. It has a different design from the main VR and generates a non-stabilized and non-thermally-compensated voltage of approximately 1.6V. The output current is not generally sufficient for the device to run in normal operation. Because of this limitation, the PLL is automatically disabled when the Main VR is switched off and the maximum allowed operating frequency is 1 MHz.

The Low Power VR can be switched off as well when an external regulator provides a 1.8V supply to the chip through the V_{18BKP} pin for use by Real Time Clock and Wake-Up logic during Standby mode (See *Section 2.1.1 on page 25*.)

For both the Main VR and the Low Power, VR stabilization is achieved by external capacitors, connected respectively to the V_{18} pins (Main Regulator) and V_{18BKP} pin (Low Power Regulator). The minimum recommended value is 10µF (Tantalum, low series resistance) plus 33nF ceramic for the Main Voltage Regulator, and 1µF for the Low-Power Voltage Regulator. Precaution should be taken to minimize the distance between the chip and the capacitors. Care should also be taken to limit the serial inductance to less than 60 nH.

Note: Both the Main Voltage Regulator and the Low Power Voltage Regulator contain each a low voltage detection circuitry which keep the device under reset when the corresponding controlled voltage value (V_{18} or V_{18BKP}) falls below 1.35V (+/- 10%).

2.3 Reset

At power on, the nRSTIN pin must be held low by an external reset circuit until V_{33} reaches the minimum specified in the datasheet.

The Reset Manager resets the MCU when one of the following events occurs:

- A Hardware reset, initiated by a low level on the nRSTIN pin
- A Software reset, forced by setting the HALT bit in the RCCU_SMR register (when enabled with SRESEN bit and the ENHALT bit in the RCCU_CCR register)
- A supply voltage drop under the threshold of the low voltage detection circuit of either of the Voltage Regulators (refer to the STR71x datasheet)
- A Watchdog end of count condition.
- Activation of WAKEUP pin when the device is in STANDBY mode (see *Section 2.5.5 on page 37*)
- Activation of the Real Time Clock Alarm when the device is in STANDBY mode (see *Section 2.5.5 on page 37*)

The event causing the last Reset is flagged in the RCCU_CFR register: the corresponding bit is set. A hardware-initiated reset or a Voltage Regulator low voltage detector reset will leave all these bits reset.

The hardware reset overrides all other conditions and forces the system to reset state. During the Reset phase, the internal registers are set to their reset values, where these are defined, and the I/O pins go into their reset configuration.

A Reset from the nRSTIN pin is asynchronous: as soon as it is driven low, a Reset cycle is initiated.

The on-chip Timer/Watchdog generates a reset condition if Watchdog mode is enabled and if the programmed period elapses without the specific code being written to the appropriate register (refer to Watchdog specifications).

When the nRSTIN pin goes high again, 2048 clock CLK plus 8 CLK2 (refer to *Section 2.4*) cycles are counted before exiting Reset state (plus possibly one more CLK period, depending on the delay between the rising edge of the nRSTIN pin and the first rising edge of CLK). Refer to *Figure 7*.

At the end of the Reset phase, the Program Counter is set to the location specified in the Reset Vector located in memory location 0x0h.

Figure 7. Reset General Timing

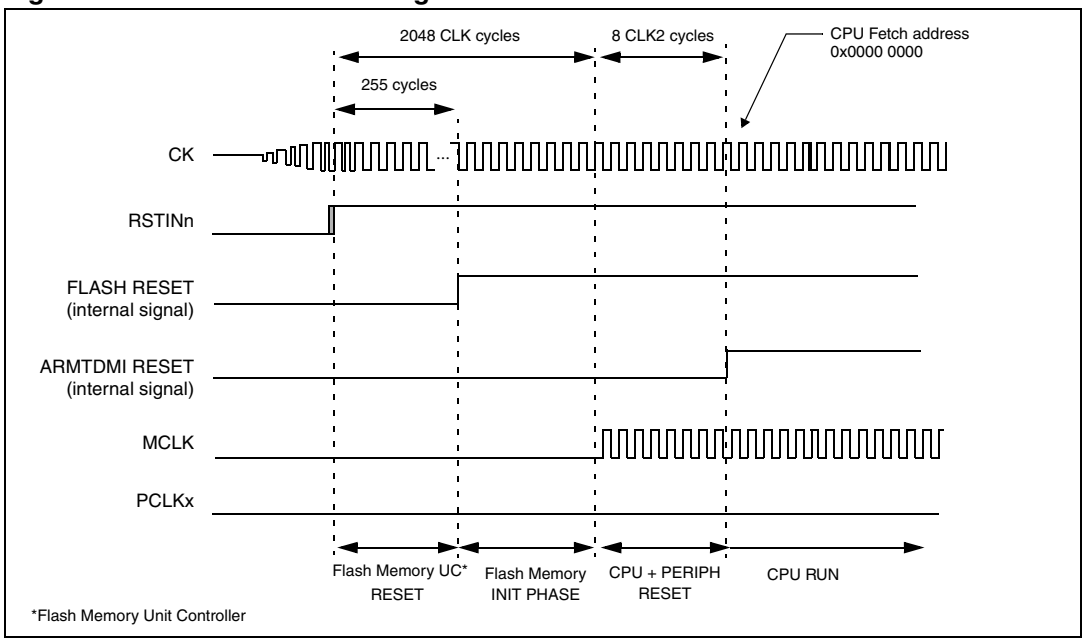
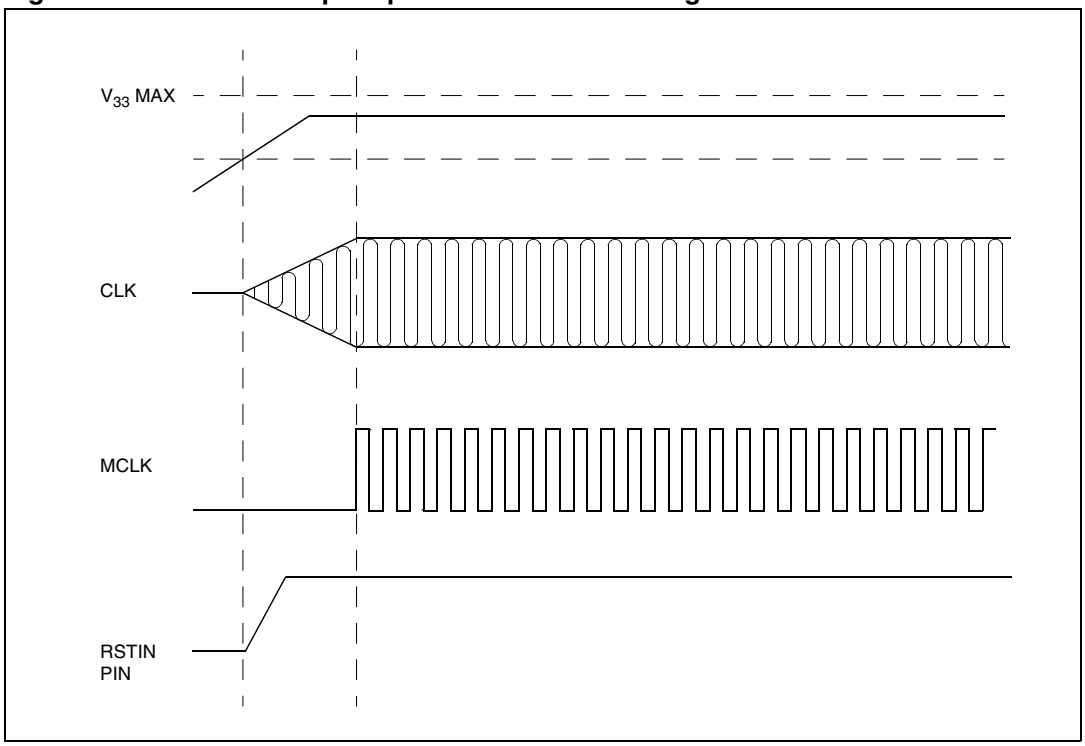


Figure 8. Clock Start-up Sequence and Reset Timing

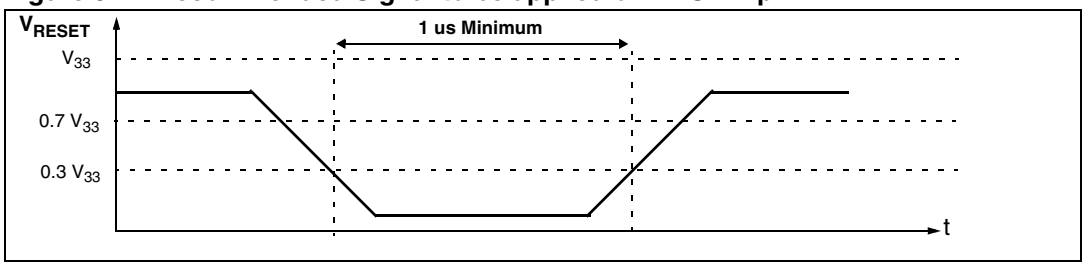


2.3.1 Reset pin timing

To improve the noise immunity of the device, the RESET input pin (nRSTIN) has a Schmitt trigger input circuit with hysteresis. Spurious RESET events are masked by an analog filter which guarantees that all the glitches (single pulse and burst) on the nRSTIN pin shorter than 100ns are not recognized by the system as valid RESET pulses. On the other hand, it

is recommended to provide a valid pulse on nRSTIN with a duration of at least 1 µs to be sure that the asynchronous pulse is properly latched. This means that all pulses longer than 100ns and shorter than 1 µs can have an unpredictable effect on the device: they can either be recognized as valid or filtered.

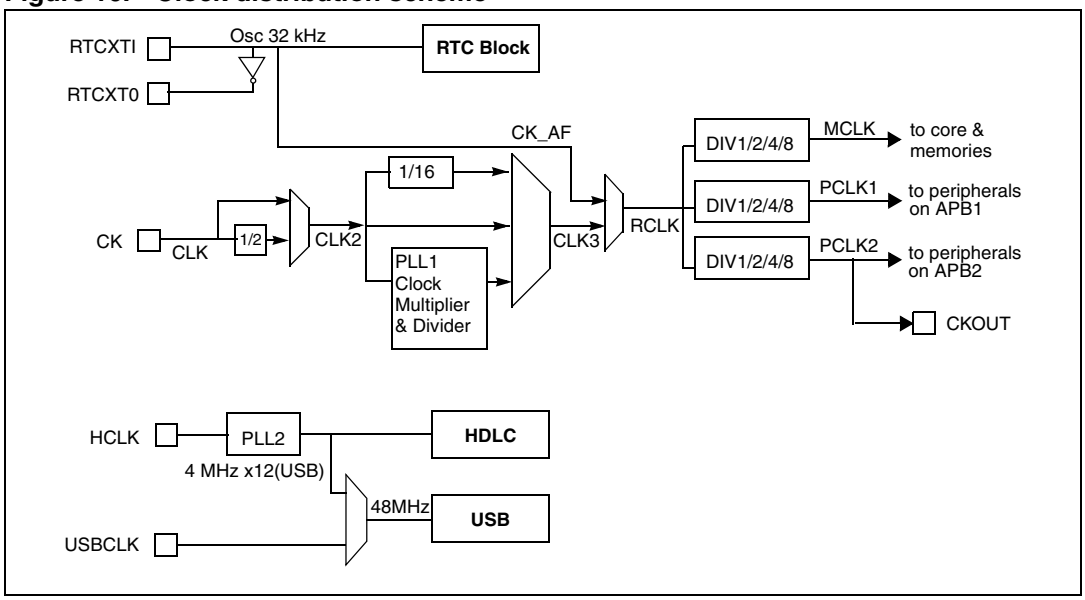
Figure 9. Recommended Signal to be applied on nRSTIN pin



2.4 Clocks

The following figure gives the STR71x clock distribution scheme:

Figure 10. Clock distribution scheme



The source clock CLK is derived from an external oscillator, through the CK pin. This clock may be turned off during Low power modes. (refer to *Section 2.5*)

The system PLL (PLL1) is used to multiply the input internally, to generate the appropriate operating frequency. RCLK is the output of the system PLL or if enabled the CK_AF alternate source (32KHz RTC clock).

Several clock domains exist in the device:

- Main Clock MCLK, including CPU, internal memories, External Memory Interface, PRCCU Registers (except RCCU registers see note 2 below)
- PCLK1, including APB1 peripherals (serial communication peripherals), as listed in Memory Map table
- PCLK2, including APB2 peripherals (system peripherals), as listed in Memory Map table

Each domain may use different frequencies independently by programming the various clock dividers. Having a divider dedicated to the CPU subsystem allows software to dynamically change CPU operating frequency, tailoring computing speed and power consumption to application needs, while maintaining a stable operation of all the peripherals.

On-chip peripherals, mapped in the APB memory space, make use of the RCLK output divided, independently from MCLK, by 2, 4 or 8. Wait states are automatically added by the bus bridge when accessing their registers.

Note that clock may be enabled/disabled independently to each peripheral. Each peripheral may also be reset under software control (Refer to *Section 15*)

Note: 1 It is forbidden to access peripheral registers if the CPU clock MCLK is slower than the related peripheral clocks PCLK1 and PCLK2.

2 If the MCLK clock divider is set to a prescaling value other than 1 (i.e. if RCLK frequency differs from MCLK), it is not possible to access the RCCU registers since they are always clocked by RCLK.

3 For non-intensive operations, PLL1 may be disabled; in addition CLK2 may be divided by a factor of 16, to allow low-power operation while maintaining fast interrupt response.

4 System blocks (ARM7TDMI®, PRCCU, on-chip memories and bridges) are driven by MCLK (Bus clock) and cannot be disabled by software in order to guarantee basic functionalities.

A 32-kHz oscillator is present to maintain a real-time-clock (RTC), with programmable WAKEUP alarm. It may be deactivated if not required; when active it is not influenced by any low-power-mode switch.

The HDLC peripheral can optionally receive its reference clock from an external pin, and may dynamically change its frequency independently from CPU operation. An internal PLL (PLL2) allows the use of a low-frequency external signal, thus reducing power consumption and generated noise.

The USB Interface needs a precise 48 MHz clock reference. This may be generated either externally through the USBCLK pin, or by the internal PLL2, multiplying an external reference at lower speed, if PLL2 is not used by the HDLC interface.

Note: If the USB interface is not used, set bits 0, 1 and 2 in the PCU_PLL2CR register to switch off PLL2 (and reduce power consumption).

Caution: To reduce power consumption, bits 0, 1 and 3 in the RCCU_PER register must be reset by the application software in the initialization phase. These bits are enabled by hardware at reset for factory test purposes only.

2.4.1 PLL1 clock multiplier

The CLK signal drives a programmable divide-by-two circuit. If the DIV2 control bit in RCCU_CFR register is set (reset condition), CLK2, is equal to CLK divided by two; if DIV2 is reset, CLK2 is identical to CLK. In practice, the divide-by-two is used in order to ensure a 50% duty cycle signal.

When PLL1 is active, it multiplies CLK2 by 12, 16, 20 or 24, depending on the status of the MX[1:0] bits in the RCCU_PLL1CR register. The multiplied clock is then divided by a factor in the range 1 to 7, determined by the status of the DX[2:0] bits in the RCCU_PLL1CR register.

When the DX[2:0] bits are programmed to 111, and the FREEN bit in the RCCU_PLL1CR is set to '1', the PLL loop is open and the PLL provides a slow frequency back-up clock which depends on the MX[1:0] and FREF_RANGE bits (refer to *Clock Configuration Reset State on page 32* and *Table 9*). If instead DX[2:0]='111' and FREEN is '0', the PLL is switched off.

The frequency multiplier contains a frequency comparator between CLK2 and the PLL clock output that verifies if the PLL clock has stabilized (locked status). The LOCK bit in the RCCU_CFR register becomes set when this condition occurs and maintains this value as long as the PLL is locked, going back to 0 if for some reason (change of MX[1:0] bits value, stop and restart of PLL or CLK2 and so on) loses the programmed frequency in which it was locked. It is possible to select the PLL clock as system clock only when the LOCK bit is '1'. If the LOCK bit return to '0' the system clock switches back to CLK2 even if the CSU_CKSEL bit is '1'. The PLL selection is further conditioned by the status of the Main Voltage regulator: only when VROK bit in PCU_PWRCR register is '1', that is the Voltage Regulator is providing a stabilized supply voltage, the PLL can be selected (refer to Voltage Regulator specifications). Setting the CSU_CKSEL bit in the RCCU_CFR register allows to select the multiplier clock as system clock, but the two previous conditions must be matched.

Care is required, when programming the PLL multiplier and divider factors, not to exceed the maximum allowed operating frequency for each clock. Refer to the datasheet specifications.

There is no lower limit for MCLK. However, some peripherals can show incorrect operation when the system clock has a too low frequency.

PLL Free Running Mode

The PLL is able to provide a low-precision clock PLLCLK, usable for slow program execution. The frequency range is from 125 kHz to 500 kHz, depending on the MX[1:0] bits and the FREF_RANGE. This mode enabled by the FREEN and DX[2:0] bits in the RCCU_PLL1CR register: when PLL is off and FREEN bit is '1', that is, all these four bits are set, the PLL provides this clock. The selection of this clock is still managed by the CSU_CKSEL bit, but is not conditioned by bits LOCK in RCCU_CFR and VROK in the PCU_PWRCR register. To avoid unpredictable behavior of the PLL clock, the user must set and reset the Free Running mode only when the PLL clock is not the system clock, i.e when the CSU_CKSEL bit is '0'.

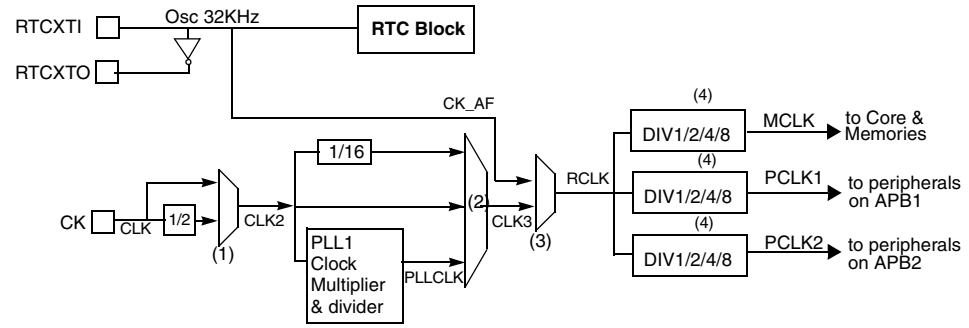
Table 9. PLL1 Free Running Mode Frequency

	Free Running Mode frequency	
MX[1:0]	**FREF_RANGE=0**	**FREF_RANGE=1**
'01, '11'	125 kHz	250 kHz
'00, '10'	250 kHz	500 kHz

2.4.2 Configuring the clocks

The following figure describes the PRCCU registers programming for configuring the clocks:

Figure 11. PRCCU programming



(1)

Div2	CLK2
0	CLK
1	CLK/2

(2)

CSU_CKSEL	CK2_16	CLK3
x	0	CLK2/16
0	1	CLK2
1	1	PLL1 output

(1)&(2) RCCU_CFR Register

(3)

CKAF_SEL	RCLK
0	CLK3
1	CK_AF

(3) RCCU_CCR Register

(4)

FACT[1.0]	MCLK
FACT1[1.0]	PCLK1
FACT2[1.0]	PCLK2
00	RCLK
01	RCLK/2
10	RCLK/4
11	RCLK/8

(4) PCU_MDIV and PDIVR Registers

Clock Configuration Reset State

In reset state, the RCCU_CFR value is 8008h and The PCU_MDIVR and PCU_PDIVR register values (FACT bits) are 0000h. Consequently, in reset state the clock configuration is DIV2 = 1, CK2_16=1 and therefore MCLK, PCLK1 and PCLK2 operate at half the external clock frequency CLK

Typical Clock Configuration Example

For example, to obtain a 48 MHz MCLK with an external CK of 16 MHz,

- Set MX[1:0] = 01 to multiply CLK2 by 12
- Set DX[2:0] = 001 to divide by 2
- Set FREF_RANGE = 1

PRCCU Operating modes

Table 10. PRCCU Operating modes

MODE	RCLK	DIV2	CSU_ CKSEL	MX[1:0]	DX[2-0]	CK2_16	CKAF_ SEL	WFI_CK SEL
PLL1 x 24	CLK2 x 24/N	1	1	1 0	N={DX}+1	1	0	X
PLL1 x 20	CLK2 x 20/N	1	1	0 0	N={DX}+1	1	0	x

Table 10. PRCCU Operating modes

MODE	RCLK	DIV2	CSU_ CKSEL	MX[1:0]	DX[2-0]	CK2_16	CKAF_ SEL	WFI_CK SEL
PLL1 x 16	CLK2 x 16/N	1	1	1 1	N={DX}+1	1	0	x
PLL1 x 12	CLK2 x 12/N	1	1	0 1	N={DX}+1	1	0	x
SLOW 1	CLK2	1	0	X	X	1	0	x
SLOW 2	CLK2 /16	1	X	X	X	0	0	x
SLOW3	CK_AF	X	X	X	X	X	1	x
WFI	If LPOWFI=0, no changes occur on RCLK							
LOWPOWER WFI 1	CLK2 /16	1	X	X	X	X	X	0
LOWPOWER WFI 2	CK_AF	1	X	X	X	X	X	1
RESET	CLK2	1	0	00	111	1	0	0

Note: Refer to Section 2.5 for Low power modes: SLOW, Low Power WFI.

2.4.3 Interrupt generation

The PRCCU generates an interrupt request on the following events:

Table 11. PRCCU interrupts

Event	Description	Event trigger	Interrupt Mask	Event Flag
CK_AF Switching	CK_AF selected or deselected as RCLK source	CK_AF bit in RCCU_CCR register toggles	EN_CKAF bit in RCCU_CCR register	CKAF_I bit in RCCU_CFR register
CLK2/16 Switching	CLK2/16 selected or deselected as RCLK source	CK2_16 bit in RCCU_CFR register toggles	EN_CK2_16 bit in RCCU_CCR register	CK2_16_I bit in RCCU_CFR register
Lock	PLL1 becomes locked or unlocked	LOCK bit in RCCU_CFR register toggles	EN_LOCK bit in RCCU_CCR register	LOCK_I bit in RCCU_CFR register
Stop	CLK restarts after waking up from Stop mode		EN_STOP bit in RCCU_CCR register	STOP_I bit in RCCU_CFR register

When any of these events occur, the corresponding pending bit in the RCCU_CFR register becomes ‘1’ and the interrupt request is forwarded to the interrupt controller. It is up to the user to reset the pending bit as the first instruction of the interrupt routine. The pending bits are clear-only (cleared only by writing ‘1’). Each interrupt can be masked by resetting the corresponding mask bit in the RCCU_CCR register.

2.5 Low power modes

2.5.1 Slow mode

In Slow mode, you reduce power consumption by slowing down the main clock. In Slow mode you can continue to use all the device functions of the chip, but at reduced speed.

To enter Slow mode, RCLK frequency can be forced to CLK2, CLK2 divided by 16, or to CK_AF (32KHz clock) provided CKAF_ST is set, indicating that the Real Time Clock is selected and actually present.

To reduce power consumption, you can turn off the PLL1 by setting bits DX[2:0] in the RCCU_PLL1CR register.

Note: 1 PLL1 can be configured to switch off automatically when the 32KHz CK_AF clock is selected, this can be done by setting the CKSTOP_EN bit in the RCCU_CFR register.

2 When selecting the 32KHz as system clock, you can reduce power consumption by stopping the external oscillator using a GPIO pin.

2.5.2 WFI mode

In WFI mode, you reduce power consumption by stopping the core. The program stops executing, but peripherals are kept running and the register contents are preserved. The device resumes, and execution restarts when an interrupt request is sent to the EIC.

To enter WFI mode, software must write a 0 in the WFI bit of the RCCU_SMR register.

To wakeup from WFI mode an interrupt request must be acknowledged by the EIC.

2.5.3 LPWFI mode

LPWFI (Low power wait for interrupt) is a combination of WFI and Slow modes. In fact, when entering in this low power mode, the following occurs:

- The MCLK clock to the core is stopped
- The peripherals are clocked by the CLK2_16 clock or the CK_AF (32KHz) clock
- The PLL is automatically disabled

To wakeup from LPWFI mode, an interrupt request must be acknowledged by the EIC.

Note: 1 In LPWFI mode the MCU consumption can be reduced by:
- Stopping the Main Voltage Regulator by setting bit LPVRWFI in the PCU_PWRCR register.
- Putting the FLASH in power-down mode by setting bit PWD in the FLASH_CR0 register (refer to the STR7 Flash programming reference manual)
- Stopping the external oscillator when CK_AF is selected

2 After wakeup from LPWFI mode, if the clock selected during LPWFI mode is CK2_16, the system clock (RCLK) switches automatically to CK2.
After wakeup from LPWFI mode, if the clock selected during LPWFI mode is CK_AF, this clock remains the system clock (RCLK).
Refer to Figure 12 and Figure 13 for examples.

Figure 12. Example of LPWFI mode using CK_AF

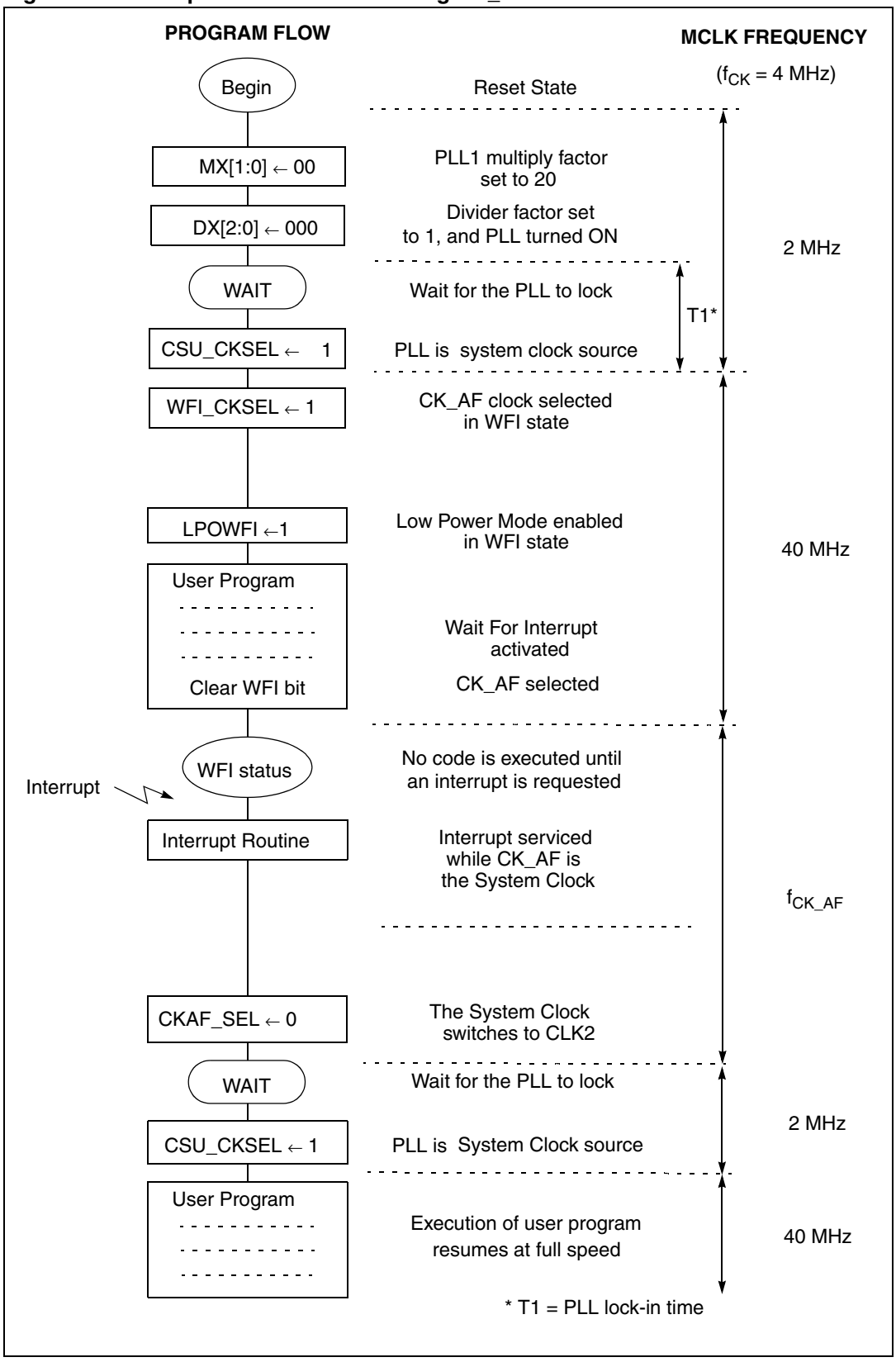
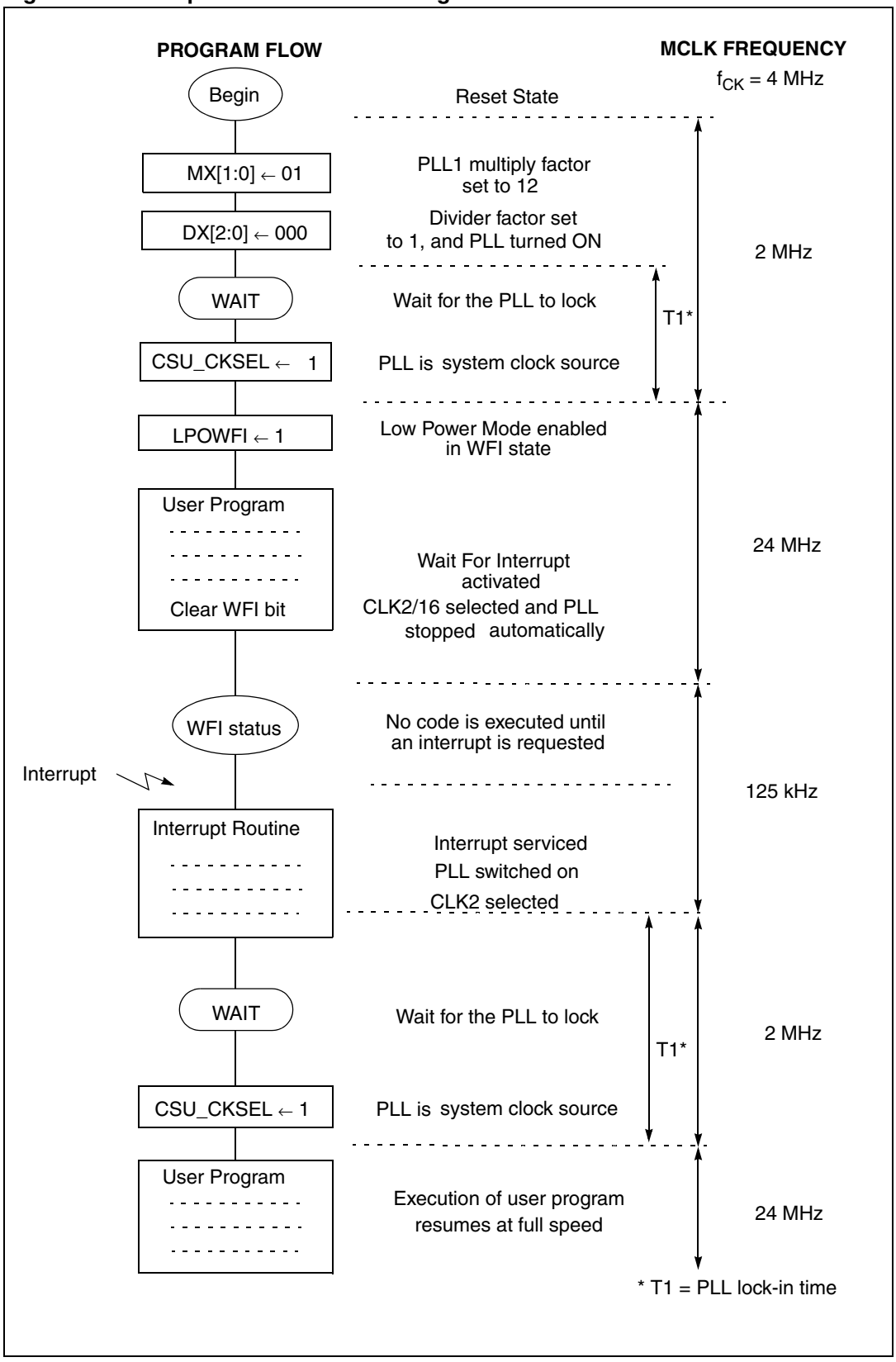


Figure 13. Example of LPWFI mode using CLK2/16



2.5.4 Stop mode

In Stop mode you stop the RCLK (core and peripherals clocks) without resetting the device, hence preserving the MCU status (except the CSU_CKSEL and the STOP_I bits in the RCCU_CFR register).

To enter Stop Mode you have to execute the Stop bit setting sequence described in the XTI chapter. The device will remain in Stop mode until a wake-up line is asserted to restart the program execution.

The MCU resumes program execution after a delay of 2048 CLK clock periods after the Stop mode wakeup event.

On wake-up from Stop mode, the STOP_I bit in the RCCU_CFR register is set and an interrupt is generated if enabled.

Notes:

Note: 1 PLL1 is automatically disabled during STOP mode

2 The MCU power consumption during STOP mode can be reduced by:
- Stopping the Main Voltage Regulator by setting bit LPVRWFI in the PCU_PWRCR register
- Putting the FLASH in power-down mode by setting bit PWD in the FLASH_CR0 register (refer to the STR7 Flash programming reference manual)
- Stopping the external oscillator using a GPIO pin

3 The external oscillator must be enabled when exiting from STOP mode

If a reset occurs while the device is in Stop mode, the clock restarts.

*Note: Assuming MCLK=PCLK, a STOP mode Wake-up event cannot occur less than (N+ 6)*MCLK periods after the start of the STOP bit setting sequence, where N is the number of cycles needed to perform the stop bit setting sequence (refer to STOP Mode Entry Conditions on page 86 for further details). Otherwise the wakeup event will be ignored.*

2.5.5 Standby mode

The Main Voltage Regulator control logic manages the power-down/wake-up sequence to ensure a smooth transition to and from the ultra low-power Standby state.

In Standby mode, the power supply is given through V_{33} pins as normal. The Main Voltage Regulator is switched off and the V_{18} domain (the kernel of the device) is powered off (the voltage on external V_{18} pins falls to zero). The Backup block, including Real Time Clock and Wake-Up logic, is independently powered by the Low-Power Voltage Regulator.

See also *Section 2.1.1: Optional use of external V18BKP supply on page 25*

The power-down sequence is initiated either by a software command, by setting the PWRDWN bit in the PCU_PWRCR register (Software Standby Entry), or by externally forcing the nSTDBY pin to ‘0’ (Hardware Standby Entry).

Note: 1 Since nSTDBY is a bidirectional open-drain pin, an external pull-up is required.

2 Since in Standby mode, the V_{18} supply of the device kernel is switched off, the content of the system RAM is lost and the Watchdog is disabled.

3 When entering Standby mode the V_{18} domain (the kernel of the device) is powered off while the logic belonging to the Backup block is kept powered-on. When leaving Standby mode, all the logic belonging to the V_{18} domain starts from the reset state. This means that the registers (PCU_PWRCR, RTC_ALR and RTC_PRL) that contain bits, which affect the behaviour of Backup logic will in general not reflect its status. These bits are in fact latched

in the Backup domain to retain their value during Standby but on leaving Standby mode, their value is not transferred back to the logic in the V_{18} domain which in turn shows the reset value.

4 *Since any write operation to the Real Time Clock registers requires at least two 32 kHz clock cycles to complete, if software configures the Real Time Clock registers, Standby mode can only be entered after the write operation to the Real Time Clock register is over. This can be monitored by polling the RTOFF bit in the RTC_CRL register.*

5 *Ensure that the Wake-up pin is pulled down when entering Standby mode.*

Software standby entry

Figure 14 and *Figure 15*, show the sequence of events that occur at external signal level and at software level when Standby mode is entered via software.

- All the signals connecting the main kernel (V_{18} domain) and the Backup block are forced to ground to avoid electrical damage.
- The V_{18BKP} domain is disconnected from the V_{18} domain and therefore the Backup block is supplied only by the Low Power Regulator.
- The Main Voltage Regulator is switched off.
- All the I/Os belonging to the kernel are forced into High Impedance.
- The nSTDBY pin is forced low by the device.

Figure 14. Software Standby Mode Entry with WAKEUP Exit

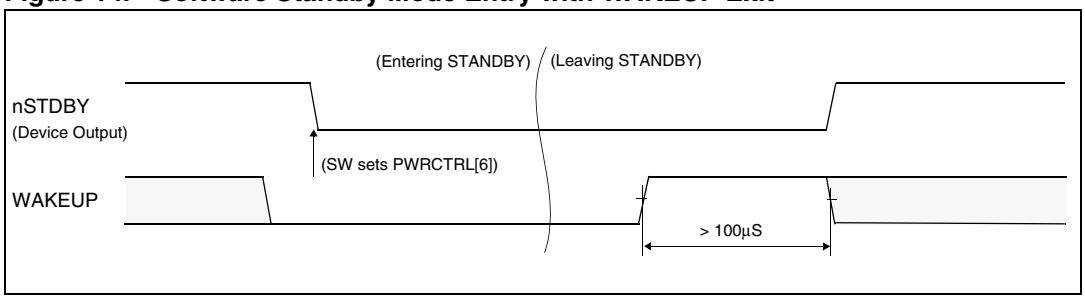


Figure 15. Software Standby Mode Entry with nRSTIN Exit

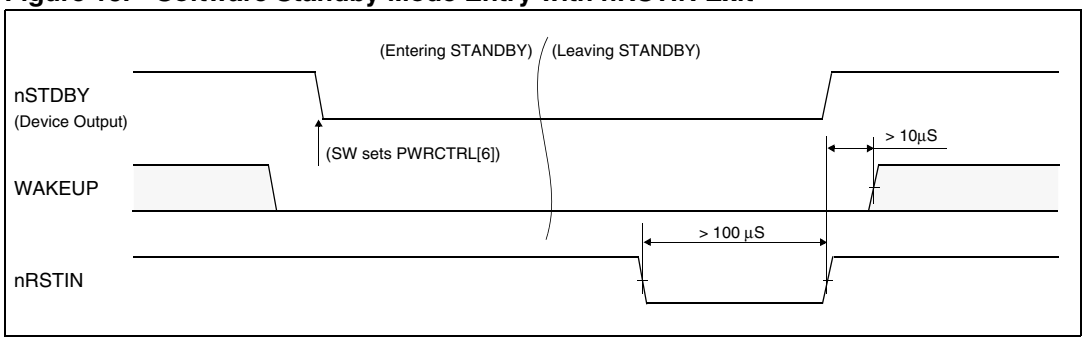
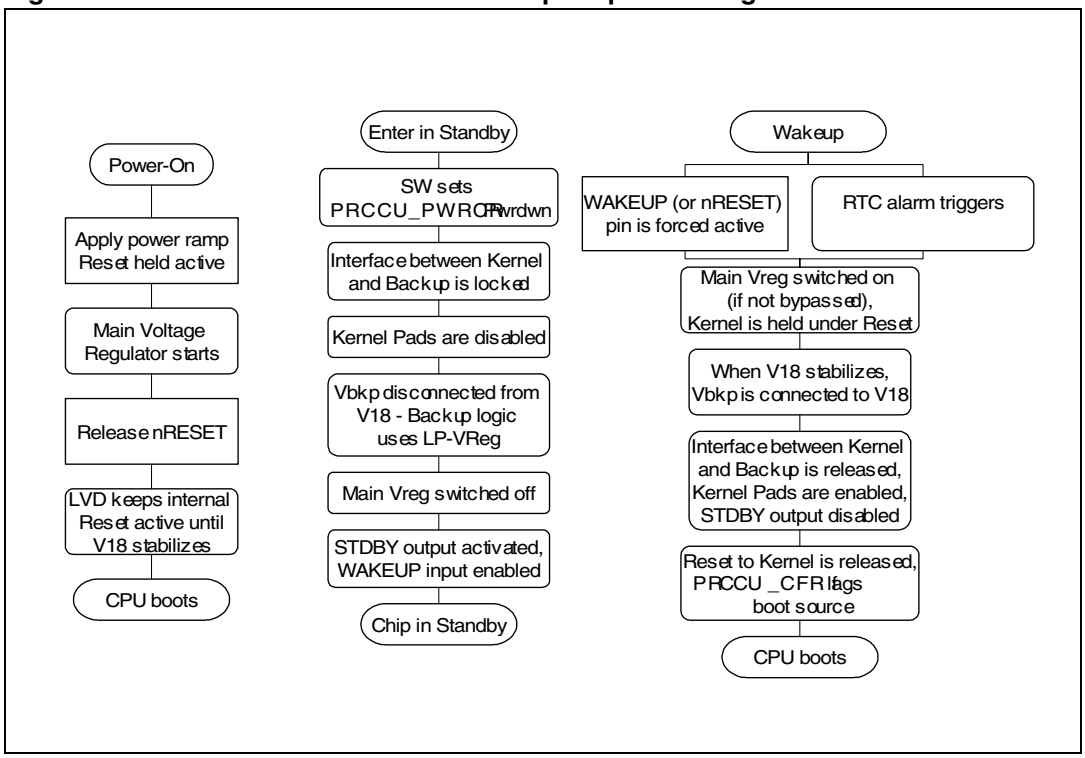


Figure 16. Software Power Down - Wakeup Sequence Diagram



A wake-up event may be generated by the RTC alarm or can be generated externally by a wake-up pin. In this case, a pulse of at least 100µS is necessary. The other source of wake-up is the external nRSTIN pin.

A wake-up event switches the power to the kernel back on. The kernel is kept under reset up to when the internal voltage is correctly regulated. At this point, the interface between the kernel and the Backup block is reconnected, and the CPU restarts from its reset sequence. The flags in the RCCU_CFR register will indicate the wake-up source (RTC, WAKEUP pin, nRSTIN pin, Watchdog, Software).

Note: 1 If the WAKEUP pin is at high logic level, it is not allowed to enter Standby mode.

2 In Standby mode, the RTC alarm will always cause a wake-up event regardless of the mask bit configuration.

Hardware Standby Entry

In case of Standby mode entered via hardware, the sequence is the same but it is started by the external activation of nSTDBY pin, which is hence operating as an input. In this case, rising edges on both WAKEUP and nSTDBY pin are requested to exit from Standby mode (see *Figure 17* and *Figure 18*). The WAKEUP rising edge switches the Main Voltage Regulator back on, while the nSTDBY pin rising edge releases the internal Reset to the V_{18} domain of the device (powered by the Main VR).

Note: 1 In Hardware Standby mode entry, the minimum time between the WAKEUP and nSTDBY rising edges is 100µs.

2 In Standby mode, the RTC alarm will always cause a wake-up event regardless of the mask bit configuration.

3 If WAKEUP pin is at high logic level, it is not allowed to enter Standby mode. It is forbidden to keep WAKEUP pin high while forcing nSTDBY low. It is also forbidden to force nSTDBY high before or after the pulse on both WAKEUP and nRSTIN.

4 A Reset event (nRSTIN activation) has priority over nSTDBY. Therefore, reset activation will force exit from Standby mode. If nRSTIN is activated while nSTDBY is active, the device exits from Standby mode. If a Reset pulse is given while nSTDBY is kept at constant low level, the device will enter Standby mode again after nRSTIN rising edge.

5 In order to wake-up the system using the WAKEUP pin, the WAKEUP signal pulse width must be held active high for at least 100us. A pulse width of less than 100us may affect the system.

Figure 17. HW Standby Mode Entry with WAKEUP Exit

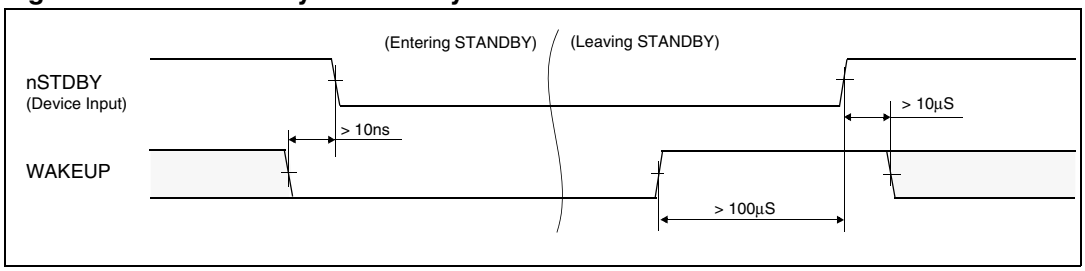
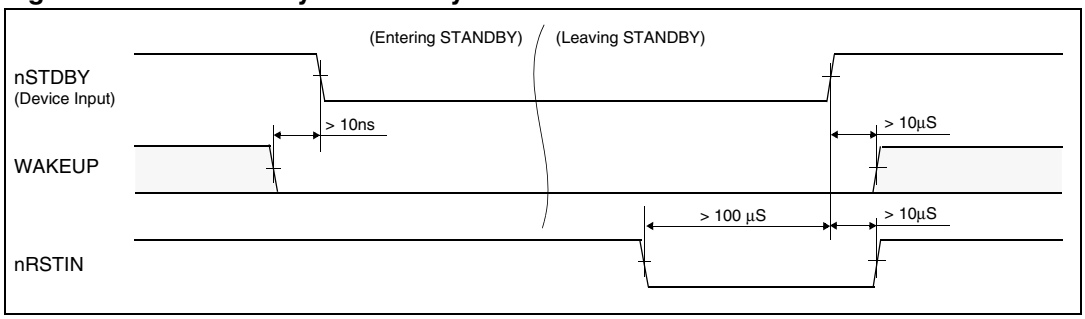


Figure 18. HW Standby Mode Entry with nRSTIN Exit



Wakeup / RTC alarm reset

When a Wakeup event restarts the device from the Standby Mode and power supply is reapplied to the V_{18} domain, the reset is activated. The reset source is indicated by a bit set in RCCU_CFR register.

2.6 Register description

In this section, the following abbreviations are used:

read/write (rw)	Software can read and write this bit
read only (r)	Software can only read this bit, writing to it has no effect.
read/clear (rc_w0)	Software can read as well as clear this bit by writing ‘0’. Writing ‘1’ has no effect.
read/clear (rc_w1)	Software can read as well as clear this bit by writing ‘1’. Writing ‘0’ has no effect.

2.6.1 Clock control register (RCCU_ CCR)

Base address: 0xA000 0000h
Address offset: 0x00h
Reset value:0x0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				EN_ HALT	EN_ STOP	EN_ CK2_16	EN_ CKAF	EN_ LOCK	Reserved			SRES EN	CKAF _SEL	WFI_ CKSEL	LOP_ WFI
				rw	rw	rw	rw	rw				rw	rw	rw	rw

Bits 31:12	Reserved, always read as 0.
Bit 11	**EN_HALT**: *Enable Halt bit.* 0: No Software Reset. 1: Software Reset when software sets the HALT bit in the RCCU_SMR register and if SRESEN=1
Bit 10	**EN_STOP**: *Stop Interrupt Masking bit.* 0: Stop interrupt request disabled 1: Stop interrupt request enabled.
Bit 9	**EN_CK2_16**: *CK2_16 Interrupt Masking bit.* 0: CK2_16 switching interrupt disabled 1: CK2_16 switching interrupt enabled
Bit 8	**EN_CKAF**: *CKAF Interrupt Masking bit.* 0: CKAF switching interrupt request disabled 1: CKAF switching interrupt request enabled
Bit 7	**EN_LOCK**: *Lock Interrupt Masking bit.* 0: LOCK switching interrupt request disabled 1: LOCK switching interrupt request enabled
Bits 6:4	Reserved, always read as 0.
Bit 3	**SRESEN**: *Software Reset Enable.* 0: No Software Reset. 1: Software Reset when software sets the HALT bit in the RCCU_SMR register and if EN_HALT=1.

Bit 2	**CKAF_SEL**: *Alternate Function Clock Select.* 0: Deselect CK_AF clock 1: Select CK_AF clock. **Note:** To check if the selection has actually occurred, check that CKAF_ST is set. If no Real Time Clock is present, the selection will not occur.
Bit 1	**WFI_CKSEL**: *WFI Clock Select.* This bit selects the clock used in Low power WFI mode if LPOWFI = 1. 0: MCLK during Low Power WFI is CLK2/16 1: MCLK during Low Power WFI is CK_AF, providing it is present. In effect this bit sets CKAF_SEL in WFI mode
Bit 0	**LPOWFI**: *Low Power mode during Wait For Interrupt.* 0: Low Power mode during WFI disabled. When WFI is executed, MCLK is unchanged 1: The device enters Low Power mode when the WFI instruction is executed. The clock during this state depends on WFI_CKSEL

2.6.2 Clock flag register (RCCU_CFR)

Base address: 0xA000 0000h
Address offset: 0x08h
Reset Value: 0000 8408h after an external WakeUp Reset
0000 8208h after a Voltage Regulator low voltage detector Reset
0000 8088h after a RTC Alarm WakeUp Reset
0000 8048h after a Watchdog Reset
0000 8028h after a Software Reset
0000 8008h after an External (Power-On) Reset

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DIV2	STOP_I	CK2_16_I	CKAF_I	LOCK_I	WKP_RES	LVD_RES	Res.	RTC_ALARM	WDG RES	SOFT RES	CKSTOP_EN	CK2_16	CKAF_ST	LOCK	CSU_CKSEL
rw	rc_w1	rc_w1	rc_w1	rc_w1	r	r		r	r	r	rw	rw	r	r	rw

Bits 31:16	Reserved, always read as 0.
Bit 15	**DIV2**: *OSCIN Divided by 2.* This bit controls the divide-by-2 circuit which operates on the CLK signal. 0: No division of CLK frequency. 1: CLK is divided by 2.
Bit 14	**STOP_I**: *Stop Interrupt pending bit.* This bit is clear only. 0: No Stop interrupt request pending. 1: Stop Interrupt request is pending.
Bit 13	**CK2_16_I**: *CK2_16 switching Interrupt pending bit.* This bit is clear only. 0: No CK2_16 Interrupt request pending. 1: CK2_16 Interrupt request pending.

Bit 12	**CKAF_I**: *CK_AF switching Interrupt pending bit.* This bit is clear only. 0: No CK_AF switching Interrupt request pending. 1: CK_AF switching Interrupt request pending.
Bit 11	**LOCK_I**: *Lock Interrupt pending bit.* This bit is clear only. 0: No Lock Interrupt request pending. 1: Lock Interrupt request pending.
Bit 10	**WKP_RES**: *External wakeup flag.* This bit is read only. 0: No WakeUp reset occurred. 1: Reset was generated by an External WakeUp event in Standby mode.
Bit 9	**LVD_RES**: *Voltage Regulator low voltage detector reset flag.* This bit is read only. 0: No voltage regulator low voltage detector reset occurred. 1: Voltage regulator LVD reset occurred.
Bit 8	Reserved, always read as 0.
Bit 7	**RTC_ALARM**: *Real-Time-Clock alarm reset flag.* This bit is read only. 0: No RTC alarm event occurred. 1: Reset was generated by RTC alarm during Standby mode.
Bit 6	**WDG_RES**: *Watchdog reset flag.* This bit is read only. 0: No Watchdog reset occurred. 1: Watchdog reset occurred.
Bit 5	**SOFTRES**: *Software Reset Flag.* This bit is read only. 0: No software reset occurred. 1: Software reset occurred.
Bit 4	**CKSTOP_EN**: *Clock Stop Enable* This bit is set and cleared by software 0: PLL1 is kept enabled even if CK_AF has been selected. 1: PLL1 is disabled if CK_AF has been selected.
Bit 3	**CK2_16**: *CLK2/16 Selection* 0: CLK2/16 is selected as RCLK source and the PLL is off. 1: The RCLK source is CLK2 (or the PLL output depending on the value of CSU_CKSEL)
Bit 2	**CKAF_ST**: *CK _AF Status.* This bit is read only. 0: The PLL clock, CLK2 or CLK2/16 is the RCLK source (depending on CSU_CKSEL and CK2_16 bits). 1: CK_AF is the RCLK source

Bit 1	**LOCK**: *PLL locked-in* This bit is read only. 0: The PLL is turned off or not locked and cannot be selected as RCLK source. 1: The PLL is locked
Bit 0	**CSU_CKSEL**: *CSU Clock Select* This bit is set and cleared by software. It is kept reset by hardware when: –bits DX[2:0] (RCCU_PLL1CR) are set to 111 –the quartz is stopped (by hardware or software) –the CK2_16 bit (RCCU_CFR) is forced to '0' 0: CLK2 provides the system clock 1: The PLL Multiplier provides the system clock if LOCK and VROK bits are '1' If the FREEN bit is set, this bit selects this clock independently of the LOCK and VROK bits. **Note**: Setting the CKAF_SEL bit overrides any other clock selection. The clearing the CK2_16 bit overrides the CSU_CKSEL selection (see *Figure 11 on page 32*)

2.6.3 PLL1 Configuration Register (RCCU_PLL1CR)

Base address: 0xA000 0000h
Address offset: 0x18h
Reset value: 0000 0007h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								FREEN	FREF_RANGE	MX1	MX0	-	DX2	DX1	DX0
								rw	rw	rw	rw		rw	rw	rw

Bits 31:8	Reserved, always read as 0.
Bit 7	**FREEN**: *PLL Free Running Mode Enable* 0: Free Running mode disabled. In this case, PLL1 operation depends only on the MX[1:0] and DX[2:0] bits. 1: Free Running mode enabled. In this mode, when all three DX[2:0] bits are set, the PLL is not stopped but provides a slow frequency back-up clock, selected by the CSU_CKSEL bit; Operation in this mode does not require the LOCK and VROK bits to be set.
Bit 6	**FREF_RANGE**: *Reference Frequency Range selector bit* 0: Configure PLL1 for input frequency (CLK2) of 1.5 to 3 MHz 1: Configure PLL1 for input frequency (CLK2) of greater than 3 MHz
Bits 5:4	**MX[1:0]**: *PLL1 Multiplication Factor* These bits are written by software to define the PLL1 multiplication factor 00: CLK2 * 20 01: CLK2 * 12 10: CLK2 * 24 11: CLK2 * 16 **Note:** It is recommended to deselect and switch-off PLL1 before changing MX values.
Bit 3	Reserved, always read as ‘0’.
Bits 2:0	**DX[2:0]**: *PLL1 output clock division factor* These bits are written by software to define the PLL1 division factor. 000: PLLCK / 1 001: PLLCK / 2 010: PLLCK / 3 011: PLLCK / 4 100: PLLCK / 5 101: PLLCK / 6 110: PLLCK / 7 111: FREEN = 0: CLK2 (PLL1 OFF, Reset State) FREEN = 1: PLL1 in Free Running mode

2.6.4 Peripheral enable register (RCCU_PER)

Base address: A000 0000h
Address offset: 1Ch
Reset value: 0001 FFFFh

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PH_CK[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PH_CK[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0	**PH_CK[32:0]**: *Peripheral Clock enable*. 0: the peripheral clock is stretched, stopping the peripheral. 1: the peripheral clock is enabled, allowing peripheral operation. The relation between register bits and peripherals is shown in *Table 12*.

Table 12. Peripheral clock management

PH_CKEN Reg. Status	Peripheral Stopped
PH_CK[1:0] = 0	not used[1)]
PH_CK[2] = 0	EMI
PH_CK[3] = 0	not used[1)]
PH_CK[4] = 0	USB KERNEL
PH_CK[16:5] = 0	not used[1)]
PH_CK[31:17] = 0	not used

[1)] To reduce power consumption, these bits should be cleared by software after reset.

2.6.5 System mode register (RCCU_SMR)

Base address: A000 0000h
Address offset: 20h
Reset value: 0000 0001h

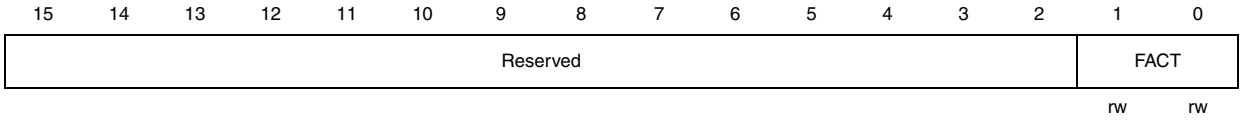
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved														HALT	WFI
														rw	rw

Bits 31:2	Reserved, always read as 0.
Bit 1	**HALT**: *Halt* 0: No effect 1: Generate a Software reset if the SRESEN bit and the ENHALT bit in the RCCU_CCR register is set.
Bit 0	**WFI**: *Wait For Interrupt mode* 0: Enter WFI (Wait For Interrupt) mode. In this mode the CPU remains in idle state until an interrupt request is acknowledged by the EIC. When this occurs, the bit is set to ‘1’ again. This means that this bit, once reset, can only be set to ‘1’ by hardware. 1: No effect **Caution**: If all EIC interrupt channels are masked, clearing this bit will stop program execution indefinitely unless the device is reset. Hence you must ensure that at least one interrupt channel is enabled before clearing the WFI bit.

2.6.6 MCLK divider control (PCU_MDIVR)

Address offset: 40h
Reset value: 0000h

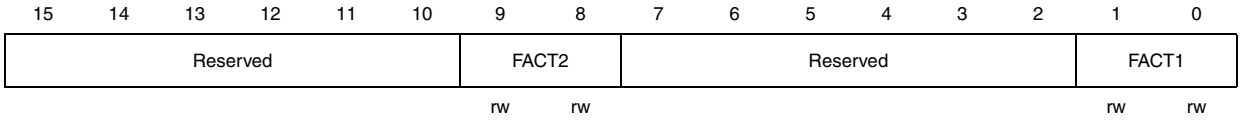


This register sets the prescaling factor for the Main System Clock MCLK. It may be written by software at any time, to dynamically adjust the operation frequency.

Bits 15:2	Reserved, always read as 0.
Bits 1:0	**FACT[1:0]:** *Division factor* These bits are written by software to define the MCLK prescaler division factor. 00: Default, no prescaling, MCLK = RCLK 01: MCLK = RCLK / 2 10: MCLK = RCLK / 4 11: MCLK = RCLK / 8

2.6.7 Peripheral clock divider control register (PCU_PDIVR)

Address offset: 44h
Reset value: 0000h



This register sets the prescaling factor for the two APB Clocks PCLK1, for peripherals belonging to the APB1 group, and PCLK2, for peripherals belonging to the APB2 group. It may be written by software at any time.
FACT1 and FACT2 may be chosen independently.

Note: Peripheral clock speed must be equal or lower than CPU clock speed, but lower or equal to the values specified in the datasheet. It is up to the user to ensure this condition is always met. Unexpected behaviour may occur otherwise.

Bits 15:10	Reserved, always read as 0.
Bits 9:8	**FACT2[1:0]:** *Division factor for APB2 peripherals* 00: Default, no prescaling, PCLK2 = RCLK 01: PCLK2 = RCLK / 2 10: PCLK2 = RCLK / 4 11: PCLK2 = RCLK / 8

Bits 7:2	Reserved, always read as 0.
Bits 1:0	**FACT1[1:0]:** *Division factor for APB1 peripherals* 00: Default, no prescaling, PCLK1 = RCLK 01: PCLK1 = RCLK / 2 10: PCLK1 = RCLK / 4 11: PCLK1 = RCLK / 8

2.6.8 Peripheral reset control register (PCU_RSTR)

Address offset: 48h
Read/Write
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													EMI RST	Reserved	
												rw	rw		

This register allows to force a Reset activation individually to most system blocks. Not all system blocks may be reset by software, to guarantee consistent behaviour of the device.

Bits 15:4	Reserved, always read as 0.
Bit 3	Reserved for factory test. To reduce power consumption, this bit must be set by software.
Bit 2	**EMIRST:** *External Memory interface reset* 0: Normal EMI operation 1: Force External Memory Interface to reset state. Reset activation/ deactivation is synchronous with system clock MCLK.
Bits 1:0	Reserved for factory test. To reduce power consumption, these bits must be set by software.

2.6.9 PLL2 control register (PCU_PLL2CR)

Address offset: 4Ch
Reset value: 0033h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LOCK	Reserved				IRQ PEND	IRQ MASK	USB EN	PLL EN	FRQ RNG	MX (1:0)		-	DX(2:0)		
r					rc_w1	rw	rw	rw	rw	rw			rw		

This register controls operation of the PLL2, dedicated to HDLC or USB blocks.

Bit 15	**LOCK**: *PLL2 Locked* Read Only. This bit is set by HW when the PLL2 is locked on the input reference clock and provides a stable frequency. Any change of this bit may generate an interrupt request, if enabled by setting bit 9, IRQ MASK.
Bits 14:11	Reserved.

Bit 10	**IRQ PEND**: *Interrupt Request to CPU on LOCK transition Pending* Set by hardware, Clear Only by software. When this bit is set a Lock Status Change Interrupt request is pending. This interrupt request is mapped on the PRCCU Interrupt vector. The pending request is removed by WRITING this bit TO ONE.
Bit 9	**IRQ MASK**: *Enable Interrupt Request to CPU on LOCK transition* When this bit is reset (default) no interrupt is generated by PLL2; when it is set any change in LOCK status will generate an Interrupt request. The pending request is cleared by reading this register.
Bit 8	**USBEN**: *Enable PLL clock to USB* When this bit is reset (default) the 48 MHz reference clock for USB is connected to USBCLK pin; when it is set the reference clock for USB is provided by HCLK pin, through PLL2. Input frequency, Multiplication and Division factors must be appropriately chosen to guarantee the precision required by USB standard.
Bit 7	**PLLEN**: *Select PLL2* When this bit is reset (default) PLL2 is bypassed, but not switched off, and HCLK drives the internal logic directly (HDLC, or USB if selected setting bit 8, USBEN. When PLLEN is set the PLL output is selected as source clock for the logic. It is forbidden to set this bit if the bit 15, LOCK, is reset. If the PLL unlocks for any reason, this bit is reset by Hardware. **Note:** To switch off PLL2, and reduce power consumption, set the DX[2:0] bits.
Bit 6	**FRQRNG**: *PLL2 frequency range selection* This bit has to be set by software when the PLL input frequency (HCLK pin) is in the range 3-5 MHz; This bit has to be cleared by software when the PLL input frequency (HCLK pin) is in the range 1.5-3 MHz
Bits 5:4	**MX[1:0]**: *PLL2 Multiplication Factor.* These bits are written by software to define the PLL2 multiplication factor 00: CLK2 * 20 01: CLK2 * 12 10: CLK2 * 28 11: CLK2 * 16
Bit 3	Reserved, always read as 0.
Bits 2:0	**DX[2:0]**: *PLL2 output clock divider factor.* These bits are written by software to define the PLL division factor. 000: PLLCK / 1 001: PLLCK / 2 010: PLLCK / 3 011: PLLCK / 4 100: PLLCK / 5 101: PLLCK / 6 110: PLLCK / 7 111: BYPASS (PLL OFF)

2.6.10 Boot configuration register (PCU_BOOTCR)

Address offset: 50h

Reset value: 0000 00p1 1100 00bb
where p: depending on package - b: depending on BOOT pin values (see below)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						PKG64	res.	HDLC	CAN	ADC EN	LPOW DBG EN	USB FILT EN	BSPI0 EN	BOOT	
						r	r	r	r	rw	rw	rw	rw	rw	

This register includes Boot options and other global configuration controls.
Boot options are values latched on external BOOT pins at reset time, and visible to sofware through these bits; they can be modified afterward to restore a different configuration (memory map). See *Section 1.2 on page 18* for full description.
Configuration bits are fixed (hardware) device options, not available to user modification. These bits allow reading the configuration value by the software.

Bit 15:10	Reserved.
Bit 9	**PKG64**: *Die is hosted in 64-pin package* *HW Configuration. Read only.* When this bit is cleared, the chip is mounted on a 144-pin package. When this bit is set, the chip is mounted on a 64-pin package, and all non-bonded pins are forced to idle mode (input and output disabled), irrespective of the values in IOPort control registers.
Bit 8	Reserved.
Bit 7	**HDLC**: *HDLC Active* *HW Configuration. Read only.* When this bit is cleared, the HDLC interface controller is disabled. When this bit is set, the HDLC interface controller is enabled.
Bit 6	**CAN**: *CAN Active* *HW Configuration. Read only.* When this bit is cleared, the CAN interface controller is disabled. When this bit is set, the CAN interface controller is enabled.
Bit 5	**ADC EN**: *Enable ADC* *Read/Write.* When this bit is cleared (Reset value), the Analog part of the Analog-to-digital converter is disabled (Power Down), minimizing static power consumption. When this bit is set, the Analog part of the ADC is enabled, and can be used by the application. After setting this bit, a start-up time of 1 ms (TBC) is required before the first valid conversion. This bit should be reset by software to switch off the ADC in any low-power mode (WFI, STOP)

Bit 4	**LPOWDBGEN**: *Enable Reserved Debug features for STOP mode* *Read/Write.* When the device is in STOP mode, all internal clocks are frozen, including MCLK to ARM7, which cannot answer to a Debug Request from the Emulator (DBGRQS pin, or command through JTAG interface). When this bit is set, asserting the Debug Request input forces immediate exit from STOP mode, enabling internal clocks, so allowing the emulator to take control of the system. When this bit is cleared (Reset value), this feature is disabled, and a Debug Request while in STOP mode is ignored.
Bit 3	**USBFILT EN**: *Enable USB Standby Filter* *Read/Write.* The USB Transceiver features a low-pass filter to improve noise immunity while the bus is in STANDBY mode. When this bit is cleared (Reset value), the filter is disabled. When this bit is set, the filter is enabled (subject to STANDBY condition of USB bus).
Bit 2	**BSPI0 EN**: *Enable BSPI0* *Read/Write.* When this bit is cleared (Reset value), the BSPI0 interface controller is disabled, and pins P0.0 to P0.3 can be used by UART3 and I2C1. When this bit is set, the BSPI0 interface controller is enabled, and will access pins P0.0 to P0.3 (if they are programmed as Alternate Function). In this state UART3 and I2C1 are not available to the application.
Bits 1:0	**BOOT[1:0]:** *Boot Mode* These bits report the Boot configuration as selected by the user on Boot[1:0] pins or through JTAG programming. 00: Flash memory boot mode (default) 01: Reserved 10: RAM boot mode 11: External memory boot mode The memory bank corresponding to BOOT value is mapped at address 0000.0000 in addition to its normal position in the memory map, and the CPU starts fetching code from this block after end of Reset sequence. These bits may subsequently be modified by software, to map whatever memory at address 0000.0000.

2.6.11 Power control register (PCU_PWRCR)

Address offset: 54h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WR EN	BUSY	WKUP ALRM	VR OK	Reserved		FLASH LP	LVD_DIS	OSC BYP	PWR DWN	LPVR BYP	LPVR WFI	VR BYP	Reserved		
rw	r	r	r			rw	rws	rw	rw	rw	rw	rw			

Bit 15	**WREN**: *Register Write Enable* This bit acts as a security mechanism to avoid spurious writes into this register. When WREN = '0' (default), all bits in the register are write protected, and cannot be changed. This bit can only be set by software, not cleared. When WREN = '1', bits listed as RW become writable. A timeout is activated, allowing write operations to the register for 64 MCLK cycles. After this interval, the bit is reset by HW. This bits does not affect read operations.
Bit 14	**BUSY**: *Backup logic Busy - programming ongoing* *Read only* The Backup Logic is asynchronous, and this bits flags the handshake status between the register and the related logic. When BUSY = '0' (default), it is possible to program the Backup Logic by writing into the register. When BUSY = '1', a previous write operation is not yet completed, and it is forbidden to write to the register.
Bit 13	**WKUP-ALRM**: *WakeUp or Alarm active* *Read only* This bit is set to logical one by the HW when external WAKEUP pin, or when an internal wake-up source is active. In this case a powerdown command has no effect. When WKUP-ALRM = '0' (default), it is possible to start the power down sequence.
Bit 12	**VROK**: *Main Regulator OK* *Read Only.* When VROK = '0', the Main Voltage Regulator is not stable, and the supply to the device is not guaranteed to be in specification range. When VROK = '1', the Main Voltage Regulator is stable, in the specification range **Note**: The software should check this bit before and after any change of configuration which impacts clock and power status of the device.
Bits 11: 10	Reserved, always read as 0.

Bit 9	**FLASH LP**: *Flash low-power (low-speed) mode select* This bit may be written only when bit 15, WREN, is set. When this bit is cleared (Default Value), the Flash works in FAST mode, using BURST mode for sequential accesses, to allow zero wait state operation up to the maximum device frequency, and generating a WAIT cycle on non-sequential memory accesses. When this bit is set, the Flash enters Low-Power mode, and BURST is disabled. WAIT states are never generated. LP mode may be used for operation frequency (MCLK) up to 33 MHz. **Note:** Correct operation of the device if this bit is set while MCLK runs faster than the maximum rated speed is not guaranteed.
Bit 8	**LVD DIS**: *Voltage Regulator Low Voltage Detection Disable* This bit may be written only when bit 15, WREN, is set. Sticky bit: once set, it cannot be reset by software anymore - only a Reset will clear it. When this bit is cleared (Default Value), a Reset will be generated if the supply voltage drops below the threshold of the low voltage detector of either of the Voltage Regulators. When this bit is set, no Reset will be generated when a voltage drop occurs. **Note:** For security reasons, it is allowed to set this bit only when the Low Power voltage regulator is disabled (LPVRBYP bit = 1).
Bit 7	**OSC BYP**: *32-kHz Oscillator Bypass Enable* This bit may be written only when bit 15, WREN, is set. When this bit is cleared (Default Value), the 32-KHz oscillator is enabled, providing a clock source for the Real Time Clock, and a Backup clock source to the whole system. When this bit is set, the 32-KHz oscillator is stopped and bypassed, featuring zero power consumption and allowing an external reference clock to feed the Real Time Clock, or a Backup clock source to the whole system.
Bit 6	**PWRDWN**: *Activate Standby Mode* This bit may be written only when bit 15, WREN, is set. When PWRDWN = '0' (default), the chip is working in normal mode. When PWRDWN = '1', the chip enters Standby mode. The Main Voltage Regulator is switched off, and the power supply to the kernel of the device is disconnected. The Backup (Low-Power) Voltage Regulator is still active, supplying the Backup section: Real Time Clock and wake-up logic.
Bit 5	**LPVRBYP**: *Low Power Regulator Bypass* This bit may be written only when bit 15, WREN, is set. When LPVRBYP = '0' (default), the Backup (Low-Power) Voltage Regulator is active, ready to take over in low-power modes. When LPVRBYP = '1', the Backup (Low-Power) Voltage Regulator is switched off (bypassed), and the Backup logic is supplied by an external source at 1.8V through the V18bkp pin. NOTE: if this bit is set without a proper external supply connection, unexpected operation may result, including permanent damage to the device.
Bit 4	**LPVRWFI**: *Low Power Regulator in Wait-For-Interrupt mode.* This bit may be written only when bit 15, WREN, is set. When LPVRWFI = '0' (default), the main Voltage Regulator is always active, except in Standby mode. When LPVRWFI = '1', the main Voltage Regulator is switched off (bypassed) in Low-Power modes STOP, LP_WFI (see *Section 2.5 on page 34*), in which the Backup (Low-Power) Voltage Regulator supplies the device.

Bit 3	**VRBYP**: *Main Regulator Bypass* This bit may be written only when bit 15, WREN, is set. When VRBYP = '0' (default), the Main Voltage Regulator is active, supplying the device. When VRBYP = '1', the Main Voltage Regulator is unconditionally switched off. In this case, the device is only powered by the Low Power Voltage Regulator. In this configuration the maximum allowed operation frequency is 1MHz and the PLL is disabled.
Bits 2:0	Reserved, always read as 0.

2.7 PRCCU register map

Table 13. PRCCU register map

Addr. Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	RCCU_CCR	-	-	-	-	EN_HALT	EN_STOP	EN_CK2_16	EN_CKAF	EN_LOCK	-	-	-	SRESEN	CKAF_SEL	WFI_CKSEL	LOP_WFI
8	RCCU_CFR	DIV2	STOP_I	CK2_16_I	CKAF_I	LOCK_I	WKP_RES	LVD_RES	-	RTC_ALARM	WDGRES	SOFTRES	CKSTOP_EN	CK2_16	CKAF_ST	LOCK	CSU_CKSEL
18	RCCU_PLL1CR	-	-	-	-	-	-	-	-	FREEN	FREF_RANGE	MX1	MX0	-	DX2	DX1	DX0
1C	RCCU_PER	-	-	-	-	-	-	-	-	-	-	-	PH_CK4	PH_CK3	PH_CK2	PH_CK1	PH_CK0
20	RCCU_SMR															HALT	WFI
40	PCU_MDIVR	reserved														FACT	
44	PCU_PDIVR	reserved						FACT2		reserved						FACT1	
48	PCU_RSTR	RST[15:0]															
4C	PCU_PLL2CR	LOCK	reserved				IRQ PEND	IRQ MASK	USB EN	PLL EN	FRQ RNG	MX (1:0)		-	DX(2:0)		
50	PCU_BOOTCR	reserved						PKG 64	-	HDLC	CAN	ADC EN	LPOWDBG EN	USBFILT EN	BSPI0 EN	BOOT	
54	PCU_PWRCR	WR EN	BUSY	WKUP ALRM	VR OK	-		FLASH LP	LVD DIS	OSC BYP	PWR DWN	LPVR BYP	LPVR WFI	VR BYP	-	-	-

See *Table 1* for base address

3 I/O ports

3.1 Functional description

Each of the General Purpose I/O Ports has three 16-bit configuration registers (PC0, PC1, PC2) and one 16-bit Data register (PD).

Subject to the specific hardware characteristics of each I/O port listed in the datasheet device pin description table, you can configure each port bit individually as input, output, alternate function etc.

Each I/O port bit is freely programmable, however the I/O port registers have to be accessed as 16-bit words. Word (32-bit) or byte accesses are not allowed.

Figure 19 shows the basic structure of an I/O Port bit.

Figure 19. Basic Structure of an I/O Port Bit

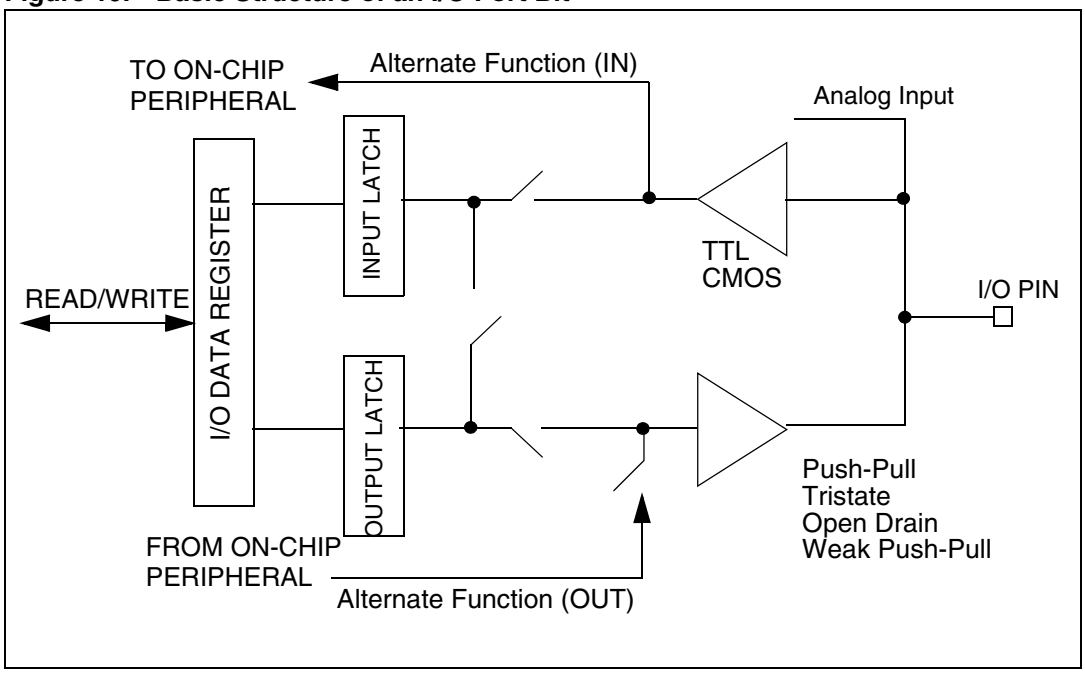


Table 14. Port bit configuration table

Port Configuration Registers (bit)	Values							
PC0(n)	0	1	0	1	0	1	0	1
PC1(n)	0	0	1	1	0	0	1	1
PC2(n)	0	0	0	0	1	1	1	1
Configuration	HiZ/AIN	IN	IN	IPUPD	OUT	OUT	AF	AF
Output	TRI	TRI	TRI	WP	OD	PP	OD	PP
Input	AIN	TTL	CMOS	CMOS	N.A.	N.A.	CMOS	CMOS

Notes:
AF: Alternate Function
AIN: Analog Input
CMOS: CMOS Input levels
HiZ: High impedance
IN: Input
IPUPD: Input Pull Up /Pull Down
N.A. not applicable. In Output mode, a read access the port will get the output latch value). See *Figure 22*.
OD:
OUT: Output
PP: Push-Pull
TRI: Tristate
TTL: TTL Input levels
WP: Weak Push-Pull

3.1.1 General purpose I/O (GPIO)

At reset the I/O ports are configured as general purpose (memory mapped I/O).

When you write to the I/O Data register the data is always loaded in the Output Latch. The Output Latch holds the data to be output while the Input Latch captures the data present on the I/O pin.

A read access to the I/O Data register reads the Input Latch or the Output Latch depending on whether the Port bit is configured as input or output.

3.1.2 Bit-wise write operations

The bit-wise instructions proposed by the "ARM7 Instruction Set" can only apply on the internal ARM7 Ri registers. Consequently, it is not possible to perform directly bit-wise write operations (like a bit set or a bit clear) of an I/O Port register. This has to be done in three operations:

- Load the whole 16-bit Port Data register into an Ri register
- Modify the Ri register using the bitwise ARM7 instruction.
- Store back the whole 16-bit result into the Port Data Register.

Because this is not an atomic operation, it is possible that an Interrupt Subroutine (ISR) is served between the Load and the Store access. If this ISR sets or clears some other bits of the Port register, storing back the Port register can corrupt the Port. Consequently, you need to disable the interrupts during these operations if the Interrupt Sub Routines are susceptible to modify the other bits of the I/O port.

3.1.3 Alternate function I/O (AF)

The alternate functions for each pin are listed in the datasheet. If you configure a port bit as Alternate Function, this disconnects the output latch and connects the pin to the output signal of an on-chip peripheral.

- For alternate function inputs, the port must be configured in Input mode and the input pin must be driven externally.

Note: It is also possible to emulate by software the AFI input pin by programming the GPIO controller. In this case, the port should be configured in Alternate Function Output mode. And obviously, the corresponding port should not be driven externally as it will be driven by the software using the GPIO controller.

- For AF output or input-output, the port bit must be in AF configuration

External Interrupts/Wakeup lines

Some ports have external interrupt capability (see datasheet). To use external interrupts, the port must be configured in input mode. For more information on interrupts and wakeup lines, refer to *Section 4*.

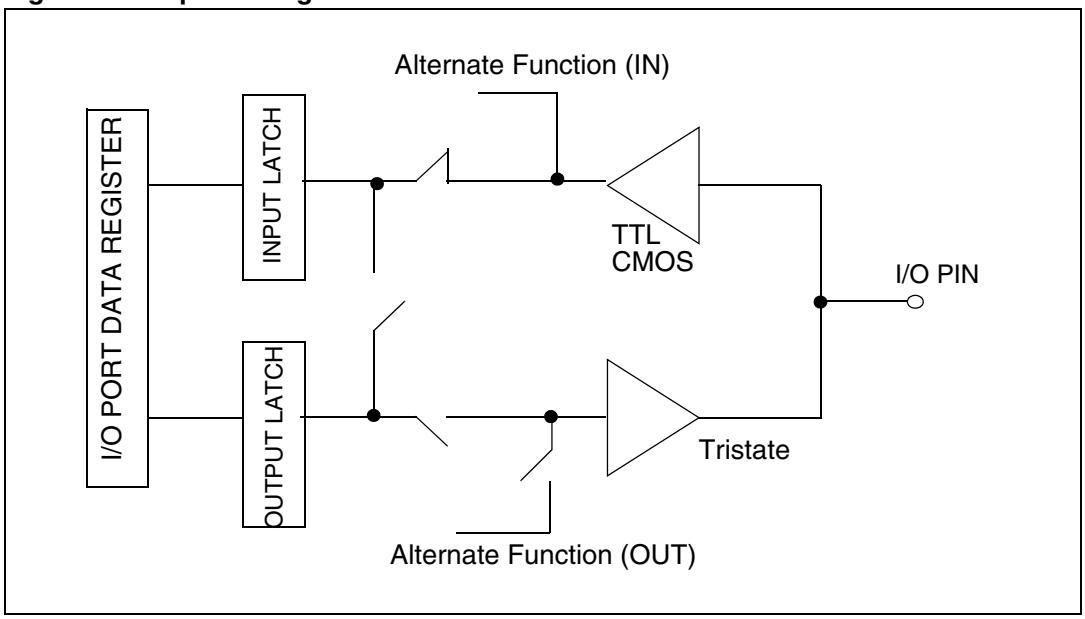
3.1.4 Input configuration

When the I/O Port is programmed as Input:

- The Output Buffer is forced tristate
- The data present on the I/O pin is sampled into the Input Latch every clock cycle
- A read access to the Data register gets the value in the Input Latch.

The *Figure 20* shows the Input Configuration of the I/O Port bit.

Figure 20. Input Configuration



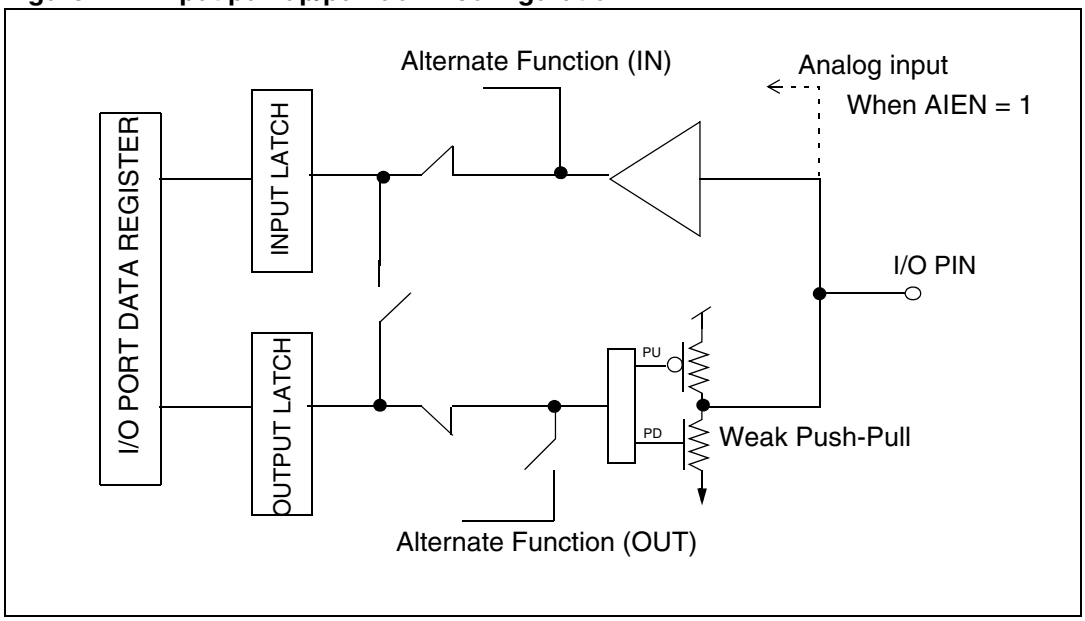
3.1.5 Input pull up/pull down configuration

When the I/O Port is programmed as Input Pull Up/Pull Down:

- The Output Buffer is turned on in Weak Push-Pull configuration and software can write the appropriate level in the output latch to activate the weak pull-up or pull-down as required.
- The data in the Output Latch drives the I/O pin (a logic zero activates a weak pull-down, a logic one activates a weak pull-up)
- A read access to the I/O Data register gets the Input Latch value.

The *Figure 21* shows the Input PUPD Configuration of the I/O Port.

Figure 21. Input pull up/pull down configuration



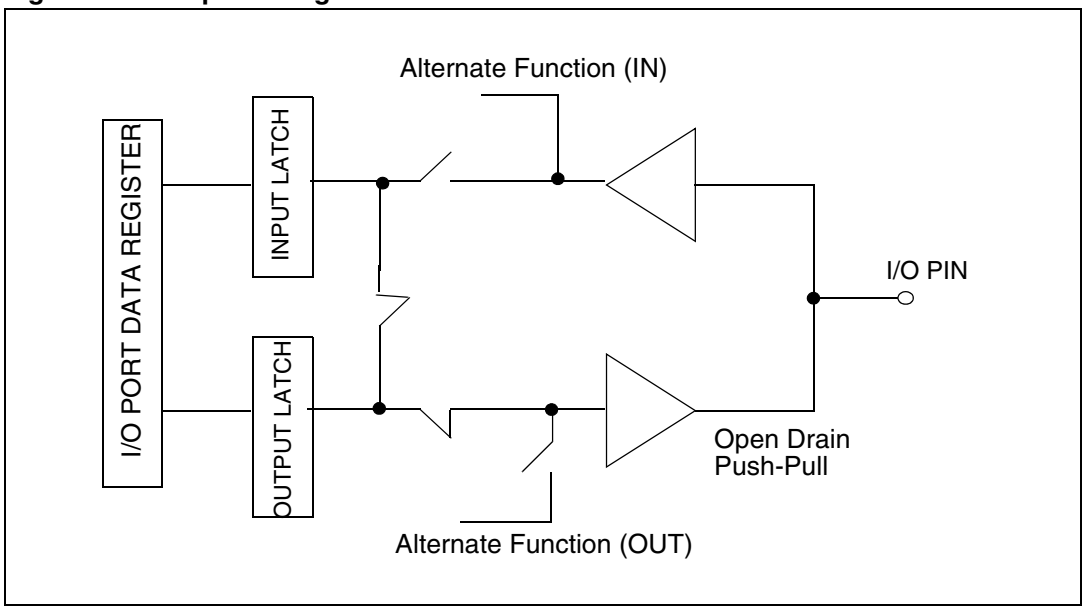
3.1.6 Output configuration

When the I/O Port is programmed as Output:

- The Output Buffer is turned on in Open Drain or Push-Pull configuration
- The data in the Output Latch drives the I/O pin
- A read access to the I/O Data register gets the Output Latch value.

The *Figure 22* shows the Output Configuration of the I/O Port bit.

Figure 22. Output configuration



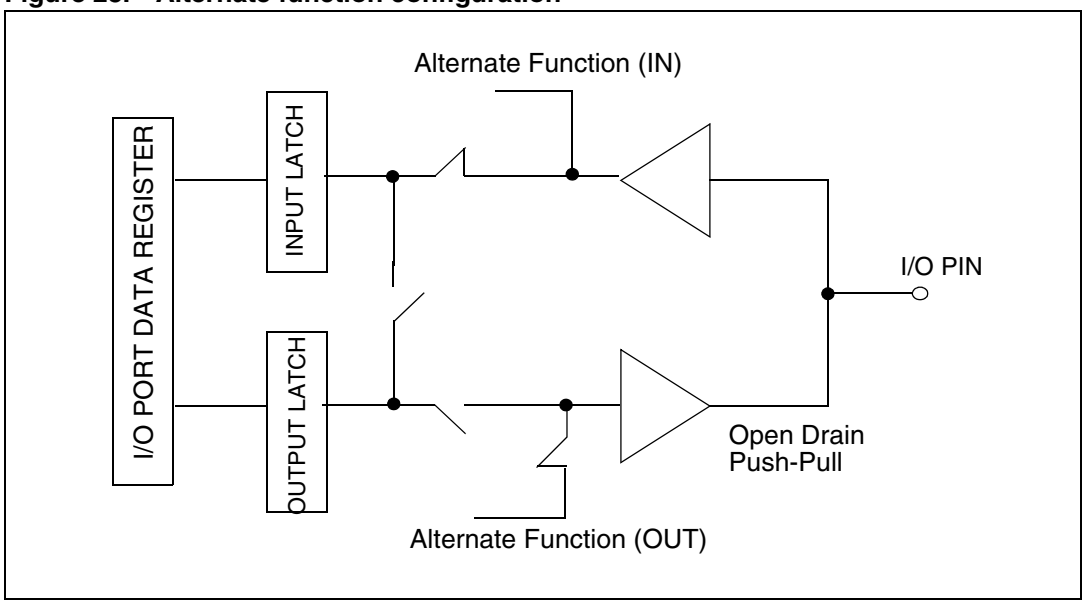
3.1.7 Alternate function configuration

When the I/O Port is programmed as Alternate Function:

- The Output Buffer is turned on in Open Drain or Push-Pull configuration
- The Output Buffer is driven by the signal coming from the peripheral (alternate function out)
- The data present on the I/O pin is sampled into the Input Latch every clock cycle
- A read access to the Data register gets the value in the Input Latch.

The *Figure 23* shows the Alternate Function Configuration of the I/O Port bit.

Figure 23. Alternate function configuration



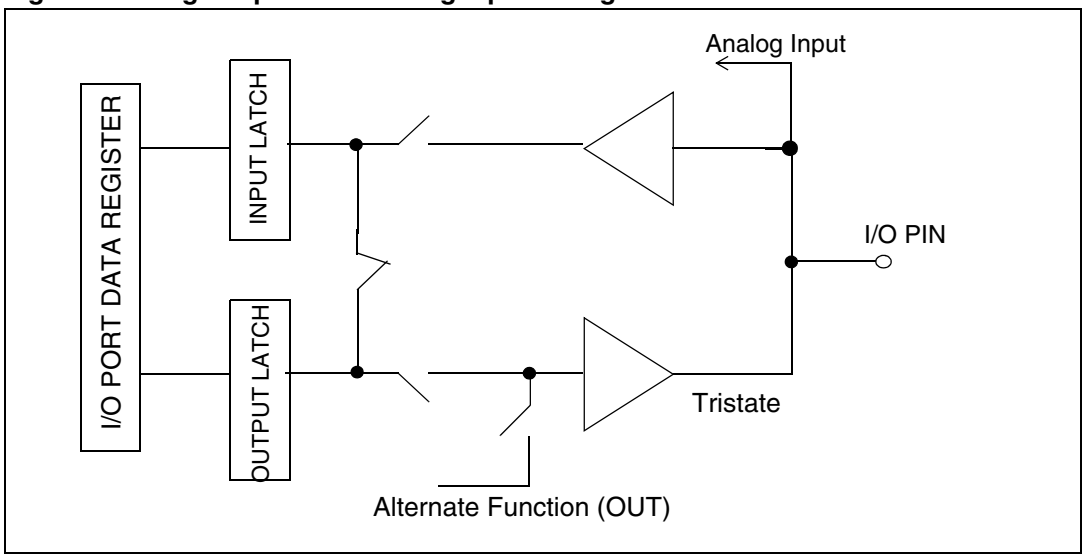
3.1.8 High impedance-analog input configuration

When the I/O Port is programmed as High impedance-Analog Input Configuration:

- The Output Buffer is forced tristate
- The Input Buffer is disabled (the Alternate Function Input is forced to a constant value)
- The Analog Input can be input to an Analog peripheral
- A read access to the I/O Data register gets the Output Latch value

The *Figure 24* shows the High impedance-Analog Input Configuration of the I/O Port bit.

Figure 24. High impedance-analog input configuration



3.2 Register description

The I/O port registers cannot be accessed by byte.

3.2.1 Port configuration register 0 (PC0)

Address Offset: 00h

Reset value: FFFFh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C015	C014	C013	C012	C011	C010	C09	C08	C07	C06	C05	C04	C03	C02	C01	C00
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 15:0 = **C0[15:0]:** *Port Configuration bits*
See *Table 14 on page 57* to configure the I/O Port.

3.2.2 Port configuration register 1 (PC1)

Address Offset: 04h

Reset value: FFFFh (7FFFh for GPIO0)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C115	C114	C113	C112	C111	C110	C19	C18	C17	C16	C15	C14	C13	C12	C11	C10
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 15:0 = **C1[15:0]:** *Port configuration bits*
See *Table 14 on page 57* to configure the I/O Port.

3.2.3 Port configuration register 2 (PC2)

Address Offset: 08h

Reset value: 0000h (0001h for GPIO2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C215	C214	C213	C212	C211	C210	C29	C28	C27	C26	C25	C24	C23	C22	C21	C20
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 15:0 = **C2[15:0]:** *Port Configuration bits*
See *Table 14 on page 57* to configure the I/O Port.

3.2.4 I/O data register (PD)

Address Offset: 0Ch

Reset value: (*)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 15:0 = **D[15:0]:** *I/O Data bits*
A writing access to this register always writes the data in the Output Latch.
A reading access reads the data from the Input Latch in Input and Alternate function configurations or from the Output Latch in Output and High impedance configurations.

(*) Depends on external hardware configuration

3.2.5 I/O port register map

The following table summarizes the registers implemented in each I/O port.

Table 15. I/O-port register map

Addr. Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	PC0	C0[15:0]															
4	PC1	C1[15:0]															
8	PC2	C2[15:0]															
C	PD	D[15:0]															

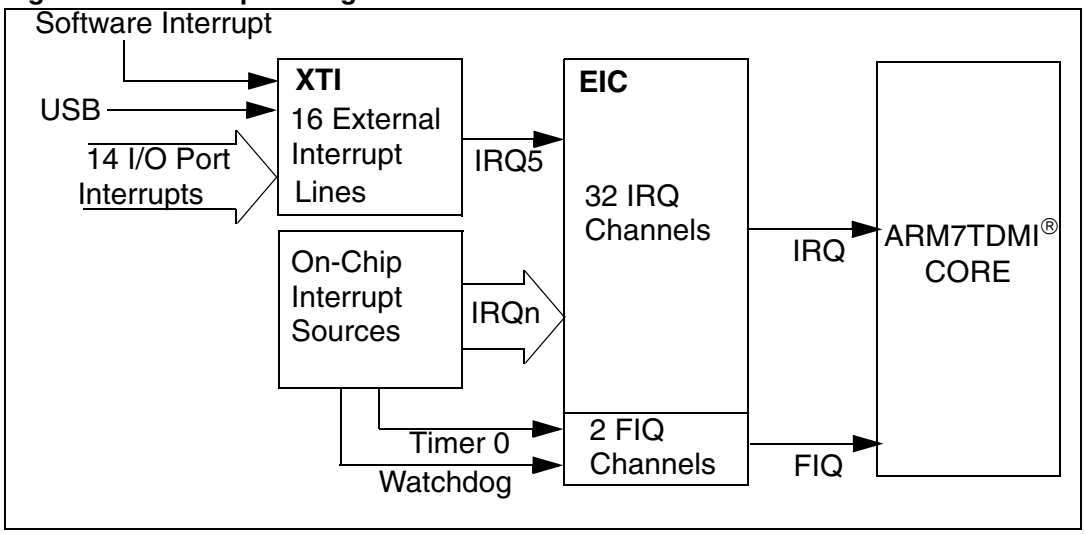
See *Table 3 on page 14* for base address

4 Interrupts

The ARM7 CPU provides two levels of interrupt, FIQ (Fast Interrupt Request) for fast, low latency interrupt handling and IRQ (Interrupt Request) for more general interrupts.

The STR71x interrupt management system provides two interrupt management blocks: the EIC and XTI. Refer to *Figure 25*.

Figure 25. Interrupt Management Overview



4.1 Interrupt latency

As soon as an interrupt request is generated (either from external interrupt source or from an on-chip peripheral), the request itself must go through three different stages before the interrupt handler routine can start. The interrupt latency can be seen as the sum of three different contributions:

- Latency due to the synchronisation of the input stage. This logic can be present (e.g. synchronization stage on external interrupts input lines) or not (e.g. on-chip interrupt request), depending on the interrupt source. Either zero or two clock cycles delay are related to this stage.
- Latency due to the EIC itself. Two clock cycles are related to this stage.
- Latency due to the ARM7TDMI interrupt handling logic (refer to the documentation available on www.arm.com).

Table 16. EIC Interrupt latency (in clock cycles)

	SYNCH. STAGE		EIC STAGE
	min	max	
FIQ	0	2	2
IRQ			

4.2 Enhanced interrupt controller (EIC)

The Enhanced Interrupt Controller (EIC) performs hardware handling of multiple interrupt channels, interrupt priority arbitration and vectorization. It provides:

- 32 maskable interrupt channels, mapped on the IRQ interrupt request line of the ARM CPU
- 16 programmable priority levels for each interrupt channel mapped on IRQ
- Hardware support for interrupt nesting (up to 16 interrupt requests can be nested), with internal hardware nesting stack
- 2 maskable interrupt channels, mapped on FIQ interrupt request line of the ARM CPU, with neither priority nor vectorization
- at register offset 0x18h, the jump instruction to the start address (defined by the user) of the ISR of the highest priority interrupt.
- 16 external interrupts from the XTI block are mapped on IRQ5.

The EIC performs the following operations without software intervention:

- Rejects/accepts an interrupt request according to the related channel mask bit,
- Compares all pending IRQ requests with the current priority level. The IRQ is asserted to the ARM7 if the priority of the current interrupt request is higher than the stored current priority,
- Loads the address vector of the highest priority IRQ to the Interrupt Vector Register (offset 0x18h)
- Saves the previous interrupt priority in the HW priority stack whenever a new IRQ is accepted
- Updates the Current Interrupt Priority Register with the new priority whenever a new interrupt is accepted

If multiple interrupt sources are mapped on the same interrupt vector, software has read the peripheral interrupt flag register to determine the exact source of interrupt (see Interrupt Flags column in *Table 17*)

Table 17. IRQ Interrupt vector table

Vector	Acronym	Peripheral Interrupt	Peripheral Interrupt Flags
IRQ0	T0.EFTI	Timer 0 global interrupt	5
IRQ1	FLASH	FLASH global interrupt	
IRQ2	PRCCU	PRCCU global interrupt	
IRQ3	RTC	Real Time Clock global interrupt	2
IRQ4	WDG.IRQ	Watchdog timer interrupt	1
IRQ5	XTI.IRQ	XTI external interrupt	16
IRQ6	USB.HPIRQ	USB high priority event interrupt	0-7
IRQ7	I2C0.ITERR	I2C 0 error interrupt	
IRQ8	I2C1.ITERR	I2C 1 error interrupt	
IRQ9	UART0.IRQ	UART 0 global interrupt	9
IRQ10	UART1.IRQ	UART 1 global interrupt	9

Table 17. IRQ Interrupt vector table

Vector	Acronym	Peripheral Interrupt	Peripheral Interrupt Flags
IRQ11	UART2.IRQ	UART 2 global interrupt	9
IRQ12	UART3.IRQ	UART 3 global interrupt	9
IRQ13	BSPI0.IRQ	BSPI 0 global interrupt	5
IRQ14	BSPI1.IRQ	BSPI 1 global interrupt	5
IRQ15	I2C0.IRQ	I2C 0 tx/rx interrupt	
IRQ16	I2C1.IRQ	I2C 1 tx/rx interrupt	
IRQ17	CAN.IRQ	CAN module global interrupt	32
IRQ18	ADC.IRQ	ADC sample ready interrupt	1
IRQ19	T1.GI	Timer 1 global interrupt	5
IRQ20	T2.GI	Timer 2 global interrupt	5
IRQ21	T3.GI	Timer 3 global interrupt	5
IRQ22	Reserved		
IRQ23	Reserved		
IRQ24	Reserved		
IRQ25	HDLC.IRQ	HDLC global interrupt	
IRQ26	USB.LPIRQ	USB low priority event interrupt	7-15
IRQ27	Reserved		
IRQ28	Reserved		
IRQ29	T0.TOI	Timer 0 Overflow interrupt	1
IRQ30	T0.OCMPA	Timer 0 Output Compare A interrupt	1
IRQ31	T0.OCMPB	Timer 0 Output Compare B interrupt	1

Two maskable interrupt sources are mapped on FIQ vectors, as shown in *Table 18*:

Table 18. FiQ Vector table

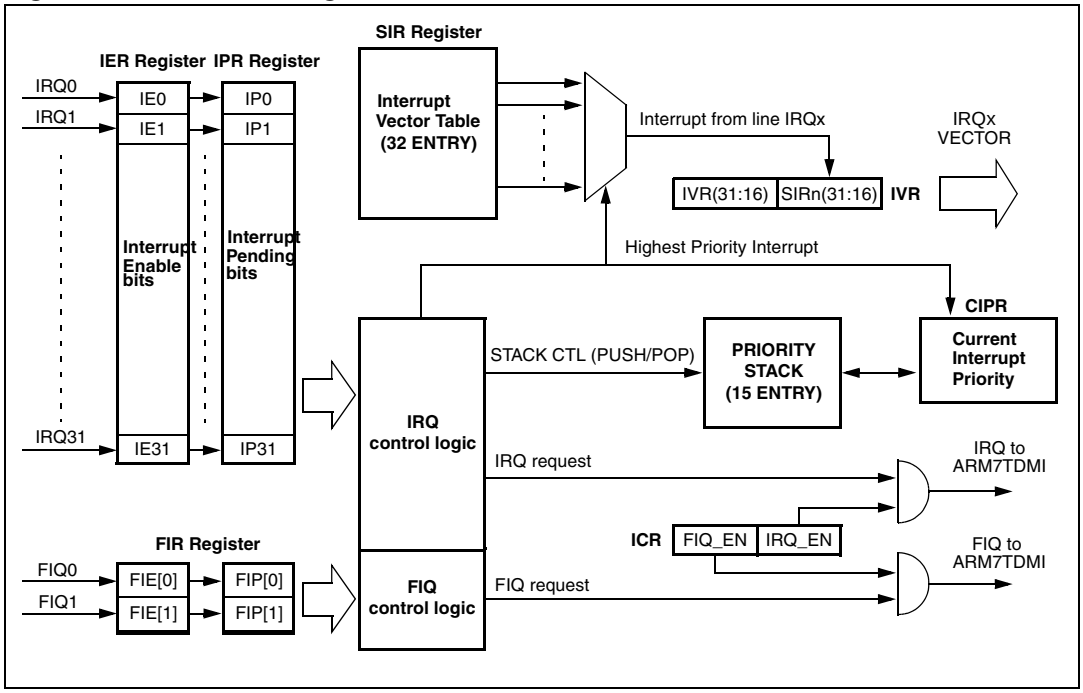
Vector	Interrupt Source
FIQ0	T0.GI - Timer 0 Global Interrupt
FIQ1	WDG.IRQ - Watchdog timer interrupt

These sources are also available as normal IRQs. In most cases, you should only enable one FIQ source in your application. If you enable both FIQ sources, then you can determine the source of the interrupt by reading the FIQ pending bits in the the EIC register. Bear in mind that FIQ has no priority mechanism, so if simultaneous FIQ events occur, software has to manage the priority.

The ARM7TDMI CPU provides two levels of interrupt, FIQ (Fast Interrupt Request) for fast, low latency interrupt handling and IRQ (Interrupt Request) for more general interrupts.

Hardware handling of multiple interrupt channels, interrupt priority and automatic vectorization require therefore a separate Enhanced Interrupt Controller (EIC).

Figure 26. EIC block diagram



- 32 maskable interrupt channels, mapped on ARM's interrupt request pin IRQ
- 16 programmable priority levels for each interrupt channels mapped on IRQ
- hardware support for interrupt nesting (15 levels)
- 2 maskable interrupt channels, mapped on ARM's interrupt request pin FIQ, with neither priority nor vectorization

4.2.1 IRQ mechanism

The EIC is composed of a priority decoder, a finite state machine and a stack.

Priority Decoder

The priority decoder is a combinational block continuously calculating the highest priority pending IRQ. If there is a winner, it updates the EIC_IVR (Interrupt Vector Register) with the address of the IRQ interrupt routine that has won the arbitration, and asserts the nIRQ internal signal low. The nIRQ internal signal ORed with the inverted EIC IRQ enable bit (IRQ_EN) corresponds to the ARM7TDMI® nIRQ signal.

Each channel has a 4-bit field, the SIPL (Source Interrupt Priority Level) in the EIC_SIRn (Source Interrupt Register 0-31) defining the channel priority level in the range of 0 (lowest priority) to 15 (highest).

If several channels are assigned with the same priority level, an internal hardware daisy chain fixes the priority between them. The higher the channel address, the higher the priority. If channel 2 and channel 6 are assigned to the same software priority level, and if they are both pending at the same time, channel 6 will be served first.

In order to declare a channel as a winner, the channel must:

- Be pending (EIC_IPR0-1 - Interrupt Pending Register, 32 pending bits, one per channel). In order to be pending, a channel has to be enabled (EIC_IER0-1 - Interrupt Enable Register, 32 enable bits, one per channel).
- Have the highest priority level, higher than the current one (EIC_CIPR - Current Interrupt Priority Register) and higher than any other pending interrupt channel.
- Have the highest position in the interrupt channel chain if there are multiple pending interrupt channels with the same priority level.

The EIC_CIPR register provides the priority of the interrupt routine currently being served. At reset, the EIC_CIPR is cleared. During an interrupt routine, it can be modified by software from the initialized priority value stored in the EIC_SIRn (Source Interrupt Register 0-31) up to 15. Attempting to write a lower value that the one in EIC_SIRn will have no effect.

For safe operation, it is recommended to disable the global IRQ before modifying EIC_CIPR EIC_SIR, or EIC_IPR pending IRQ clearing, to avoid dangerous race conditions. Moreover, if IRQ_EN is cleared in an interrupt service routine, the pending bit related to the IRQ currently being served must not be cleared, otherwise it will no longer be possible to recover EIC status before popping the stack.

Finite State Machine

The Finite State Machine (FSM) has two states, READY and WAIT. The two states correspond to the ARM7TDMI® nIRQ line being asserted (WAIT) or not (READY). The state of nIRQ will be unconditionally masked (deasserted high) by the EIC global enable bit IRQ_EN being cleared. After a reset the FSM is in READY state (EIC nIRQ line is high). When the priority decoder elects a new winner, the FSM moves from READY to WAIT state and the EIC nIRQ line is asserted low.

To move the FSM back to the ready state, it is mandatory to read the EIC_IVR register or to reset the EIC cell. The EIC can be reset by a global reset resetting the entire device or by clearing bit 14 in the APB2_SWRES register.

Reading the EIC_IVR always moves the FSM from WAIT to READY state, assuming that the FSM was in WAIT state, and automatically releases the EIC nIRQ line.

There is no flag indicating the FSM state.

Stack

The stack is up to 15 events deep corresponding to the maximum number of nested interrupts. It is used to push and pop the previous EIC state. The data pushed onto the stack are the EIC_CICR (Current Interrupt Channel Register) and EIC_CIPR (Current Interrupt Priority Register).

When the FSM is in WAIT state, reading the EIC_IVR raises an internal flag. This pushes the previous EIC_CICR and EIC_CIPR onto the EIC stack. This happens on the next internal clock cycle after reading the EIC_IVR. In the meantime, the internal flag is cleared. The EIC_CICR and EIC_CIPR are updated with the value corresponding to the interrupt channel read in EIC_IVR.

If EIC_IVR is read while the FSM is in READY state, the internal flag is not raised and no operation is performed on the EIC internal stack.

A routine can only be interrupted by a event having a higher priority. Consequently, the maximum number of nested interrupts is 15, corresponding to the 15 priority levels, from 1

to 15. An interrupt with priority level 0 can never be executed. In order for the stack to be full, up to 15 interrupts must be nested and each interrupt event must appear sequentially from priority level 1 up to 15. The main program must have priority level 0.

Having all interrupt sources with a priority level 0 could be useful in applications that only use polling.

To pop the stack, the EIC pending bit corresponding to interrupt in the EIC_CICR register has to be cleared. Clearing any other pending bits will not pop the stack. Take care not to clear a pending bit corresponding to an event still in the stack, otherwise it will not be possible to pop the stack when reaching this stack stage. When the stack is popped, the EIC_CICR and EIC_CIPR are restored with the values corresponding to the previous interrupt event.

EIC Interrupt Vectoring

When the ARM7TDMI® decodes an IRQ interrupt request, the instruction at address 0x18 is executed. By this time, the EIC_IVR register is updated with the address of the highest pending interrupt routine. In order to get the maximum advantage from the EIC mechanism, the instruction at address 0x18 can load the program counter with the address located in the EIC_IVR. In this way, the CPU vector points directly to the right interrupt routine without any software overhead.

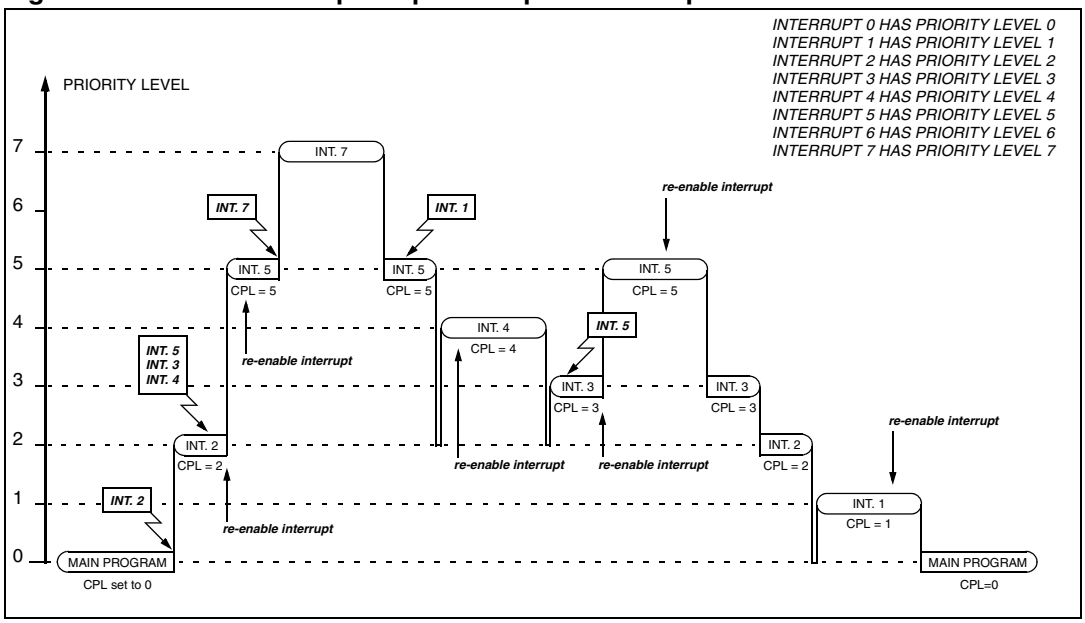
As the priority decoder is always active, the arbitration is never stopped. It may happen that an interrupt event asserts low the ARM7TDMI® nIRQ line, and if between the nIRQ line asserted low and the EIC_IVR read operation a new highest priority event appears the EIC_IVR will have the value corresponding to the highest priority pending interrupt when the EIC_IVR is read.

It is not mandatory to read the EIC_IVR and to branch directly to the right interrupt routine with the instruction at address 0x18. An alternative solution could be to branch to a single interrupt entry point and to read the EIC_IVR register later on. The only mandatory operations are to first read the EIC_IVR once only, then to clear the corresponding pending bit. From an EIC standpoint, the interrupt is acknowledged when the EIC_IVR is read and is completed when the corresponding pending bit is cleared. From an ARM7TDMI® core standpoint, the interrupt is acknowledged when the ARM7TDMI® nIRQ line is asserted low and is completed when the exception return sequence is executed.

The EIC nIRQ line is equivalent to the ARM7TMDI nIRQ line but in addition it is masked by the EIC global enable bit (IRQ_EN). The EIC nIRQ line can be asserted low but if the global EIC enable bit is not set, the ARM7TDMI® nIRQ line will not be asserted low.

The EIC_IPR is a read/clear register, so writing a ‘0’ has no effect, while writing a ‘1’ resets the related bit. Therefore, refrain from using read-modify instructions to avoid corruption of the EIC_IPR status. Most of the EIC pending bits are related to a peripheral pending bit. The peripheral pending bit must be cleared prior to clearing the EIC pending bit. Otherwise the EIC pending bit will be set again and the interrupt routine will be executed twice.

Figure 27. Nested Interrupt Request Sequence Example



EIC IRQ Notes

Reading the EIC_IVR, while the FSM is in READY state, will have no effect. The value read will be unpredictable. Actually, the EIC assigns the default IRQ routine addresses to the EIC_IVR.

As a consequence of a bad programming procedure, the EIC_IVR could also have an unpredictable value while the FSM is in WAIT state. This case has obviously to be avoided as the CPU, when subsequently executing an interrupt subroutine, would execute it while the value in the EIC_IVR register is not relevant. This would result in a stack EIC corruption as the corresponding pending bit could be reset.

There are several cases where this can happen:

- When lowering a pending channel priority level in the EIC_SIRn register to a value equal or lower than the current program priority.
- When the software clears some pending bits without taking care to execute the standard interrupt routine sequence (see below).

In such cases, the priority decoder loses the winner while the EIC nIRQ line is still being asserted. Only reading EIC_IVR will release the EIC nIRQ line.

The CPU will execute the interrupt routine without having a relevant value in the EIC_IVR register, possibly corrupting the stack. If the corresponding pending bit is reset, it will not be possible to execute the EIC stack pop operation.

The normal way to process an interrupt event is to read the EIC_IVR register only once in the interrupt routine. Before exiting the interrupt routine, the corresponding peripheral and the EIC pending bits must be cleared. As soon as the EIC_IVR is read, the application software can read the EIC_CICR register to know which interrupt routine is currently executing, as long as the EIC_IVR register is not used as a routine pointer.

If the EIC_IVR is not read in the interrupt routine the nIRQ line will not be released and the interrupt will be executed twice. If the pending bit is already cleared, the EIC stack will be corrupted as it will not be able to perform the pop operation.

Inside an interrupt routine, it is not an issue to clear pending bits having a lower priority level than the current one, as the nIRQ line is not asserted low in this case. It can be an issue to clear a pending bit that has a higher priority level because the EIC nIRQ line is already asserted low, and when interrupts are re-enabled, the EIC stack will be corrupted.

Clearing a pending bit of an interrupt already in the stack will corrupt the stack.

In the main program, if the global interrupts are disabled, all interrupt sources are disabled and all pending bits are cleared. If the nIRQ was already asserted low, as soon as the global interrupts are enabled the CPU executes an interrupt routine. In this situation, the EIC_IVR read will have an unpredictable value, corrupting the EIC stack.

There is only one safe way to clear pending bits without executing the corresponding interrupt routine. This is to clear them from an IRQ routine that has a higher priority level. In this way, the EIC nIRQ line is guaranteed to be released.

All EIC pending bits can be cleared including the ones that the user application wants to address later on. The user code needs to make sure that, for those interrupts, the peripheral pending bit is not cleared. By this way, the corresponding EIC pending bits will be set again. As all EIC pending bits are cleared, the EIC stack is guaranteed to pop properly. An alternative solution is to make sure that the EIC pending bit corresponding to the EIC_IVR read is cleared.

4.2.2 FIQ mechanism

Compared to the EIC IRQ mechanism, the EIC FIQ mechanism does not provide any automatic vectoring and software priority level to each FIQ interrupt source. It provides a global FIQ enable bit, an enable and a pending bit per FIQ channel.

There are several differences between the global F bit in the CPSR ARM core register and the global FIQ enable bit in the EIC. The F bit can not be modified in user mode while the EIC global FIQ enable bit is always accessible in all modes. In addition, the F bit does not modify the nFIQ internal signal level. It just masks the signal to the core while the EIC global FIQ enable bit acts on the nFIQ signal level. The nFIQ signal is always inactive as soon as the global EIC FIQ enable bit is reset, and is active as soon as the global EIC FIQ enable bit is set along with at least one FIQ pending bit.

In order for an FIQ channel source to be pending, the corresponding FIQ enable bit must be set. Clearing the FIQ channel enable bit while the corresponding pending bit is set will not clear the pending bit and the channel will stay active until the pending bit is cleared.

In order for the CPU to vector at the address 0x1C (FIQ exception vector address), the F bit and the global EIC FIQ enable bit must be enabled and at least one FIQ pending bit must be active. Otherwise, the CPU will not enter the FIQ exception routine.

4.3 Register description

In this section, the following abbreviations are used:

read/write (rw)	Software can read and write to these bits.
read-only (r)	Software can only read these bits.
read/clear (rc_w1)	Software can read as well as reset this bit by writing ‘1’. Writing ‘0’ has no effect on the bit value.

4.3.1 Interrupt control register (EIC_ICR)

Address Offset: 00h

Reset value: 0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

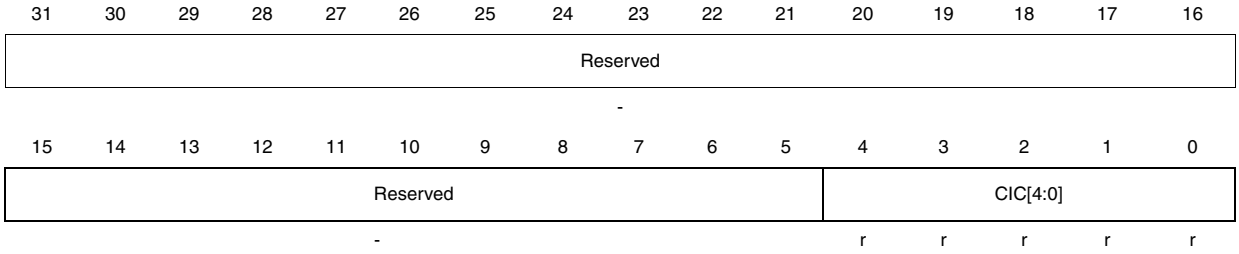
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved														FIQ_EN	IRQ_EN
														rw	rw

Bits 31:2	Reserved, always read as 0.
Bit 1	**FIQ_EN:** *FIQ output Enable bit* Software can read and write to this bit. 0: Enhanced Interrupt Controller FIQ output request to CPU is disabled, even if the EIC has detected valid and enabled fast interrupt requests at its inputs. 1: Enhanced Interrupt Controller FIQ output request to CPU is enabled.
Bit 0	**IRQ_EN**: *IRQ output Enable bit* Software can read and write to this bit. 0: Enhanced Interrupt Controller IRQ output request to CPU is disabled, even if the EIC has detected valid and enabled interrupt requests at its inputs. 1: Enhanced Interrupt Controller IRQ output request to CPU is enabled.

4.3.2 Current interrupt channel register (EIC_CICR)

Address Offset: 04h

Reset value: 0000 0000h



The EIC_CICR reports the number of the interrupt channel currently being serviced. There are 32 possible channel IDs (0 to 31), so the significant register bits are only five (4 down to 0).

After reset, the EIC_CICR value is set to '0' and is updated by the EIC only after the processor has started servicing a valid IRQ interrupt request, i.e. one clock cycle after having read IVR.

To make this happen, EIC registers must be configured as below:

- EIC_ICR IRQ_EN bit =1 (to have the nIRQ signal to ARM7TDMI active)'
- EIC_IER0 not all '0' (at least one interrupt channel must be enabled)
- Among the interrupt channels enabled by the IERx registers, at least one must have the SIPL field of the related SIR register not set to 0 because the EIC generates a processor interrupt request (asserting the nIRQ line) **ONLY IF** it detects an enabled interrupt request whose **priority value is greater than the EIC_CIPR (Current Interrupt Priority Register) value**.

When the nIRQ signal to ARM7TDMI® is activated, the software will read the EIC_IVR (Interrupt Vector Register). This read operation will advise the EIC logic that the ISR (Interrupt Service Routine) has been initiated and that the CICR can be updated

The EIC_CICR value can not be modified by the software (read only register).

Bits 31:5	Reserved, always read as 0.
Bits 4:0	**CIC[4:0]:** *Current Interrupt Channel* Number of the interrupts whose service routine is currently in execution phase. These are read-only bits.

4.3.3 Current interrupt priority register (EIC_CIPR)

Address Offset: 08h

Reset value: 0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												CIP[3:0]			
-												rw	rw	rw	rw

The EIC_CIPR register reports the priority value of the interrupt currently being serviced. There are 16 possible priority values (0 to 15), so the significant register bits are only four (3 down to 0).

After reset, the Current Interrupt Priority (CIP) value is set to '0' and is updated by the EIC only after the processor has started servicing a valid IRQ interrupt request.

To make this happen, EIC registers must be configured as below:

- IRQ_EN bit in the EIC_ICR register is set
- Not all the bits in EIC_IER0 registers should be '0' (at least one interrupt channel must be enabled)
- At least one of the interrupt channels, enabled by the EIC_IER register, must have the SIPL field of the related EIC_SIR register not set to 0, because the EIC generates a processor interrupt request (asserting the nIRQ line) only if it detects an enabled interrupt request whose priority value is bigger than the EIC_CIPR (Current Interrupt Priority Register) value.

When the nIRQ signal to ARM7TDMI® is activated, the processor will read the EIC_IVR. This read operation will inform the EIC that the ISR has been initiated and the EIC_CIPR register can be properly updated.

The EIC_CIPR value can be modified by software only to promote a running ISR to a higher level and only during an ISR. The EIC logic will allow a write to the CIP field of any value equal or greater than the priority value associated with the interrupt channel currently being serviced.

e.g.: suppose the IRQ signal is set because of an enabled interrupt request on channel #4, whose priority value is 7 (i.e. SIPL of SIR7 is 7); after software reads the EIC_IVR register, the EIC will load the CIP field with 7. Until the interrupt service procedure is completed, writes of values 7 up to 15 will be allowed, while attempts to modify the CIP content with priority lower than 7 will have no effect.

The user software has to avoid a situation where the FSM is in WAIT state and the EIC_IVR has an unpredictable value.

Bits 31:4	Reserved, always read as 0.
Bits 3:0	**CIP[3:0]:** *Current Interrupt Priority* Priority value of the interrupt which is currently in execution phase. The software can read and write to these bits.

4.3.4 Interrupt vector register (EIC_IVR)

Address Offset: 18h

Reset value: 0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IVR[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IVR[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

The EIC_IVR is the EIC register that the sofware has to read after detecting the nIRQ signal assertion.

The EIC_IVR read operation informs the EIC that the interrupt service routine (ISR) corresponding to the pending request has been initiated.

This means that:

- the IRQ signal can be de-asserted
- the EIC_CIPR and EIC_CICR can be updated
- no interrupt requests, whose priority is lower or equal than the current one can be processed.

Bits 31:16	**IVR[31:16]:** *Interrupt Vector (High portion)* This register value does not depend on the interrupts to be serviced. It has to be programmed by the user (see Note) at the time of initialization. It is common to all the interrupt channels. Software can read and write to these bits.
Bits 15:0	**IVR[15:0]:** *Interrupt Vector (Low portion)* This register value depends on the interrupts to be serviced (i.e. one of the enabled interrupts with the highest priority), and it is a copy of the Source Interrupt Vector (SIV) value of the EIC_SIR corresponding to the channel to be serviced. These are read only bits.

Note: The EIC does not care about the IVR content: from the controller point of view it is a simple concatenation of two 16-bit fields.

IVR = IVR(31:16) & SIRn(31:16)

What has to be written in the IVR(31:16) is the higher part of the address pointing to the memory location where the interrupt service routine begins. The field SIRn(31:16) will contain the lower 16 bits (offset) of the memory address related to the channel specific ISR.

Reading the IVR is acknowledged only when the CPU is not in debug mode and the user code is executing in ARM IRQ mode.

4.3.5 Fast interrupt register (EIC_FIR)

Address Offset: 1Ch

Reset value: 0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												FIP[1:0]		FIE[1:0]	
												rw	rw	rw	rw

In order for the controller to react to the 2 fast-interrupt (FIQ) channels, the enable bits 1 and 0 must be set to 1. Bits 3 and 2 indicate which channel is the source of the interrupt.

Bits 31:4	Reserved, always read as 0.
Bits 3:2	**FIP[1:0]:** *Channel 1 and 0 Fast Interrupt Pending Bit* These bits are set by hardware by a Fast interrupt request on the corresponding channel. These bits are cleared only by software, i.e. writing a ‘0’ has no effect, whereas writing a ‘1’ clears the bit (forces it to ‘0’). 0: No Fast interrupt pending on channel n. 1: Fast Interrupt pending on channel n.
Bits 1:0	**FIE[1:0]:** *FIQ Channel 1 and 0 Interrupt Enable bit* In order to have the controller responding to a FIQ on a specific channel, the corresponding FIE bit must be set. 0: Fast Interrupt request on FIQ channel n disabled. 1: Fast Interrupt request on FIQ channel n enabled.

4.3.6 Interrupt enable register 0 (EIC_IER0)

Address Offset: 20h

Reset value: 0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IER[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IER[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0	**IER[31:0]:** *Channel 31 to 0 Interrupt Enable bits* The EIC_IER0 is a 32 bit register: it provides an enable bit for each of the 32 EIC interrupt input channels. In order to enable the interrupt response to a specific interrupt input channel the corresponding bit in the EIC_IER0 register must be set to '1'. A '0' value prevents the corresponding pending bit geing set. 0: Input channel disabled. 1: Input channel enabled.

4.3.7 Interrupt pending register 0 (EIC_IPR0)

Address Offset: 40h

Reset value: 0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IPR[31:16]															
rc_w1															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IPR[15:0]															
rc_w1															

Bits 31:0	**IPR[31:0]:** *Channel 31 to 0 Interrupt Pending bit* The EIC_IPR0 is a 32 bit register, which provides a pending bit for each of the 32 EIC interrupt input channels. This is where the information about the channel interrupt status is kept. If the corresponding bit in the enable register EIC_IER0 has been set, the EIC_IPR0 bit set high implies that the related channel has asserted an interrupt request that has not been serviced yet. The bits are Read/Clear, i.e. writing a '0' has no effect, whereas writing a '1' clears the bit. 0: No interrupt pending. 1: Interrupt pending.

Note: Before exiting an ISR, the software must have cleared the EIC_IPR0 bit related to the executed routine. This bit clear operation will be interpreted by the EIC as End of Interrupt (EOI) sequence and will allow the interrupt stack pop and processing of new interrupts.

Note: The Interrupt Pending bits must be carefully handled because the EIC state machine and its internal priority hardware stack could be forced to a non recoverable condition if unexpected pending bit clear operations are performed.

Example 1:

- Suppose that one or more interrupt channels are enabled, with a priority higher than zero. As soon as an interrupt request arises, the EIC FSM processes the new input and asserts the nIRQ signal. If before reading the EIC_IVR, for any reason, software clears the pending bits, the nIRQ signal will remain asserted the same, even if no more interrupts are pending.
  The only way to reset the nIRQ line logic is to read the EIC_IVR (0x18) register or to send a software reset to the EIC.

Example 2:

- Suppose that one or more interrupt channels are enabled, with a priority higher than zero. As soon as an interrupt request arises, the EIC FSM processes the new input and asserts the nIRQ signal. If after reading the EIC_IVR, for any reason, software clears the pending bit related to the serviced channel before completing the ISR, the EIC will detect an End Of Interrupt command, will send a pop request to the priority stack and a new interrupt, even of lower priority, will be processed.
  To close an interrupt handling section (EOI), the interrupt pending clear operation must be performed at the end of the related ISR, on the pending bit related to the serviced channel. On the other hand, as soon as the pending bit of the serviced channel is cleared (even by mistake) by the software, the EOI sequence is entered by the EIC.

*Note: In order to safely clear a pending bit of an IRQ **not** currently serviced, bit IRQ_EN of EIC_ICR register should be cleared first. If this is not done, the EIC FSM can enter an unrecoverable state.*
In general, while in the main program, clearing a pending bit has no drawbacks. When this operation is instead performed inside an IRQ routine it is very important not to clear by mistake the IPR bit related to the IRQ currently being serviced. Since IRQ_EN bit freezes the Stack, the pop operation for the current IRQ will not be performed and will not be even possible in future when IRQs will be re-enabled.

4.3.8 Source interrupt registers - channel n (EIC_SIRn)

Address Offset: 60h to DCh

Reset value: 0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SIV[31:16]															
rw															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												SIPL[3:0]			
												rw			

There are 32 different EIC_SIRn registers for each input interrupt channel.

Bits 31:16	**SIV[31:16]:** *Source Interrupt Vector for interrupt channel n (n=0... 31)* This field contains the interrupt channel dependent part of the interrupt vector that will be provided to the processor when the EIC_IVR (address 0x18) is read. Depending on what the processor expects (32 bit address or opcode, see IVR description), the SIV will have to be loaded with the interrupt channel ISR address offset or with the lower part (including the jump offset) of the first ISR instruction opcode.
Bits 15:4	Reserved, always read as 0.
Bits 3:0	**SIPL[3:0]:** *Source Interrupt Priority Level for interrupt channel n (n=0... 31)* These 4 bits allow to associate the interrupt channel to a priority value between 0 and 15. The reset value is 0.

Note: To be processed by the EIC logic an interrupt channel must have a priority level higher than the current interrupt priority (CIP). The lowest value CIP can have is 0 so all the interrupt sources that have a priority level equal to 0 will never generate an IRQ request, even if properly enabled.

4.3.9 EIC register map

Table 19. EIC register map

Addr. Offset	Register Name	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00h	EIC_ICR	reserved																														FIQ_EN	IRQ_EN
04h	EIC_CICR	reserved																												CIC[4:0]			
08h	EIC_CIPR	reserved																												CIP[3:0]			
18h	EIC_IVR	Jump Instruction Opcode or Jump Base Address																Jump Offset															
1Ch	EIC_FIR	reserved																												FIP [2:0]		FIE [1:0]	
20h	EIC_IER	IER[31:0]																															
40h	EIC_IPR	IPR[31:0]																															
60h	EIC_SIR0	SIV0[31:16]																reserved												SIPL0[3:0]			
64h	EIC_SIR1	SIV1[31:16]																reserved												SIPL1[3:0]			
68h	EIC_SIR2	SIV2[31:16]																reserved												SIPL2[3:0]			
6Ch	EIC_SIR3	SIV3[31:16]																reserved												SIPL3[3:0]			
70h	EIC_SIR4	SIV4[31:16]																reserved												SIPL4[3:0]			
74h	EIC_SIR5	SIV5[31:16]																reserved												SIPL5[3:0]			
78h	EIC_SIR6	SIV6[31:16]																reserved												SIPL6[3:0]			
7Ch	EIC_SIR7	SIV7[31:16]																reserved												SIPL7[3:0]			
80h	EIC_SIR8	SIV8[31:16]																reserved												SIPL8[3:0]			
84h	EIC_SIR9	SIV9[31:16]																reserved												SIPL9[3:0]			
88h	EIC_SIR10	SIV10[31:16]																reserved												SIPL10[3:0]			
8Ch	EIC_SIR11	SIV11[31:16]																reserved												SIPL11[3:0]			
90h	EIC_SIR12	SIV12[31:16]																reserved												SIPL12[3:0]			
94h	EIC_SIR13	SIV13[31:16]																reserved												SIPL13[3:0]			
98h	EIC_SIR14	SIV14[31:16]																reserved												SIPL14[3:0]			
9Ch	EIC_SIR15	SIV15[31:16]																reserved												SIPL15[3:0]			
A0h	EIC_SIR16	SIV16[31:16]																reserved												SIPL16[3:0]			
A4h	EIC_SIR17	SIV17[31:16]																reserved												SIPL17[3:0]			
A8h	EIC_SIR18	SIV18[31:16]																reserved												SIPL18[3:0]			
ACh	EIC_SIR19	SIV19[31:16]																reserved												SIPL19[3:0]			
B0h	EIC_SIR20	SIV20[31:16]																reserved												SIPL20[3:0]			
B4h	EIC_SIR21	SIV21[31:16]																reserved												SIPL21[3:0]			
B8h	EIC_SIR22	SIV22[31:16]																reserved												SIPL22[3:0]			

Table 19. EIC register map

Addr. Offset	Register Name	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BCh	EIC_SIR23	SIV23[31:16]																reserved												SIPL23[3:0]			
C0h	EIC_SIR24	SIV24[31:16]																reserved												SIPL24[3:0]			
C4h	EIC_SIR25	SIV25[31:16]																reserved												SIPL25[3:0]			
C8h	EIC_SIR26	SIV26[31:16]																reserved												SIPL26[3:0]			
CCh	EIC_SIR27	SIV27[31:16]																reserved												SIPL27[3:0]			
D0h	EIC_SIR28	SIV28[31:16]																reserved												SIPL28[3:0]			
D4h	EIC_SIR29	SIV29[31:16]																reserved												SIPL29[3:0]			
D8h	EIC_SIR30	SIV30[31:16]																reserved												SIPL30[3:0]			
DCh	EIC_SIR31	SIV31[31:16]																reserved												SIPL31[3:0]			

4.3.10 Programming considerations

Here are a few guidelines on how to program the EIC registers in order to get up and running quickly. In the following, it is assumed we are dealing with standard interrupts and that we want, for example, to detect an interrupt on channel #22, which has a priority of 5.

First of all, you have to assign the priority and the jump address for the interrupt channel #22. Therefore:

- Write the binary value “0101” in the SIPL field of the SIR22 register, i.e. priority 5 (it must be non-zero to allow IRQ to be generated).

Two registers are used to supply the channel interrupt vector to the EIC controller (IVR[31:16] and SIR22[31:16]):

- write in SIR22[31:16], i.e. in the upper part of the SIR register related to channel #22, the memory address offset (or the jump offset) where the Interrupt Service Routine, related to interrupt channel #7, starts.
- insert the base jump address (or the jump opcode) in the most significant half of the IVR register, i.e. IVR[31:16].

Finally, you have to enable interrupts both at the global level and at the interrupt channel level. To do this, perform these steps:

- set the IRQ_EN bit of ICR to 1
- set bit # 22 of IER to 1

As far as the FIQ interrupts are concerned, since they have no vectorization or priority, only the first two steps above are involved. Supposing you want to enable FIQ channel #1:

- set the FIQ_EN bit of ICR to 1.
- set bit #1 of FIE in FIR register to 1.

4.3.11 Application note

Every Interrupt Service Routine (ISR) should have the following blocks of code.

- A **Header routine** to enter ISR. It must be:

```
1) STMFD sp!,{r5,lr}The workspace r5 plus the current return address lr_irq is pushed
into the system stack.
2) MRS r5,spsr Save the spsr into r5
3) STMFD sp!,{r5}Save r5
4) MSR cpsr_c,#0x1FReenable IRQ, go into system mode
5) STMFD sp!,{lr}Save lr_sys for the system mode
```

Note: r5 is a generic register chosen in this example from the available registers r0 to r12. Since there is no way to save SPSR directly into the ARM stack, the operation is executed in two steps using r5 as a temporary register.

- The ISR **Body routines**.
- A **Footer routine** to exit ISR. It must be:

```
1) LDMFD sp!,{lr}Restore lr_sys for system mode
2) MSR cpsr_c,#0xD2 Disable IRQ, move to IRQ mode
3) Clear pending bit in EIC (using the proper IPRx)
4) LDMFD sp!,{r5}Restore r5
5) MSR spsr,r5 Restore Status register spsr
6) LDMFD sp!,{r5,lr}Restore status lr_irq and workspace
7) SUBS pc,lr,#4 Return from IRQ and interrupt reenabled
```

The following two sections give some comments on the above code and hints on calling subroutines from an ISR.

Avoiding LR_sys and r5 register content loss

This first example refers to a LR_sys content loss problem: it is assumed that an ISR without instruction 5) in the header routine (and consequently without instruction 1) in the footer routine) has just started; the following happens:

- Instruction 4) is executed (so system mode is entered)
- A subroutine is called with a BL instruction and LR_sys now contains the return address A: the first subroutine instruction should store register LR_sys in the stack (the address of this instruction is called B)
- A higher priority interrupt starts before the previous operation could be executed
- A new ISR stores address B in LR_irq and enters system mode
- A new subroutine is called with a BL instruction: LR_sys is loaded with the new return address C (this overwrites the previous value A!) which is now stored in the stack.
- The highest priority ISR ends and address B is restored: now the LR_sys value can be put in the stack but its value has changed to address C (instead of A).

The work-around to avoid such a dangerous situation is to insert line 5) at the end of the header routine and consequently line 1) at the beginning of the footer routine.

Similar reasons could lead register **r5** to be corrupted. To fix this problem, lines 3) in header and 4) in footer should be added.

Hints on calling subroutines from within ISRs

This section discusses a case where a subroutine is called by an ISR.

Supposing this type of procedure starts with an instruction like:

```
STMFDSP!, { ... , LR }
```

probably it will end with:

```
LDMFDSP!, { ... , LR }
MOVPC, LR
```

If a higher priority IRQ occurs between the last two instructions, and the new ISR then calls another subroutine, the LR content will be lost: so, when the last IRQ ends, the previously interrupted subroutine will not return to the correct address.

To avoid this, the previous two instructions must be replaced with the single instruction:

```
LDMFDSP!, { ... , PC }
```

which automatically moves the stored link register directly into the program counter, making the subroutine return correctly.

4.4 External interrupts (XTI)

The main function of the External Interrupts Unit (XTI) is to manage the external interrupt lines. The XTI is connected to the IRQ5 channel of the EIC module.

Using the XTI registers, 14 I/O ports can be programmed as external interrupt lines or as wake-up lines, able to wake-up the MCU from STOP mode.

Note: Only the WAKEUP pin (P0.15) can be used to wake-up from STANDBY mode

Two additional XTI lines are used by specific interrupt sources:

- Software interrupt
- USB End Suspend event.

Refer to *Table 20*.

Some external interrupt lines are mapped to I/O ports that can be enabled as inputs to the CAN, I^2C, BSPI or UART peripherals. This means you can program it so that any activity on these serial buses will generate an interrupt and wake-up the MCU STOP mode.

Table 20. External Interrupt Line mapping

Wake-up line #	Wake-up line source
0	SW interrupt - no HW connection.
1	USB wake-up event: generated while exiting from suspend mode
2	Port 2.8 - External Interrupt
3	Port 2.9 - External Interrupt
4	Port2.10 - External Interrupt
5	Port 2.11 - External Interrupt
6	Port 1.11 - CAN module receive pin (CANRX).

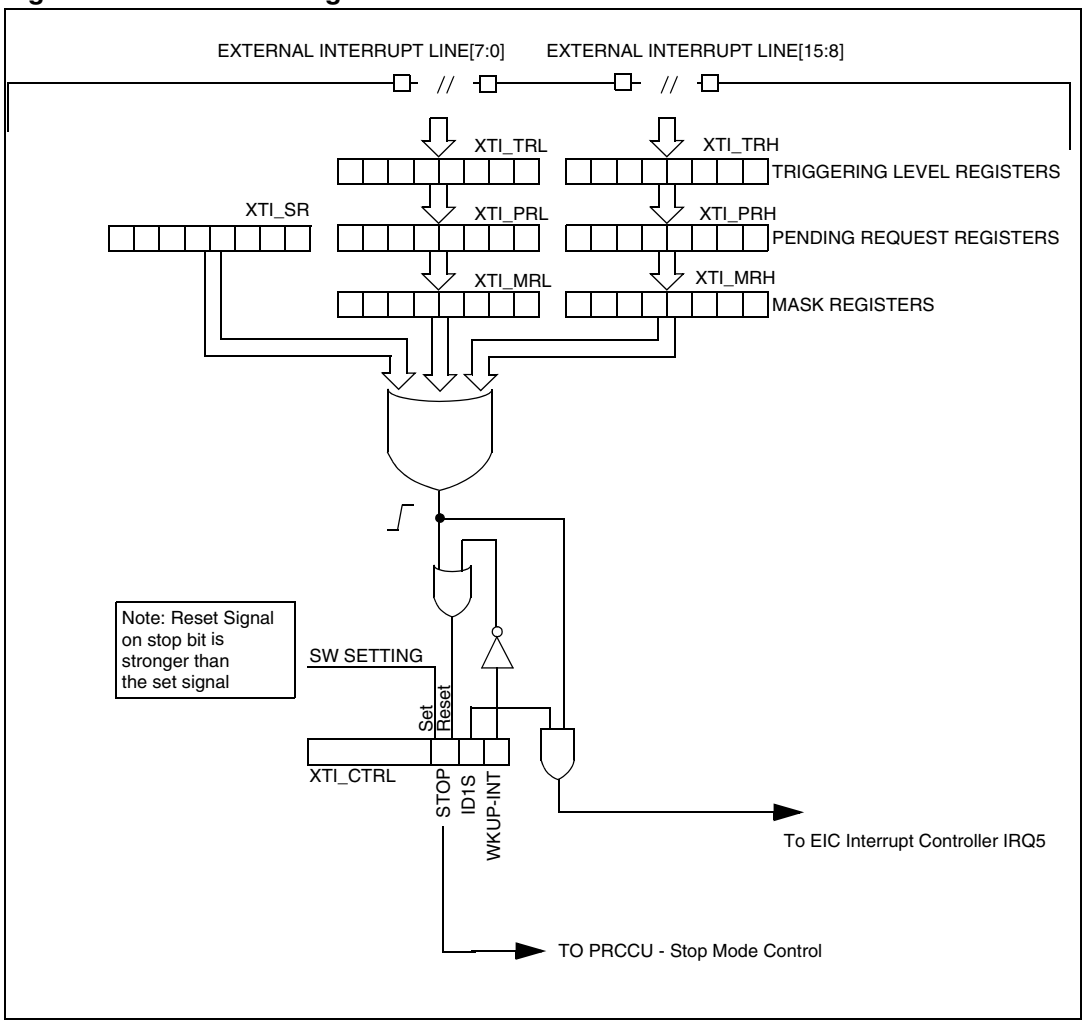
Table 20. External Interrupt Line mapping

Wake-up line #	Wake-up line source
7	Port 1.13 - HDLC clock (HCLK) ***or*** I2C.0 Clock (I0.SCL)
8	Port 1.14 - HDLC receive pin (HRXD) ***or*** I2C.0 Data (SDA)
9	Port 0.1 - BSPI0 Slave Input data (S0.MOSI) ***or*** UART3 Receive Data Input (U3.Rx)
10	Port 0.2 - BSPI0 Slave Input serial clock (S0.SCLK) ***or*** I2C.1 Clock (I1.SCL)
11	Port 0.6 - BSPI1 Slave Input serial clock (S1.SCLK)
12	Port 0.8 - UART0 Receive Data Input (U0.Rx)
13	Port 0.10 - UART1 Receive Data Input (U1.Rx)
14	Port 0.13 - UART2 Receive Data Input (U2.Rx)
15	Port 0.15 - WAKEUP pin ***or*** RTC ALARM

4.4.1 Features

- External interrupt lines can be used to wake-up the system from STOP mode
- Programmable selection of Wake-up or Interrupt
- Programmable Wake-up trigger edge polarity
- All Wake-up Lines individually maskable
- Wake-up interrupt generated by software

Figure 28. XTI Block Diagram



4.4.2 Functional description

Interrupt mode

To configure the 16 lines as interrupt sources, use the following procedure:

1. Configure the mask bits of the 16 wake-up lines (XTI_MRL, XTI_MRH).
2. Configure the triggering edge registers of the wake-up lines (XTI_PRL, XTI_TRH).
3. In the EIC registers, enable the IRQ5 interrupt channel so an interrupt coming from one of the 16 wake-up lines can be correctly acknowledged.
4. Clear the WKUP-INT bit in the XTI_CTRL register to disable Wake-up Mode and enable interrupt mode
5. Set the ID1S bit in the XTI_CTRL register to enable the 16 wake-up lines as external interrupt source lines.

Wake-up mode selection

To configure the 16 lines as wake-up sources, use the following procedure:

1. Configure the mask bits of the 16 wake-up lines (XTI_MRL, XTI_MRH).
2. Configure the triggering edge registers of the wake-up lines (XTI_TRL, XTI_TRH).
3. If an interrupt routine is to be executed after a wake-up event, then enable the IRQ5 interrupt channel using the EIC registers. Otherwise, if the wake-up event only restarts executing of the code from where it was stopped, the IRQ5 interrupt channel must be masked.
4. Since the PRCCU can generate an interrupt request when exiting from STOP mode, take care to mask it if the wake-up event is only to restart code execution.
5. Set the WKUP-INT bit in the XTI_CTRL register to select Wake-up Mode.
6. Set the ID1S bit in the XTI_CTRL register to enable the 16 wake-up lines as external interrupt source lines.

4.4.3 Programming considerations

The following paragraphs give some guidelines for designing an application program.

Procedure for entering/exiting STOP mode

1. Program the polarity of the trigger event of external wake-up lines by writing registers XTI_TRH and XTI_TRL.
2. Check that at least one mask bit (registers XTI_MRH, XTI_MRL) is equal to 1 (so at least one external wake-up line is not masked).
3. Reset at least the unmasked pending bits: if unmasked pending bits are not cleared STOP Mode cannot be entered.
4. Set the ID1S t and the WKUP-INT bits in the XTI_CTRL register.
5. To generate an interrupt on the associated channel (IRQ5), set the related enable, mask and priority bits in the EIC registers.
6. Reset the STOP bit in register XTI_CTRL and STOP_I bit in CLK_FLAG register (PRCCU).
7. To enter STOP mode, write the sequence 1, 0, 1 to the STOP bit in the XTI_CTRL register. As already said, the three write operations are effective even though not executed in a strict sequence (intermediate instructions are allowed): to reset the sequence it is sufficient to write twice a logic ‘0’ to the STOP bit of XTI_CTRL register (corresponding anyway to a bad sequence).
8. The code to be executed just after the STOP sequence must check the status of the STOP and PRCCU STOP_I bits to determine if the device entered STOP mode or not. If the device did not enter in STOP mode it is necessary to re-loop the procedure from the beginning, otherwise the procedure continues from next point.
9. Poll the wake-up pending bits to determine which wake-up line caused the exit from STOP mode.
10. Clear the wake-up pending bit that was set.

Simultaneous Setting of Pending Bits

It is possible that several simultaneous wake-up events set different pending bits. In order to accept subsequent events on external wake-up/interrupt lines, once the first interrupt routine has been executed, the corresponding pending bit in XTI_PRx register it is necessary to clear at least one pending bit: this operation allows a rising edge to be generated on the internal line (if there is at least one more pending bit set and not masked) and so to set the interrupt controller pending bit again. A further interrupt on the same channel of the interrupt

controller will be serviced depending on the status of the mask bit. Two possible situations may arise:

1. The user chooses to reset all pending bits: no further interrupt requests will be generated on the channel. In this case the user has to:
   - Reset the interrupt controller mask bit (to avoid generating a spurious interrupt request during the next reset operations)
   - Reset the XTI_PRH register
   - Reset the XTI_PRL register
2. The user chooses to keep at least one pending bit active: at least one additional interrupt request will be generated on the interrupt controller channel. In this case the user has to reset the desired pending bits. This operation will generate a rising edge on the interrupt controller channel and the corresponding pending bit will be set again. An interrupt on the this channel will be serviced depending on the status of corresponding mask bit.

STOP Mode Entry Conditions

Assuming the device is in Run mode: during the STOP bit setting sequence the following cases may occur:

Case 1: Wrong STOP bit setting sequence

This can happen if an Interrupt request is acknowledged during the STOP bit setting sequence. In this case polling the STOP and STOP_I bits will give:

STOP = 0, STOP_I = 0

This means that the device did not enter STOP mode due to a bad STOP bit setting sequence: the user must retry the sequence.

Case 2: Correct STOP bit setting sequence

In this case the device enters STOP mode.

To exit STOP mode, a wake-up interrupt must be acknowledged. This implies:

STOP = 0, STOP_I = 1

This means that the device entered and exited STOP mode due to an external wake-up line event.

Case 3: A wake-up event on the external wake-up lines occurs during the STOP bit setting sequence

There are two possible cases:

1. Interrupt requests to the CPU are disabled: in this case the device will not enter STOP mode, no interrupt service routine will be executed and the program execution continues from the instruction following the STOP bit setting sequence. The status of STOP and STOP_I bits will be again:

   STOP = 0, STOP_I = 0

   The application can determine why the device did not enter STOP mode by polling the pending bits of the external lines (at least one must be at 1).
2. Interrupt requests to CPU are enabled: in this case the device will not enter STOP mode and the interrupt service routine will be executed. The status of STOP and STOP_I bits will be again:

STOP = 0, STOP_I = 0

The interrupt service routine can determine why the device did not enter STOP mode by polling the pending bits of the external lines (at least one must be at 1).

If the device really exits from STOP Mode, the PRCCU STOP_I bit is still set and must be reset by software. Otherwise, if an Interrupt request was acknowledged during the STOP bit setting sequence, the PRCCU STOP_I bit is reset. This means that the system has filtered the STOP Mode entry request.

The WKUP-INT bit can be used by an interrupt routine to detect and to distinguish events coming from Interrupt Mode or from Wake-up Mode, allowing the code to execute different procedures.

To exit STOP mode, it is sufficient that one of the 16 wake-up lines (not masked) generates an event: the clock restarts after the delay needed for the oscillator to restart.

Note: After waking-up from STOP Mode, software can successfully reset the pending bits (edge sensitive), even though the corresponding wake-up line is still active (high or low, depending on the Trigger Event register programming); the user must poll the external pin status to detect and distinguish a short event from a long one (for example keyboard input with keystrokes of varying length).

4.4.4 Register description

XTI software interrupt register (XTI_SR)

Address Offset: 1Ch
Reset value: 00h

7	6	5	4	3	2	1	0
XTIS7	XTIS6	XTIS5	XTIS4	XTIS3	XTIS2	XTIS1	XTIS0
rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:0	**XTIS[7:0]:** *Software Interrupt Pending Bits.* These bits can be set by software to implement a software interrupt. The interrupt routine must clear any pending bits that are set. These interrupts are masked by the global interrupt enable (bit ID1S in XTI_CTRL register) . 0: No software interrupt pending. 1: Software interrupt pending.

Wake-up Control Register (XTI_CTRL)

Address Offset: 24h
Reset value: 00h

7	6	5	4	3	2	1	0
reserved					STOP	ID1S	WKUP-INT
-					rw	rw	rw

Bits 7:3	Reserved.
Bit 2	**STOP:** *Stop bit.* To enter STOP Mode, write the sequence 1,0,1 to this bit with three (not necessarily consecutive) write operations. When a correct sequence is recognized, the STOP bit is set and the PRCCU puts the MCU in STOP Mode. The software sequence succeeds only if the following conditions are true: The WKUP-INT bit is 1, At least one mask bit is equal to 1 (at least one external wake-up line is not masked) in the XTI_MRL and XTI_MRH registers. All unmasked pending bits are reset in the XTI_PRL and XTI_PRH registers, Otherwise the device cannot enter STOP mode, the program code continues executing and the STOP bit remains cleared. The bit is reset by hardware if, while the device is in STOP mode, a wake-up interrupt comes from any of the unmasked wake-up lines. The STOP bit is at 1 in the two following cases: After the first write instruction of the sequence (a 1 is written to the STOP bit) At the end of a successful sequence (i.e. after the third write instruction of the sequence) **Caution:** If interrupt requests are acknowledged during the sequence, the system will not enter STOP mode (since the sequence is not completed). At the end of the interrupt service routine, it is recommended to reset the sequence state machine by twice writing a logic ‘0’ to the STOP bit of XTI_CTRL register (corresponding anyway to a bad sequence). Otherwise, the incomplete sequence will wait to be completed, (STOP bit is set only after the third correct writing instruction of the sequence). **Caution:** Whenever a STOP request is issued to the device,several clock cycles are needed to enter STOP mode (see PRCCU chapter for further details). Hence the execution of the instruction following the STOP bit setting sequence might start before entering STOP mode (consider the ARM7 three-stage pipeline as well). In order to avoid executing any valid instructions after a correct STOP bit setting sequence and before entering STOP mode, it is mandatory to execute a few (at least 6) dummy instructions after the STOP bit setting sequence (after the third valid STOP bit write operation. Additionally, if an interrupt routine is executed when exiting from STOP mode , another set of dummy instructions (at least 3) must be added, to take into account of the latency period. This takes into account that when STOP mode is entered, the pipeline content is frozen as well, and when the system restarts the first executed instruction was already fetched and decoded before entering STOP mode. Below is some example code for managing STOP mode entering/exiting.

	`LDR R0, = APB0 + APB_XTIP; Base address of Wake-up Module setting`
	
	`MOV R1, #3 ; Setting of Control Register`
	`STR R1, [R0, #XTI_CTRL]`
	
	`MOV R3, #0x08; Unmask wake-up line 3`
	`STR R3, [R0, #XTI_MRL]`
	
	`MOV R2, #0x07; R2 used to write '1' into STOP bit`
	
	`STR R2, [R0, #XTI_CTRL]; 1st sequence instruction (writing '1')`
	`STR R1, [R0, #XTI_CTRL]; 2nd sequence instruction (writing '0')`
	`STR R2, [R0, #XTI_CTRL]; 3rd sequence instruction (writing '1')`
	
	`MOV R1, R1 ; Set of 9 dummy instructions`
	`MOV R1, R1`
	`MOV R1, R1`
	`MOV R1, R1`
	`MOV R1, R1`
	`MOV R1, R1`
	`MOV R1, R1`
	`MOV R1, R1`
	`MOV R1, R1`
	If you want the system to restart from STOP mode without entering an interrupt service routine, but simply by executing the first valid instruction just after the STOP sequence, only the first six dummy instructions are needed.

Bit 1	**ID1S:** *XTI Global Interrupt Mask.* This bit is set and cleared by software. 0: XTI interrupts disabled. 1: XTI interrupts enabled. **Caution:** To avoid spurious interrupt requests on the IRQ5 channel of the EIC, it is recommended to clear the corresponding enable bit in the EIC IER register before modifying the ID1S bit.
Bit 0	**WKUP-INT:** *Wake-up Interrupt.* This bit is set and cleared by software. 0: The 16 wake-up lines can be used to generate interrupt requests on the IRQ5 channel of the EIC interrupt controller 1: The 16 wake-up lines work as wake-up sources for exiting from STOP mode.

XTI Mask Register High (XTI_MRH)

Address Offset: 28h
Reset value: 00h

7	6	5	4	3	2	1	0
XTIM15	XTIM14	XTIM13	XTIM12	XTIM11	XTIM10	XTIM9	XTIM8
rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:0	**XTIM[15:8]:** *Wake-Up Mask bits.* If XTIMx is set, an interrupt and/or a wake-up event (depending on ID1S and WKUP-INT bits) are generated if the corresponding XTIPx pending bit is set. More precisely, if XTIMx=1 and XTIPx=1 then: – If ID1S=1 and WKUP-INT=1 then an interrupt and a wake-up events are generated. – If ID1S=1 and WKUP-INT=0 only an interrupt is generated. – If ID1S=0 and WKUP-INT=1 only a wake-up event is generated. – If ID1S=0 and WKUP-INT=0 neither interrupts nor wake-up events are generated. If XTIMx is reset, no wake-up events can be generated.

XTI Mask Register Low (XTI_MRL)

Address Offset: 2Ch
Reset value: 00h

7	6	5	4	3	2	1	0
XTIM7	XTIM6	XTIM5	XTIM4	XTIM3	XTIM2	XTIM1	XTIM0
rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:0	**XTIM[7:0]:** *Wake-Up Mask bits.* If XTIMx is set, an interrupt and/or a wake-up event (depending on ID1S and WKUP-INT bits) are generated if the corresponding XTIPx pending bit is set. More precisely, if XTIMx=1 and XTIPx=1 then: – If ID1S=1 and WKUP-INT=1 then an interrupt and a wake-up events are generated. – If ID1S=1 and WKUP-INT=0 only an interrupt is generated. – If ID1S=0 and WKUP-INT=1 only a wake-up event is generated. – If ID1S=0 and WKUP-INT=0 neither interrupts nor wake-up events are generated. If XTIMx is reset, no wake-up events can be generated.

XTI Trigger Polarity Register High (XTI_TRH)

Address Offset: 30h
Reset value: 00h

7	6	5	4	3	2	1	0
XTIT15	XTIT14	XTIT13	XTIT12	XTIT11	XTIT10	XTIT9	XTIT8
rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:0	**XTIT[15:8]:** *XTI Trigger Polarity Bits* These bits are set and cleared by software. 0: The corresponding XTIPx pending bit will be set on the falling edge of the input wake-up line. 1: The corresponding XTIPx pending bit will be set on the rising edge of the input wake-up line.

XTI Trigger Polarity Register Low (XTI_TRL)

Address Offset: 34h
Reset value: 00h

7	6	5	4	3	2	1	0
XTIT7	XTIT6	XTIT5	XTIT4	XTIT3	XTIT2	XTIT1	XTIT0
rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:0	**XTIT[7:0]:** *XTI Trigger Polarity Bits* These bits are set and cleared by software. 0: The corresponding XTIPx pending bit will be set on the falling edge of the input wake-up line. 1: The corresponding XTIPx pending bit will be set on the rising edge of the input wake-up line. **Caution:** As the external wake-up lines are edge triggered, no glitches must be generated on these lines. If either a rising or a falling edge on the external wake-up lines occurs while writing the XTI_TRH or XTI_TRL registers, the pending bit will not be set.

XTI Pending Register High (XTI_PRH)

Address Offset: 38h
Reset value: 00h

7	6	5	4	3	2	1	0
XTIP15	XTIP14	XTIP13	XTIP12	XTIP11	XTIP10	XTIP9	XTIP8
rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

Bits 7:0	**XTIP[15:8]:** *XTI Pending Bits.* These bits are set by hardware on occurrence of the trigger event on the corresponding wake-up line. These bits can be written by software only to ‘0’. 0: No Wake-up Trigger event occurred. 1: Wake-up Trigger event occurred.

XTI Pending Register Low (XTI_PRL)

Address Offset: 3Ch
Reset value: 00h

7	6	5	4	3	2	1	0
XTIP7	XTIP6	XTIP5	XTIP4	XTIP3	XTIP2	XTIP1	XTIP0
rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

Bits 7:0	**XTIP[7:0]:** *XTI Pending Bits.* These bits are set by hardware on occurrence of the trigger event on the corresponding wake-up line. These bits can be written by software only to ‘0’. 0: No Wake-up Trigger event occurred. 1: Wake-up Trigger event occurred.

4.4.5 XTI register map

Table 21. XTI register map

Address Offset	Register Name	7	6	5	4	3	2	1	0
1C	XTI_SR	XTIS(7:0)							
24	XTI_CTRL	reserved					STOP	ID1S	WKUP-INT
28	XTI_MRH	XTIM(15:8)							
2C	XTI_MRL	XTIM(7:0)							
30	XTI_TRH	XTIT(15:8)							
34	XTI_TRL	XTIT(7:0)							
38	XTI_PRH	XTIP(15:8)							
3C	XTI_PRL	XTIP(7:0)							

See *Table 3 on page 14* for base address

5 Real time clock (RTC)

5.1 Introduction

The RTC provides a set of continuously running counters which can be used, with suitable software, to provide a clock-calendar function. The counter values can be written to set the current time/date of the system.

The RTC includes an APB slave interface, to provide access by word to internal 32-bit registers; this interface is disconnected from the APB bus when the main power supply is removed.

5.2 Main features

- Programmable prescaler: external clock divided up to 2^{20}
- 32-bit programmable counter for long term measurement
- External clock input (must be at least 4 times slower than PCLK2 clock, usually 32 kHz)
- Separate power supply
- 4 dedicated maskable interrupt lines:
  - Alarm interrupt, for generating a software programmable alarm interrupt
  - Seconds interrupt, for generating a periodic interrupt signal with a programmable period length (up to 1 sec.)
  - Overflow interrupt, to detect when the of the internal programmable counter rolls over to zero
  - Global interrupt: a logical OR function of all the above, to allow a single interrupt channel to manage all the interrupt sources

5.3 Functional description

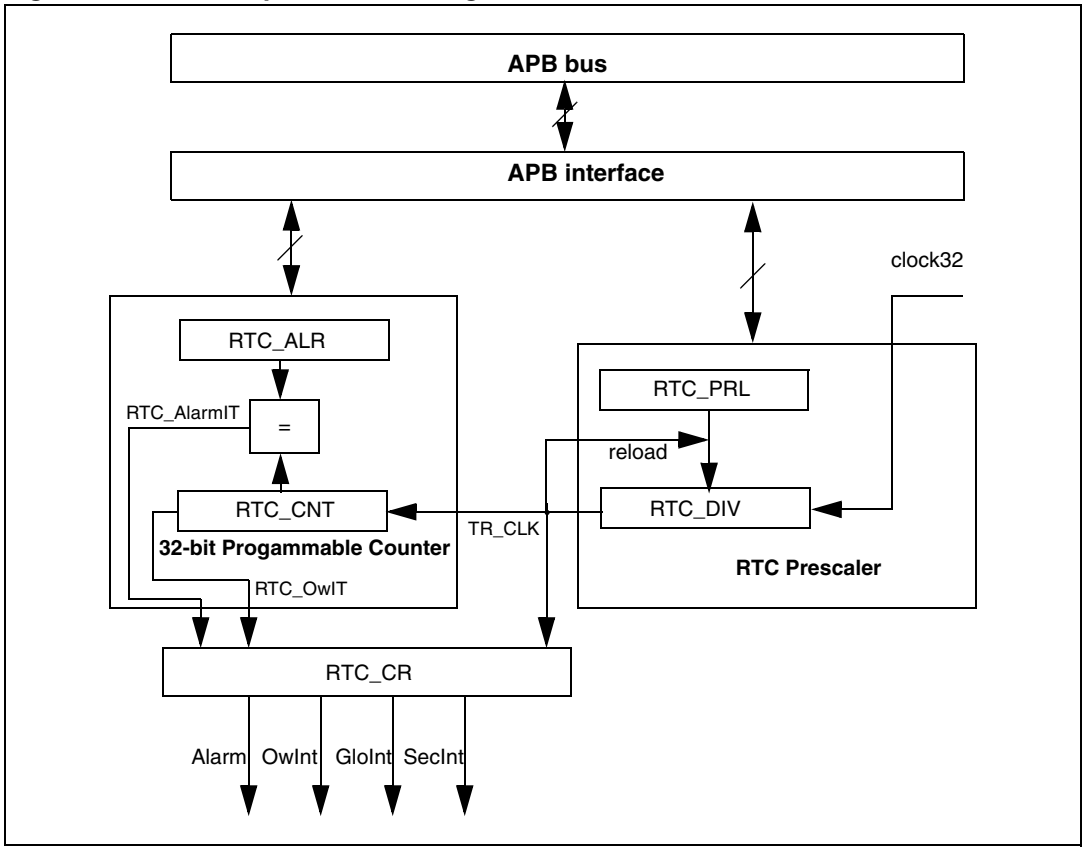
5.3.1 Overview

The RTC consists of two main units (see *Figure 29 on page 94*), the first one (APB Interface) is used to interface the APB bus. This unit also contains a set of 16-bit registers, synchronous to PCLK2 Clock and accessible from the APB bus in read or write mode (for more details refer to Register description section). The APB interface is clocked by the PCLK2 Clock.

The other unit (RTC Core) consists of a chain of programmable counters made of 2 main blocks. The first block is the RTC precaler block which generates the RTC time base TR_CLK which can be programmed to have a period of up 1 second. It includes a 20-bit programmable divider (RTC Prescaler). Every TR_CLK period, the RTC generates an interrupt (SecInt) if it is enabled in the RTC_CR register. The second block is a 32-bit programmable counter that can be initialised to the current system time. The system time is incremented at the TR_CLK rate and compared with a programmable date (stored in the RTC_ALR register) in order to generate an alarm interrupt, if enabled in RTC_CR control register.

Important note: Due to the fact the RTC has 2 different clock domains, after wake-up from STOP mode, the APB interface (controlled by PCLK2) does not match the registers/counter values of the RTC domain (clocked by the 32 kHz osc) that keeps running while the system is in STOP mode. Therefore to avoid reading wrong values in the APB interface, the application should wait for at least 1 RTC clock period or 31.25µs after exiting STOP mode before reading the RTC registers.

Figure 29. RTC simplified block diagram



5.3.2 Reset procedure

All system registers are asynchronously reset by the System Reset or the Software Reset, except RTC_ALR, RTC_CNT, RTC_DIV. These registers and the Real-time clock counter are reset only by the Low Voltage Detector (Power-on reset). They are not affected by any other reset source, nor by Standby mode.

5.3.3 Free-running mode

After Power-on reset, the peripheral enters free-running mode. In this operating mode, the RTC Prescaler and the Programmable counter start counting. Interrupt flags are activated too but, since interrupt signals are masked, there is no interrupt generation. Interrupt signals must be enabled by setting the appropriate bits in the RTC_CR register. In order to avoid spurious interrupt generation it is recommended to clear old interrupt requests before enabling them.

5.3.4 RTC flag assertion

The RTC Second Interrupt Request (SIR) is asserted at each RTC Core clock cycle before the update of the RTC Counter.

The RTC Overflow Interrupt Request (OWIR) is asserted at the last RTC Core clock cycle before the counter reaches the 0x0000 value.

The RTC_Alarm and RTC Alarm Interrupt Request (AIR), see *Figure 30*, are asserted at the last RTC Core clock cycle before the counter reaches the RTC Alarm value stored in the Alarm register increased by one (RTC_ALR + 1). To set the RTC Alarm value, you must be sure that this write is synchronized with the RTC Second flag. For this purpose, two alternative methods can be used:

- Use the RTC Alarm interrupt and update the RTC Alarm and/or RTC Counter registers in the RTC interrupt service routine.
- Wait for the SIR flag until it is set and then update the RTC Alarm and/or RTC Counter registers.

Note: If RTC interrupts are used during Run, Slow, WFI or LPWFI modes the RTC clock must be at least 4 times slower than PCLK2 clock. However, it still possible to use the RTC alarm but through the XTI interrupt (XTI line 15).

Figure 30. RTC second and alarm waveform example with PR=0003, ALARM=00004

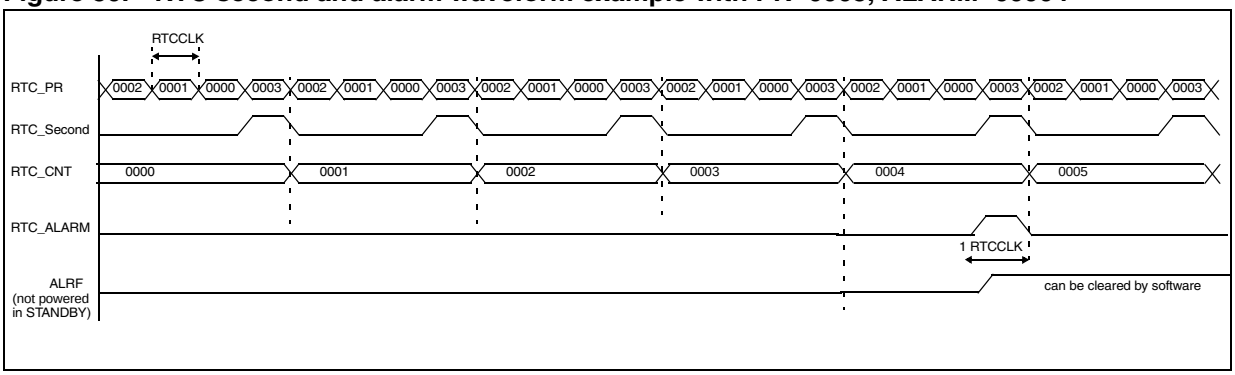
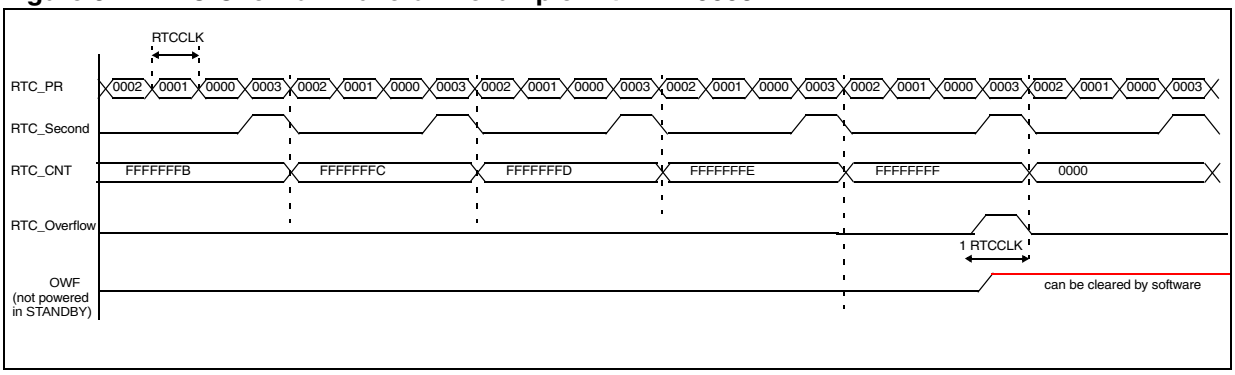


Figure 31. RTC Overflow waveform example with PR=0003



5.3.5 Configuration mode

To write in RTC_PRL, RTC_CNT, RTC_ALR registers, the peripheral must enter Configuration mode. This is done setting the CNF bit in the RTC_CRL register.

In addition, writing to any RTC register is only enabled if the previous write operation is finished. To enable the software to detect this situation, the RTOFF status bit is provided in the RTC_CR register to indicate that an update of the registers is in progress. A new value can be written to the RTC counters only when the status bit value is '1'.

Configuration Procedure:

1. Poll RTOFF, wait until its value goes to '1'
2. Set CNF bit to enter configuration mode
3. Write to one or more RTC registers
4. Clear CNF bit to exit configuration mode

The write operation only executes when the CNF bit is cleared and it takes at least two Clock32 cycles to complete.

5.4 Register description

The RTC registers cannot be accessed by byte. The reserved bits can not be written and they are always read as '0'.

5.4.1 RTC control register high (RTC_CRH)

Address Offset: 00h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												GEN	OWEN	AEN	SEN
-												rw	rw	rw	rw

These bits are used to mask interrupt requests. Note that at reset all interrupts are disabled, so it is possible to write to the RTC registers to ensure that no interrupt requests are pending after initialization. It is not possible to write RTC_CRH register when the peripheral is completing a previous write operation (flagged by RTOFF=0, see *Section 5.3.5: Configuration mode on page 96*).

The functions of the RTC are controlled by this control register. Some bits must be written using a specific configuration procedure (see *Section 5.3.5: Configuration mode on page 96*).

Bits 15:4	Reserved, always read as 0.
Bit 3	**GEN**: *Global interrupt Enable* 0: Global interrupt is masked. 1: Global interrupt is enabled.
Bit 2	**OWEN**: *Overflow interrupt Enable* 0: Overflow interrupt is masked. 1: Overflow interrupt is enabled.

Bit 1	**AEN**: *Alarm interrupt Enable* 0: Alarm interrupt is masked. 1: Alarm interrupt is enabled.
Bit 0	**SEN**: *Second interrupt Enable* 0: Second interrupt is masked. 1: Second interrupt is enabled.

5.4.2 RTC control register low (RTC_CRL)

Address Offset: 04h
Reset value: 0020h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										RTOFF	CNF	GIR	OWIR	AIR	SIR
-										r	rw	rc_w0	rc_w0	rc_w0	rc_w0

The functions of the RTC are controlled by this control register. It is not possible to write RTC_CR register when the peripheral is completing a previous write operation (flagged by RTOFF=0, see *Section 5.3.5: Configuration mode on page 96*).

Bits 15:6	Reserved, always read as 0.
Bit 5	**RTOFF**: *RTC operation OFF* With this bit the RTC reports the status of the last write operation performed on its registers, indicating if it has been completed or not. If its value is '0' then it is not possible to write to any of the RTC registers. This bit is read only. 0: Last write operation on RTC registers is still ongoing. 1: Last write operation on RTC registers terminated.
Bit 4	**CNF**: *Configuration Flag* This bit must be set by software to enter configuration mode so as to allow new values to be written in the RTC_CNT, RTC_ALR or RTC_PRL registers. The write operation is only executed when , the CNF bit is reset by software after has been set. 0: Exit configuration mode (start update of RTC registers). 1: Enter configuration mode.
Bit 3	**GIR**: *Global Interrupt Request* This bit contains the status of global interrupt request signal, which is goes high when at least one of the other interrupt lines is active. When this bit is set, the corresponding interrupt will be generated only if GEN bit is set. The GIR bit can be set only by hardware and can be cleared only by software, while writing '1' will left it unchanged. 0: GloInt interrupt condition not met. 1: GloInt interrupt request pending.
Bit 2	**OWIR**: *Overflow Interrupt Request* This bit stores the status of periodic interrupt request signal (*RTC_OwIT*) generated by the overflow of the 32-bit programmable counter. This interrupt may be used to wake-up the system from a long lasting Standby condition, if the system time has to be kept up-to-date. When this bit is at '1', the corresponding interrupt will be generated only if OWEN bit is set to '1'. OWEN bit can be set at '1' only by hardware and can be cleared only by software, while writing '1' will left it unchanged. 0: Overflow interrupt condition not met. 1: Overflow interrupt request pending.

Bit 1	**AIR**: *Alarm Interrupt Request* This bit contains the status of periodic interrupt request signal (*RTC_AlarmIt*) generated by the 32 bit programmable counter when the threshold set in RTC_ALR register is reached. When this bit is at '1', the corresponding interrupt will be generated only if AEN bit is set to '1'. AIR bit can be set at '1' only by hardware and can be cleared only by software, while writing '1' will left it unchanged. 0: Alarm interrupt condition not met. 1: Alarm interrupt request pending.
Bit 0	**SIR**: *Second Interrupt Request* This bit contains the status of second interrupt request signal (*RTC_SecIt*) generated by the overflow of the 20-bit programmable prescaler which increments the RTC counter. Hence this Interrupt provides a periodic signal with a period corresponding to the resolution programmed for the RTC counter (usually one second). When this bit is at '1', the corresponding interrupt will be generated only if SEN bit is set to '1'. SIR bit can be set at '1' only by hardware and can be cleared only by software, while writing '1' will left it unchanged. 0: 'Second' interrupt condition not met. 1: 'Second' interrupt request pending. Any interrupt request remains pending until the appropriate RTC_CR request bit is reset by software, notifying that the interrupt request has been granted. **Note:** At reset the interrupts are disabled, it is possible to write the RTC registers and no interrupt requests are pending.

5.4.3 RTC prescaler load register high (RTC_PRLH)

Address Offset: 08h
Write only (see *Section 5.3.5: Configuration mode on page 96*)
Reset value: 0000h

The Prescaler Load registers keep the period counting value of the RTC prescaler. They are write protected by the RTOFF bit in the RTC_CR register, write operation is allowed if RTOFF value is '1'.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												PRL(19:16)			
-												w	w	w	w

Bits 15:4	Reserved, always read as 0.
Bits 3:0	**PRL[19:16]:** *RTC Prescaler reload value high* These bits are used to define the counter clock frequency according to the following formula: $f_{TR\_CLK} = f_{RTC}/(PRL[19:0]+1)$. **Caution:** The zero value is not recommended, otherwise RTC interrupts and flags cannot be asserted correctly.

5.4.4 RTC prescaler load register low (RTC_PRLL)

Address Offset: 0Ch
Write only (see *Section 5.3.5: Configuration mode on page 96*)
Reset value: 8000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PRL(15:0)															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 15:0	**PRL[15:0]:** *RTC Prescaler reload value low* These bits are used to define the counter clock frequency according to the following formula: $f_{TR\_CLK} = f_{RTC}/(PRL[19:0]+1)$

Note: If the input clock frequency (f_{RTC}) is 32.768 kHz, write 7FFFh in this register to get a signal period of 1 second.

5.4.5 RTC prescaler divider register high (RTC_DIVH)

Address Offset: 10h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												RTCDIV[19:16]			
-												r	r	r	r

Every period of TR_CLK the counter inside RTC prescaler is reloaded with the value stored in the RTC_PRL register. To get an accurate time measurement it is possible to read the current value of the prescaler counter, stored into the RTC_DIV register, without stopping it. This register is read only and it is reloaded by hardware after any change in RTC_PRL or RTC_CNT registers.

Bits 15:4	Reserved, always read as 0.
Bits 3:0	**RTCDIV[19:16]:** *RTC Clock Divider High*

5.4.6 RTC prescaler divider register low (RTC_DIVL)

Address Offset: 14h
Reset value: 8000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTCDIV[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 15:0	**RTCDIV[15:0]:** *RTC Clock Divider Low*

5.4.7 RTC counter register high (RTC_CNTH)

Address Offset: 18h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTC_CNT[31:16]															
rw															

The RTC core has one 32-bit programmable counter, accessed through 2 16-bit registers; the count rate is based on the TR_Clock time reference, generated by the prescaler. RTC_CNT registers keep the counting value of this counter. They are write protected by bit RTOFF in the RTC_CR register, write operation is allowed if RTOFF value is '1'. A write operation on the upper (RTC_CNTH) or lower (RTC_CNTL) registers directly loads the corresponding programmable counter and reloads the RTC Prescaler. When reading, the current value in the counter (system date) is returned. The counters keep on running while the external clock oscillator is working even if the main system is powered down (Standby mode).

Bits 15:0	**RTC_CNT[31:16]:** *RTC Counter High* Reading RTC_CNTH register, the current value of the high part of RTC Counter register is returned. To write this register it is required to enter configuration mode using the RTOFF bit in the RTC_CR register.

5.4.8 RTC counter register low (RTC_CNTL)

Address Offset: 1Ch
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTC_CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0	**RTC_CNT[15:0]:** *RTC Counter Low* Reading RTC_CNTL register, the current value of the lower part of RTC Counter register is returned. To write this register it is required to enter configuration mode using the RTOFF bit in the RTC_CR register.

5.4.9 RTC alarm register high (RTC_ALRH)

Address Offset: 20h
Write only (see *Section 5.3.5: Configuration mode on page 96*)
Reset value: FFFFh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTC_ALR[31:16]															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

When the programmable counter reaches the 32-bit value stored in the RTC_ALR register, an alarm is triggered and the RTC_alarmIT interrupt request is generated. This register is write protected by the RTOFF bit in the RTC_CR register, write operation is allowed if the RTOFF value is '1'.

Bits 15:0	**RTC_ALR[31:16]:** *RTC Alarm High* The high part of alarm time is written by software in this register. To write this register it is required to enter configuration mode using the RTOFF bit in the RTC_CR register.

5.4.10 RTC alarm register low (RTC_ALRL)

Address Offset: 24h
Write only (see *Section 5.3.5: Configuration mode on page 96*)
Reset value: FFFFh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTC_ALR[15:0]															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 15:0	**RTC_ALR[15:0]:** *RTC Alarm Low* The low part of alarm time is written by software in this register. To write this register it is required to enter configuration mode using the RTOFF bit in the RTC_CR register.

5.5 RTC register map

RTC registers are mapped as 16-bit addressable registers as described in the table below:

Table 22. RTC Register Map

Address Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00h	RTC_CRH	---												GEN	OWEN	AEN	SEN
04h	RTC_CRL	---										RTOFF	CNF	GIR	OWIR	AIR	SIR
08h	RTC_PRLH	---												PRL			
Ch	RTC_PRLL	PRL															
10h	RTC_DIVH	---												DIV			
14h	RTC_DIVL	DIV															
18h	RTC_CNTH	CNTH															
1Ch	RTC_CNTL	CNTL															
20h	RTC_ALRH	ALARMH															
24h	RTC_ALRL	ALARML															

See *Table 3 on page 14* for base address

6 Watchdog timer (WDG)

6.1 Introduction

The Watchdog Timer peripheral can be used as free-running timer or as Watchdog to resolve processor malfunctions due to hardware or software failures.

6.2 Main features

- 16-bit down Counter
- 8-bit clock Prescaler
- Safe Reload Sequence
- Free-running Timer mode
- End of Counting interrupt generation

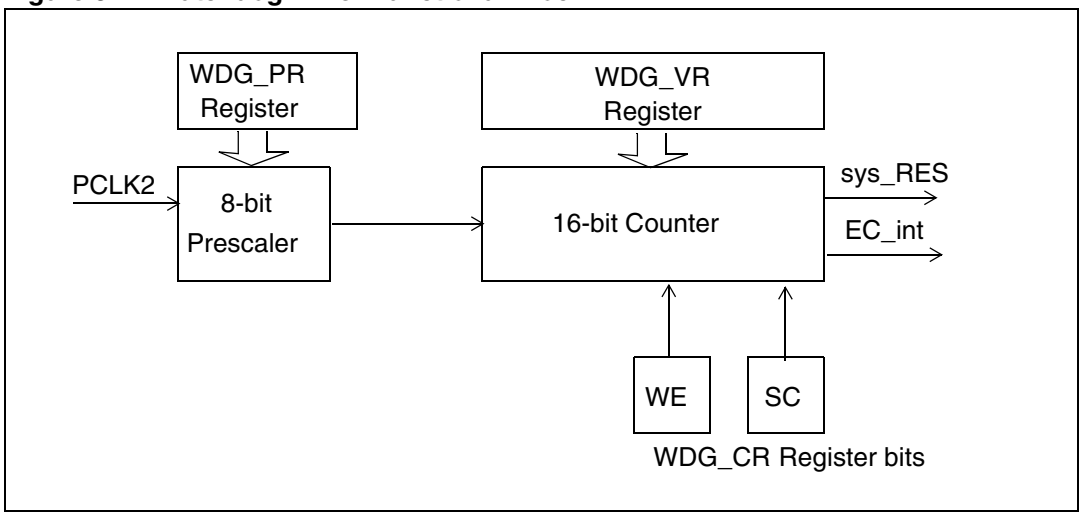
6.3 Functional description

Figure 32 shows the functional blocks of the Watchdog Timer module. The module can work as Watchdog or as Free-running Timer. In both working modes the 16-bit Counter value can be accessed through a reading of the WDG_CNT register.

6.3.1 Free-running timer mode

If the WE bit of WDG_CR register is not written to '1' by software, the peripheral enters Free-running Timer mode.
When in this operating mode as the SC bit of WDG_CR register is written to '1' the WDG_VR value is loaded in the Counter and the Counter starts counting down.

Figure 32. Watchdog Timer Functional Block



When it reaches the end of count value (0000h) an End of Count interrupt is generated (EC_int) and the WDG_VR value is re-loaded. The Counter runs until the SC bit is cleared.

6.3.2 Watchdog mode

If the WE bit of WDG_CR register is written to '1' by software, the peripheral enters Watchdog mode. This operating mode can not be changed by software (the SC bit has no effect and WE bit cannot be cleared).
As the peripheral enters in this operating mode, the WDG_VR value is loaded in the Counter and the Counter starts counting down. When it reaches the end of count value (0000h) a system reset signal is generated (sys_RES).

If a sequence of two consecutive values (0xA55A and 0x5AA5) is written in the WDG_KR register see *Section 6.4*, the WDG_VR value is re-loaded in the Counter, so the End of count can be avoided.

6.4 Register description

The Watchdog Timer registers can not be accessed by byte.
The reserved bits can not be written and they are always read at '0'.

6.4.1 WDG control register (WDG_CR)

Address Offset: 00h

Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved														SC	WE
-														rw	rw

Bits 15:2	Reserved, must be kept cleared.
Bit 1	**SC**: *Start Counting bit* 0: The counter is stopped. 1: The counter loads the Timer pre-load value and starts counting These functions are permitted only in Timer Mode (WE bit = 0).
Bit 0	**WE**: *Watchdog Enable bit* 0: Timer Mode is enabled 1: Watchdog Mode is enabled This bit can't be reset by software. When WE bit is high, SC bit has no effect.

6.4.2 WDG prescaler register (WDG_PR)

Address Offset: 04h

Reset value: 00FFh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved								PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
-								rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:8	Reserved.
Bits 7:0	**PR[7:0]**: *Prescaler value* The clock to Timer Counter is divided by PR[7:0]+1. This value takes effect when Watchdog mode is enabled (WE bit is put to '1') or the re-load sequence occurs or the Counter starts (SC) bit is put to '1' in Timer mode.

6.4.3 WDG preload value register (WDG_VR)

Address Offset: 08h

Reset value: FFFFh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TV15	TV14	TV13	TV12	TV11	TV10	TV9	TV8	TV7	TV6	TV5	TV4	TV3	TV2	TV1	TV0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0	**TV[15:0]**: *Timer Pre-load Value* This value is loaded in the Timer Counter when it starts counting or a re-load sequence occurs or an End of Count is reached. The time (µs) need to reach the end of count is given by: (PR[7:0]+1)*(TV[15:0]+1)*t_{PCLK2}/1000 (µs) where t_{PCLK2} is the Clock period measured in ns. I.e. if PCLK2 = 20 MHz the default timeout set after the system reset is 256*65535*50/1000 = 838800 µs.

6.4.4 WDG counter register (WDG_CNT)

Address Offset: 0Ch

Reset value: FFFFh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT15	CNT14	CNT13	CNT12	CNT11	CNT10	CNT9	CNT8	CNT7	CNT6	CNT5	CNT4	CNT3	CNT2	CNT1	CNT0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 15:0	**CNT[15:0]:** *Timer Counter Value* The current counting value of the 16-bit Counter is available reading this register.

6.4.5 WDG status register (WDG_SR)

Address Offset: 10h

Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved															EC
-															r-c

Bits 15:1	Reserved.
Bit 0	**EC**: *End of Count pending bit* 0: No End of Count has occurred 1: The End of Count has occurred In Watchdog Mode (WE = 1) this bit has no effect. This bit can be set only by hardware and must be reset by software.

6.4.6 WDG mask register (WDG_MR)

Address Offset: 14h

Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved															ECM
-															rw

Bits 15:1	Reserved.
Bit 0	**ECM**: *End of Count Mask bit* 0: End of Count interrupt request is disabled 1: End of Count interrupt request is enabled

6.4.7 WDG key register (WDG_KR)

Address Offset: 18h

Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
K15	K14	K13	K12	K11	K10	K9	K8	K7	K6	K5	K4	K3	K2	K1	K0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0	**K[15:0]:** *Key Value* When Watchdog Mode is enabled, writing in this register two consecutive values (refer to device specification) the Counter is initialized to TV[15:0] value and the Prescaler value in WTDPR register take effect. Any number of instructions can be executed between the two writes. If Watchdog Mode is disabled (WE = 0) a writing in this register has no effect. The reading value of this register is 0000h.

6.5 WDG register map

Table 23. Watchdog-Timer Register Map

Addr. Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00h	WDG_CR	reserved														SC	WE
04h	WDG_PR	reserved								PR(7:0)							
08h	WDG_VR	TV(15:0)															
0Ch	WDG_CNT	TV(15:0)															
10h	WDG_SR	reserved															EC
14h	WDG_MR	reserved															MEC
18h	WDG_KR	K[15:0]															

See *Table 3 on page 14* for base address

7 Timer (TIM)

7.1 Introduction

A timer consists of a 16-bit counter driven by a programmable prescaler.

It may be used for a variety of purposes, including pulse length measurement of up to two input signals (input capture) or generation of up to two output waveforms (output compare and PWM).

Pulse lengths and waveform periods can be modulated from a very wide range using the timer prescaler.

7.2 Main features

- Programmable prescaler: f_{PCLK2} divided from 1 to 256, Prescaler register (0 to 255) value +1.
- Overflow status flag and maskable interrupts
- External clock input (must be at least 4 times slower than the PCLK2 clock speed) with the choice of active edge
- Output compare functions with
  - 2 dedicated 16-bit registers
  - 2 dedicated programmable signals
  - 2 dedicated status flags
  - 2 dedicated interrupt flags.
- Input capture functions with
  - 2 dedicated 16-bit registers
  - 2 dedicated active edge selection signals
  - 2 dedicated status flags
  - 2 dedicated interrupt flags.
- Pulse width modulation mode (PWM)
- One pulse mode (OPM)
- PWM input mode
- Timer global interrupt (5 internally OR'ed or separated sources, depending on device)
  - ICIA: Timer Input capture A interrupt
  - ICIB: Timer Input capture B interrupt
  - OCIA: Timer Output compare A interrupt
  - OCIB: Timer Output compare B interrupt
  - TOI: Timer Overflow interrupt.

The Block Diagram is shown in *Figure 33*.

7.3 Special features

Timer0:

- T0.EXTCLK input is directly connected to CK pin through a prescaler, which divides the input frequency by eight.
- T0.ICAP_B is connected to RTC ALARM; this allows to synchronize Timer0 and the Real Time Clock.

Timer2:

- T2.ICAP_B (Input Capture B) is connected to HDLC_HRMC (HDLC Reception Message Complete) from HDLC, which is the Interrupt Request generated on completion of a correct data frame.
- T2.OCMP_B (Output Compare B) is connected to HDLC_HTEN (Transmit Enable). A rising edge of T2.OCMP_B allows to trigger by Hardware the start of Transmission, enabling a precise timing for this event, or (in combination with the ICAP_B above) a precise delay between the reception of an incoming frame and the start of the answer.

7.4 Functional description

7.4.1 Counter

The principal block of the Programmable Timer is a 16-bit counter and its associated 16-bit registers.

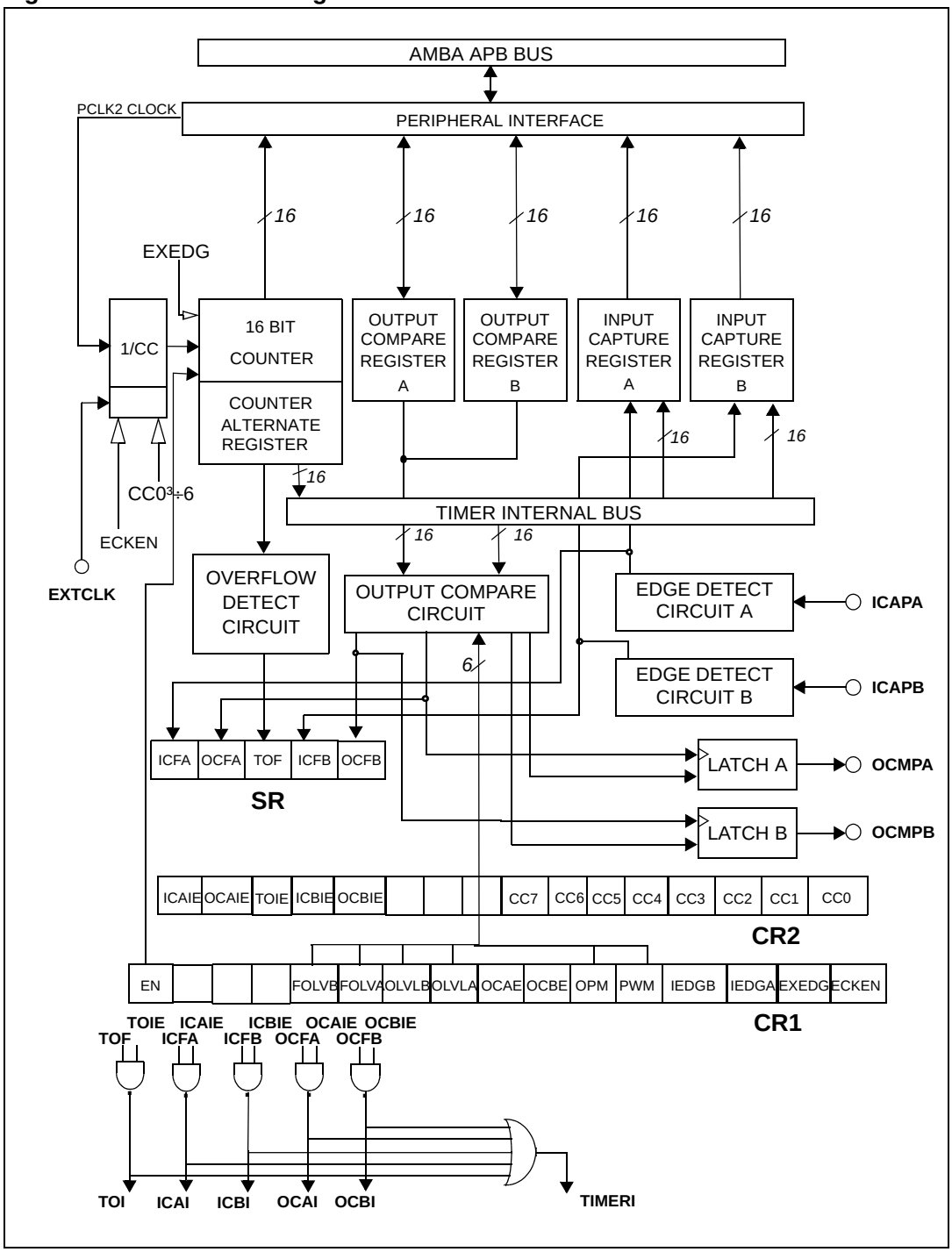
Writing in the Counter Register (CNTR) resets the counter to the FFFCh value.

The timer clock source can be either internal or external selecting ECKEN bit of CR1 register. When ECKEN = 0, the frequency depends on the prescaler division bits (CC7-CC0) of the CR2 register.

An overflow occurs when the counter rolls over from FFFFh to 0000h then the TOF bit of the SR register is set. An interrupt is generated if TOIE bit of the CR2 register is set; if this condition is false, the interrupt remains pending to be issued as soon as it becomes true.

Clearing the overflow interrupt request is done by a write access to the SR register while the TOF bit is set with the data bus 13-bit at ‘0’, while all the other bits shall be written to ‘1’ (the SR register is clear only, so writing a ‘1’ in a bit has no effect: this makes possible to clear a pending bit without risking to clear a new coming interrupt request from another source).

Figure 33. Timer Block Diagram



7.4.2 External clock

The external clock is selected if ECKEN = 1 in CR1 register.

The status of the EXEDG bit determines the type of level transition on the external clock pin EXTCLK that will trigger the counter.

The counter is synchronized with the rising edge of the internal clock coming from the PCLK2 block.

At least four rising edges of the PCLK2 clock must occur between two consecutive active edges of the external clock; thus the external clock frequency must be less than a quarter of the PCLK2 clock frequency.

Note: the external clock is not available for Timer2.

7.4.3 Internal clock

Figure 34. Counter timing diagram, internal clock divided by 2

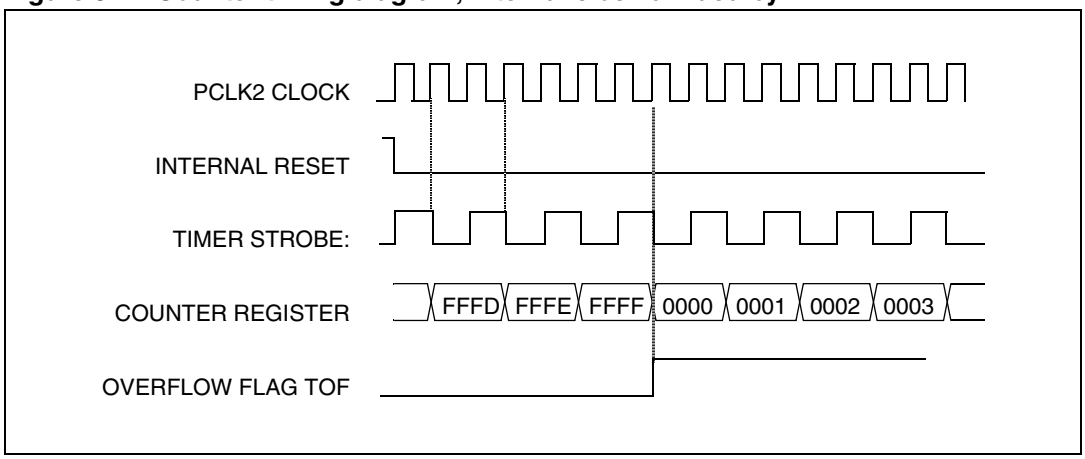


Figure 35. Counter timing diagram, internal clock divided by 4

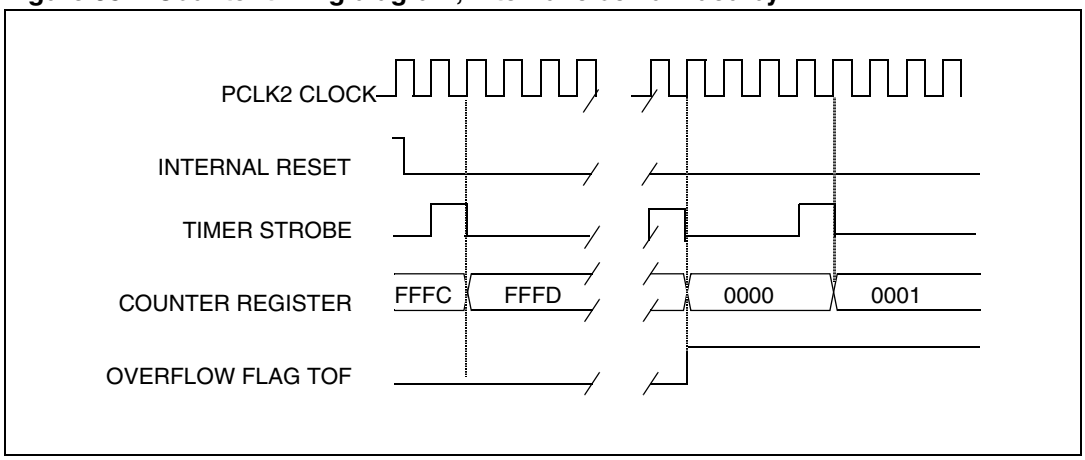
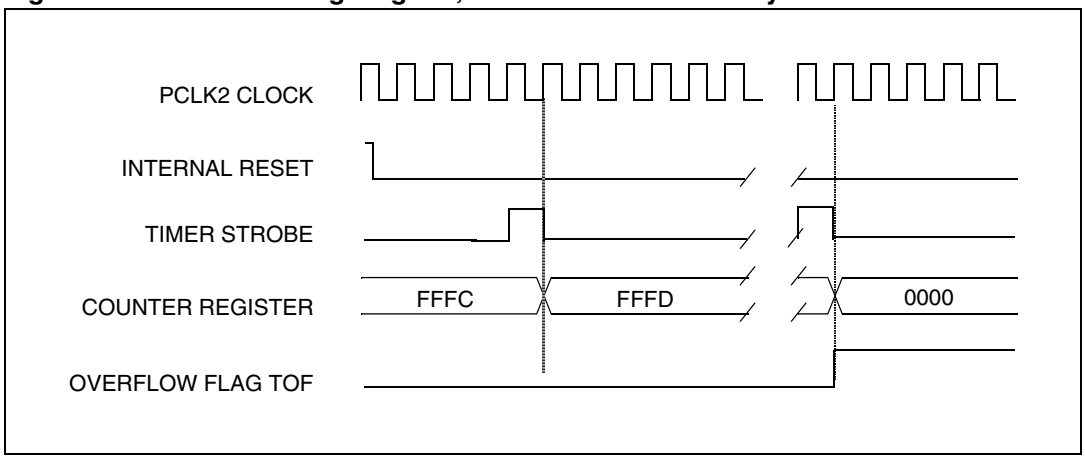


Figure 36. Counter timing diagram, internal clock divided by n



7.4.4 Input capture

In this section, the index "*i*", may be A or B.

The two input capture 16-bit registers (ICAR and ICBR) are used to latch the value of the counter after a transition detected by the ICAP*i* pin (see *Figure 37*).

IC*i*R register are read-only registers.

The active transition is software programmable through the IEDG*i* bit of the Control Register (CR1).

Timing resolution is one count of the counter: ($f_{PCLK2/(CC7 \div CC0+1)}$).

Procedure

To use the input capture function select the following in the CR1 and CR2 registers:

- Select the timer clock source (ECKEN).
- Select the timer clock division factor (CC7÷CC0) if internal clock is used.
- Select the edge of the active transition on the ICAPA pin with the IEDGA bit, if ICAPA is active.
- Select the edge of the active transition on the ICAPB pin with the IEDGB bit, if ICAPB is active.
- Set ICAIE (or ICBIE) when ICAPA (or ICAPB) is active, to generate an interrupt after an input capture.

When an input capture occurs:

- ICF*i* bit is set.
- The IC*i*R register contains the value of the counter on the active transition on the ICAP*i* pin (see *Figure 38*).
- A timer interrupt is generated if ICAIE is set (if only ICAPA is active) or ICBIE is set (if only ICAPB is active); otherwise, the interrupt remains pending until concerned enable bits are set.

Clearing the Input Capture interrupt request is done by:

- A write access to the SR register while the ICF*i* bit is cleared, 15-bit at '0' for ICAPA and 12-bit at '0' for ICAPB.

Figure 37. Input capture block diagram

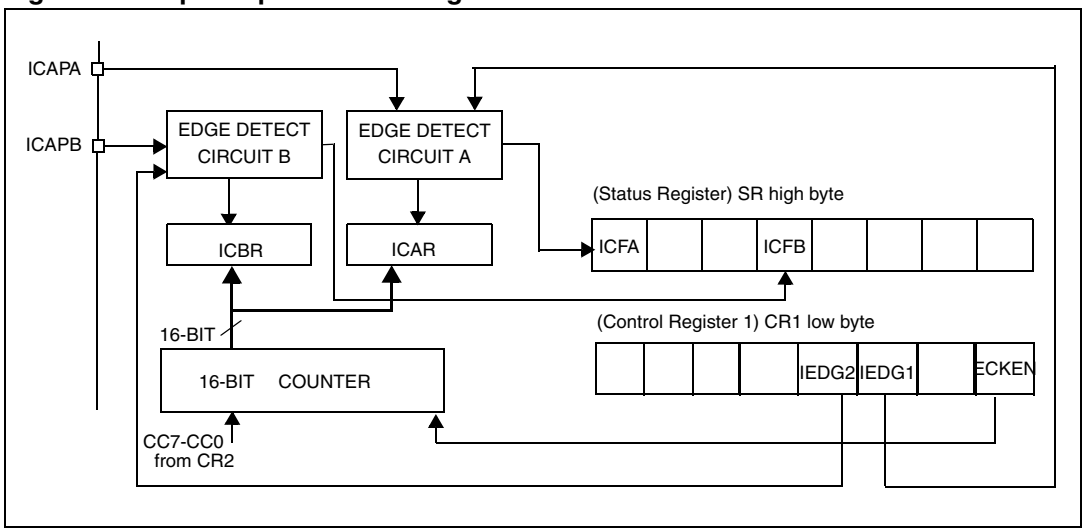
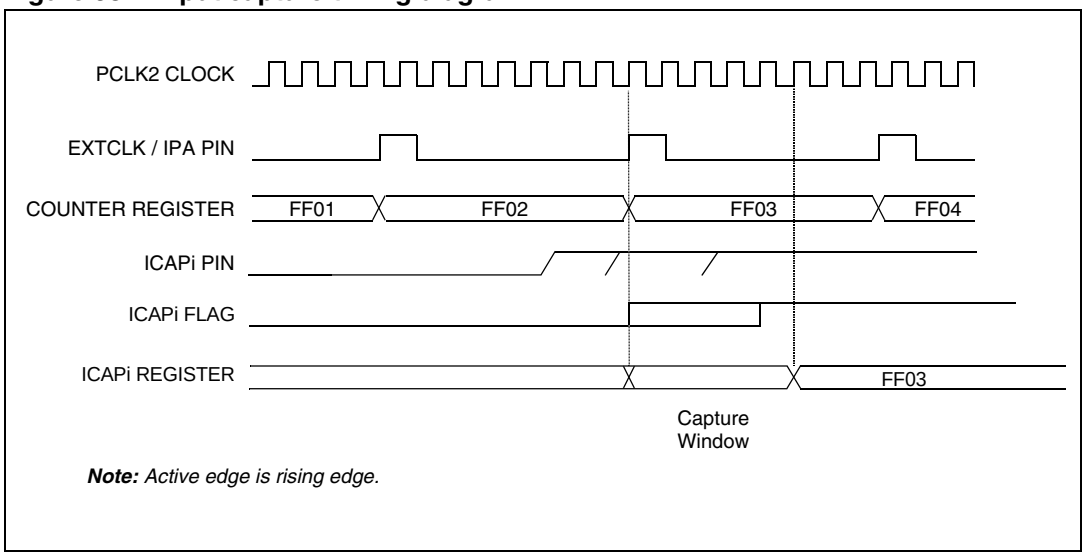


Figure 38. Input capture timing diagram



Note: *Active edge is rising edge.*

7.4.5 Output compare

In this section, the index "*i*", may be A or B.

This function can be used to control an output waveform or indicating when a period of time has elapsed.

When a match is found between the Output Compare register and the counter, the output compare function:

- Assigns pins with a programmable value if the OC*i*E bit is set
- Sets a flag in the status register
- Generates an interrupt if enabled

Two 16-bit registers Output Compare Register A (OCAR) and Output Compare Register B (OCBR) contain the value to be compared to the counter each timer clock cycle.

These registers are readable and writable and are not affected by the timer hardware. A reset event changes the OC*i*R value to 8000h.

Timing resolution is one count of the counter: ($f_{PCLK2/(CC7 \div CC0+1)}$).

Procedure

To use the output compare function, select the following in the CR1/CR2 registers:

- Set the OC*i*E bit if an output is needed then the OCMP*i* pin is dedicated to the output compare *i* function.
- Select the timer clock (ECKGEN) and the prescaler division factor (CC7÷CC0).

Select the following in the CR1/CR2 registers:

- Select the OLVL*i* bit to applied to the OCMP*i* pins after the match occurs.
- Set OCAIE (OCBIE) if only compare A (compare B) needs to generate an interrupt.

When match is found:

- OCF*i* bit is set.
- The OCMP*i* pin takes OLVL*i* bit value (OCMP*i* pin latch is forced low during reset and stays low until valid compares change it to OLVL*i* level).
- A timer interrupt is generated if the OCAIE (or OCBIE) bit in CR2 register is set, the OCAR (or OCBR) matches the timer counter (i.e. OCFA or OCFB is set).

Clearing the output compare interrupt request is done by a write access to the SR register while the OCF*i* bit is cleared, 14-bit at '0' for OCAR and 12-bit at '0' for OCBR.

If the OC*i*E bit is not set, the OCMP*i* pin is at '0' and the OLVL*i* bit will not appear when match is found.

The value in the 16-bit OC*i*R register and the OLVL*i* bit should be changed after each successful comparison in order to control an output waveform or establish a new elapsed timeout.

The OC*i*R register value required for a specific timing application can be calculated using the following formula:

$$\Delta \text{ OC}i\text{R} = \frac{\Delta t * f_{PCLK2}}{(CC7 \div CC0+1)}$$

Where:

Δt = Desired output compare period (in seconds)
f_{PCLK2} = Internal clock frequency
CC7÷CC0 = Timer clock prescaler

Figure 39. Output compare block diagram

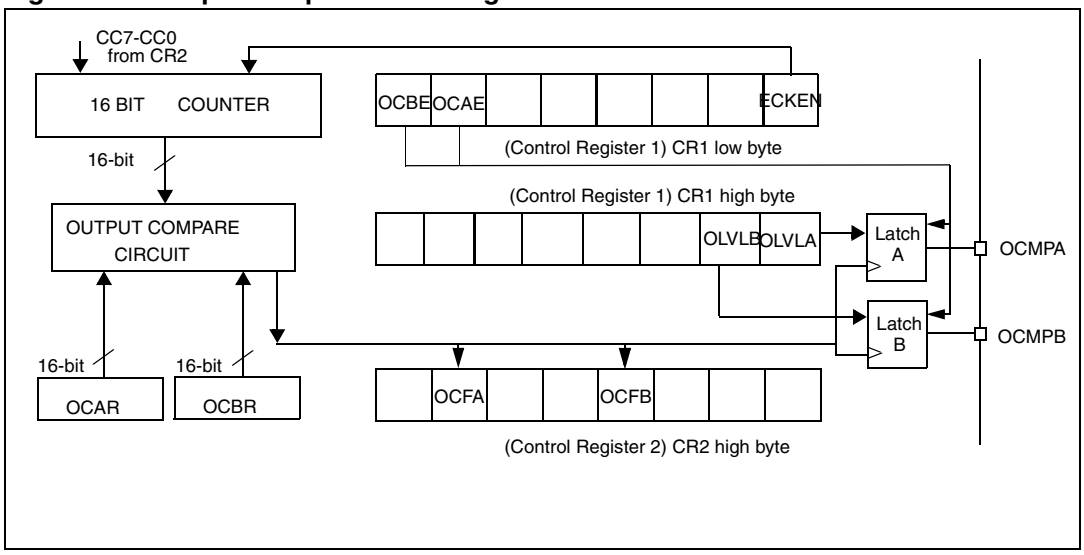
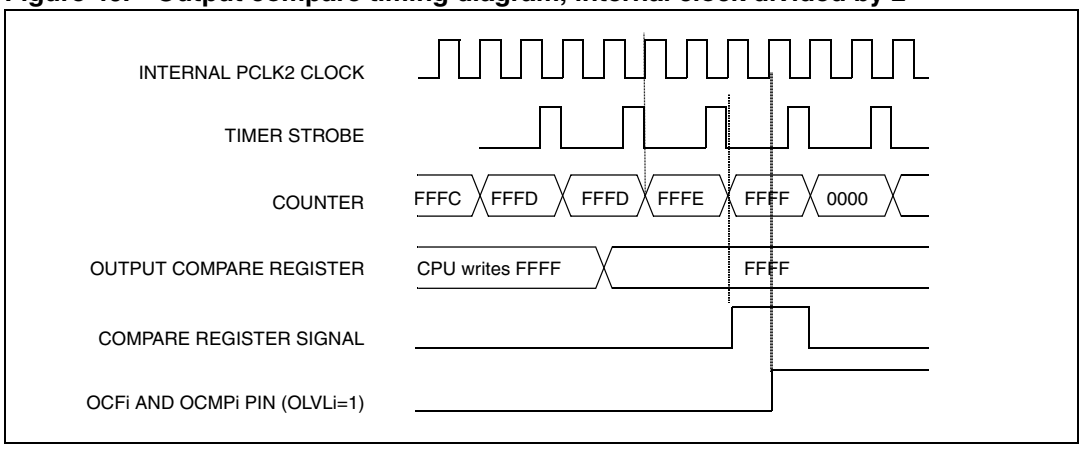


Figure 40. Output compare timing diagram, internal clock divided by 2



7.4.6 Forced compare mode

In this section the index "*i*" may represent A or B.

Bits 11:8 of CR1 register and bits 7:0 of CR2 are used (Refer to *Section 7.6* for detailed Register Description).

When the FOLVA bit is set, the OLVLA bit is copied to the OCMPA pin if PWM and OPM are both cleared. When FOLVB bit is set, the OLVLB bit is copied to the OCMPB pin.

The OLVL*i* bit has to be toggled in order to toggle the OCMP*i* pin when it is enabled (OC*i*E bit=1).

Note: When FOLVi is set, no interrupt request is generated.

Nevertheless the OCFi bit can be set if OCiR = Counter, an interrupt can be generated if enabled.

Input capture function works in Forced Compare mode.

7.4.7 One pulse mode

One Pulse mode enables the generation of a pulse when an external event occurs. This mode is selected via the OPM bit in the CR1 register.

The one pulse mode uses the Input Capture A function (trigger event) and the Output Compare A function.

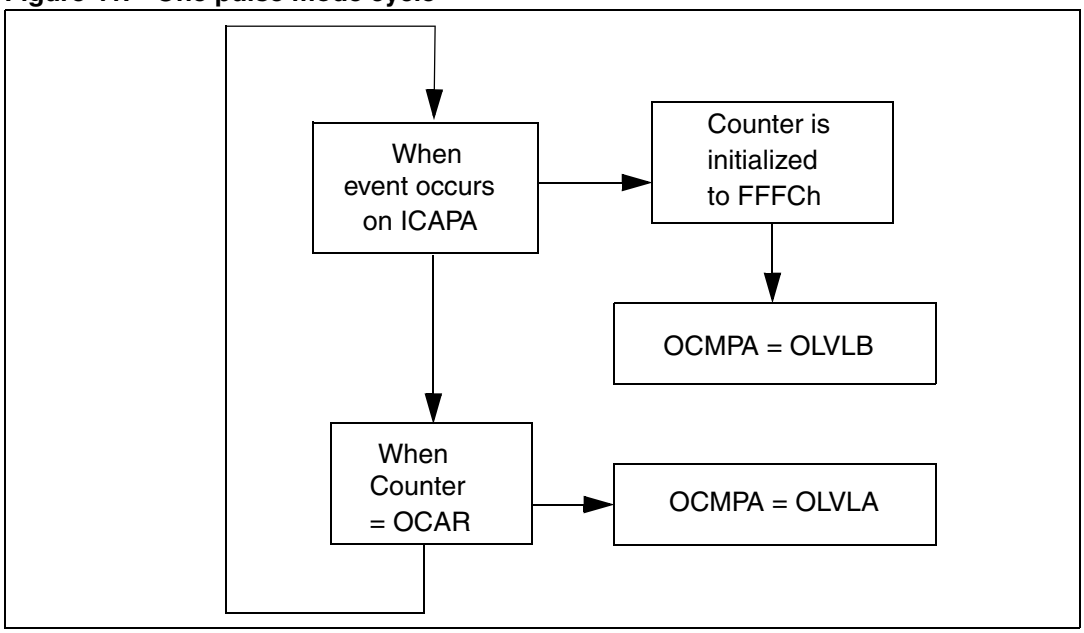
Procedure

To use one pulse mode, select the following in the CR1 register:

- Using the OLVLA bit, select the level to be applied to the OCMPA pin after the pulse.
- Using the OLVLB bit, select the level to be applied to the OCMPA pin during the pulse.
- Select the edge of the active transition on the ICAPA pin with the IEDGA bit.
- Set the OCAE bit, the OCMPA pin is then dedicated to the Output Compare A function.
- Set the OPM bit.
- Select the timer clock (ECKGEN) and the prescaler division factor (CC7-CC0).

Load the OCAR register with the value corresponding to the length of the pulse (see the formula in *Section 7.4.8: Pulse width modulation mode on page 118*).

Figure 41. One pulse mode cycle

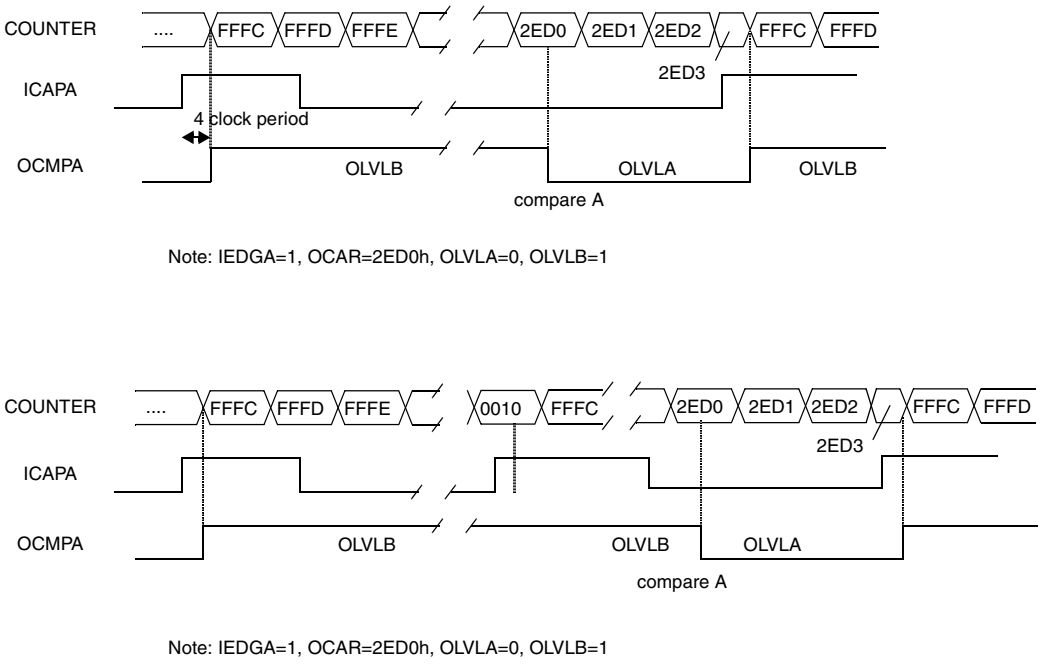


Then, on a valid event on the ICAPA pin, the counter is initialized to FFFCh and OLVLB bit is loaded on the OCMPA pin after four clock period. When the value of the counter is equal to the value of the contents of the OCAR register, the OLVLA bit is output on the OCMPA pin (See *Figure 42*).

Notes

- The OCFA bit cannot be set by hardware in one pulse mode but the OCFB bit can generate an Output Compare interrupt.
- The ICFA bit is set when an active edge occurs and can generate an interrupt if the ICAIE bit is set. The ICAR register will have the value FFFCh.
- When the Pulse Width Modulation (PWM) and One Pulse Mode (OPM) bits are both set with FOLVA= 1, the OPM mode is the only active one, otherwise the PWM mode is the only active one.
- Forced Compare B mode works in OPM
- Input Capture B function works in OPM
- When OCAR = FFFBh in OPM, then a pulse of width FFFFh is generated
- If event occurs on ICAPA again before the Counter reaches the value of OCAR, then the Counter will be reset again and the pulse generated might be longer than expected as in *Figure 42*.
- If a write operation is performed on the counter register before the Counter reaches the value of OCAR, then the Counter will be reset again and the pulse generated might be longer than expected.
- If a write operation is performed on the counter register after the Counter reaches the value of OCAR, then there will have no effect on the waveform.

Figure 42. One pulse mode timing



7.4.8 Pulse width modulation mode

Pulse Width Modulation mode enables the generation of a signal with a frequency and pulse length determined by the value of the OCAR and OCBR registers.

The pulse width modulation mode uses the complete Output Compare A function plus the OCBR register.

Procedure

To use pulse width modulation mode select the following in the CR1 register:

- Using the OLVLA bit, select the level to be applied to the OCMPA pin after a successful comparison with OCAR register.
- Using the OLVLB bit, select the level to be applied to the OCMPA pin after a successful comparison with OCBR register.
- Set OCAE bit: the OCMPA pin is then dedicated to the output compare A function.
- Set the PWM bit.
- Select the timer clock (ECKGEN) and the prescaler division factor (CC7-CC0).

Load the OCBR register with the value corresponding to the period of the signal.

Load the OCAR register with the value corresponding to the length of the pulse if (OLVLA=0 and OLVLB=1).

If OLVLA=1 and OLVLB=0 the length of the pulse is the difference between the OCBR and OCAR registers.

The OC*i*R register value required for a specific timing application can be calculated using the following formula:

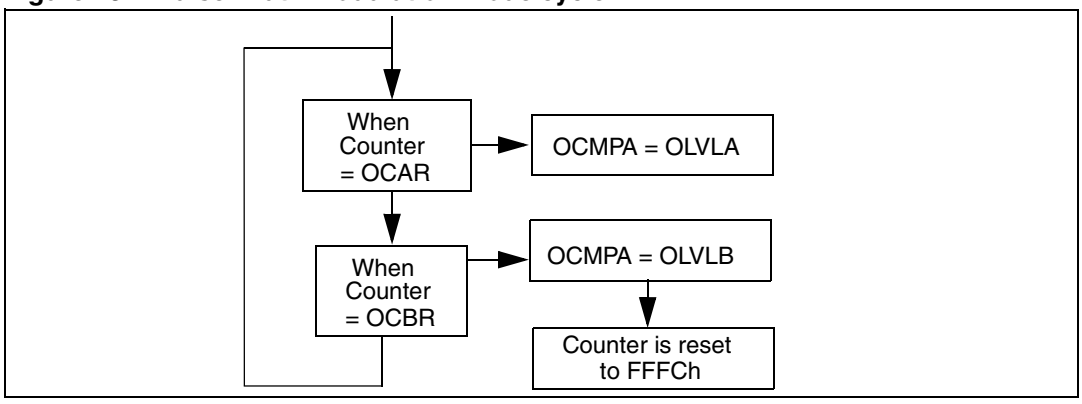
$$\text{OC}i\text{R Value} = \frac{t * f_{PCLK2}}{t_{PRESC}} - 5$$

Where:

t = Desired output compare period (seconds)
f_{PCLK2} = Internal clock frequency (Hertz)
t_{PRESC} = Timer clock prescaler (1, 2 ... , 256)

The Output Compare B event causes the counter to be initialized to FFFCh (See *Figure 44*)

Figure 43. Pulse width modulation mode cycle



Notes

- The OCFA bit cannot be set by hardware in PWM mode, but OCFB is set every time counter matches OCBR.
- The Input Capture function is available in PWM mode.
- When Counter = OCBR, then OCFB bit will be set. This can generate an interrupt if OCBIE is set. This interrupt will help any application where pulse-width or period needs to be changed interactively.
- When the Pulse Width Modulation (PWM) and One Pulse Mode (OPM) bits are both set with FOLVA = 0, the PWM mode is the only active one, otherwise the OPM mode is the only active one.
- The value loaded in OCBR **must always be greater than** that in OCAR to produce meaningful waveforms. Note that 0000h is considered to be greater than FFFCh or FFFDh or FFFEh or FFFFh.
- When OCAR > OCBR, no waveform will be generated.
- When OCBR = OCAR, a **square** waveform with 50% duty cycle will be generated as in *Figure 44*.
- When OCBR > OCAR:

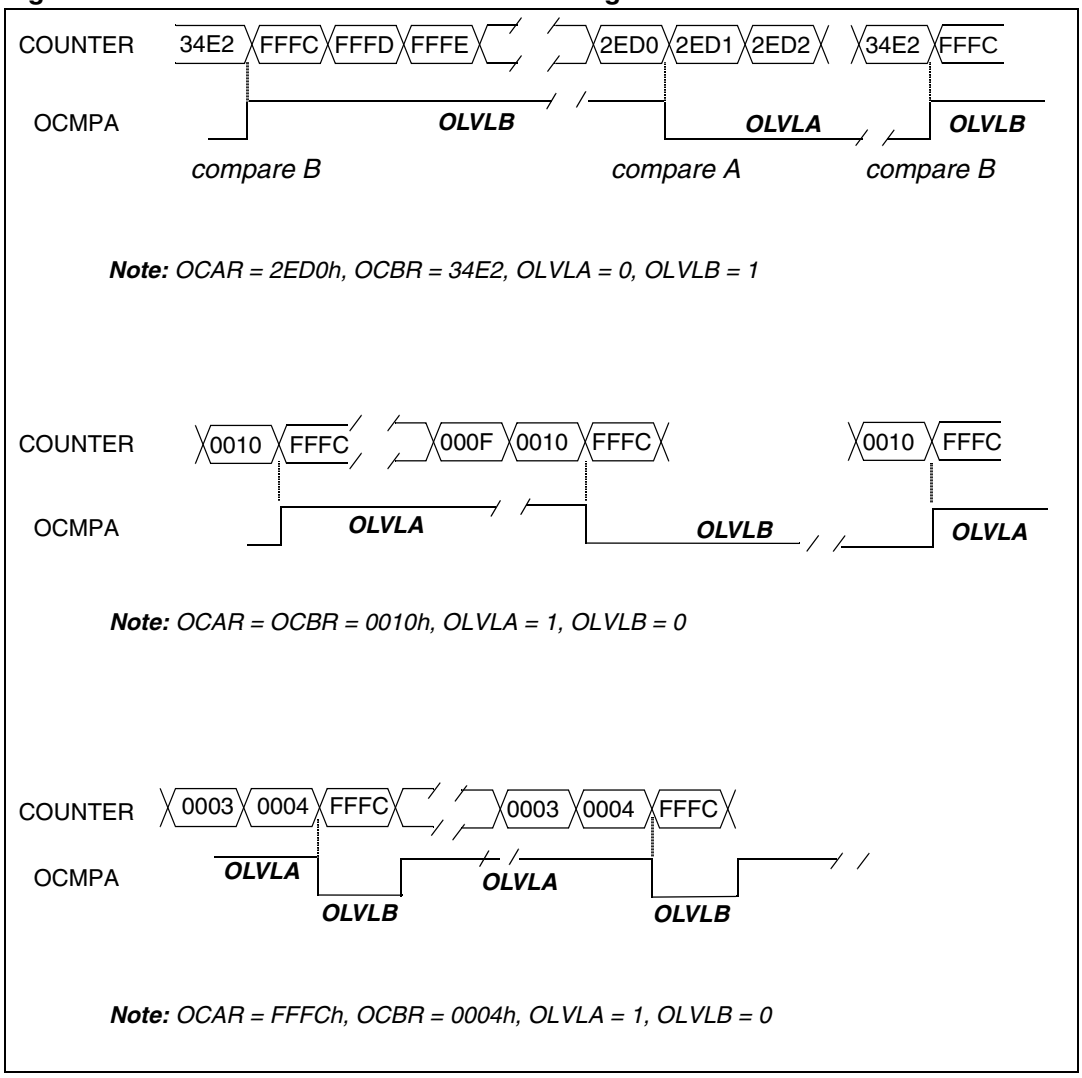
$$Period = t_{APB2} \times (PRESC + 1) \times (OCBR - FFFC + 1)$$

- When OCBR and OCAR are loaded with FFFCh (the counter reset value) then a square waveform will be generated & the counter will remain stuck at FFFCh. The period will be calculated using the following formula:

$$Period = t_{APB2} \times (PRESC + 1) \times (OCBR - FFFC + 1) \times 2$$

- When OCAR is loaded with FFFCh (the counter reset value) then the waveform will be generated as in *Figure 44*.
- When FOLVA bit is set and PWM bit is set, then PWM mode is the active one. But if FOLVB bit is set then the OLVLB bit will appear on OCMPB (when OCBE bit = 1).
- When a write is performed on CNTR register in PWM mode, then the Counter will be reset and the pulse-width/period of the waveform generated may not be as desired.

Figure 44. Pulse width modulation mode timing



7.4.9 Pulse width modulation input

The PWM Input functionality enables the measurement of the period and the pulse width of an external waveform. The initial edge is programmable.

It uses the two Input Capture registers and the Input signal of the Input Capture A module.

Procedure

The CR2 register must be programmed as needed for Interrupt generation. To use pulse width modulation mode select the following in the CR1 register:

- set the PWMI bit
- Select the first edge in IEDGA
- Select the second edge IEDGB as the negated of IEDGA
- Program the clock source and prescaler as needed
- Enable the counter setting the EN bit.

To have a coherent measure the interrupt should be linked to the Input Capture A Interrupt, reading in ICAR the period value and in ICBR the pulse width.

To obtain the time values:

$$\text{Period} = \frac{t_{PRESC} * ICAR}{f_{PCLK2}}$$

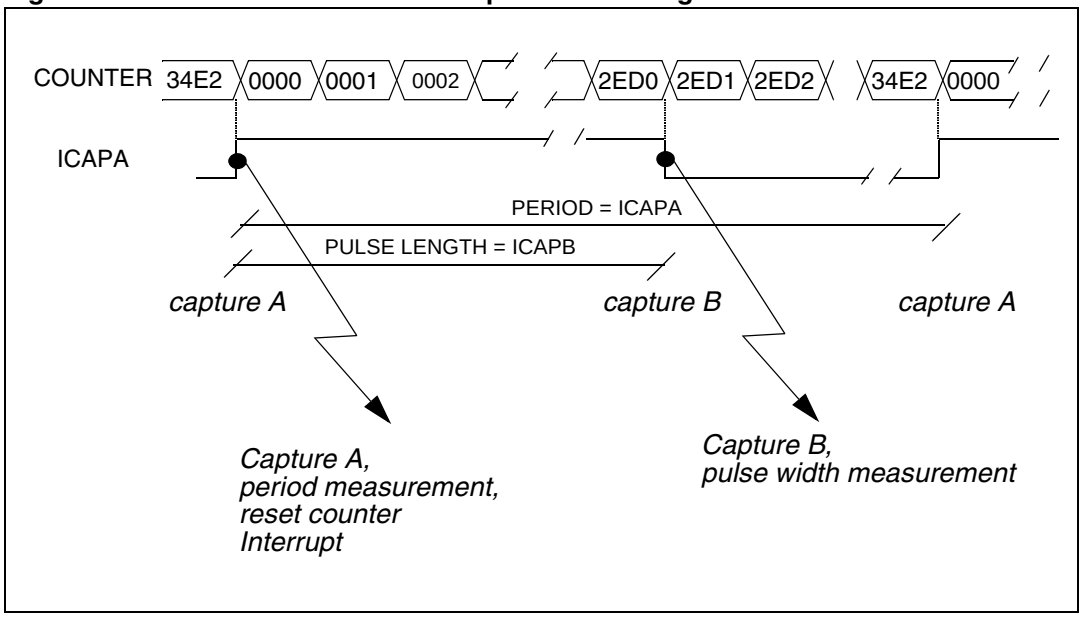
$$\text{Pulse} = \frac{t_{PRESC} * ICBR}{f_{PCLK2}}$$

Where:

f_{PCLK2} = Internal clock frequency
t_{PRESC}= Timer clock prescaler

The Input Capture A event causes the counter to be initialized to 0000h, allowing a new measure to start. The first Input Capture on ICAPA do not generate the corresponding interrupt request.

Figure 45. Pulse width modulation input mode timing



7.5 Interrupt management

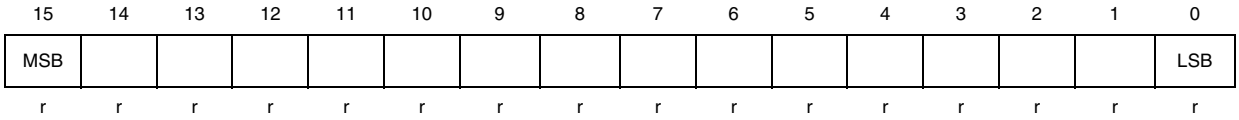
To use the interrupt features, set the OC*i*IE and/or IC*i*IE and/or TOIE bits in the CR2 register to enable the peripheral to perform interrupt requests on the desired events

7.6 Register description

Each Timer is associated with two control and one status registers, and with six pairs of data registers (16-bit values) relating to the two input captures, the two output compares, the counter. Every register can have only an access by 16 bits, that means is not possible to read or write only a byte.

7.6.1 Input capture A register (TIMn_ICAR)

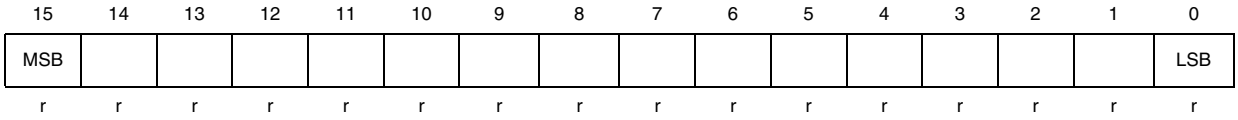
Address Offset: 00h
Reset value: xxxxh



This is a 16-bit read only register that contains the counter value transferred by the Input Capture A event.

7.6.2 Input capture B register (TIMn_ICBR)

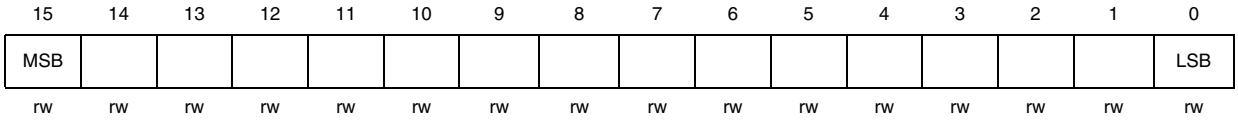
Address Offset: 04h
Reset value: xxxxh



This is a 16-bit read only register that contains the counter value transferred by the Input Capture B event.

7.6.3 Output compare A register (TIMn_OCAR)

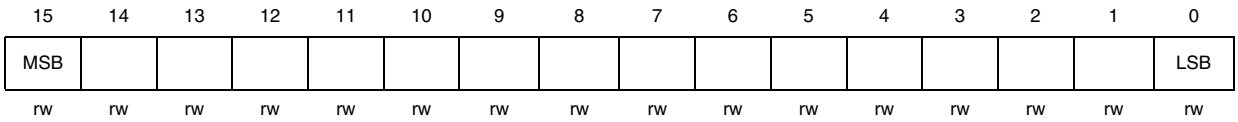
Address Offset: 08h
Reset value: 8000h



This is a 16-bit register that contains the value to be compared to the CNTR register and signalled on OCMPA output.

7.6.4 Output compare B register (TIMn_OCBR)

Address Offset: 0Ch
Reset value: 8000h



This is a 16-bit register that contains the value to be compared to the CNTR register and signalled on OCMPB output.

7.6.5 Counter register (TIMn_CNTR)

Address Offset: 10h
Reset value: FFFCh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MSB															LSB
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

This is a 16-bit register that contains the counter value. By writing in this register the counter is reset to the FFFCh value.

7.6.6 Control register 1 (TIMn_CR1)

Address Offset: 14h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EN	PWMI	Reserved		FOLVB	FOLVA	OLVLB	OLVLA	OCBE	OCAE	OPM	PWM	IEDGB	IEDGA	EXEDG	ECKEN
rw	rw	-		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 15	**EN**: *Timer Count Enable* 0: Timer counter is stopped. 1: Timer counter is enabled.
Bit 14	**PWMI**: *Pulse Width Modulation Input* 0: PWM Input is not active. 1: PWM Input is active.
Bits 13:12	Reserved. These bits must be always written to 0.
Bit 11	**FOLVB**: *Forced Output Compare B* 0: No effect. 1: Forces OLVLB to be copied to the OCMPB pin.
Bit 10	**FOLVA**: *Forced Output Compare A* 0: No effect. 1: Forces OLVLA to be copied to the OCMPA pin.
Bit 9	**OLVLB**: *Output Level B* This bit is copied to the OCMPB pin whenever a successful comparison occurs with the OCBR register and OCBE is set in the CR2 register. This value is copied to the OCMPA pin in One Pulse Mode and Pulse Width Modulation mode.
Bit 8	**OLVLA**: *Output Level A* The OLVLA bit is copied to the OCMPA pin whenever a successful comparison occurs with the OCAR register and the OCAE bit is set in the CR2 register.
Bit 7	**OCBE**: *Output Compare B Enable* 0: Output Compare B function is enabled, but the OCMPB pin is a general I/O. 1: Output Compare B function is enabled, the OCMPB pin is dedicated to the Output Compare B capability of the timer.

Bit 6	**OCAE**: *Output Compare A Enable* 0: Output Compare A function is enabled, but the OCMPA pin is a general I/O. 1: Output Compare A function is enabled, the OCMPA pin is dedicated to the Output Compare A capability of the timer.
Bit 5	**OPM**: *One Pulse Mode* 0: One Pulse Mode is not active. 1: One Pulse Mode is active, the ICAPA pin can be used to trigger one pulse on the OCMPA pin; the active transition is given by the IEDGA bit. The length of the generated pulse depends on the contents of the OCAR register
Bit 4	**PWM**: *Pulse Width Modulation* 0: PWM mode is not active. 1: PWM mode is active, the OCMPA pin outputs a programmable cyclic signal; the length of the pulse depends on the value of OCAR register; the period depends on the value of OCBR register.
Bit 3	**IEDGB**: *Input Edge B* This bit determines which type of level transition on the ICAPB pin will trigger the capture. 0: A falling edge triggers the capture. 1: A rising edge triggers the capture.
Bit 2	**IEDGA**: *Input Edge A* This bit determines which type of level transition on the ICAPA pin will trigger the capture. 0: A falling edge triggers the capture. 1: A rising edge triggers the capture.
Bit 1	**EXEDG**: *External Clock Edge* This bit determines which type of level transition on the external clock pin (or internal signal) EXTCLK will trigger the counter. 0: A falling edge triggers the counter. 1: A rising edge triggers the counter.
Bit 0	**ECKEN**: *External Clock Enable* 0: Internal clock, divided by prescaler division factor, is used to feed timer clock. 1: External source is used for timer clock.

7.6.7 Control register 2 (TIMn_CR2)

Address Offset: 18h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ICAIE	OCAIE	TOIE	ICBIE	OCBIE	reserved			CC7	CC6	CC5	CC4	CC3	CC2	CC1	CC0
rw	rw	rw	rw	rw	-			rw	rw	rw	rw	rw	rw	rw	rw

Bit 15	**ICAIE**: *Input Capture A Interrupt Enable* 0: No interrupt on input capture A. 1: Generate interrupt if ICFA flag is set.
Bit 14	**OCAIE**: *Output Compare A Interrupt Enable* 0: No interrupt on OCFA set. 1: Generate interrupt if OCFA flag is set.
Bit 13	**TOIE**: *Timer Overflow Interrupt Enable* 0: Interrupt is inhibited. 1: A timer interrupt is enabled whenever the TOF bit of the SR register is set.
Bit 12	**ICBIE**: *Input Capture B Interrupt Enable* 0: No interrupt on input capture B. 1: Generate interrupt if ICFB flag is set.
Bit 11	**OCBIE**: *Output Compare B Interrupt Enable* 0: No interrupt on OCFB set. 1: Generate interrupt if OCFB flag is set.
Bits 10:8	Reserved. These bits must be always written to 0.
Bits 7:0	**CC[7:0]**: *Prescaler division factor* This 8-bit string is the factor used by the prescaler to divide the internal clock. Timer clock will be equal to f_{PCLK2} / (CC[7:0] +1).

7.6.8 Status register (TIMn_SR)

Address Offset: 1Ch
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ICFA	OCFA	TOF	ICFB	OCFB	Reserved										
rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	-										

Bit 15	**ICFA**: *Input Capture Flag A* 0: No input capture (reset value). 1: An input capture has occurred. To clear this bit, write the SR register, with a '0' in bit 15 (and '1' in all the other bits, just to avoid an unwanted clearing of another pending bit).
Bit 14	**OCFA**: *Output Compare Flag A* 0: No match (reset value). 1: The content of the counter has matched the content of the OCAR register. This bit is not set in the PWM mode even if counter matches OCAR. To clear this bit, write the SR register, with a '0' in bit 14 (and '1' in all the other bits, just to avoid an unwanted clearing of another pending bit).
Bit 13	**TOF**: *Timer Overflow* 0: No timer overflow (reset value). 1:The counter rolled over from FFFFh to 0000h. To clear this bit, write the SR register, with a '0' in bit 13 (and '1' in all the other bits, just to avoid an unwanted clearing of another pending bit).
Bit 12	**ICFB**: *Input Capture Flag B* 0: No input capture (reset value). 1: An input capture has occurred. To clear this bit, write the SR register, with a '0' in bit 12 (and '1' in all the other bits, just to avoid an unwanted clearing of another pending bit).
Bit 11	**OCFB**: *Output Compare Flag B* 0: No match (reset value). 1: The content of the counter has matched the content of the OCBR register. It is set in PWM mode too. To clear this bit, write the SR register, with a '0' in bit 11 (and '1' in all the other bits, just to avoid an unwanted clearing of another pending bit).
Bits 10:0	Reserved. These bits must be always written to 0.

7.7 Timer register map

Table 24. Timer Register Map

Addr. Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00h	TIMn_ICAR	Input Capture A															
04h	TIMn_ICBR	Input Capture B															
08h	TIMn_OCAR	Output Compare A															
0Ch	TIMn_OCBR	Output Compare B															
10h	TIMn_CNTR	Counter Value															
14h	TIMn_CR1	EN	PWMI	Reserved		FOLVB	FOLVA	OLVLB	OLVLA	OCBE	OCAE	OPM	PWM	IEDGB	IEDGA	EXEDG	ECKEN
18h	TIMn_CR2	ICAIE	OCAIE	TOE	ICBIE	OCBIE	reserved			CC7	CC6	CC5	CC4	CC3	CC2	CC1	CC0
1Ch	TIMn_SR	ICFA	OCFA	TOF	ICFB	OCFB	reserved										

See *Table 3 on page 14* for base address

8 Controller area network (CAN)

8.1 Introduction

The C_CAN consists of the CAN Core, Message RAM, Message Handler, Control Registers and Module Interface (Refer to *Figure 46*).

The CAN Core performs communication according to the CAN protocol version 2.0 part A and B. The bit rate can be programmed to values up to 1MBit/s. For the connection to the physical layer, additional transceiver hardware is required.

For communication on a CAN network, individual Message Objects are configured. The Message Objects and Identifier Masks for acceptance filtering of received messages are stored in the Message RAM.

All functions concerning the handling of messages are implemented in the Message Handler. These functions include acceptance filtering, the transfer of messages between the CAN Core and the Message RAM, and the handling of transmission requests as well as the generation of the module interrupt.

The register set of the C_CAN can be accessed directly by the CPU through the module interface. These registers are used to control/configure the CAN Core and the Message Handler and to access the Message RAM.

8.2 Main features

- Supports CAN protocol version 2.0 part A and B
- Bit rates up to 1 MBit/s
- 32 Message Objects
- Each Message Object has its own identifier mask
- Programmable FIFO mode (concatenation of Message Objects)
- Maskable interrupt
- Disabled Automatic Re-transmission mode for Time Triggered CAN applications
- Programmable loop-back mode for self-test operation
- Two 16-bit module interfaces to the APB bus

8.3 Block diagram

The C_CAN interfaces with the AMBA APB bus. *Figure 46* shows the block diagram of the C_CAN.

CAN core

CAN Protocol Controller and Rx/Tx Shift Register.

Message RAM

Stores Message Objects and Identifier Masks.

Registers

All registers used to control and to configure the C_CAN.

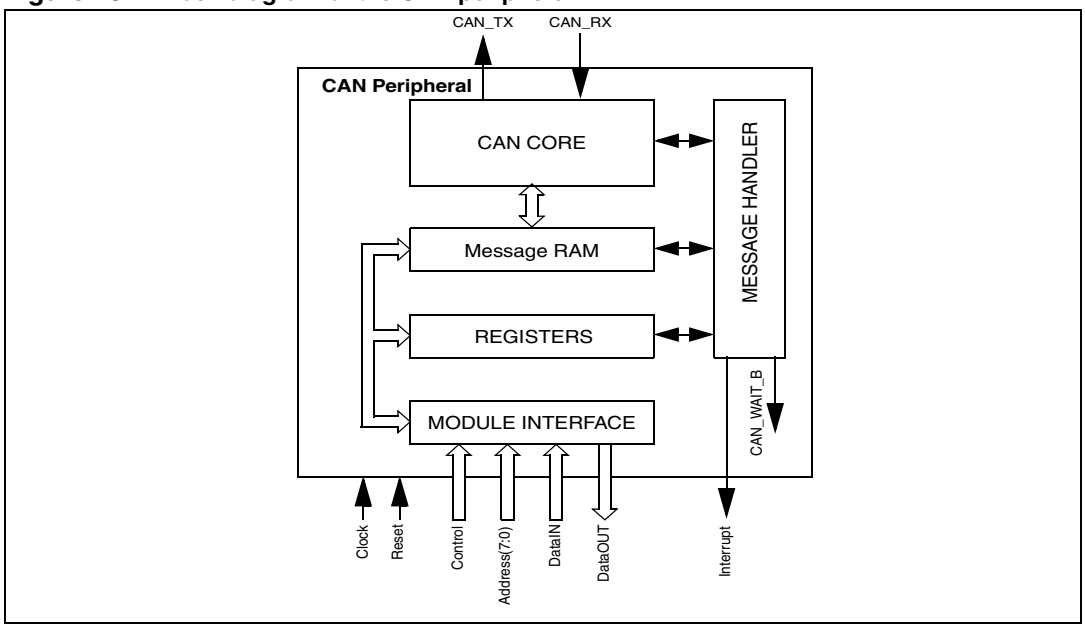
Message handler

State Machine that controls the data transfer between the Rx/Tx Shift Register of the CAN Core and the Message RAM as well as the generation of interrupts as programmed in the Control and Configuration Registers.

Module interface

The module interface provides the interface between the APB 16-bit bus and the C_CAN registers.

Figure 46. Block diagram of the CAN peripheral



8.4 Functional description

8.4.1 Software initialization

The software initialization is started by setting the Init bit in the CAN Control Register, either by a software or a hardware reset, or by going to Bus_Off state.

While the Init bit is set, all message transfers to and from the CAN bus are stopped and the status of the CAN_TX output pin is recessive (HIGH). The Error Management Logic (EML) counters are unchanged. Setting the Init bit does not change any configuration register.

To initialize the CAN Controller, software has to set up the Bit Timing Register and each Message Object. If a Message Object is not required, the corresponding MsgVal bit should be cleared. Otherwise, the entire Message Object has to be initialized.

Access to the Bit Timing Register and to the Baud Rate Prescaler (BRP) Extension Register for configuring bit timing is enabled when the Init and Configuration Change Enable *(*CCE) bits in the CAN Control Register are both set.

Resetting the Init bit (by CPU only) finishes the software initialization. Later, the Bit Stream Processor (BSP) (see *Section 8.7.10: Configuring the bit timing on page 160*) synchronizes itself to the data transfer on the CAN bus by waiting for the occurrence of a sequence of 11

consecutive recessive bits (≡ Bus Idle) before it can take part in bus activities and start the message transfer.

The initialization of the Message Objects is independent of Init and can be done on the fly, but the Message Objects should all be configured to particular identifiers or set to not valid before the BSP starts the message transfer.

To change the configuration of a Message Object during normal operation, software has to start by resetting the corresponding MsgVal bit. When the configuration is completed, MsgVal is set again.

8.4.2 CAN message transfer

Once the C_CAN is initialized and Init bit is cleared, the C_CAN Core synchronizes itself to the CAN bus and starts the message transfer.

Received messages are stored in their appropriate Message Objects if they pass the Message Handler's acceptance filtering. The whole message including all arbitration bits, DLC and eight data bytes are stored in the Message Object. If the Identifier Mask is used, the arbitration bits which are masked to "don't care" may be overwritten in the Message Object.

Software can read or write each message any time through the Interface Registers and the Message Handler guarantees data consistency in case of concurrent accesses.

Messages to be transmitted are updated by the application software. If a permanent Message Object (arbitration and control bits are set during configuration) exists for the message, only the data bytes are updated and the TxRqst bit with NewDat bit are set to start the transmission. If several transmit messages are assigned to the same Message Object (when the number of Message Objects is not sufficient), the whole Message Object has to be configured before the transmission of this message is requested.

The transmission of any number of Message Objects may be requested at the same time. Message objects are transmitted subsequently according to their internal priority. Messages may be updated or set to not valid any time, even when their requested transmission is still pending. The old data will be discarded when a message is updated before its pending transmission has started.

Depending on the configuration of the Message Object, the transmission of a message may be requested autonomously by the reception of a remote frame with a matching identifier.

8.4.3 Disabled automatic re-transmission mode

In accordance with the CAN Specification (see ISO11898, 6.3.3 Recovery Management), the C_CAN provides means for automatic re-transmission of frames that have lost arbitration or have been disturbed by errors during transmission. The frame transmission service will not be confirmed to the user before the transmission is successfully completed. This means that, by default, automatic retransmission is enabled. It can be disabled to enable the C_CAN to work within a Time Triggered CAN (TTCAN, see ISO11898-1) environment.

Disabled Automatic Retransmission mode is enabled by setting the Disable Automatic Retransmission (DAR) bit in the CAN Control Register. In this operation mode, the

programmer has to consider the different behaviour of bits TxRqst and NewDat in the Control Registers of the Message Buffers:

- When a transmission starts, bit TxRqst of the respective Message Buffer is cleared, while bit NewDat remains set.
- When the transmission completed successfully, bit NewDat is cleared.
- When a transmission fails (lost arbitration or error), bit NewDat remains set.

To restart the transmission, the CPU should set the bit TxRqst again.

8.4.4 Test mode

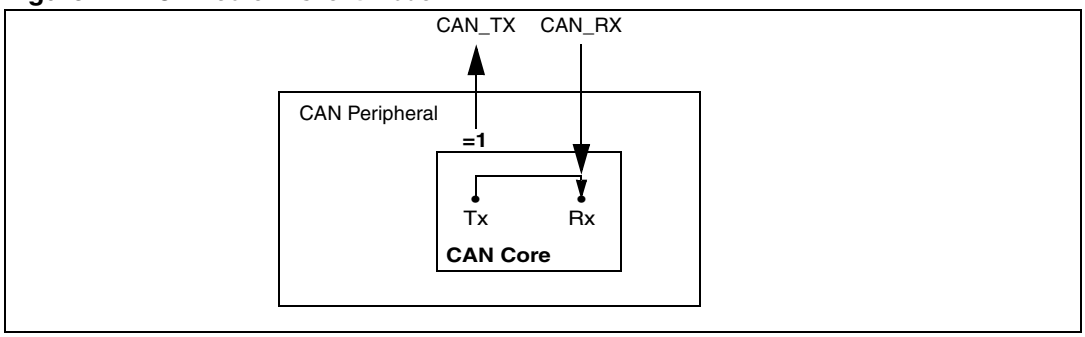
Test Mode is entered by setting the Test bit in the CAN Control Register. In Test Mode, bits Tx1, Tx0, LBack, Silent and Basic in the Test Register are writeable. Bit Rx monitors the state of the CAN_RX pin and therefore is only readable. All Test Register functions are disabled when the Test bit is cleared.

Silent mode

The CAN Core can be set in Silent Mode by programming the Silent bit in the Test Register to one.

In Silent Mode, the C_CAN is able to receive valid data frames and valid remote frames, but it sends only recessive bits on the CAN bus and it cannot start a transmission. If the CAN Core is required to send a dominant bit (ACK bit, Error Frames), the bit is rerouted internally so that the CAN Core monitors this dominant bit, although the CAN bus may remain in recessive state. The Silent Mode can be used to analyse the traffic on a CAN bus without affecting it by the transmission of *dominant* bits. *Figure 47* shows the connection of signals CAN_TX and CAN_RX to the CAN Core in Silent Mode.

Figure 47. CAN core in silent mode

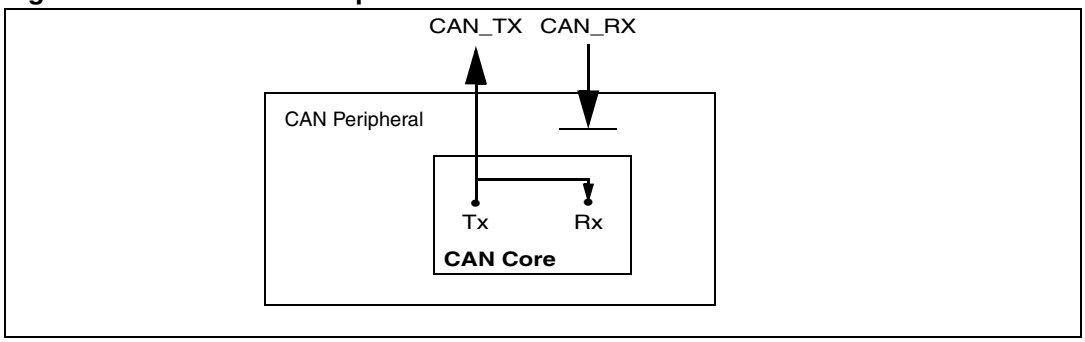


In ISO 11898-1, Silent Mode is called Bus Monitoring Mode.

Loop back mode

The CAN Core can be set in Loop Back Mode by programming the Test Register bit LBack to one. In Loop Back Mode, the CAN Core treats its own transmitted messages as received messages and stores them in a Receive Buffer (if they pass acceptance filtering). *Figure 48* shows the connection of signals, CAN_TX and CAN_RX, to the CAN Core in Loop Back Mode.

Figure 48. CAN core in loop back mode

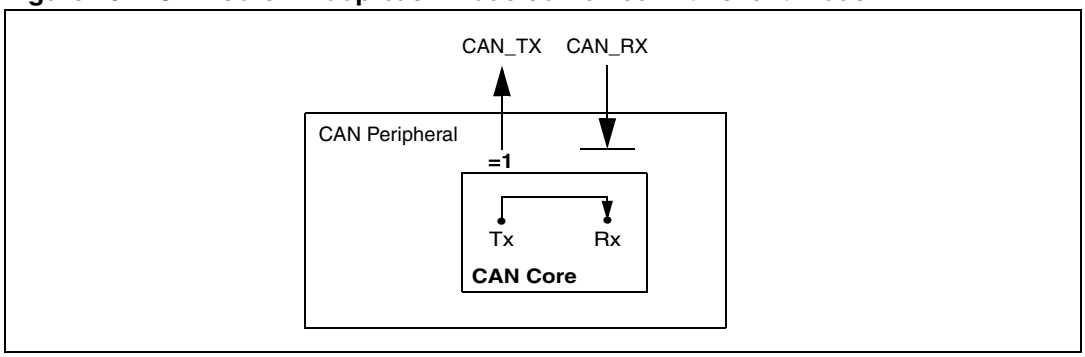


This mode is provided for self-test functions. To be independent from external stimulation, the CAN Core ignores acknowledge errors (recessive bit sampled in the acknowledge slot of a data/remote frame) in Loop Back Mode. In this mode, the CAN Core performs an internal feedback from its Tx output to its Rx input. The actual value of the CAN_RX input pin is disregarded by the CAN Core. The transmitted messages can be monitored on the CAN_TX pin.

Loop back combined with silent mode

It is also possible to combine Loop Back Mode and Silent Mode by programming bits LBack and Silent to one at the same time. This mode can be used for a “Hot Selftest”, which means that C_CAN can be tested without affecting a running CAN system connected to the CAN_TX and CAN_RX pins. In this mode, the CAN_RX pin is disconnected from the CAN Core and the CAN_TX pin is held recessive. *Figure 49* shows the connection of signals CAN_TX and CAN_RX to the CAN Core in case of the combination of Loop Back Mode with Silent Mode.

Figure 49. CAN core in loop back mode combined with silent mode



Basic mode

The CAN Core can be set in Basic Mode by programming the Test Register bit Basic to one. In this mode, the C_CAN runs without the Message RAM.

The IF1 Registers are used as Transmit Buffer. The transmission of the contents of the IF1 Registers are requested by writing the Busy bit of the IF1 Command Request Register to one. The IF1 Registers are locked while the Busy bit is set. The Busy bit indicates that the transmission is pending.

As soon the CAN bus is idle, the IF1 Registers are loaded into the shift register of the CAN Core and the transmission is started. When the transmission has been completed, the Busy bit is reset and the locked IF1 Registers are released.

A pending transmission can be aborted at any time by resetting the Busy bit in the IF1 Command Request Register while the IF1 Registers are locked. If the CPU has reset the Busy bit, a possible retransmission in case of lost arbitration or in case of an error is disabled.

The IF2 Registers are used as a Receive Buffer. After the reception of a message the contents of the shift register is stored into the IF2 Registers, without any acceptance filtering.

Additionally, the actual contents of the shift register can be monitored during the message transfer. Each time a read Message Object is initiated by writing the Busy bit of the IF2 Command Request Register to one, the contents of the shift register are stored in the IF2 Registers.

In Basic Mode, the evaluation of all Message Object related control and status bits and the control bits of the IFn Command Mask Registers are turned off. The message number of the Command request registers is not evaluated. The NewDat and MsgLst bits in the IF2 Message Control Register retain their function, DLC3-0 indicate the received DLC, and the other control bits are read as '0'.

Software control of CAN_TX pin

Four output functions are available for the CAN transmit pin, CAN_TX. In addition to its default function (serial data output), the CAN transmit pin can drive the CAN Sample Point signal to monitor CAN_Core's bit timing and it can drive constant dominant or recessive values. The latter two functions, combined with the readable CAN receive pin CAN_RX, can be used to check the physical layer of the CAN bus.

The output mode for the CAN_TX pin is selected by programming the Tx1 and Tx0 bits of the CAN Test Register.

The three test functions of the CAN_TX pin interfere with all CAN protocol functions. CAN_TX must be left in its default function when CAN message transfer or any of the test modes (Loop Back Mode, Silent Mode, or Basic Mode) are selected.

8.5 Register description

The C_CAN allocates an address space of 256 bytes. The registers are organized as 16-bit registers.

The two sets of interface registers (IF1 and IF2) control the CPU access to the Message RAM. They buffer the data to be transferred to and from the RAM, avoiding conflicts between CPU accesses and message reception/transmission.

In this section, the following abbreviations are used:

read/write (rw)	The software can read and write to these bits.
read-only (r)	The software can only read these bits.
write-only (w)	The software should only write to these bits.

The CAN registers are listed in *Table 25*.

Table 25. CAN registers

Register Name	Address Offset	Reset Value
CAN Control Register (CAN_CR)	00h	0001h
Status Register (CAN_SR)	04h	0000h
Error Counter (CAN_ERR)	08h	0000h
Bit Timing Register (CAN_BTR)	0Ch	2301h
Test Register (CAN_TESTR)	14h	0000 0000 R000 0000 b **Note:** R=current value of the RX pin
BRP Extension Register (CAN_BRPR)	18h	0000h
IFn Command Request Registers (CAN_IFn_CRR)	20h (CAN_IF1_CRR), 80h (CAN_IF2_CRR)	0001h
IFn Command Mask Registers (CAN_IFn_CMR)	24h (CAN_IF1_CMR), 84h (CAN_IF2_CMR)	0000h
IFn Mask 1 Register (CAN_IFn_M1R)	28h (CAN_IF1_M1R), 88h (CAN_IF2_M1R)	FFFFh
IFn Mask 2 Register (CAN_IFn_M2R)	2Ch (CAN_IF1_M2R), 8Ch (CAN_IF2_M2R)	FFFFh
IFn Message Arbitration 1 Register (CAN_IFn_A1R)	30h (CAN_IF1_A1R), 90h (CAN_IF2_A1R)	0000h
IFn Message Arbitration 2 Register (CAN_IFn_A2R)	34h (CAN_IF1_A2R), 94h (CAN_IF2_A2R)	0000h
IFn Message Control Registers (CAN_IFn_MCR)	38h (CAN_IF1_MCR), 98h (CAN_IF2_MCR)	0000h
IFn Data A/B Registers (CAN_IFn_DAnR and CAN_IFn_DBnR)	3Ch (CAN_IF1_DA1R), 40h (CAN_IF1_DA2R), 44h (CAN_IF1_DB1R), 48h (CAN_IF1_DB2R), 9Ch (CAN_IF2_DA1R), A0h (CAN_IF2_DA2R), A4h (CAN_IF2_DB1R), A8h (CAN_IF2_DB2R)	0000h
Interrupt Identifier Register (CAN_IDR)	10h	0000h
Transmission Request Registers 1 & 2 (CAN_TxRnR)	100h (CAN_TxR1R), 104h (CAN_TxR2R)	0000h
New Data Registers 1 & 2 (CAN_NDnR)	120h (CAN_ND1R), 124h (CAN_ND2R)	0000h
Interrupt Pending Registers 1 & 2 (CAN_IPnR)	140h (CAN_IP1R), 144h (CAN_IP2R)	0000h
Message Valid Registers 1 & 2 (CAN_MVnR)	160h (CAN_MV1R), 164h (CAN_MV2R)	0000h

8.5.1 CAN interface reset state

After the hardware reset, the C_CAN registers hold the reset values given in *Table 25* and the register descriptions below.

Additionally the *busoff* state is reset and the output CAN_TX is set to recessive (HIGH). The value 0x0001 (Init = '1') in the CAN Control Register enables the software initialization. The C_CAN does not influence the CAN bus until the CPU resets the Init bit to '0'.

The data stored in the Message RAM is not affected by a hardware reset. After powering on, the contents of the Message RAM are undefined.

8.5.2 CAN protocol related registers

These registers are related to the CAN protocol controller in the CAN Core. They control the operating modes and the configuration of the CAN bit timing and provide status information.

CAN control register (CAN_CR)

Address Offset: 00h
Reset value: 0001h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Test	CCE	DAR	res	EIE	SIE	IE	Init
-	-	-	-	-	-	-	-	rw	rw	rw	-	rw	rw	rw	rw

Bits 15:8	Reserved, , forced by hardware to 0.
Bit 7	**Test:** *Test Mode Enable* 0: Normal Operation. 1: Test Mode.
Bit 6	**CCE:** *Configuration Change Enable* 0: No write access to the Bit Timing Register. 1: Write access to the Bit Timing Register allowed (while bit Init=1).
Bit 5	**DAR:** *Disable Automatic Re-transmission* 0: Automatic Retransmission of disturbed messages enabled. 1: Automatic Retransmission disabled.
Bit 4	Reserved, forced by hardware to 0.
Bit 3	**EIE:** *Error Interrupt Enable* 0: Disabled - No Error Status Interrupt will be generated. 1: Enabled - A change in the bits BOff or EWarn in the Status Register will generate an interrupt.
Bit 2	**SIE:** *Status Change Interrupt Enable* 0: Disabled - No Status Change Interrupt will be generated. 1: Enabled - An interrupt will be generated when a message transfer is successfully completed or a CAN bus error is detected.

Bit 1	**IE:** *Module Interrupt Enable* 0: Disabled. 1: Enabled.
Bit 0	**Init:** *Initialization* 0: Normal Operation. 1: Initialization is started.

*Note: The busoff recovery sequence (see CAN Specification Rev. 2.0) cannot be shortened by setting or resetting the Init bit. If the device goes in the busoff state, it will set Init of its own accord, stopping all bus activities. Once Init has been cleared by the CPU, the device will then wait for 129 occurrences of Bus Idle (129 * 11 consecutive recessive bits) before resuming normal operations. At the end of the busoff recovery sequence, the Error Management Counters will be reset.*

During the waiting time after resetting Init, each time a sequence of 11 recessive bits has been monitored, a Bit0Error code is written to the Status Register, enabling the CPU to readily check up whether the CAN bus is stuck at dominant or continuously disturbed and to monitor the proceeding of the busoff recovery sequence.

Status register (CAN_SR)

Address Offset: 04h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								BOff	EWarn	EPass	RxOk	TxOk	LEC		
-	-	-	-	-	-	-	-	r	r	r	rw	rw	rw	rw	rw

Bit 15:8	Reserved, forced by hardware to 0.
Bit 7	**BOff:** *Busoff Status* 0: The CAN module is not in busoff state. 1: The CAN module is in busoff state.
Bit 6	**EWarn:** *Warning Status* 0: Both error counters are below the error warning limit of 96. 1: At least one of the error counters in the EML has reached the error warning limit of 96.
Bit 5	**EPass:** *Error Passive* 0: The CAN Core is error active. 1: The CAN Core is in the error passive state as defined in the CAN Specification.
Bit 4	**RxOk:** *Received a Message Successfully* 0: No message has been successfully received since this bit was last reset by the CPU. This bit is never reset by the CAN Core. 1: A message has been successfully received since this bit was last reset by the CPU (independent of the result of acceptance filtering).

Bit 3	**TxOk:** *Transmitted a Message Successfully* 0: Since this bit was reset by the CPU, no message has been successfully transmitted. This bit is never reset by the CAN Core. 1: Since this bit was last reset by the CPU, a message has been successfully (error free and acknowledged by at least one other node) transmitted.
Bits 2:0	**LEC[2:0]:** *Last Error Code (Type of the last error to occur on the CAN bus)* The LEC field holds a code, which indicates the type of the last error to occur on the CAN bus. This field will be cleared to '0' when a message has been transferred (reception or transmission) without error. The unused code '7' may be written by the CPU to check for updates. *Table 26* describes the error codes.

Table 26. Error codes

Error Code	Meaning
0	No Error
1	Stuff Error: More than 5 equal bits in a sequence have occurred in a part of a received message where this is not allowed.
2	Form Error: A fixed format part of a received frame has the wrong format.
3	AckError: The message this CAN Core transmitted was not acknowledged by another node.
4	Bit1Error: During the transmission of a message (with the exception of the arbitration field), the device wanted to send a recessive level (bit of logical value '1'), but the monitored bus value was dominant.
5	Bit0Error: During the transmission of a message (or acknowledge bit, or active error flag, or overload flag), though the device wanted to send a dominant level (data or identifier bit logical value '0'), but the monitored Bus value was recessive. During busoff recovery, this status is set each time a sequence of 11 recessive bits has been monitored. This enables the CPU to monitor the proceedings of the busoff recovery sequence (indicating the bus is not stuck at *dominant* or continuously disturbed).
6	CRCError: The CRC check sum was incorrect in the message received, the CRC received for an incoming message does not match with the calculated CRC for the received data.
7	Unused: When the LEC shows the value '7', no CAN bus event was detected since the CPU wrote this value to the LEC.

Status interrupts

A Status Interrupt is generated by bits BOff and EWarn (Error Interrupt) or by RxOk, TxOk, and LEC (Status Change Interrupt) assuming that the corresponding enable bits in the CAN Control Register are set. A change of bit EPass or a write to RxOk, TxOk, or LEC will never generate a Status Interrupt.

Reading the Status Register will clear the Status Interrupt value (8000h) in the Interrupt Register, if it is pending.

Error counter (CAN_ERR)

Address Offset: 08h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RP	REC[6:0]							TEC[7:0]							
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bit 15	**RP:** *Receive Error Passive* 0: The Receive Error Counter is below the error passive level. 1: The Receive Error Counter has reached the error passive level as defined in the CAN Specification.
Bits 14:8	**REC[6:0]:** *Receive Error Counter* Actual state of the Receive Error Counter. Values between 0 and 127.
Bits 7:0	**TEC[7:0]:** *Transmit Error Counter* Actual state of the Transmit Error Counter. Values between 0 and 255.

Bit timing register (CAN_BTR)

Address Offset: 0Ch

Reset value: 2301h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res	TSeg2			TSeg1				SJW		BRP					
-	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 15	Reserved, forced by hardware to 0.
Bits 14:12	**TSeg2:** *Time segment after sample point* 0x0-0x7: Valid values for TSeg2 are [0 … 7]. The actual interpretation by the hardware of this value is such that one more than the value programmed here is used.
Bits 11:8	**TSeg1:** *Time segment before the sample point minus Sync_Seg* 0x01-0x0F: valid values for TSeg1 are [1 … 15]. The actual interpretation by the hardware of this value is such that one more than the value programmed is used.
Bits 7:6	**SJW:** (*Re)Synchronization Jump Width* 0x0-0x3: Valid programmed values are [0 … 3]. The actual interpretation by the hardware of this value is such that one more than the value programmed here is used.
Bits 5:0	**BRP:** *Baud Rate Prescaler* 0x01-0x3F: The value by which the oscillator frequency is divided for generating the bit time quanta. The bit time is built up from a multiple of this quanta. Valid values for the Baud Rate Prescaler are [0 … 63]. The actual interpretation by the hardware of this value is such that one more than the value programmed here is used.

Note: With a module clock APB_CLK of 8 MHz, the reset value of 0x2301 configures the C_CAN for a bit rate of 500 kBit/s. The registers are only writable if bits CCE and Init in the CAN Control Register are set.

Test register (CAN_TESTR)

Address Offset: 14h
Reset value: 0000 0000 R000 0000 b (R:current value of RX pin)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								Rx	Tx[1:0]		LBack	Silent	Basic	Res	
-	-	-	-	-	-	-	-	r	rw	rw	rw	rw	rw	-	-

Bits 15:8	Reserved, forced by hardware to 0.
Bit 7	**Rx:** *Current value of* CAN_RX Pin 0: The CAN bus is dominant (CAN_RX = '0'). 1: The CAN bus is recessive (CAN_RX = '1').
Bit 6:5	**Tx[1:0]:** CAN_TX pin control 00: Reset value, CAN_TX is controlled by the CAN Core 01: Sample Point can be monitored at CAN_TX pin 10: CAN_TX pin drives a dominant ('0') value. 11: CAN_TX pin drives a recessive ('1') value.
Bit 4	**LBack:** *Loop Back Mode* 0: Loop Back Mode is disabled. 1: Loop Back Mode is enabled.
Bit 3	**Silent:** *Silent Mode* 0: Normal operation. 1: The module is in Silent Mode.
Bit 2	**Basic:** *Basic Mode* 0: Basic Mode disabled. 1: IF1 Registers used as Tx Buffer, IF2 Registers used as Rx Buffer.
Bits 1:0	Reserved, forced by hardware to 0.

Write access to the Test Register is enabled by setting the Test bit in the CAN Control Register. The different test functions may be combined, but Tx1-0 ≠ "00" disturbs message transfer.

BRP extension register (CAN_BRPR)

Address Offset: 18h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved												BRPE			
-	-	-	-	-	-	-	-	-	-	-	-	rw	rw	rw	rw

Bits 15:4	Reserved, forced by hardware to 0.
Bits 3:0	**BRPE:** *Baud Rate Prescaler Extension* 0x00-0x0F: By programming BRPE, the Baud Rate Prescaler can be extended to values up to 1023. The actual interpretation by the hardware is that one more than the value programmed by BRPE (MSBs) and BRP (LSBs) is used.

8.5.3 Message interface register sets

There are two sets of Interface Registers, which are used to control the CPU access to the Message RAM. The Interface Registers avoid conflict between the CPU access to the Message RAM and CAN message reception and transmission by buffering the data to be transferred. A complete Message Object (see *Message object in the message memory*) or parts of the Message Object may be transferred between the Message RAM and the IF*n* Message Buffer registers (see *IFn message buffer registers*) in one single transfer.

The function of the two interface register sets is identical except for the Basic test mode. They can be used the way one set of registers is used for data transfer to the Message RAM while the other set of registers is used for the data transfer from the Message RAM, allowing both processes to be interrupted by each other. *Table 27: IF1 and IF2 message interface register set on page 140* provides an overview of the two Interface Register sets.

Each set of Interface Registers consists of Message Buffer Registers controlled by their own Command Registers. The Command Mask Register specifies the direction of the data transfer and which parts of a Message Object will be transferred. The Command Request Register is used to select a Message Object in the Message RAM as target or source for the transfer and to start the action specified in the Command Mask Register.

Table 27. IF1 and IF2 message interface register set

Address	IF1 Register Set	Address	IF2 Register Set
CAN Base + 0x20	IF1 Command Request	CAN Base + 0x80	IF2 Command Request
CAN Base + 0x24	IF1 Command Mask	CAN Base + 0x84	IF2 Command Mask
CAN Base + 0x28	IF1 Mask 1	CAN Base + 0x88	IF2 Mask 1
CAN Base + 0x2C	IF1 Mask 2	CAN Base + 0x8C	IF2 Mask 2
CAN Base + 0x30	IF1 Arbitration 1	CAN Base + 0x90	IF2 Arbitration 1
CAN Base + 0x34	IF1 Arbitration 2	CAN Base + 0x94	IF2 Arbitration 2
CAN Base + 0x38	IF1 Message Control	CAN Base + 0x98	IF2 Message Control
CAN Base + 0x3C	IF1 Data A 1	CAN Base + 0x9C	IF2 Data A 1
CAN Base + 0x40	IF1 Data A 2	CAN Base + 0xA0	IF2 Data A 2
CAN Base + 0x44	IF1 Data B 1	CAN Base + 0xA4	IF2 Data B 1
CAN Base + 0x48	IF1 Data B 2	CAN Base + 0xA8	IF2 Data B 2

IF*n* command request registers (CAN_IF*n*_CRR)

Address offset: 20h (CAN_IF1_CRR), 80h (CAN_IF2_CRR)
Reset Value: 0001h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Busy	Reserved									Message Number					
r	-	-	-	-	-	-	-	-	-	rw	rw	rw	rw	rw	rw

A message transfer is started as soon as the application software has written the message number to the Command Request Register. With this write operation, the Busy bit is automatically set to notify the CPU that a transfer is in progress. After a waiting time of 3 to 6 APB_CLK periods, the transfer between the Interface Register and the Message RAM is completed. The Busy bit is cleared.

Bit 15	**Busy:** *Busy Flag* 0: Read/write action has finished. 1: Writing to the IF*n* Command Request Register is in progress. This bit can only be read by the software.
Bits 14:6	Reserved, forced by hardware to 0.
Bits 5:0	Message Number: 0x01-0x20: Valid Message Number, the Message Object in the Message RAM is selected for data transfer. 0x00: Not a valid Message Number, interpreted as 0x20. 0x21-0x3F: Not a valid Message Number, interpreted as *0x01-0x1F*.

Note: When a Message Number that is not valid is written into the Command Request Register, the Message Number will be transformed into a valid value and that Message Object will be transferred.

IF*n* command mask registers (CAN_IF*n*_CMR)

Address offset: 24h (CAN_IF1_CMR), 84h (CAN_IF2_CMR)
Reset Value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								WR/RD	Mask	Arb	Control	ClrIntPnd	TxRqst/NewDat	Data A	Data B
-	-	-	-	-	-	-	-	rw	rw	rw	rw	rw	rw	rw	rw

The control bits of the IF*n* Command Mask Register specify the transfer direction and select which of the IF*n* Message Buffer Registers are source or target of the data transfer.

Bits 15:8	Reserved, forced by hardware to 0.
Bit 7	**WR/RD:** *Write / Read* 0: Read: Transfer data from the Message Object addressed by the Command Request Register into the selected Message Buffer Registers. 1: Write: Transfer data from the selected Message Buffer Registers to the Message Object addressed by the Command Request Register.

Bits 6:0	These bits of IFn Command Mask Register have different functions depending on the transfer direction: Direction = Write Bit 6 = Mask Access Mask Bits 0: Mask bits unchanged. 1: transfer Identifier Mask + MDir + MXtd to Message Object. Bit 5 = Arb Access Arbitration Bits 0: Arbitration bits unchanged. 1: Transfer Identifier + Dir + Xtd + MsgVal to Message Object. Bit 4 = Control Access Control Bits 0: Control Bits unchanged. 1: Transfer Control Bits to Message Object. Bit 3 = ClrIntPnd Clear Interrupt Pending Bit When writing to a Message Object, this bit is ignored. Bit 2 = TxRqst/NewDat Access Transmission Request Bit 0: TxRqst bit unchanged. 1: Set TxRqst bit. If a transmission is requested by programming bit TxRqst/NewDat in the IFn Command Mask Register, bit TxRqst in the IFn Message Control Register will be ignored. Bit 1 = Data A Access Data Bytes 3:0 0: Data Bytes 3:0 unchanged. 1: Transfer Data Bytes 3:0 to Message Object. Bit 0 = Data B Access Data Bytes 7:4 0: Data Bytes 7:4 unchanged. 1: Transfer Data Bytes 7:4 to Message Object.
Bits 6:0	Direction = Read Bit 6 = Mask: *Access Mask Bits* 0: Mask bits unchanged. 1: Transfer Identifier Mask + MDir + MXtd to IF*n* Message Buffer Register. Bit 5 = Arb: *Access Arbitration Bits* 0: Arbitration bits unchanged. 1: Transfer Identifier + Dir + Xtd + MsgVal to IF*n* Message Buffer Register. Bit 4 = Control: *Access Control Bits* 0: Control Bits unchanged. 1: Transfer Control Bits to IF*n* Message Buffer Register. Bit 3 = ClrIntPnd: *Clear Interrupt Pending Bit* 0: IntPnd bit remains unchanged. 1: Clear IntPnd bit in the Message Object. Bit 2 = TxRqst/NewDat: *Access Transmission Request Bit* 0: NewDat bit remains unchanged. 1: Clear NewDat bit in the Message Object. A read access to a Message Object can be combined with the reset of the control bits IntPnd and NewDat. The values of these bits transferred to the IFn Message Control Register always reflect the status before resetting these bits. Bit 1 = Data A Access Data Bytes 3:0 0: Data Bytes 3:0 unchanged. 1: Transfer Data Bytes 3:0 to IF*n* Message Buffer Register. Bit 0 = Data B Access Data Bytes 7:4 0: Data Bytes 7:4 unchanged. 1: Transfer Data Bytes 7:4 to IF*n* Message Buffer Register.

IF*n* message buffer registers

The bits of the Message Buffer registers mirror the Message Objects in the Message RAM. The function of the Message Objects bits is described in *Message object in the message memory on page 144*.

IF*n* mask 1 register (CAN_IF*n*_M1R)

Address offset: 28h (CAN_IF1_M1R), 88h (CAN_IF2_M1R)
Reset Value: FFFFh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Msk[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The function of the Msk bits is described in *Message object in the message memory on page 144*.

IF*n* mask 2 register (CAN_IF*n*_M2R)

Address offset: 2Ch (CAN_IF1_M2R), 8Ch (CAN_IF2_M2R)
Reset Value: FFFFh

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MXtd	MDir	Res	Msk[28:16]												
rw	rw	r	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The function of the Message Objects bits is described in the *Message object in the message memory on page 144*.

IF*n* message arbitration 1 register (CAN_IF*n*_A1R)

Address offset: 30h (CAN_IF1_A1R), 90h (CAN_IF2_A1R)
Reset Value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ID[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The function of the Message Objects bits is described in the *Message object in the message memory on page 144*.

IF*n* message arbitration 2 register (CAN_IF*n*_A2R)

Address offset: 34h (CAN_IF1_A2R), 94h (CAN_IF2_A2R)
Reset Value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MsgVal	Xtd	Dir	ID[28:16]												
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The function of the Message Objects bits is described in the *Message object in the message memory on page 144*.

IF*n* message control registers (CAN_IF*n*_MCR)

Address offset: 38h (CAN_IF1_MCR), 98h (CAN_IF2_MCR)
Reset Value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NewDat	MsgLst	IntPnd	UMask	TxIE	RxIE	RmtEn	TxRqst	EoB	Reserved			DLC[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	-	-	-	rw	rw	rw	rw

The function of the Message Objects bits is described in the *Message object in the message memory on page 144*.

IF*n* data A/B registers (CAN_IF*n*_DA*n*R and CAN_IF*n*_DB*n*R)

The data bytes of CAN messages are stored in the IF*n* Message Buffer Registers in the following order:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IF1 Message Data A1 (address 0x3C)	Data(1)								Data(0)							
IF1 Message Data A2 (address 0x40)	Data(3)								Data(2)							
IF1 Message Data B1 (address 0x44)	Data(5)								Data(4)							
IF1 Message Data B2 (address 0x48)	Data(7)								Data(6)							
IF2 Message Data A1 (address 0x9C)	Data(1)								Data(0)							
IF2 Message Data A2 (address 0xA0)	Data(3)								Data(2)							
IF2 Message Data B1 (address 0xA4)	Data(5)								Data(4)							
IF2 Message Data B2 (address 0xA8)	Data(7)								Data(6)							
	rw								rw							

In a CAN Data Frame, Data(0) is the first, Data(7) is the last byte to be transmitted or received. In CAN's serial bit stream, the MSB of each byte will be transmitted first.

Message object in the message memory

There are 32 Message Objects in the Message RAM. To avoid conflicts between CPU access to the Message RAM and CAN message reception and transmission, the CPU

cannot directly access the Message Objects, these accesses are handled through the IF*n* Interface Registers.

Table 28 provides an overview of the structures of a Message Object

Table 28. Structure of a message object in the message memory

Message Object												
UMask	Msk 28-0	MXtd	MDir	EoB	NewDat		MsgLst	RxIE	TxIE	Int Pnd	RmtEn	TxRqst
MsgVal	ID28-0	Xtd	Dir	DLC 3-0	Data 0	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7

The Arbitration Registers ID28-0, Xtd, and Dir are used to define the identifier and type of outgoing messages and are used (together with the mask registers Msk28-0, MXtd, and MDir) for acceptance filtering of incoming messages. A received message is stored in the valid Message Object with matching identifier and direction set to receive (Data Frame) or transmit (Remote Frame). Extended frames can be stored only in Message Objects with Xtd set, standard frames in Message Objects with Xtd clear. If a received message (Data Frame or Remote Frame) matches more than one valid Message Object, it is stored into that with the lowest message number. For details see *Acceptance filtering of received messages*.

MsgVal	Message Valid 1: The Message Object is configured and should be considered by the Message Handler. 0: The Message Object is ignored by the Message Handler. Note: The application software must reset the MsgVal bit of all unused Messages Objects during the initialization before it resets bit Init in the CAN Control Register. This bit must also be reset before the identifier Id28-0, the control bits Xtd, Dir, or the Data Length Code DLC3-0 are modified, or if the Messages Object is no longer required.
UMask	Use Acceptance Mask 1: Use Mask (Msk28-0, MXtd, and MDir) for acceptance filtering. 0: Mask ignored. Note: If the UMask bit is set to one, the Message Object's mask bits have to be programmed during initialization of the Message Object before MsgVal is set to one.
ID28-0	Message Identifier ID28 - ID0, 29-bit Identifier ("Extended Frame") ID28 - ID18, 11-bit Identifier ("Standard Frame")
Msk28-0	Identifier Mask 1: The corresponding identifier bit is used for acceptance filtering. 0: The corresponding bit in the identifier of the message object cannot inhibit the match in the acceptance filtering.
Xtd	Extended Identifier 1: The 29-bit ("extended") Identifier will be used for this Message Object. 0: The 11-bit ("standard") Identifier will be used for this Message Object.

MXtd	Mask Extended Identifier 1: The extended identifier bit (IDE) is used for acceptance filtering. 0: The extended identifier bit (IDE) has no effect on the acceptance filtering. Note: When 11-bit (“standard”) Identifiers are used for a Message Object, the identifiers of received Data Frames are written into bits ID28 to ID18. For acceptance filtering, only these bits together with mask bits Msk28 to Msk18 are considered.
Dir	Message Direction 1: Direction = transmit: On TxRqst, the respective Message Object is transmitted as a Data Frame. On reception of a Remote Frame with matching identifier, the TxRqst bit of this Message Object is set (if RmtEn = one). 0: Direction = *receive*: On TxRqst, a Remote Frame with the identifier of this Message Object is transmitted. On reception of a Data Frame with matching identifier, that message is stored in this Message Object.
MDir	Mask Message Direction 1: The message direction bit (Dir) is used for acceptance filtering. 0: The message direction bit (Dir) has no effect on the acceptance filtering.
EoB	End of Buffer 1: Single Message Object or last Message Object of a FIFO Buffer. 0: Message Object belongs to a FIFO Buffer and is not the last Message Object of that FIFO Buffer. Note: This bit is used to concatenate two or more Message Objects (up to 32) to build a FIFO Buffer. For single Message Objects (not belonging to a FIFO Buffer), this bit must always be set to one. For details on the concatenation of Message Objects see *Section 8.7.7: Configuring a FIFO buffer*.
NewDat	New Data 1: The Message Handler or the application software has written new data into the data portion of this Message Object. 0: No new data has been written into the data portion of this Message Object by the Message Handler since last time this flag was cleared by the application software.
MsgLst	Message Lost (only valid for Message Objects with direction = *receive*) 1: The Message Handler stored a new message into this object when NewDat was still set, the CPU has lost a message. 0: No message lost since last time this bit was reset by the CPU.
RxIE	Receive Interrupt Enable 1: IntPnd will be set after a successful reception of a frame. 0: IntPnd will be left unchanged after a successful reception of a frame.
TxIE	Transmit Interrupt Enable 1: IntPnd will be set after a successful transmission of a frame. 0: IntPnd will be left unchanged after the successful transmission of a frame.
IntPnd	Interrupt Pending 1: This message object is the source of an interrupt. The Interrupt Identifier in the Interrupt Register will point to this message object if there is no other interrupt source with higher priority. 0:This message object is not the source of an interrupt.

RmtEn	Remote Enable 1: At the reception of a Remote Frame, TxRqst is set. 0: At the reception of a Remote Frame, TxRqst is left unchanged.
TxRqst	Transmit Request 1: The transmission of this Message Object is requested and is not yet done. 0: This Message Object is not waiting for transmission.
DLC3-0	Data Length Code *0-8*: Data Frame has 0-8 data bytes. *9-15*: Data Frame has 8 data bytes Note: The Data Length Code of a Message Object must be defined the same as in all the corresponding objects with the same identifier at other nodes. When the Message Handler stores a data frame, it will write the DLC to the value given by the received message. **Data 0:** 1st data byte of a CAN Data Frame **Data 1**: 2nd data byte of a CAN Data Frame **Data 2**: 3rd data byte of a CAN Data Frame **Data 3**: 4th data byte of a CAN Data Frame **Data 4**: 5th data byte of a CAN Data Frame **Data 5**: 6th data byte of a CAN Data Frame **Data 6**: 7th data byte of a CAN Data Frame **Data 7 :** 8th data byte of a CAN Data Frame Note: The Data 0 Byte is the first data byte shifted into the shift register of the CAN Core during a reception while the Data 7 byte is the last. When the Message Handler stores a Data Frame, it will write all the eight data bytes into a Message Object. If the Data Length Code is less than 8, the remaining bytes of the Message Object will be overwritten by unspecified values.

8.5.4 Message handler registers

All Message Handler registers are read-only. Their contents, TxRqst, NewDat, IntPnd, and MsgVal bits of each Message Object and the Interrupt Identifier is status information provided by the Message Handler FSM.

Interrupt identifier register (CAN_IDR)

Address Offset: 10h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IntId[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 15:0	**IntId[15:0]:** Interrupt Identifier (*Table 29* indicates the source of the interrupt) If several interrupts are pending, the CAN Interrupt Register will point to the pending interrupt with the highest priority, disregarding their chronological order. An interrupt remains pending until the application software has cleared it. If IntId is different from 0x0000 and IE is set, the IRQ interrupt signal to the EIC is active. The interrupt remains active until IntId is back to value 0x0000 (the cause of the interrupt is reset) or until IE is reset. The Status Interrupt has the highest priority. Among the message interrupts, the Message Object' s interrupt priority decreases with increasing message number. A message interrupt is cleared by clearing the Message Object's IntPnd bit. The Status Interrupt is cleared by reading the Status Register.

Table 29. Source of interrupts

Interrupt Identifier	Cause of the Inerrupt
0x0000	No Interrupt is Pending
0x0001-0x0020	Number of Message Object which caused the interrupt.
0x0021-0x7FFF	unused
0x8000	Status Interrupt
0x8001-0xFFFF	unused

Transmission request registers 1 & 2 (CAN_TxR*n*R)

Address Offset: 100h (CAN_TxR1R), 104h (CAN_TxR2R)
Reset Value: 0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TxRqst[32:17]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TxRqst[16:1]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

These registers hold the TxRqst bits of the 32 Message Objects. By reading the TxRqst bits, the CPU can check which Message Object in a Transmission Request is pending. The TxRqst bit of a specific Message Object can be set/reset by the application software through the IFn Message Interface Registers or by the Message Handler after reception of a Remote Frame or after a successful transmission.

Bits 31:16	**TxRqst[32:17]:** *Transmission Request Bits (of all Message Objects)* 0: This Message Object is not waiting for transmission. 1: The transmission of this Message Object is requested and is not yet done. These bits are read only.
Bits 15:0	**TxRqst1[6:1]:** *Transmission Request Bits (of all Message Objects)* 0: This Message Object is not waiting for transmission. 1: The transmission of this Message Object is requested and is not yet done. These bits are read only.

New data registers 1 & 2 (CAN_ND*n*R)

Address Offset: 120h (CAN_ND1R), 124h (CAN_ND2R)
Reset Value: 0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
NewDat[32:17]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NewDat[16:1]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

These registers hold the NewDat bits of the 32 Message Objects. By reading out the NewDat bits, the CPU can check for which Message Object the data portion was updated. The NewDat bit of a specific Message Object can be set/reset by the CPU through the IFn

Message Interface Registers or by the Message Handler after reception of a Data Frame or after a successful transmission.

Bits 31:16	**NewDat[32:17]:** *New Data Bits (of all Message Objects)* 0: No new data has been written into the data portion of this Message Object by the Message Handler since the last time this flag was cleared by the application software. 1: The Message Handler or the application software has written new data into the data portion of this Message Object.
Bits 15:0	**NewDat[16:1]:** *New Data Bits (of all Message Objects)* 0: No new data has been written into the data portion of this Message Object by the Message Handler since the last time this flag was cleared by the application software. 1: The Message Handler or the application software has written new data into the data portion of this Message Object.

Interrupt pending registers 1 & 2 (CAN_IP*n*R)

Address Offset: 140h (CAN_IP1R), 144h (CAN_IP2R)
Reset Value: 0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IntPnd[32:17]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IntPnd[16:1]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

These registers contain the IntPnd bits of the 32 Message Objects. By reading the IntPnd bits, the CPU can check for which Message Object an interrupt is pending. The IntPnd bit of a specific Message Object can be set/reset by the application software through the IFn Message Interface Registers or by the Message Handler after reception or after a successful transmission of a frame. This will also affect the value of IntId in the Interrupt Register.

Bits 31:16	**IntPnd[32:17]:** *Interrupt Pending Bits (of all Message Objects)* 0: This message object is not the source of an interrupt. 1: This message object is the source of an interrupt.
Bits 15:0	**IntPnd[16:1]:** *Interrupt Pending Bits (of all Message Objects)* 0: This message object is not the source of an interrupt. 1: This message object is the source of an interrupt.

Message valid registers 1 & 2 (CAN_MV*n*R)

Address Offset: 160h (CAN_MV1R), 164h (CAN_MV2R)
Reset Value: 0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MsgVal[32:17]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MsgVal[16:1]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

These registers hold the MsgVal bits of the 32 Message Objects. By reading the MsgVal bits, the application software can check which Message Object is valid. The MsgVal bit of a specific Message Object can be set/reset by the application software via the IFn Message Interface Registers.

Bits 31:16	**MsgVal[32:17]:** *Message Valid Bits (of all Message Objects)* 0: This Message Object is ignored by the Message Handler. 1: This Message Object is configured and should be considered by the Message Handler.
Bits 15:0	**MsgVal[16:1]:** *Message Valid Bits (of all Message Objects)* 0: This Message Object is ignored by the Message Handler. 1: This Message Object is configured and should be considered by the Message Handler.

8.6 Register map

Table 30. CAN register map

Addr offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00h	CAN_CR	Reserved								Test	CCE	DAR	res	EIE	SIE	IE	Init
04h	CAN_SR	Reserved								BOff	EWarn	EPass	RxOk	TxOk	LEC		
08h	CAN_ERR	RP	REC6-0							TEC7-0							
0Ch	CAN_BTR	res	TSeg2			TSeg1				SJW		BRP					
10h	CAN_IDR	IntId15-8								IntId7-0							
14h	CAN_TESTR	Reserved								Rx	Tx1	Tx0	LBack	Silent	Basic	Reserved	
18h	CAN_BRPR	Reserved												BRPE			
20h	CAN_IF1_CRR	Busy	Reserved									Message Number					
24h	CAN_IF1_CMR	Reserved								WR/RD	Mask	Arb	Control	ClrIntPnd	TxRqst/ NewDat	Data A	Data B

Table 30. CAN register map

Addr offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
28h	CAN_IF1_M1R	Msk15-0															
2Ch	CAN_IF1_M2R	MXtd	MDir	res	Msk28-16												
30h	CAN_IF1_A1R	ID15-0															
34h	CAN_IF1_A2R	Msg Val	Xtd	Dir	ID28-16												
38h	CAN_IF1_MCR	NewDat	MsgLst	IntPnd	UMask	TxIE	RxIE	RmtEn	TxRqst	EoB	Reserved			DLC3-0			
3Ch	CAN_IF1_DA1R	Data(1)								Data(0)							
40h	CAN_IF1_DA2R	Data(3)								Data(2)							
44h	CAN_IF1_DB1R	Data(5)								Data(4)							
48h	CAN_IF1_DB2R	Data(7)								Data(6)							
80h	CAN_IF2_CRR	Busy	Reserved									Message Number					
84h	CAN_IF2_CMR	Reserved								WR/RD	Mask	Arb	Control	ClrIntPnd	TxRqst/ NewDat	Data A	Data B
88h	CAN_IF2_M1R	Msk15-0															
8Ch	CAN_IF2_M2R	MXtd	MDir	res	Msk28-16												
90h	CAN_IF2_A1R	ID15-0															
94h	CAN_IF2_A2R	Msg Val	Xtd	Dir	ID28-16												
98h	CAN_IF2_MCR	NewDat	MsgLst	IntPnd	UMask	TxIE	RxIE	RmtEn	TxRqst	EoB	Reserved			DLC3-0			
9Ch	CAN_IF2_DA1R	Data(1)								Data(0)							
A0h	CAN_IF2_DA2R	Data(3)								Data(2)							
A4h	CAN_IF2_DB1R	Data(5)								Data(4)							
A8h	CAN_IF2_DB2R	Data(7)								Data(6)							
100h	CAN_TxR1R	TxRqst16-1															
104h	CAN_TxR2R	TxRqst32-17															
120h	CAN_ND1R	NewDat16-1															
124h	CAN_ND2R	NewDat32-17															
140h	CAN_IP1R	IntPnd16-1															
144h	CAN_IP2R	IntPnd32-17															
160h	CAN_MV1R	MsgVal16-1															

Table 30. CAN register map

Addr offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
164h	CAN_MV2R	MsgVal32-17															

Note: Reserved bits are read as 0' except for IFn Mask 2 Register where they are read as '1'.

Refer to *Table 2* for the register base addresses.

8.7 CAN communications

8.7.1 Managing message objects

The configuration of the Message Objects in the Message RAM (with the exception of the bits MsgVal, NewDat, IntPnd, and TxRqst) will not be affected by resetting the chip. All the Message Objects must be initialized by the application software or they must be "not valid" (MsgVal = '0') and the bit timing must be configured before the application software clears the Init bit in the CAN Control Register.

The configuration of a Message Object is done by programming Mask, Arbitration, Control and Data fields of one of the two interface registers to the desired values. By writing to the corresponding IFn Command Request Register, the IFn Message Buffer Registers are loaded into the addressed Message Object in the Message RAM.

When the Init bit in the CAN Control Register is cleared, the CAN Protocol Controller state machine of the CAN_Core and state machine of the Message Handler control the internal data flow of the C_CAN. Received messages that pass the acceptance filtering are stored in the Message RAM, messages with pending transmission request are loaded into the CAN_Core's Shift Register and are transmitted through the CAN bus.

The application software reads received messages and updates messages to be transmitted through the IF*n* Interface Registers. Depending on the configuration, the CPU is interrupted on certain CAN message and CAN error events.

8.7.2 Message handler state machine

The Message Handler controls the data transfer between the Rx/Tx Shift Register of the CAN Core, the Message RAM and the IF*n* Registers.

The Message Handler FSM controls the following functions:

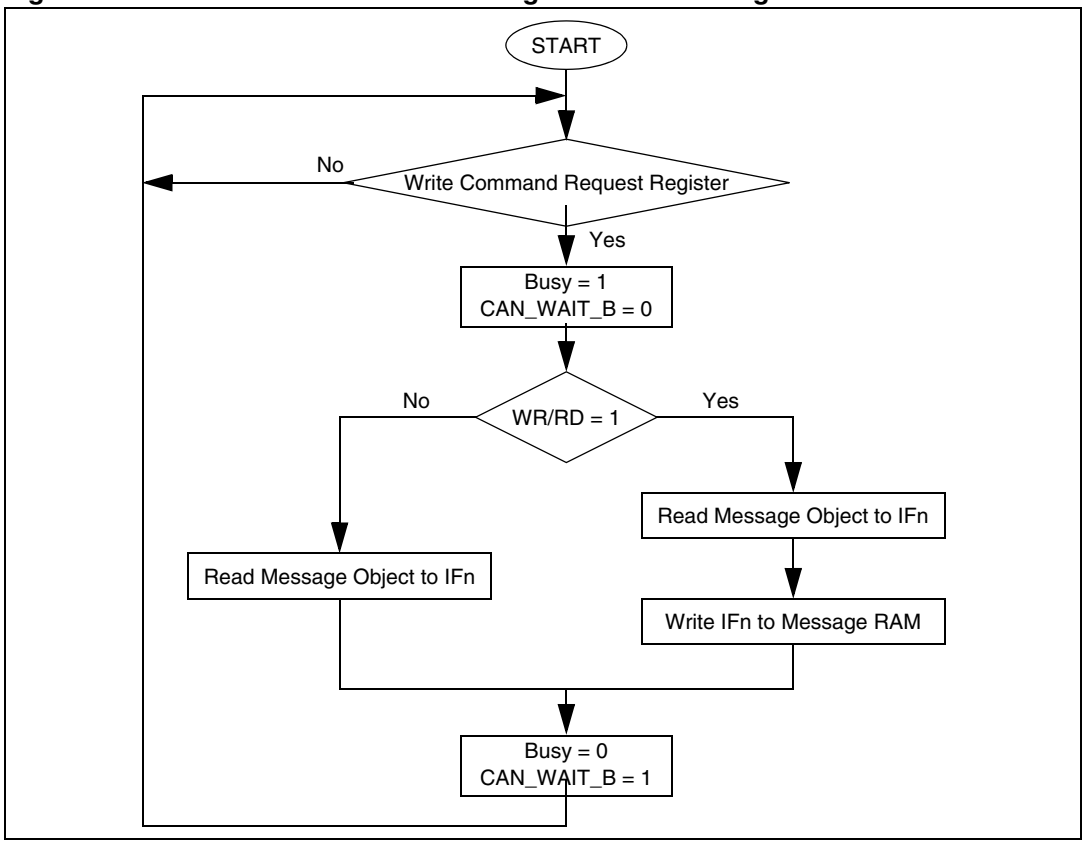
- Data Transfer from IF*n* Registers to the Message RAM
- Data Transfer from Message RAM to the IF*n* Registers
- Data Transfer from Shift Register to the Message RAM
- Data Transfer from Message RAM to Shift Register
- Data Transfer from Shift Register to the Acceptance Filtering unit
- Scanning of Message RAM for a matching Message Object
- Handling of TxRqst flags
- Handling of interrupts.

Data transfer from/to message RAM

When the CPU initiates a data transfer between the IF*n* Registers and Message RAM, the Message Handler sets the Busy bit in the respective Command Request Register (CAN_IFn_CRR). After the transfer has completed, the Busy bit is again cleared (see *Figure 50*).

The respective Command Mask Register specifies whether a complete Message Object or only parts of it will be transferred. Due to the structure of the Message RAM, it is not possible to write single bits/bytes of one Message Object. It is always necessary to write a complete Message Object into the Message RAM. Therefore, the data transfer from the IF*n* Registers to the Message RAM requires a read-modify-write cycle. First, those parts of the Message Object that are not to be changed are read from the Message RAM and then the complete contents of the Message Buffer Registers are written into the Message Object.

Figure 50. Data transfer between IF*n* Registers and Message RAM



After a partial write of a Message Object, the Message Buffer Registers that are not selected in the Command Mask Register will set the actual contents of the selected Message Object.

After a partial read of a Message Object, the Message Buffer Registers that are not selected in the Command Mask Register will be left unchanged.

Message transmission

If the shift register of the CAN Core cell is ready for loading and if there is no data transfer between the IFn Registers and Message RAM, the MsgVal bits in the Message Valid Register and TxRqst bits in the Transmission Request Register are evaluated. The valid

Message Object with the highest priority pending transmission request is loaded into the shift register by the Message Handler and the transmission is started. The NewDat bit of the Message Object is reset.

After a successful transmission and also if no new data was written to the Message Object (NewDat = '0') since the start of the transmission, the TxRqst bit of the Message Control register (CAN_IFn_MCR) will be reset. If TxIE bit of the Message Control register (CAN_IFn_MCR) is set, IntPnd bit of the Interrupt Identifier register (CAN_IDR) will be set after a successful transmission. If the C_CAN has lost the arbitration or if an error occurred during the transmission, the message will be retransmitted as soon as the CAN bus is free again. Meanwhile, if the transmission of a message with higher priority has been requested, the messages will be transmitted in the order of their priority.

Acceptance filtering of received messages

When the arbitration and control field (Identifier + IDE + RTR + DLC) of an incoming message is completely shifted into the Rx/Tx Shift Register of the CAN Core, the Message Handler FSM starts the scanning of the Message RAM for a matching valid Message Object.

To scan the Message RAM for a matching Message Object, the Acceptance Filtering unit is loaded with the arbitration bits from the CAN Core shift register. The arbitration and mask fields (including MsgVal, UMask, NewDat, and EoB) of Message Object 1 are then loaded into the Acceptance Filtering unit and compared with the arbitration field from the shift register. This is repeated with each following Message Object until a matching Message Object is found or until the end of the Message RAM is reached.

If a match occurs, the scan is stopped and the Message Handler FSM proceeds depending on the type of frame (Data Frame or Remote Frame) received.

Reception of data frame

The Message Handler FSM stores the message from the CAN Core shift register into the respective Message Object in the Message RAM. Not only the data bytes, but all arbitration bits and the Data Length Code are stored in the corresponding Message Object. This is done to keep the data bytes connected with the identifier even if arbitration mask registers are used.

The NewDat bit is set to indicate that new data (not yet seen by the CPU) has been received. The application software should reset NewDat bit when the Message Object has been read. If at the time of reception, the NewDat bit was already set, MsgLst is set to indicate that the previous data (supposedly not seen by the CPU) is lost. If the RxIE bit is set, the IntPnd bit is set, causing the Interrupt Register to point to this Message Object.

The TxRqst bit of this Message Object is reset to prevent the transmission of a Remote Frame, while the requested Data Frame has just been received.

Reception of Remote Frame

When a Remote Frame is received, three different configurations of the matching Message Object have to be considered:

1. **Dir** = ‘1’ (direction = *transmit*), **RmtEn** = ‘1’, **UMask** = ‘1’ or’0’
   At the reception of a matching Remote Frame, the TxRqst bit of this Message Object is set. The rest of the Message Object remains unchanged.
2. **Dir** = ‘1’ (direction = *transmit*), **RmtEn** = ‘0’, **UMask** =’0’
   At the reception of a matching Remote Frame, the TxRqst bit of this Message Object remains unchanged; the Remote Frame is ignored.
3. **Dir** = ‘1’ (direction = *transmit*), **RmtEn** = ‘0’, **UMask** =’1’
   At the reception of a matching Remote Frame, the TxRqst bit of this Message Object is reset. The arbitration and control field (Identifier + IDE + RTR + DLC) from the shift register is stored in the Message Object of the Message RAM and the NewDat bit of this Message Object is set. The data field of the Message Object remains unchanged; the Remote Frame is treated similar to a received Data Frame.

Receive/transmit priority

The receive/transmit priority for the Message Objects is attached to the message number. Message Object 1 has the highest priority, while Message Object 32 has the lowest priority. If more than one transmission request is pending, they are serviced due to the priority of the corresponding Message Object.

8.7.3 Configuring a transmit object

Table 31 shows how a Transmit Object should be initialized.

Table 31. Initialization of a Transmit Object

Msg Val	Arb	Data	Mask	EoB	Dir	New Dat	Msg Lst	RxIE	TxIE	IntPnd	RmtEn	Tx Rqst
1	appl.	appl.	appl.	1	1	0	0	0	appl.	0	appl.	0

The Arbitration Register values (ID28-0 and Xtd bit) are provided by the application. They define the identifier and type of the outgoing message. If an 11-bit Identifier (“Standard Frame”) is used, it is programmed to ID28 - ID18. The ID17 - ID0 can then be disregarded.

If the TxIE bit is set, the IntPnd bit will be set after a successful transmission of the Message Object.

If the RmtEn bit is set, a matching received Remote Frame will cause the TxRqst bit to be set; the Remote Frame will autonomously be answered by a Data Frame.

The Data Register values (DLC3-0, Data0-7) are provided by the application, TxRqst and RmtEn may not be set before the data is valid.

The Mask Registers (Msk28-0, UMask, MXtd, and MDir bits) may be used (UMask=’1’) to allow groups of Remote Frames with similar identifiers to set the TxRqst bit. The Dir bit should not be masked.

8.7.4 Updating a transmit object

The CPU may update the data bytes of a Transmit Object any time through the IFn Interface registers, neither MsgVal nor TxRqst have to be reset before the update. Even if only a part of the data bytes are to be updated, all four bytes of the corresponding IFn Data A Register or IFn Data B Register have to be valid before the contents of that register are transferred to the Message Object. Either the CPU has to write all four bytes into the IFn Data Register or

the Message Object is transferred to the IFn Data Register before the CPU writes the new data bytes.

When only the (eight) data bytes are updated, first 0x0087 is written to the Command Mask Register and then the number of the Message Object is written to the Command Request Register, concurrently updating the data bytes and setting TxRqst.

To prevent the reset of TxRqst at the end of a transmission that may already be in progress while the data is updated, NewDat has to be set together with TxRqst. For details see *Message transmission on page 154*.

When NewDat is set together with TxRqst, NewDat will be reset as soon as the new transmission has started.

8.7.5 Configuring a receive object

Table 32 shows how a Receive Object should be initialized.

Table 32. Initialization of a Receive Object

Msg Val	Arb	Data	Mask	EoB	Dir	New Dat	Msg Lst	RxIE	TxIE	IntPnd	RmtEn	TxRqst
1	appl.	appl.	appl.	1	0	0	0	appl.	0	0	0	0

The Arbitration Registers values (ID28-0 and Xtd bit) are provided by the application. They define the identifier and type of accepted received messages. If an 11-bit Identifier (“Standard Frame”) is used, it is programmed to ID28 - ID18. Then ID17 - ID0 can be disregarded. When a Data Frame with an 11-bit Identifier is received, ID17 - ID0 will be set to ‘0’.

If the RxIE bit is set, the IntPnd bit will be set when a received Data Frame is accepted and stored in the Message Object.

The Data Length Code (DLC3-0) is provided by the application. When the Message Handler stores a Data Frame in the Message Object, it will store the received Data Length Code and eight data bytes. If the Data Length Code is less than 8, the remaining bytes of the Message Object will be overwritten by unspecified values.

The Mask Registers (Msk28-0, UMask, MXtd, and MDir bits) may be used (UMask=’1’) to allow groups of Data Frames with similar identifiers to be accepted. The Dir bit should not be masked in typical applications.

8.7.6 Handling received messages

The CPU may read a received message any time via the IF*n* Interface registers. The data consistency is guaranteed by the Message Handler state machine.

Typically, the CPU will write first 0x007F to the Command Mask Register and then the number of the Message Object to the Command Request Register. This combination will transfer the whole received message from the Message RAM into the Message Buffer Register. Additionally, the bits NewDat and IntPnd are cleared in the Message RAM (not in the Message Buffer).

If the Message Object uses masks for acceptance filtering, the arbitration bits shows which of the matching messages have been received.

The actual value of NewDat shows whether a new message has been received since the last time this Message Object was read. The actual value of MsgLst shows whether more than one message has been received since the last time this Message Object was read. MsgLst will not be automatically reset.

By means of a Remote Frame, the CPU may request another CAN node to provide new data for a receive object. Setting the TxRqst bit of a receive object will cause the transmission of a Remote Frame with the receive object's identifier. This Remote Frame triggers the other CAN node to start the transmission of the matching Data Frame. If the matching Data Frame is received before the Remote Frame could be transmitted, the TxRqst bit is automatically reset.

8.7.7 Configuring a FIFO buffer

With the exception of the EoB bit, the configuration of Receive Objects belonging to a FIFO Buffer is the same as the configuration of a (single) Receive Object, see *Section 8.7.5: Configuring a receive object on page 157*.

To concatenate two or more Message Objects into a FIFO Buffer, the identifiers and masks (if used) of these Message Objects have to be programmed to matching values. Due to the implicit priority of the Message Objects, the Message Object with the lowest number will be the first Message Object of the FIFO Buffer. The EoB bit of all Message Objects of a FIFO Buffer except the last have to be programmed to zero. The EoB bits of the last Message Object of a FIFO Buffer is set to one, configuring it as the End of the Block.

8.7.8 Receiving messages with FIFO buffers

Received messages with identifiers matching to a FIFO Buffer are stored in a Message Object of this FIFO Buffer starting with the Message Object with the lowest message number.

When a message is stored in a Message Object of a FIFO Buffer, the NewDat bit of this Message Object is set. By setting NewDat while EoB is zero, the Message Object is locked for further write access by the Message Handler until the application software has written the NewDat bit back to zero.

Messages are stored into a FIFO Buffer until the last Message Object of this FIFO Buffer is reached. If none of the preceding Message Objects is released by writing NewDat to zero, all further messages for this FIFO Buffer will be written into the last Message Object of the FIFO Buffer and therefore overwrite previous messages.

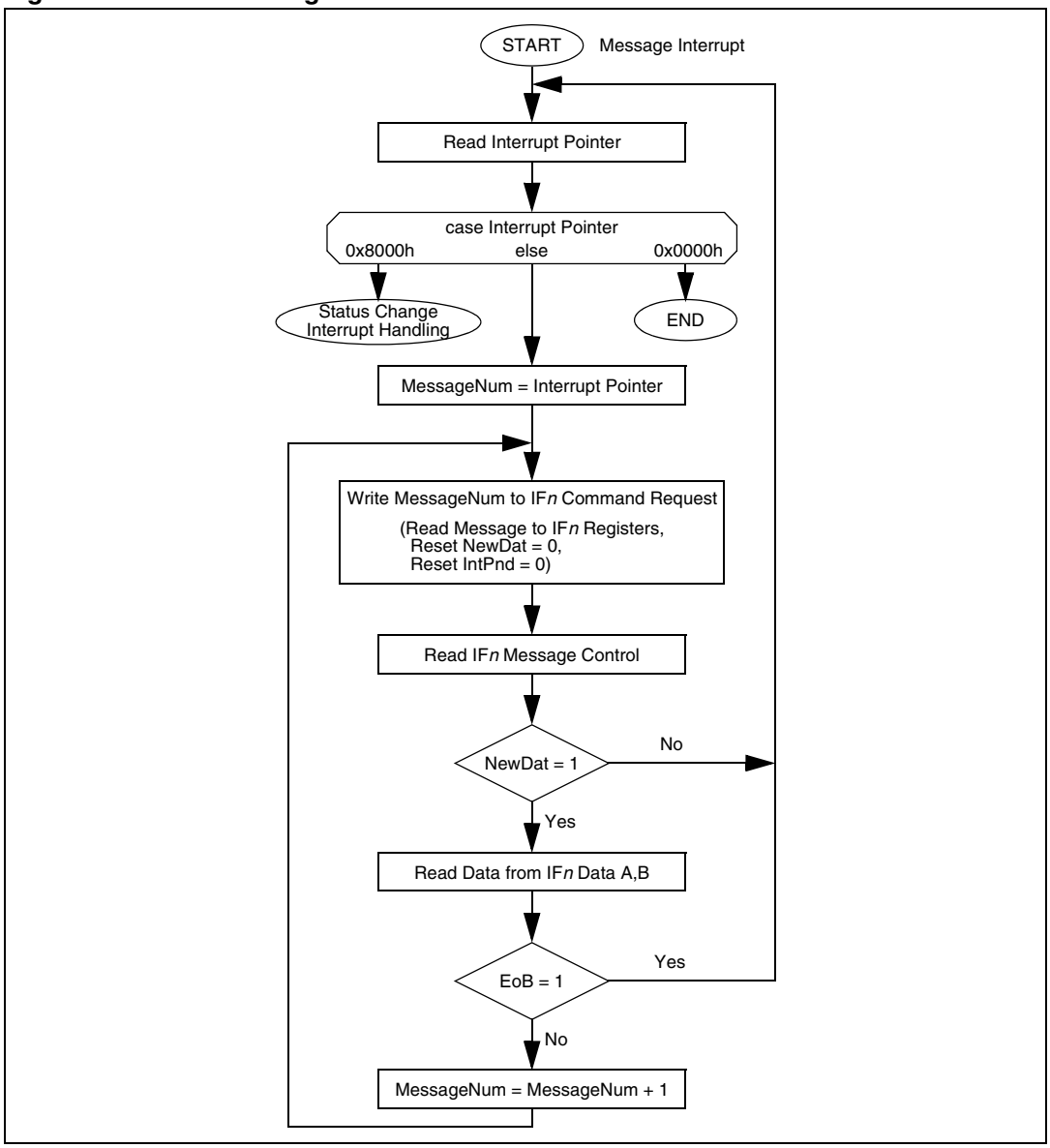
Reading from a FIFO buffer

When the CPU transfers the contents of a Message Object to the IFn Message Buffer register by writing its number to the IFn Command Request Register, the corresponding Command Mask Register should be programmed in such a way that bits NewDat and IntPnd are reset to zero (TxRqst/NewDat = '1' and ClrIntPnd = '1'). The values of these bits in the Message Control Register always reflect the status before resetting the bits.

To assure the correct function of a FIFO Buffer, the CPU should read the Message Objects starting at the FIFO Object with the lowest message number.

Figure 51 shows how a set of Message Objects which are concatenated to a FIFO Buffer can be handled by the CPU.

Figure 51. CPU handling of a FIFO buffer



8.7.9 Handling interrupts

If several interrupts are pending, the CAN Interrupt Register will point to the pending interrupt with the highest priority, disregarding their chronological order. An interrupt remains pending until the application software has cleared it.

The Status Interrupt has the highest priority. Among the message interrupts, interrupt priority of the Message Object decreases with increasing message number.

A message interrupt is cleared by clearing the IntPnd bit of the Message Object. The Status Interrupt is cleared by reading the Status Register.

The interrupt identifier, IntId, in the Interrupt Register, indicates the cause of the interrupt. When no interrupt is pending, the register will hold the value zero. If the value of the Interrupt Register is different from zero, then there is an interrupt pending and, if IE is set,

the IRQ interrupt signal to the EIC is active. The interrupt remains active until the Interrupt Register is back to value zero (the cause of the interrupt is reset) or until IE is reset.

The value 0x8000 indicates that an interrupt is pending because the CAN Core has updated (not necessarily changed) the Status Register (Error Interrupt or Status Interrupt). This interrupt has the highest priority. The CPU can update (reset) the status bits RxOk, TxOk and LEC, but a write access of the CPU to the Status Register can never generate or reset an interrupt.

All other values indicate that the source of the interrupt is one of the Message Objects. IntId points to the pending message interrupt with the highest interrupt priority.

The CPU controls whether a change of the Status Register may cause an interrupt (bits EIE and SIE in the CAN Control Register) and whether the interrupt line becomes active when the Interrupt Register is different from zero (bit IE in the CAN Control Register). The Interrupt Register will be updated even when IE is reset.

The CPU has two possibilities to follow the source of a message interrupt. First, it can follow the IntId in the Interrupt Register and second it can poll the Interrupt Pending Register (see *Interrupt pending registers 1 & 2 (CAN_IPnR) on page 150*).

An interrupt service routine that is reading the message that is the source of the interrupt may read the message and reset the Message Object's IntPnd at the same time (bit ClrIntPnd in the Command Mask Register). When IntPnd is cleared, the Interrupt Register will point to the next Message Object with a pending interrupt.

8.7.10 Configuring the bit timing

Even if minor errors in the configuration of the CAN bit timing do not result in immediate failure, the performance of a CAN network can be reduced significantly.

In many cases, the CAN bit synchronization will amend a faulty configuration of the CAN bit timing to such a degree that only occasionally an error frame is generated. However, in the case of arbitration, when two or more CAN nodes simultaneously try to transmit a frame, a misplaced sample point may cause one of the transmitters to become error passive.

The analysis of such sporadic errors requires a detailed knowledge of the CAN bit synchronization inside a CAN node and interaction of the CAN nodes on the CAN bus.

Bit time and bit rate

CAN supports bit rates in the range of lower than 1 kBit/s up to 1000 kBit/s. Each member of the CAN network has its own clock generator, usually a quartz oscillator. The timing parameter of the bit time (i.e. the reciprocal of the bit rate) can be configured individually for each CAN node, creating a common bit rate even though the oscillator periods of the CAN nodes (f_{osc}) may be different.

The frequencies of these oscillators are not absolutely stable, small variations are caused by changes in temperature or voltage and by deteriorating components. As long as the variations remain inside a specific oscillator tolerance range (df), the CAN nodes are able to compensate for the different bit rates by re-synchronizing to the bit stream.

According to the CAN specification, the bit time is divided into four segments (see *Figure 52*). The Synchronization Segment, the Propagation Time Segment, the Phase Buffer Segment 1 and the Phase Buffer Segment 2. Each segment consists of a specific, programmable number of time quanta (see *Table 33*). The length of the time quantum ($t_{q)}$,

which is the basic time unit of the bit time, is defined by the CAN controller's system clock f_{APB} and the BRP bit of the Bit Timing Register (CAN_BTR): t_q = BRP / f_{APB}.

The Synchronization Segment, Sync_Seg, is that part of the bit time where edges of the CAN bus level are expected to occur. The distance between an edge, that occurs outside of Sync_Seg, and the Sync_Seg is called the phase error of that edge. The Propagation Time Segment, Prop_Seg, is intended to compensate for the physical delay times within the CAN network. The Phase Buffer Segments Phase_Seg1 and Phase_Seg2 surround the Sample Point. The (Re-)Synchronization Jump Width (SJW) defines how far a re-synchronization may move the Sample Point inside the limits defined by the Phase Buffer Segments to compensate for edge phase errors.

Figure 52. Bit timing

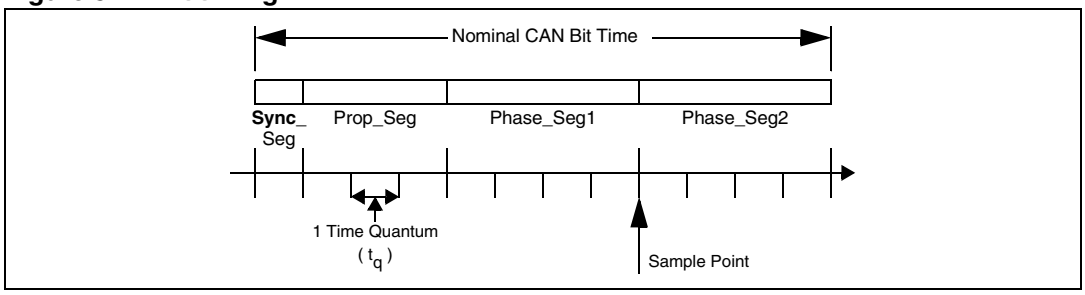


Table 33. CAN bit time parameters

Parameter	Range	Remark
BRP	[1 .. 32]	defines the length of the time quantum t_q
Sync_Seg	1 t_q	fixed length, synchronization of bus input to system clock
Prop_Seg	[1.. 8] t_q	compensates for the physical delay times
Phase_Seg1	[1..8] t_q	may be lengthened temporarily by synchronization
Phase_Seg2	[1.. 8] t_q	may be shortened temporarily by synchronization
SJW	[1 .. 4] t_q	may not be longer than either Phase Buffer Segment
This table describes the minimum programmable ranges required by the CAN protocol		

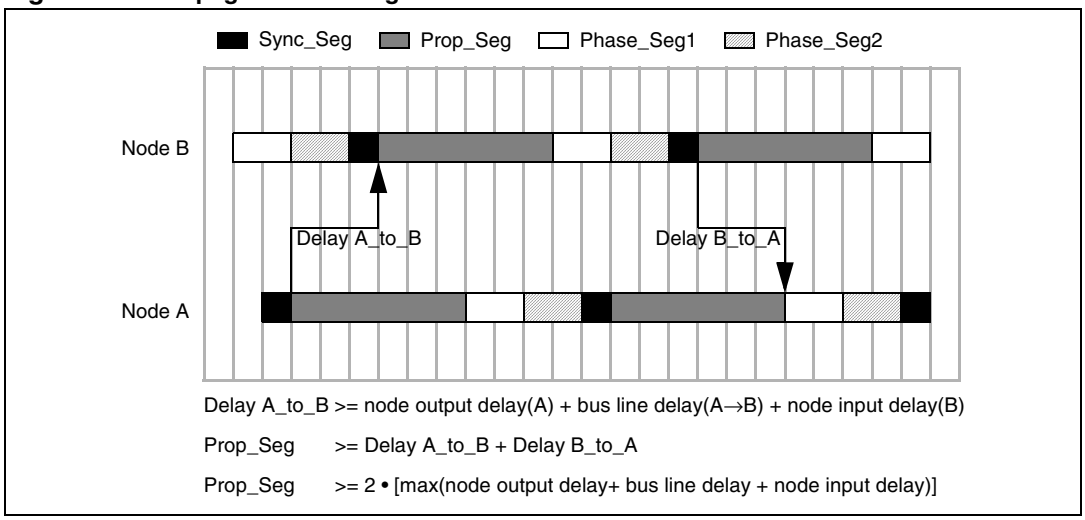
A given bit rate may be met by different bit time configurations, but for the proper function of the CAN network the physical delay times and the oscillator's tolerance range have to be considered.

Propagation time segment

This part of the bit time is used to compensate physical delay times within the network. These delay times consist of the signal propagation time on the bus and the internal delay time of the CAN nodes.

Any CAN node synchronized to the bit stream on the CAN bus will be out of phase with the transmitter of that bit stream, caused by the signal propagation time between the two nodes. The CAN protocol's non-destructive bitwise arbitration and the dominant acknowledge bit provided by receivers of CAN messages requires that a CAN node transmitting a bit stream must also be able to receive dominant bits transmitted by other CAN nodes that are synchronized to that bit stream. The example in *Figure 53* shows the phase shift and propagation times between two CAN nodes.

Figure 53. Propagation time segment



In this example, both nodes A and B are transmitters, performing an arbitration for the CAN bus. Node A has sent its Start of Frame bit less than one bit time earlier than node B, therefore node B has synchronized itself to the received edge from recessive to dominant. Since node B has received this edge delay (A_to_B) after it has been transmitted, B's bit timing segments are shifted with respect to A. Node B sends an identifier with higher priority and so it will win the arbitration at a specific identifier bit when it transmits a dominant bit while node A transmits a recessive bit. The dominant bit transmitted by node B will arrive at node A after the delay (B_to_A).

Due to oscillator tolerances, the actual position of node A's Sample Point can be anywhere inside the nominal range of node A's Phase Buffer Segments, so the bit transmitted by node B must arrive at node A before the start of Phase_Seg1. This condition defines the length of Prop_Seg.

If the edge from recessive to dominant transmitted by node B arrives at node A after the start of Phase_Seg1, it can happen that node A samples a recessive bit instead of a dominant bit, resulting in a bit error and the destruction of the current frame by an error flag.

The error occurs only when two nodes arbitrate for the CAN bus that have oscillators of opposite ends of the tolerance range and that are separated by a long bus line. This is an example of a minor error in the bit timing configuration (Prop_Seg to short) that causes sporadic bus errors.

Some CAN implementations provide an optional 3 Sample Mode but the C_CAN does not. In this mode, the CAN bus input signal passes a digital low-pass filter, using three samples and a majority logic to determine the valid bit value. This results in an additional input delay of 1 t_q, requiring a longer Prop_Seg.

Phase buffer segments and synchronization

The Phase Buffer Segments (Phase_Seg1 and Phase_Seg2) and the Synchronization Jump Width (SJW) are used to compensate for the oscillator tolerance. The Phase Buffer Segments may be lengthened or shortened by synchronization.

Synchronizations occur on edges from recessive to dominant, their purpose is to control the distance between edges and Sample Points.

Edges are detected by sampling the actual bus level in each time quantum and comparing it with the bus level at the previous Sample Point. A synchronization may be done only if a recessive bit was sampled at the previous Sample Point and if the bus level at the actual time quantum is dominant.

An edge is synchronous if it occurs inside of Sync_Seg, otherwise the distance between edge and the end of Sync_Seg is the edge phase error, measured in time quanta. If the edge occurs before Sync_Seg, the phase error is negative, else it is positive.

Two types of synchronization exist, Hard Synchronization and Re-synchronization.

A Hard Synchronization is done once at the start of a frame and inside a frame only when Re-synchronizations occur.

- Hard Synchronization
  After a hard synchronization, the bit time is restarted with the end of Sync_Seg, regardless of the edge phase error. Thus hard synchronization forces the edge, which has caused the hard synchronization to lie within the synchronization segment of the restarted bit time.
- Bit Re-synchronization
  Re-synchronization leads to a shortening or lengthening of the bit time such that the position of the sample point is shifted with regard to the edge.
  When the phase error of the edge which causes Re-synchronization is positive, Phase_Seg1 is lengthened. If the magnitude of the phase error is less than SJW, Phase_Seg1 is lengthened by the magnitude of the phase error, else it is lengthened by SJW.
  When the phase error of the edge, which causes Re-synchronization is negative, Phase_Seg2 is shortened. If the magnitude of the phase error is less than SJW, Phase_Seg2 is shortened by the magnitude of the phase error, else it is shortened by SJW.

When the magnitude of the phase error of the edge is less than or equal to the programmed value of SJW, the results of Hard Synchronization and Re-synchronization are the same. If the magnitude of the phase error is larger than SJW, the Re-synchronization cannot compensate the phase error completely, an error (phase error - SJW) remains.

Only one synchronization may be done between two Sample Points. The Synchronizations maintain a minimum distance between edges and Sample Points, giving the bus level time to stabilize and filtering out spikes that are shorter than (Prop_Seg + Phase_Seg1).

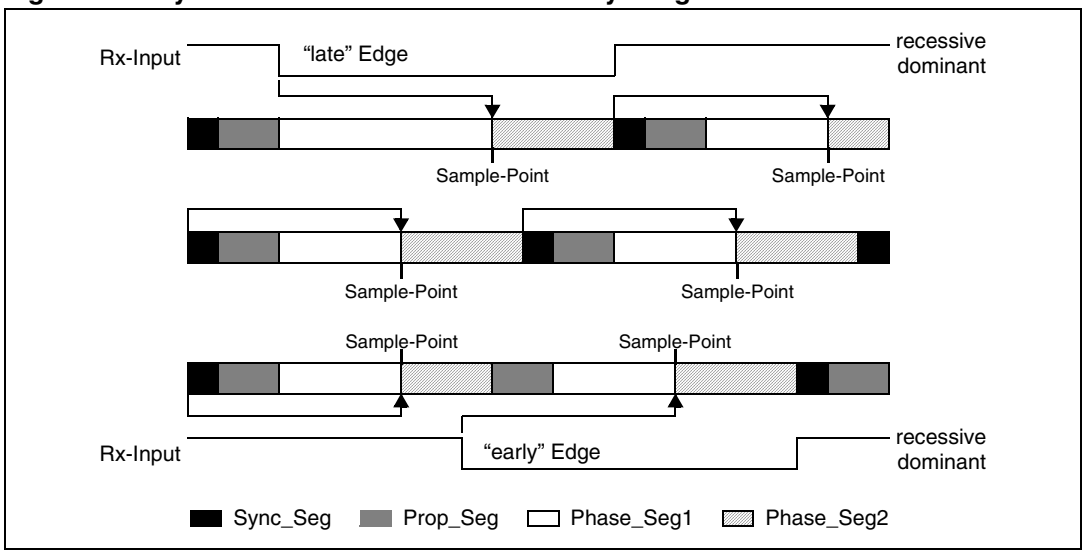
Apart from noise spikes, most synchronizations are caused by arbitration. All nodes synchronize “hard” on the edge transmitted by the “leading” transceiver that started transmitting first, but due to propagation delay times, they cannot become ideally synchronized. The “leading” transmitter does not necessarily win the arbitration, therefore the receivers have to synchronize themselves to different transmitters that subsequently “take the lead” and that are differently synchronized to the previously “leading” transmitter. The same happens at the acknowledge field, where the transmitter and some of the receivers will have to synchronize to that receiver that “takes the lead” in the transmission of the dominant acknowledge bit.

Synchronizations after the end of the arbitration will be caused by oscillator tolerance, when the differences in the oscillator’s clock periods of transmitter and receivers sum up during the time between synchronizations (at most ten bits). These summarized differences may not be longer than the SJW, limiting the oscillator’s tolerance range.

The examples in *Figure 54* show how the Phase Buffer Segments are used to compensate for phase errors. There are three drawings of each two consecutive bit timings. The upper

drawing shows the synchronization on a “late” edge, the lower drawing shows the synchronization on an “early” edge, and the middle drawing is the reference without synchronization.

Figure 54. Synchronization on “late” and “early” Edges



In the first example, an edge from recessive to dominant occurs at the end of Prop_Seg. The edge is “late” since it occurs after the Sync_Seg. Reacting to the “late” edge, Phase_Seg1 is lengthened so that the distance from the edge to the Sample Point is the same as it would have been from the Sync_Seg to the Sample Point if no edge had occurred. The phase error of this “late” edge is less than SJW, so it is fully compensated and the edge from dominant to recessive at the end of the bit, which is one nominal bit time long, occurs in the Sync_Seg.

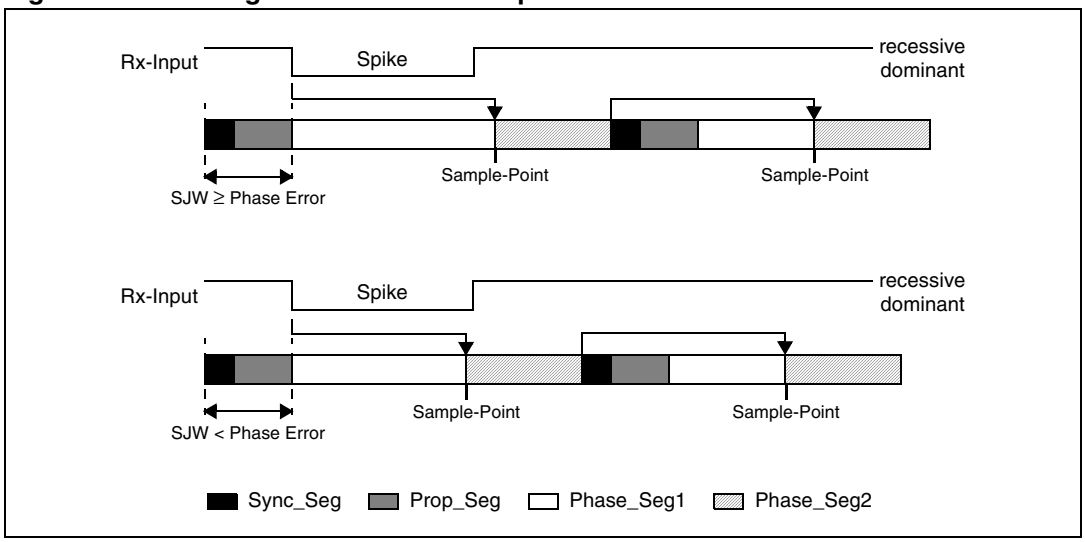
In the second example, an edge from recessive to dominant occurs during Phase_Seg2. The edge is “early” since it occurs before a Sync_Seg. Reacting to the “early” edge, Phase_Seg2 is shortened and Sync_Seg is omitted, so that the distance from the edge to the Sample Point is the same as it would have been from an Sync_Seg to the Sample Point if no edge had occurred. As in the previous example, the magnitude of phase error of this “early” edge’s is less than SJW, so it is fully compensated.

The Phase Buffer Segments are lengthened or shortened temporarily only; at the next bit time, the segments return to their nominal programmed values.

In these examples, the bit timing is seen from the point of view of the CAN state machine, where the bit time starts and ends at the Sample Points. The state machine omits Sync_Seg when synchronizing on an “early” edge, because it cannot subsequently redefine that time quantum of Phase_Seg2 where the edge occurs to be the Sync_Seg.

The examples in *Figure 55* show how short dominant noise spikes are filtered by synchronizations. In both examples the spike starts at the end of Prop_Seg and has the length of “Prop_Seg + Phase_Seg1”.

Figure 55. Filtering of short dominant spikes



In the first example, the Synchronization Jump Width is greater than or equal to the phase error of the spike's edge from recessive to dominant. Therefore the Sample Point is shifted after the end of the spike; a recessive bus level is sampled.

In the second example, SJW is shorter than the phase error, so the Sample Point cannot be shifted far enough; the dominant spike is sampled as actual bus level.

Oscillator tolerance range

The oscillator tolerance range was increased when the CAN protocol was developed from version 1.1 to version 1.2 (version 1.0 was never implemented in silicon). The option to synchronize on edges from dominant to recessive became obsolete, only edges from recessive to dominant are considered for synchronization. The only CAN controllers to implement protocol version 1.1 have been Intel 82526 and Philips 82C200, both are superseded by successor products. The protocol update to version 2.0 (A and B) had no influence on the oscillator tolerance.

The tolerance range df for an oscillator frequency f_{osc} around the nominal frequency f_{nom} is:

$$(1 - df) \bullet f_{nom} \leq f_{osc} \leq (1 + df) \bullet f_{nom}$$

It depends on the proportions of Phase_Seg1, Phase_Seg2, SJW, and the bit time. The maximum tolerance df is the defined by two conditions (both shall be met):

$$\text{I:} \quad df \leq \frac{\min(\text{Phase\_Seg1}, \text{Phase\_Seg2})}{2 \cdot (13 \cdot \text{bit\_time} - \text{Phase\_Seg2})}$$

$$\text{II:} \quad df \leq \frac{\text{SJW}}{20 \cdot \text{bit\_time}}$$

It has to be considered that SJW may not be larger than the smaller of the Phase Buffer Segments and that the Propagation Time Segment limits that part of the bit time that may be used for the Phase Buffer Segments.

The combination Prop_Seg = 1 and Phase_Seg1 = Phase_Seg2 = SJW = 4 allows the largest possible oscillator tolerance of 1.58%. This combination with a Propagation Time Segment of only 10% of the bit time is not suitable for short bit times; it can be used for bit rates of up to 125 kBit/s (bit time = 8 µs) with a bus length of 40 m.

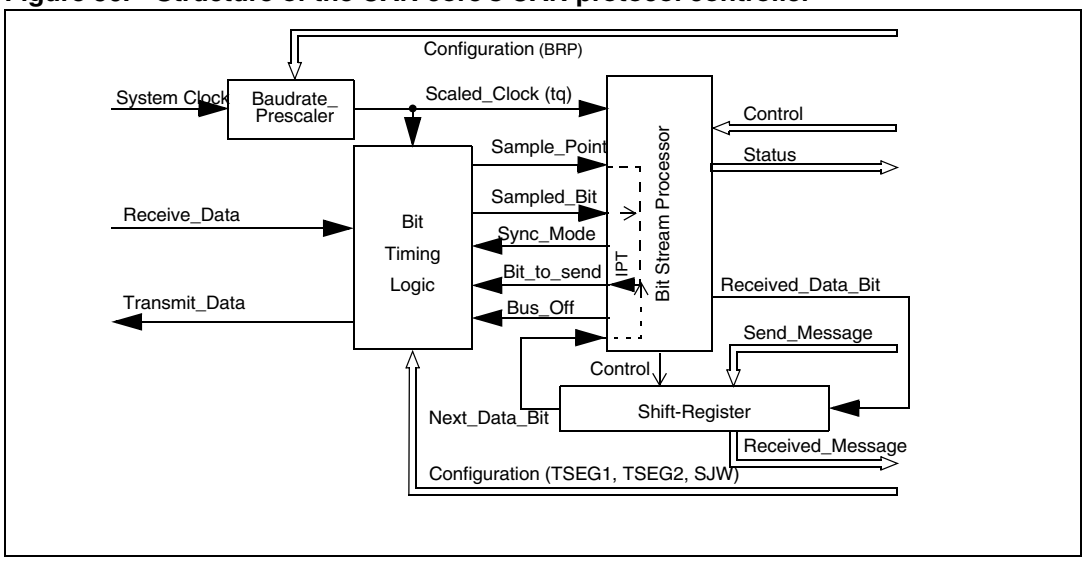
Configuring the CAN protocol controller

In most CAN implementations and also in the C_CAN, the bit timing configuration is programmed in two register bytes. The sum of Prop_Seg and Phase_Seg1 (as TSEG1) is combined with Phase_Seg2 (as TSEG2) in one byte, SJW and BRP are combined in the other byte (see *Figure 56 on page 166*).

In these bit timing registers, the four components TSEG1, TSEG2, SJW, and BRP have to be programmed to a numerical value that is one less than its functional value. Therefore, instead of values in the range of [1..n], values in the range of [0..n-1] are programmed. That way, e.g. SJW (functional range of [1..4]) is represented by only two bits.

Therefore the length of the bit time is (programmed values) [TSEG1 + TSEG2 + 3] t_q or (functional values) [Sync_Seg + Prop_Seg + Phase_Seg1 + Phase_Seg2] t_q.

Figure 56. Structure of the CAN core's CAN protocol controller



The data in the bit timing registers is the configuration input of the CAN protocol controller. The Baud Rate Prescaler (configured by BRP) defines the length of the time quantum, the basic time unit of the bit time; the Bit Timing Logic (configured by TSEG1, TSEG2, and SJW) defines the number of time quanta in the bit time.

The processing of the bit time, the calculation of the position of the Sample Point, and occasional synchronizations are controlled by the BTL state machine, which is evaluated once each time quantum. The rest of the CAN protocol controller, the BSP state machine is evaluated once each bit time, at the Sample Point.

The Shift Register sends the messages serially and receives the messages parallely. Its loading and shifting is controlled by the BSP.

The BSP translates messages into frames and vice versa. It generates and discards the enclosing fixed format bits, inserts and extracts stuff bits, calculates and checks the CRC code, performs the error management, and decides which type of synchronization is to be used. It is evaluated at the Sample Point and processes the sampled bus input bit. The time that is needed to calculate the next bit to be sent after the Sample point(e.g. data bit, CRC bit, stuff bit, error flag, or idle) is called the Information Processing Time (IPT).

The IPT is application specific but may not be longer than 2 t_q; the IPT for the C_CAN is 0 t_q. Its length is the lower limit of the programmed length of Phase_Seg2. In case of a

synchronization, Phase_Seg2 may be shortened to a value less than IPT, which does not affect bus timing.

Calculating bit timing parameters

Usually, the calculation of the bit timing configuration starts with a desired bit rate or bit time. The resulting bit time (1/bit rate) must be an integer multiple of the system clock period.

The bit time may consist of 4 to 25 time quanta, the length of the time quantum t_q is defined by the Baud Rate Prescaler with t_q = (Baud Rate Prescaler)/f_{sys}. Several combinations may lead to the desired bit time, allowing iterations of the following steps.

First part of the bit time to be defined is the Prop_Seg. Its length depends on the delay times measured in the system. A maximum bus length as well as a maximum node delay has to be defined for expandible CAN bus systems. The resulting time for Prop_Seg is converted into time quanta (rounded up to the nearest integer multiple of t_q).

The Sync_Seg is 1 t_q long (fixed), leaving (bit time – Prop_Seg – 1) t_q for the two Phase Buffer Segments. If the number of remaining t_q is even, the Phase Buffer Segments have the same length, Phase_Seg2 = Phase_Seg1, else Phase_Seg2 = Phase_Seg1 + 1.

The minimum nominal length of Phase_Seg2 has to be regarded as well. Phase_Seg2 may not be shorter than the IPT of the CAN controller, which, depending on the actual implementation, is in the range of [0..2] t_q.

The length of the Synchronization Jump Width is set to its maximum value, which is the minimum of 4 and Phase_Seg1.

The oscillator tolerance range necessary for the resulting configuration is calculated by the formulas given in *Oscillator tolerance range on page 165*

If more than one configuration is possible, that configuration allowing the highest oscillator tolerance range should be chosen.

CAN nodes with different system clocks require different configurations to come to the same bit rate. The calculation of the propagation time in the CAN network, based on the nodes with the longest delay times, is done once for the whole network.

The oscillator tolerance range of the CAN systems is limited by that node with the lowest tolerance range.

The calculation may show that bus length or bit rate have to be decreased or that the stability of the oscillator frequency has to be increased in order to find a protocol compliant configuration of the CAN bit timing.

The resulting configuration is written into the Bit Timing Register:

(Phase_Seg2-1)&(Phase_Seg1+Prop_Seg-1)&

(SynchronisationJumpWidth-1)&(Prescaler-1)

Example for bit timing at high baudrate

In this example, the frequency of APB_CLK is 10 MHz, BRP is 0, the bit rate is 1 MBit/s.

t_q	100	ns	= t_{APB_CLK}
delay of bus driver	50	ns	
delay of receiver circuit	30	ns	
delay of bus line (40m)	220	ns	

t_{Prop}	600	ns	$= 6 \bullet t_q$
t_{SJW}	100	ns	$= 1 \bullet t_q$
t_{TSeg1}	700	ns	$= t_{Prop} + t_{SJW}$
t_{TSeg2}	200	ns	= Information Processing Time + $1 \bullet t_q$
$t_{Sync\text{-}Seg}$	100	ns	$= 1 \bullet t_q$
bit time	1000	ns	$= t_{Sync\text{-}Seg} + t_{TSeg1} + t_{TSeg2}$
tolerance for APB_CLK	0.39	%	$= \frac{\min(PB1,PB2)}{2x(13xbit\_time-PB2)}$ $= \frac{0.1\mu s}{2x(13x1\mu s-0.2\mu s)}$

In this example, the concatenated bit time parameters are $(2\text{-}1)_3\&(7\text{-}1)_4\&(1\text{-}1)_2\&(1\text{-}1)_6$, the Bit Timing Register is programmed to= 0x1600.

Example for bit timing at low baudrate

In this example, the frequency of APB_CLK is 2 MHz, BRP is 1, the bit rate is 100 KBit/s.

t_q	1	µs	$= 2 \bullet t_{APB\_CLK}$
delay of bus driver	200	ns	
delay of receiver circuit	80	ns	
delay of bus line (40m)	220	ns	
t_{Prop}	1	µs	$= 1 \bullet t_q$
t_{SJW}	4	µs	$= 4 \bullet t_q$
t_{TSeg1}	5	µs	$= t_{Prop} + t_{SJW}$
t_{TSeg2}	4	µs	= Information Processing Time + $3 \bullet t_q$
$t_{Sync\text{-}Seg}$	1	µs	$= 1 \bullet t_q$
bit time	10	µs	$= t_{Sync\text{-}Seg} + t_{TSeg1} + t_{TSeg2}$
tolerance for APB_CLK	1.58	%	$= \frac{\min(PB1,PB2)}{2x(13xbit\_time-PB2)}$ $= \frac{4\mu s}{2x(13x10\mu s-4\mu s)}$

In this example, the concatenated bit time parameters are $(4\text{-}1)_3\&(5\text{-}1)_4\&(4\text{-}1)_2\&(2\text{-}1)_6$, the Bit Timing Register is programmed to= 0x34C1.

9 I²C interface module (I2C)

A I²C Bus Interface serves as an interface between the microcontroller and the serial I²C bus. It provides both multimaster and slave functions, and controls all I²C bus-specific sequencing, protocol, arbitration and timing. It supports fast I²C mode (400kHz).

9.1 Main features

- Parallel-bus/I²C protocol converter
- Multi-master capability
- 7-bit/10-bit Addressing
- Transmitter/Receiver flag
- End-of-byte transmission flag
- Transfer problem detection
- Standard/Fast I²C mode

I2C Master Features:

- Clock generation
- I²C bus busy flag
- Arbitration Lost Flag
- End-of-byte transmission flag
- Transmitter/Receiver Flag
- Start bit detection flag
- Start and Stop generation

I2C Slave Features:

- Start bit detection flag
- Stop bit detection
- I²C bus busy flag
- Detection of misplaced start or stop condition
- Programmable I²C Address detection
- Transfer problem detection
- End-of-byte transmission flag
- Transmitter/Receiver flag

9.2 General description

In addition to receiving and transmitting data, this interface converts them from serial to parallel format and vice versa, using either an interrupt or polled handshake. The interrupts are enabled or disabled by software. The interface is connected to the I²C bus by a data pin (SDA) and by a clock pin (SCL). It can be connected both with a standard I²C bus and a Fast I²C bus. This selection is made by software.

9.2.1 Mode selection

The interface can operate in the four following modes:

- Slave transmitter/receiver
- Master transmitter/receiver

By default, it operates in slave mode.

The interface automatically switches from slave to master after it generates a START condition and from master to slave in case of arbitration loss or a STOP generation, allowing then Multi-Master capability.

9.2.2 Communication flow

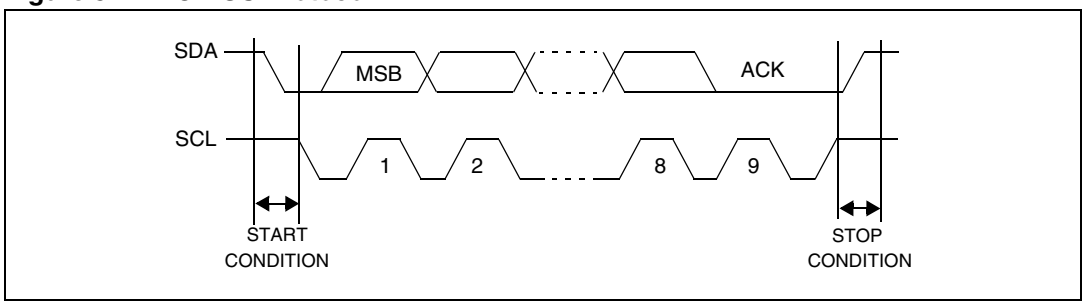
In Master mode, it initiates a data transfer and generates the clock signal. A serial data transfer always begins with a start condition and ends with a stop condition. Both start and stop conditions are generated in master mode by hardware as soon as the Master mode is selected.

In Slave mode, the interface is capable of recognizing its own address (7 or 10-bit), and the General Call address. The General Call address detection may be enabled or disabled by software.

Data and addresses are transferred as 8-bit bytes, MSB first. The first byte(s) following the start condition contain the address (one in 7-bit mode, two in 10-bit mode). The address is always transmitted in Master mode.

A 9th clock pulse follows the 8 clock cycles of a byte transfer, during which the receiver must send an acknowledge bit to the transmitter. Refer to *Figure 57*.

Figure 57. I²C BUS Protocol



Acknowledge may be enabled and disabled by software.

The I²C interface address and/or general call address can be selected by software.

The speed of the I²C interface may be selected between Standard (0-100KHz) and Fast I²C (100-400KHz).

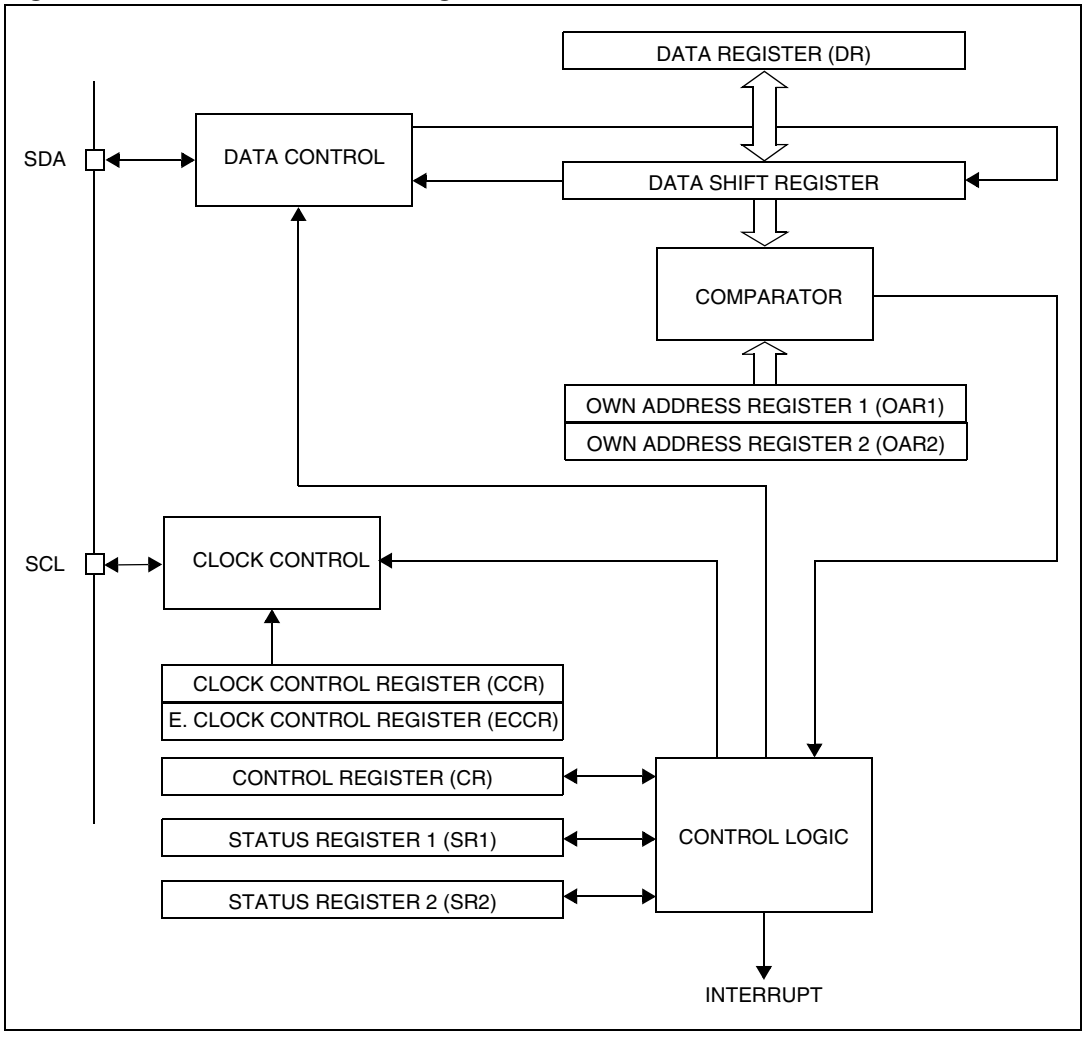
9.2.3 SDA/SCL line control

Transmitter mode: the interface holds the clock line low before transmission to wait for the microcontroller to write the byte in the Data Register.

Receiver mode: the interface holds the clock line low after reception to wait for the microcontroller to read the byte in the Data Register.

The SCL frequency (f_{SCL}) is controlled by a programmable clock divider which depends on the I^2C bus mode.

Figure 58. I^2C Interface Block Diagram



9.3 Functional description

Refer to the I2Cn_CR, I2Cn_SR1 and I2Cn_SR2 registers in *Section 9.5* for the bit definitions.

By default the I^2C interface operates in Slave mode (M/SL bit is cleared) except when it initiates a transmit or receive sequence.

First the interface frequency must be configured using the FRi bits in the I2Cn_OAR2 register.

9.3.1 Slave mode

As soon as a start condition is detected, the address is received from the SDA line and sent to the shift register; then it is compared with the address of the interface or the General Call address (if selected by software).

Note: In 10-bit addressing mode, the comparison includes the header sequence (11110xx0) and the two most significant bits of the address.

Header matched (10-bit mode only): the interface generates an acknowledge pulse if the ACK bit is set.

Address not matched: the interface ignores it and waits for another Start condition.

Address matched: the interface generates in sequence:

- Acknowledge pulse if the ACK bit is set.
- EVF and ADSL bits are set with an interrupt if the ITE bit is set.

Then the interface waits for a read of the I2Cn_SR1 register, **holding the SCL line low** (see *Figure 59* Transfer sequencing EV1).
Next, in 7-bit mode read the I2Cn_DR register to determine from the least significant bit (Data Direction Bit) if the slave must enter Receiver or Transmitter mode.

In 10-bit mode, after receiving the address sequence the slave is always in receive mode. It will enter transmit mode on receiving a repeated Start condition followed by the header sequence with matching address bits and the least significant bit set (11110xx1).

Slave receiver

Following the address reception and after I2Cn_SR1 register has been read, the slave receives bytes from the SDA line into the I2Cn_DR register via the internal shift register. After each byte the interface generates in sequence:

- Acknowledge pulse if the ACK bit is set
- EVF and BTF bits are set with an interrupt if the ITE bit is set.

Then the interface waits for a read of the I2Cn_SR1 register followed by a read of the I2Cn_DR register, **holding the SCL line low** (see *Figure 59* Transfer sequencing EV2).

Slave Transmitter

Following the address reception and after I2Cn_SR1 register has been read, the slave sends bytes from the I2Cn_DR register to the SDA line via the internal shift register.

The slave waits for a read of the I2Cn_SR1 register followed by a write in the I2Cn_DR register, **holding the SCL line low** (see *Figure 59* Transfer sequencing EV3).

When the acknowledge pulse is received:

- The EVF and BTF bits are set by hardware with an interrupt if the ITE bit is set.

Closing slave communication

After the last data byte is transferred a Stop Condition is generated by the master. The interface detects this condition and sets:

- EVF and STOPF bits with an interrupt if the ITE bit is set.

Then the interface waits for a read of the I2Cn_SR2 register (see *Figure 59* Transfer sequencing EV4).

Error cases

- **BERR**: Detection of a Stop or a Start condition during a byte transfer. In this case, the EVF and the BERR bits are set with an interrupt if the ITE bit is set.
  This detection is performed on the last 8 bits of a byte transfer but not on the first bit, as a Start or Stop condition is a normal operation at this stage in Slave mode.
  If it is a Stop then the interface discards the data, released the lines and waits for another Start condition.
  If it is a Start then the interface discards the data and waits for the next slave address on the bus.
- **AF**: Detection of a non-acknowledge bit. In this case, the EVF and AF bits are set with an interrupt if the ITE bit is set.

Note: In both cases, SCL line is not held low; however, SDA line can remain low due to possible «0» bits transmitted last. It is then necessary to release both lines by software.

How to release the SDA / SCL lines

Set and subsequently clear the STOP bit while BTF is set. The SDA/SCL lines are released after the transfer of the current byte.

9.3.2 Master mode

To switch from default Slave mode to Master mode a Start condition generation is needed.

Start condition

Setting the START bit while the BUSY bit is cleared causes the interface to switch to Master mode (M/SL bit set) and generates a Start condition.

Once the Start condition is sent:

- The EVF and SB bits are set by hardware with an interrupt if the ITE bit is set.

Then the master waits for a read of the I2Cn_SR1 register followed by a write in the I2Cn_DR register with the Slave address, **holding the SCL line low** (see *Figure 59* Transfer sequencing EV5).

Slave address transmission

Then the slave address is sent to the SDA line via the internal shift register.

In 7-bit addressing mode, one address byte is sent.

In 10-bit addressing mode, sending the first byte including the header sequence causes the following event:

- The EVF bit is set by hardware with interrupt generation if the ITE bit is set.

Then the master waits for a read of the I2Cn_SR1 register followed by a write in the I2Cn_DR register, **holding the SCL line low** (see *Figure 59* Transfer sequencing EV9).

Then the second address byte is sent by the interface.

After completion of this transfer (and acknowledge from the slave if the ACK bit is set):

- The EVF bit is set by hardware with interrupt generation if the ITE bit is set.

Then the master waits for a read of the I2Cn_SR2 register followed by a write in the I2Cn_CR register (for example set PE bit), **holding the SCL line low** (see *Figure 59* Transfer sequencing EV6).

Next the master must enter Receiver or Transmitter mode.

Note: In 10-bit addressing mode, to switch the master to Receiver mode, software must generate a repeated Start condition and re-send the header sequence with the least significant bit set (11110xx1).

Master Receiver

Following the address transmission and after I2Cn_SR1 and I2Cn_CR registers have been accessed, the master receives bytes from the SDA line into the I2Cn_DR register via the internal shift register. After each byte the interface generates in sequence:

- Acknowledge pulse if the ACK bit is set
- EVF and BTF bits are set by hardware with an interrupt if the ITE bit is set.

Then the interface waits for a read of the SR1 register followed by a read of the I2Cn_DR register, **holding the SCL line low** (see *Figure 59* Transfer sequencing EV7).

To close the communication: before reading the last byte from the I2Cn_DR register, set the STOP bit to generate the Stop condition. The interface goes automatically back to slave mode (M/SL bit cleared).

Note: In order to generate the non-acknowledge pulse after the last received data byte, the ACK bit must be cleared just before reading the second last data byte.

Master Transmitter

Following the address transmission and after I2Cn_SR1 register has been read, the master sends bytes from the I2Cn_DR register to the SDA line via the internal shift register.

The master waits for a read of the I2Cn_SR1 register followed by a write in the I2Cn_DR register, **holding the SCL line low** (see *Figure 59* Transfer sequencing EV8).

When the acknowledge bit is received, the interface sets:

- EVF and BTF bits with an interrupt if the ITE bit is set.

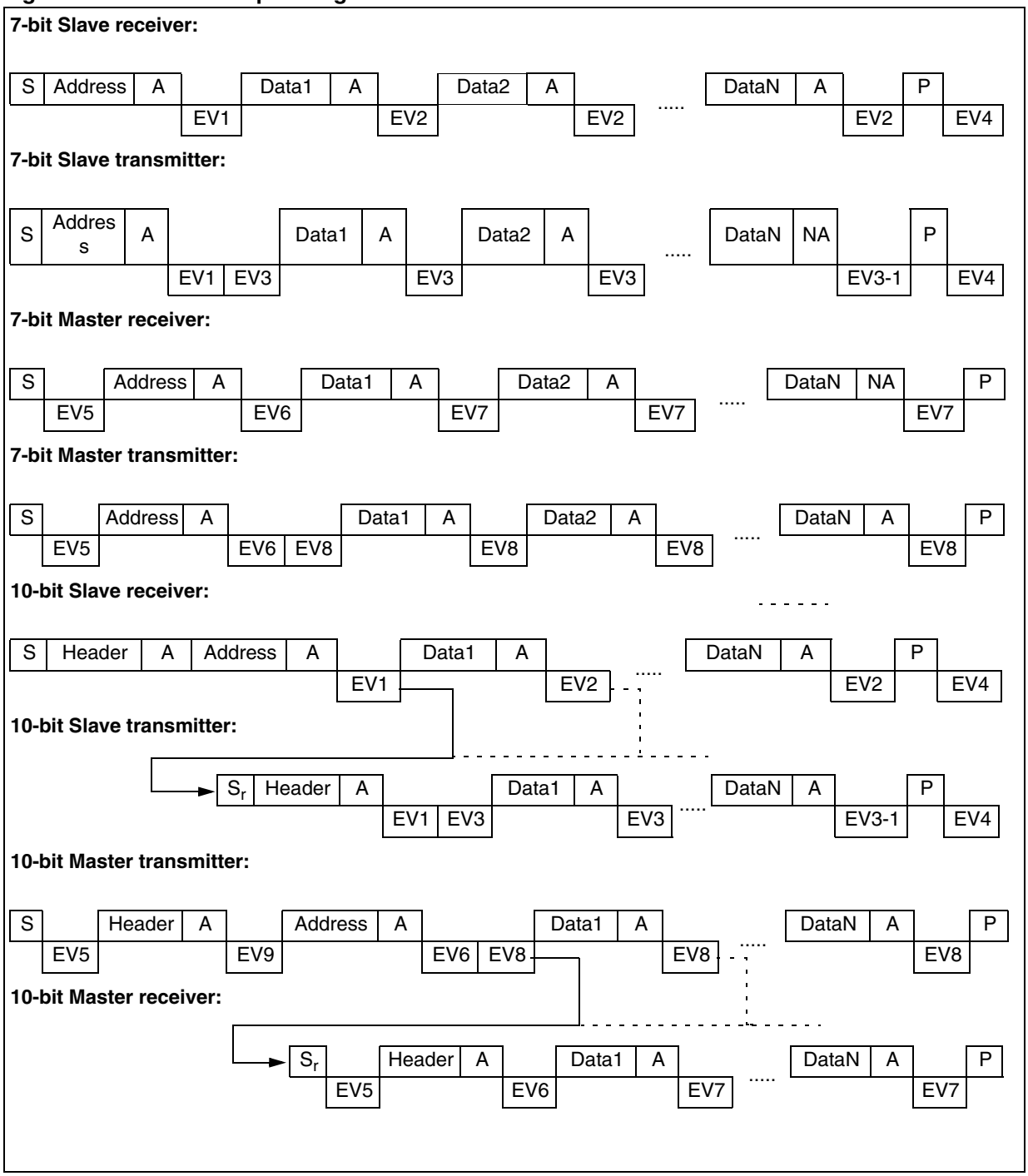
To close the communication: after writing the last byte to the I2Cn_DR register, set the STOP bit to generate the Stop condition. The interface goes automatically back to slave mode (M/SL bit cleared).

Error Cases

- **BERR**: Detection of a Stop or a Start condition during a byte transfer (on all bits). In this case, the EVF and BERR bits are set by hardware with an interrupt if ITE is set.
- **AF**: Detection of a non-acknowledge bit. In this case, the EVF and AF bits are set by hardware with an interrupt if the ITE bit is set. To resume, set the START or STOP bit.
- **ARLO:** Detection of an arbitration lost condition.
  In this case the ARLO bit is set by hardware (with an interrupt if the ITE bit is set and the interface goes automatically back to slave mode (the M/SL bit is cleared).

Note: In all these cases, the SCL line is not held low; however, the SDA line can remain low due to possible «0» bits transmitted last. It is then necessary to release both lines by software.

Figure 59. Transfer Sequencing



Legend:

S=Start, S_r = Repeated Start, P=Stop, A=Acknowledge, NA=Non-acknowledge, EVx=Event (with interrupt if ITE=1)

EV1: EVF=1, ADSL=1, cleared by reading I2Cn_SR1 register.

EV2: EVF=1, BTF=1, cleared by reading I2Cn_DR register.

EV3: EVF=1, BTF=1, cleared by reading I2Cn_SR1 register followed by writing to the DR register.

EV3-1: EVF=1, AF=1, BTF=1; AF is cleared by reading SR2 register. BTF is cleared by releasing the lines (STOP=1, STOP=0) or by writing I2Cn_DR register (DR=FFh).

Note: If lines are released bySTOP=1, STOP=0, the subsequent EV4 is not seen.

EV4: EVF=1, STOPF=1, cleared by reading SR2 register.

EV5: EVF=1, SB=1, cleared by reading I2Cn_SR1 register followed by writing I2Cn_DR register.

EV6: EVF=1, ENDAD=1 cleared by reading I2Cn_SR2 register followed by writing I2Cn_CR register (for example PE=1).

EV7: EVF=1, BTF=1, cleared by reading the I2Cn_DR register.

EV8: EVF=1, BTF=1, cleared by writing to the I2Cn_DR register.

EV9: EVF=1, ADD10=1, cleared by reading the I2Cn_SR1 register followed by writing to the I2Cn_DR register.

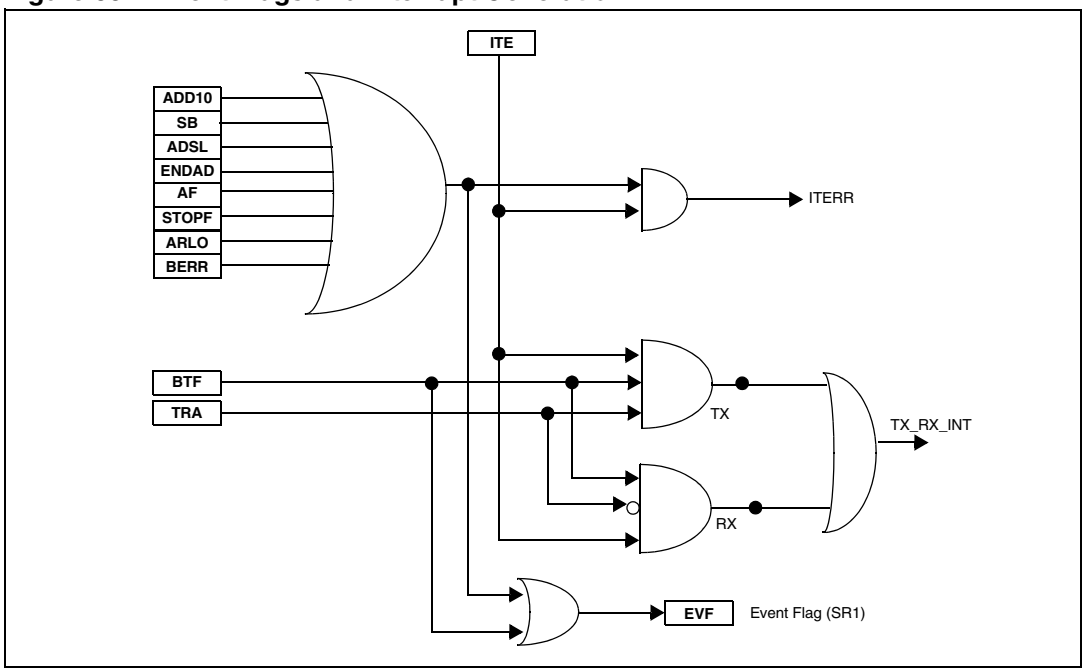
9.4 Interrupts

Several interrupt events can be flagged by the module:

- requests related to bus events, like start or stop events, arbitration lost, etc.;
- requests related to data transmission and/or reception;

These requests are issued to the interrupt controller by two different lines as described in *Figure 60*. The different flags identify the events and can be polled by the software (interrupt service routine).

Figure 60. Event Flags and Interrupt Generation



9.5 Register description

9.5.1 I²C control register (I2Cn_CR)

Address Offset: 00h
Reset value: 00h

7	6	5	4	3	2	1	0
reserved		PE	ENGC	START	ACK	STOP	ITE
-		rw	rw	rw	rw	rw	rw

Bits 7:6	Reserved, forced by hardware to '0'.
Bit 5	**PE:** *Peripheral Enable.* This bit is set and cleared by software. 0: Peripheral disabled. All the bits in the I2Cn_CR register and the I2Cn_SR register except the STOP and BUSY bit are reset. All outputs are released while PE=0. 1: Master/Slave capability enabled. The corresponding I/O pins are selected by hardware as alternate functions. **Note:** To enable the I²C interface, write the I2Cn_CR register **TWICE** with PE=1 as the first write only activates the interface (only PE is set).
Bit 4	**ENGC**: *Enable General Call.* This bit is set and cleared by software. It is also cleared by hardware when the interface is disabled (PE=0). The 00h General Call address is acknowledged (01h ignored). 0: General Call disabled. 1: General Call enabled.
Bit 3	**START**: *Generation of a Start condition.* This bit is set and cleared by software. It is also cleared by hardware when the interface is disabled (PE=0) or when the Start condition is sent (with interrupt generation if ITE=1). **In master mode:** 0: No start generation. 1: Repeated start generation. **In slave mode:** 0: No start generation. 1: Start generation when the bus is free.
Bit 2	**ACK**: *Acknowledge enable.* This bit is set and cleared by software. It is also cleared by hardware when the interface is disabled (PE=0). 0: No acknowledge returned 1: Acknowledge returned after an address byte or a data byte is received

Bit 1	**STOP**: *Generation of a Stop condition.* This bit is set and cleared by software. It is also cleared by hardware in master mode. Note: This bit is not cleared when the interface is disabled (PE=0). **In master mode:** 0: No stop generation. 1: Stop generation after the current byte transfer or after the current Start condition is sent. The STOP bit is cleared by hardware when the Stop condition is sent. **In slave mode:** 0: No stop generation. 1: Release the SCL and SDA lines after the current byte transfer (BTF=1). In this mode the STOP bit has to be cleared by software.
Bit 0	**ITE**: *Interrupt enable.* This bit is set and cleared by software and cleared by hardware when the interface is disabled (PE=0). 0: Interrupts disabled. 1: Interrupts enabled. Refer to *Figure 60* for the relationship between the events and the interrupts. SCL is held low when the ADD10, SB, BTF or ADSL flags or an EV6 event (See *Figure 59*) is detected.

9.5.2 I²C status register 1 (I2Cn_SR1)

Address Offset: 04h
Reset value: 00h

7	6	5	4	3	2	1	0
EVF	ADD10	TRA	BUSY	BTF	ADSL	M/SL	SB
r	r	r	r	r	r	r	r

Bit 7	**EVF**: *Event flag.* This bit is set by hardware as soon as an event occurs. It is cleared by software reading I2Cn_SR2 register in case of error event or as described in *Figure 59*. It is also cleared by hardware when the interface is disabled (PE=0). 0: No event 1: One of the following events has occurred: - BTF=1 (Byte received or transmitted) - ADSL=1 (Address matched in Slave mode while ACK=1) - SB=1 (Start condition generated in Master mode) - AF=1 (No acknowledge received after byte transmission) - STOPF=1 (Stop condition detected in Slave mode) - ARLO=1 (Arbitration lost in Master mode) - BERR=1 (Bus error, misplaced Start or Stop condition detected) - ADD10=1 (Master has sent header byte) - ENDAD=1 (Address byte successfully transmitted in Master mode).
Bit 6	**ADD10**: *10-bit addressing in Master mode.* This bit is set by hardware when the master has sent the first byte in 10-bit address mode. It is cleared by software reading I2Cn_SR2 register followed by a write in the I2Cn_DR register of the second address byte. It is also cleared by hardware when the peripheral is disabled (PE=0). 0: No ADD10 event occurred. 1: Master has sent first address byte (header).
Bit 5	**TRA**: *Transmitter/Receiver.* When BTF is set, TRA=1 if a data byte has been transmitted. It is cleared automatically when BTF is cleared. It is also cleared by hardware after detection of Stop condition (STOPF=1), loss of bus arbitration (ARLO=1) or when the interface is disabled (PE=0). 0: Data byte received (if BTF=1). 1: Data byte transmitted.
Bit 4	**BUSY**: *Bus busy.* This bit is set by hardware on detection of a Start condition and cleared by hardware on detection of a Stop condition. It indicates a communication in progress on the bus. This information is still updated when the interface is disabled (PE=0). 0: No communication on the bus 1: Communication ongoing on the bus

Bit 3	**BTF**: *Byte transfer finished.* This bit is set by hardware as soon as a byte is correctly received or transmitted with interrupt generation if ITE=1. It is cleared by software reading I2Cn_SR1 register followed by a read or write of I2Cn_DR register. It is also cleared by hardware when the interface is disabled (PE=0). Following a byte reception, this bit is set after transmission of the acknowledge clock pulse if ACK=1. BTF is cleared by reading I2Cn_SR1 register followed by reading the byte from I2Cn_DR register. Following a byte transmission, this bit is set after reception of the acknowledge clock pulse. In case an address byte is sent, this bit is set only after the EV6 event (See *Figure 59*). BTF is cleared by writing the next byte in I2Cn_DR register. The SCL line is held low while BTF=1. 0: Byte transfer not done 1: Byte transfer succeeded
Bit 2	**ADSL**: *Address matched (Slave mode).* This bit is set by hardware as soon as the received slave address matched with the I2Cn_OAR register content or a general call is recognized. An interrupt is generated if ITE=1. It is cleared by software reading I2Cn_SR1 register or by hardware when the interface is disabled (PE=0). The SCL line is held low while ADSL=1. 0: Address mismatched or not received. 1: Received address matched.
Bit 1	**M/SL**: *Master/Slave.* This bit is set by hardware as soon as the interface is in Master mode (writing START=1). It is cleared by hardware after detecting a Stop condition on the bus or a loss of arbitration (ARLO=1). It is also cleared when the interface is disabled (PE=0). 0: Slave mode. 1: Master mode.
Bit 0	**SB**: *Start bit (Master mode).* This bit is set by hardware as soon as the Start condition is generated (following a write START=1). An interrupt is generated if ITE=1. It is cleared by software reading I2Cn_SR1 register followed by writing the address byte in I2Cn_DR register. It is also cleared by hardware when the interface is disabled (PE=0). 0: No Start condition. 1: Start condition generated.

9.5.3 I²C status register 2 (I2Cn_SR2)

Address Offset: 08h
Reset value: 00h

7	6	5	4	3	2	1	0
Reserved		ENDAD	AF	STOPF	ARLO	BERR	GCAL
-		r	r	r	r	r	r

Bits 7:6	Reserved, forced by hardware to '0'.
Bit 5	**ENDAD**: *End of address transmission.* This bit is set by hardware when: - 7-bit addressing mode: the address byte has been transmitted; - 10-bit addressing mode: the MSB and the LSB have been transmitted during the addressing phase. When the master needs to receive data from the slave, it has to send just the MSB of the slave address once again; hence the ENDAD flag is set, without waiting for the LSB of the address.It is cleared by software by reading SR2 and a following write to the CR or by hardware when the interface is disabled (PE=0). 0: No end of address transmission 1: End of address transmission
Bit 4	**AF**: *Acknowledge failure*. This bit is set by hardware when no acknowledge is returned. An interrupt is generated if ITE=1. It is cleared by software by reading I2Cn_SR2 register or by hardware when the interface is disabled (PE=0). The SCL line is not held low while AF=1. 0: No acknowledge failure 1: Acknowledge failure
Bit 3	**STOPF**: *Stop detection (Slave mode).* This bit is set by hardware when a Stop condition is detected on the bus after an acknowledge (if ACK=1). An interrupt is generated if ITE=1. It is cleared by software reading I2Cn_SR2 register or by hardware when the interface is disabled (PE=0). The SCL line is not held low while STOPF=1. 0: No Stop condition detected 1: Stop condition detected
Bit 2	**ARLO**: *Arbitration lost*. This bit is set by hardware when the interface loses the arbitration of the bus to another master. An interrupt is generated if ITE=1. It is cleared by software reading I2Cn_SR2 register or by hardware when the interface is disabled (PE=0). After an ARLO event the interface switches back automatically to Slave mode (M/SL=0). The SCL line is not held low while ARLO=1. 0: No arbitration lost detected 1: Arbitration lost detected

Bit 1	**BERR**: *Bus error.* This bit is set by hardware when the interface detects a misplaced Start or Stop condition on all bits of a byte transfer in master mode and on the last 8 bits of a byte transfer in slave mode. An interrupt is generated if ITE=1. It is cleared by software reading I2Cn_SR2 register or by hardware when the interface is disabled (PE=0). The SCL line is not held low while BERR=1. 0: No misplaced Start or Stop condition 1: Misplaced Start or Stop condition
Bit 0	**GCAL:** *General Call (Slave mode).* This bit is set by hardware when a general call address is detected on the bus while ENGC=1. It is cleared by hardware detecting a Stop condition (STOPF=1) or when the interface is disabled (PE=0). 0: No general call address detected on bus 1: general call address detected on bus

9.5.4 I²C clock control register (I2Cn_CCR)

Address Offset: 0Ch
Reset value: 00h

7	6	5	4	3	2	1	0
FM/SM	CC6	CC5	CC4	CC3	CC2	CC1	CC0
rw	rw	rw	rw	rw	rw	rw	rw

Bit 7	**FM/SM**: *Fast/Standard I²C mode.* This bit is set and cleared by software. It is not cleared when the interface is disabled (PE=0). 0: Standard I²C mode 1: Fast I²C mode
Bits 6:0	**CC[6:0]**: *12-bit clock divider.* These bits along with CC11-CC7 of the Extended Clock Control Register select the speed of the bus (f_{SCL}) depending on the I²C mode. They are not cleared when the interface is disabled (PE=0). – Standard mode (FM/SM=0): $f_{SCL} \leq 100kHz$ $f_{SCL} = f_{PCLK1}/ (2 \times (CC[11:0] + 7))$ Given a certain f_{PCLK1} it is easy to obtain the right divider factor: $CC[11:0] = ((f_{PCLK1} / f_{SCL}) - 7) / 2 = ((t_{SCL} / t_{PCLK1}) - 7) / 2$ – Fast mode (FM/SM=1): $100kHz < f_{SCL} < 400kHz$ $f_{SCL} = f_{PCLK1}/ (3 \times ([CC11...CC0] + 9))$ Given a certain f_{PCLK} it is easy to obtain the right divider factor: $CC[11:0] = ((f_{PCLK1}/ f_{SCL}) - 9) / 3 = ((t_{SCL} / t_{PCLK1}) - 9) / 3$

Note: The programmed f_{SCL} assumes no load on SCL and SDA lines.

For a correct usage of the divider, [CC11...CC0] must be equal or greater than 0x002 (000000000010b). [CC11...CC0] equal to 0x001 (000000000001b) is not admitted.

9.5.5 I²C extended clock control register (I2Cn_ECCR)

Address Offset: 1Ch
Reset value: 00h

7	6	5	4	3	2	1	0
reserved			CC11	CC10	CC9	CC8	CC7
			rw	rw	rw	rw	rw

Bits 7:5	Reserved, forced by hardware to ‘0’.
Bit 6:0	**CC[11:7]**: *12-bit clock divider.* These bits along with those of the Clock Control Register select the speed of the bus (f_{SCL}) depending on the I²C mode. They are not cleared when the interface is disabled (PE=0)

9.5.6 I²C own address register 1 (I2Cn_OAR1)

Address Offset: 10h
Reset value: 00h

7	6	5	4	3	2	1	0
ADD7	ADD6	ADD5	ADD4	ADD3	ADD2	ADD1	ADD0
rw	rw	rw	rw	rw	rw	rw	rw

7-bit Addressing Mode

Bits 7:1	**ADD[7:1]**: *Interface address*. These bits define the I²C bus address of the interface. They are not cleared when the interface is disabled (PE=0).
Bit 0	**ADD[0]**: *Address direction bit.* This bit is don't care, the interface acknowledges either 0 or 1. It is not cleared when the interface is disabled (PE=0). Note: Address 01h is always ignored.

10-bit Addressing Mode

Bits 7:0	**ADD[7:0]**: *Interface address*. These are the least significant bits of the I²C bus address of the interface. They are not cleared when the interface is disabled (PE=0).

9.5.7 I²C own address register 2 (I2Cn_OAR2)

Address Offset: 14h
Reset value: 20h

7	6	5	4	3	2	1	0
FR2	FR1	FR0	reserved		ADD9	ADD8	res.
rw	rw	rw	-		rw	rw	-

Bits 7:5	**FR[2:0]**: *Frequency bits*. These bits are set by software only when the interface is disabled (PE=0). To configure the interface to I²C specified delays select the value corresponding to the system frequency f_{PCLK1}: 000: f_{PCLK1} = 5 to 10 MHz 001: f_{PCLK1} = 10 to 16.67 MHz 010: f_{PCLK1} = 16.67 to 26.67 MHz 011: f_{PCLK1} = 26.67 to 40 MHz 100: f_{PCLK1} = 40 to 53.33 MHz
Bits 4:3	Reserved, forced by hardware to '0'.
Bits 2:1	**ADD[9:8]**: *Interface address*. These are the most significant bits of the I²C bus address of the interface (10-bit mode only). They are not cleared when the interface is disabled (PE=0).
Bit 0	Reserved, forced by hardware to '0'.

9.5.8 I²C data register (I2Cn_DR)

Address Offset: 18h
Reset value: 00h

7	6	5	4	3	2	1	0
D7	D6	D5	D4	D3	D2	D1	D0
rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:0	**D[7:0]**: *8-bit Data Register.* These bits contain the byte to be received or transmitted on the bus. **Transmitter mode**: Byte transmission start automatically when the software writes in the I2Cn_DR register. **Receiver mode**: the first data byte is received automatically in the I2Cn_DR register using the least significant bit of the address. Then, the following data bytes are received one by one after reading the I2Cn_DR register.

9.6 I2C register map

Table 34. I²C interface register map

Address Offset	Register Name	7	6	5	4	3	2	1	0
00h	I2Cn_CR	reserved		PE	ENGC	START	ACK	STOP	ITE
04h	I2Cn_SR1	EVF	ADD10	TRA	BUSY	BTF	ADSL	M/SL	SB
08h	I2Cn_SR2	reserved		ENDAD	AF	STOPF	ARLO	BERR	GCAL
0Ch	I2Cn_CCR	FM/SM	CC6	CC5	CC4	CC3	CC2	CC1	CC0
10h	I2Cn_OAR1	ADD7	ADD6	ADD5	ADD4	ADD3	ADD2	ADD1	ADD0
14h	I2Cn_OAR2	FR2	FR1	FR0	reserved		ADD9	ADD8	res.
18h	I2Cn_DR	D7	D6	D5	D4	D3	D2	D1	D0
1Ch	I2Cn_ECCR	reserved			CC11	CC10	CC9	CC8	CC7

Refer to *Table 2 on page 13* for the register base addresses.

10 Buffered SPI (BSPI)

10.1 Introduction

A BSPI block is a standard 4-pin Serial Peripheral Interface for inter-IC control communication. It interfaces on one side to the SPI bus and on the other has a standard register data and interrupt interface.

A BSPI contains two 10-word x 16-bit FIFOs, one for receive and the other for transmit. It can directly operate with words 8 and 16 bit long and generates vectored interrupts separately for receive and transmit events.

10.2 Main features

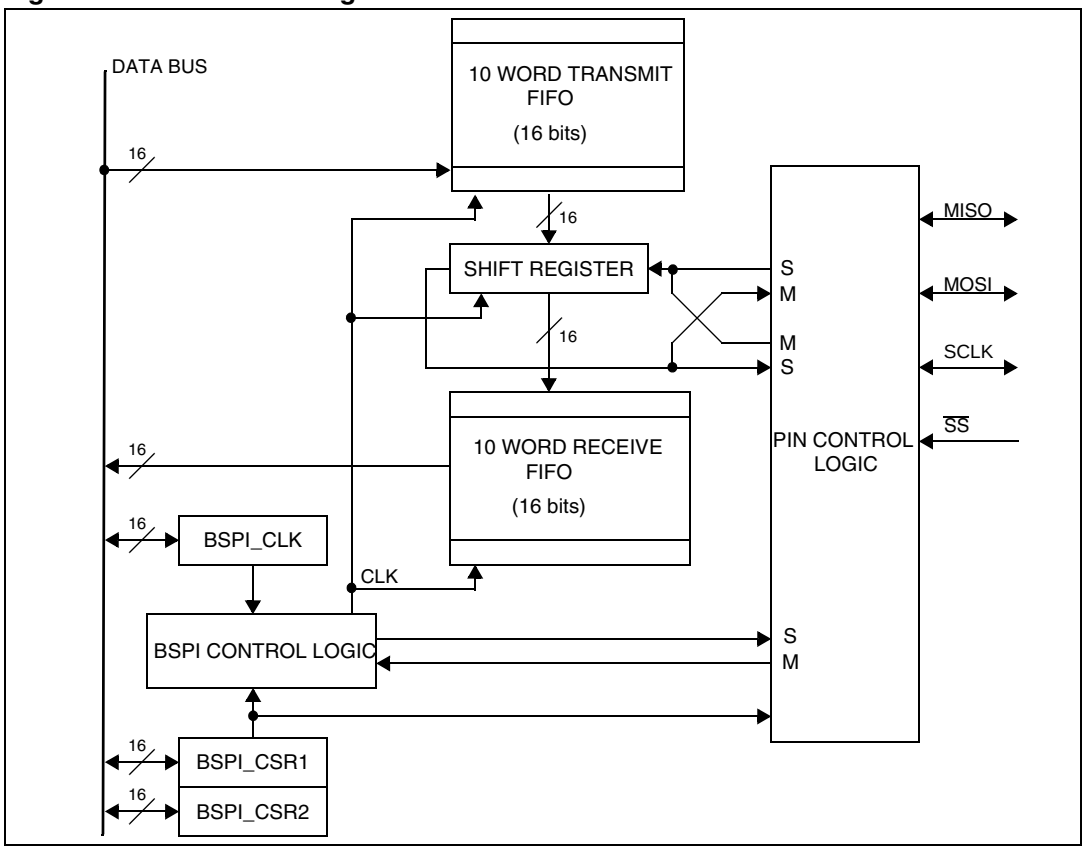
- Programmable Receive FIFO depth
- Maximum 10 word Receive FIFO
- Programmable depth Transmit FIFO
- Maximum 10 word Transmit FIFO
- Master and Slave modes supported
- Internal clock prescaler

10.3 Architecture

The processor views the BSPI as a memory mapped peripheral, which may be used by standard polling or interrupt programming techniques. Memory-mapping means processor communication can be achieved using standard instructions and addressing modes.

When an SPI transfer occurs data is transmitted and received simultaneously A serial clock line synchronizes shifting and sampling of the information on the two serial data lines. A slave select line allows individual selection of a slave device. The central elements in the BSPI system are the 16-bit shift register and the read data buffer which is 10 words x 16-bit. A block diagram of the BSPI is shown in *Figure 61 on page 187*.

Figure 61. BSPI block diagram



The BSPI is a four wire, bi-directional bus. The data path is determined by the mode of operation selected. A master and a slave mode are provided together with the associated pad control signals to control pad direction. These pins are described in *Table 35*.

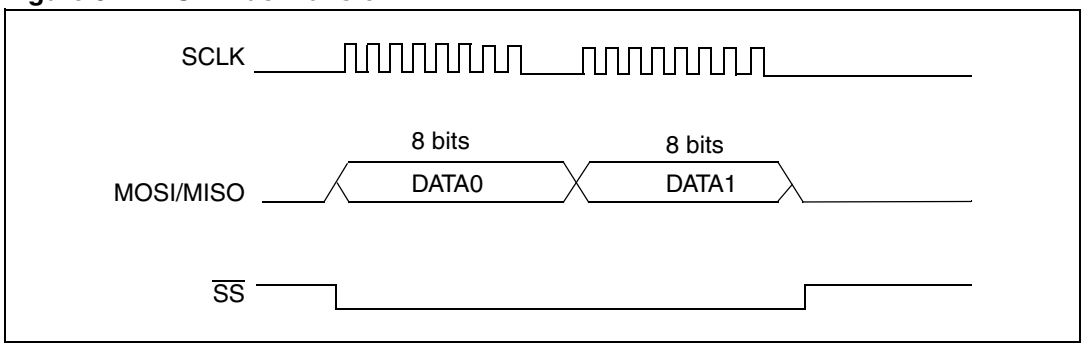
Table 35. BSPI pins

Pin Name	Description
SCLK	The bit clock for all data transfers. When the BSPI is a master the SCLK is output from the chip. When configured as a slave the SCLK is input from the external source.
MISO	Master Input/Slave Output serial data line.
MOSI	Master Output/Slave Input serial data line.
$\overline{SS}$	Slave Select. The $\overline{SS}$ input pin is used to select a slave device. Must be pulled low after the SCLK is stable and held low for the duration of the data transfer. The $\overline{SS}$ on the master must be deasserted high.

The MISO/MOSI/SCLK pins must be configured as Alternate Function push-pull and the $\overline{SS}$ Slave Select pin must be configured as Input tristate CMOS for the duration of BSPI operation. Refer to *Table 14 on page 57*.

10.4 BSPI operation

Figure 62. BSPI Bus Transfer



During a BSPI transfer (*Figure 62*), data is shifted out and shifted in (transmitted and received) simultaneously. The SCLK line synchronizes the shifting and sampling of the information. It is an output when the BSPI is configured as a master and an input when the BSPI is configured as a slave. Selection of an individual slave BSPI device is performed on the slave select line and slave devices that are not selected do not interfere with the BSPI buses.

The CPOL (clock polarity) and CPHA (clock phase) bits of the BSPIn_CSR1 register are used to select any of the four combinations of serial clock (see *Figure 63*, *Figure 64*, *Figure 65*, *Figure 66*). These bits must be the same for both the master and slave BSPI devices. The clock polarity bit selects either an active high or active low clock but does not affect transfer format. The clock phase bit selects the transfer format.

There is a 16-bit shift register which interfaces directly to the BSPI bus lines. As transmit data goes out from the register, received data fills the register.

Note: When the BSPI cell is configured in Slave mode, the SCLK_IN clock must be divided by a factor of 8 or more compared with the system clock (APB1 clock).

Figure 63. BSPI Clocking Scheme (CPOL=0, CPHA=0)

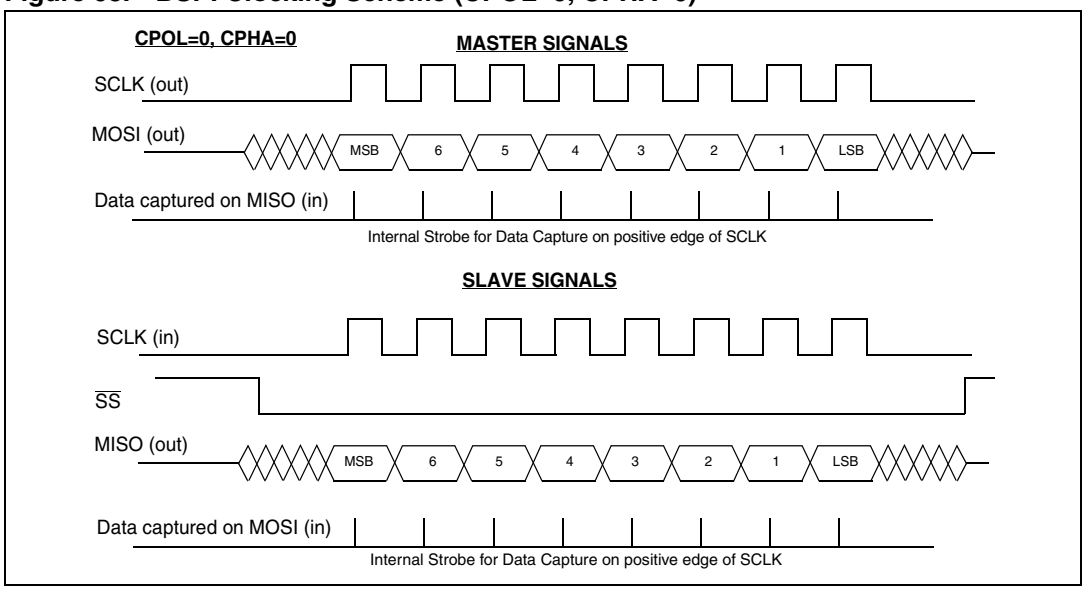


Figure 64. BSPI Clocking Scheme (CPOL=0, CPHA=1)

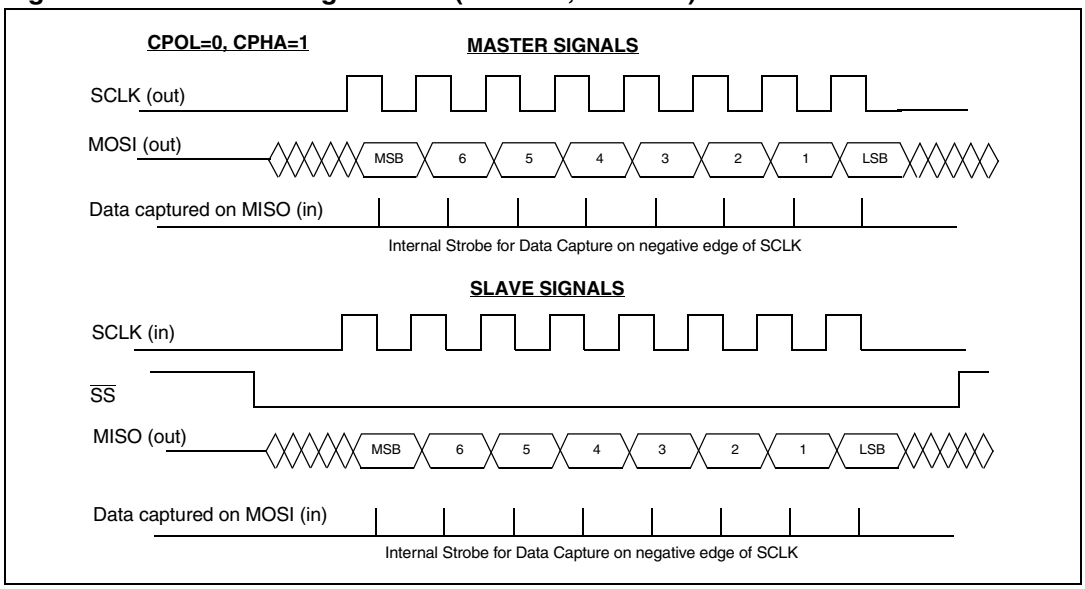


Figure 65. BSPI Clocking Scheme (CPOL=1, CPHA=0)

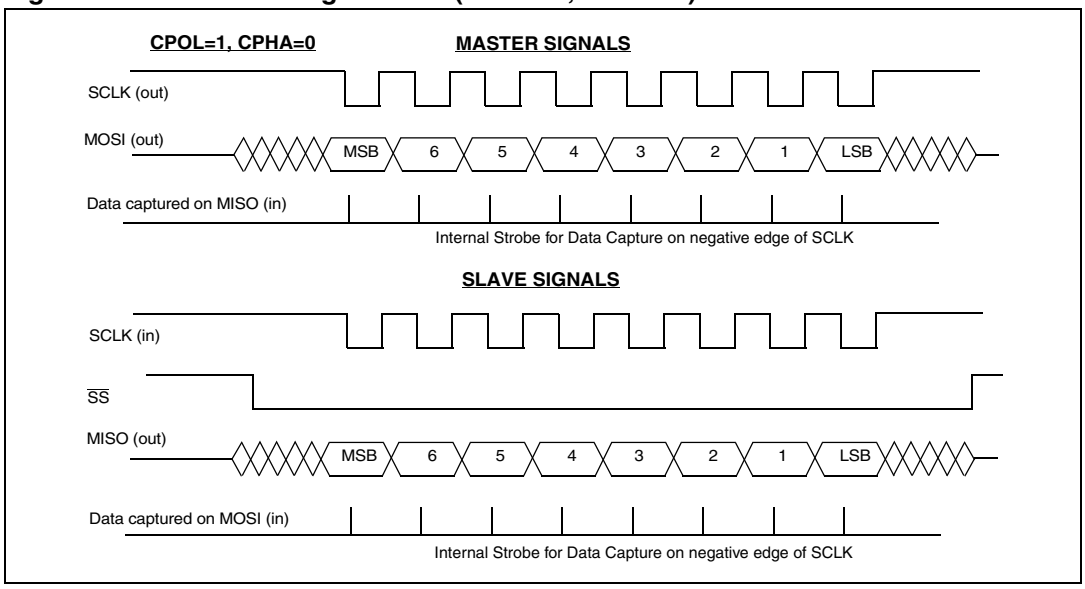
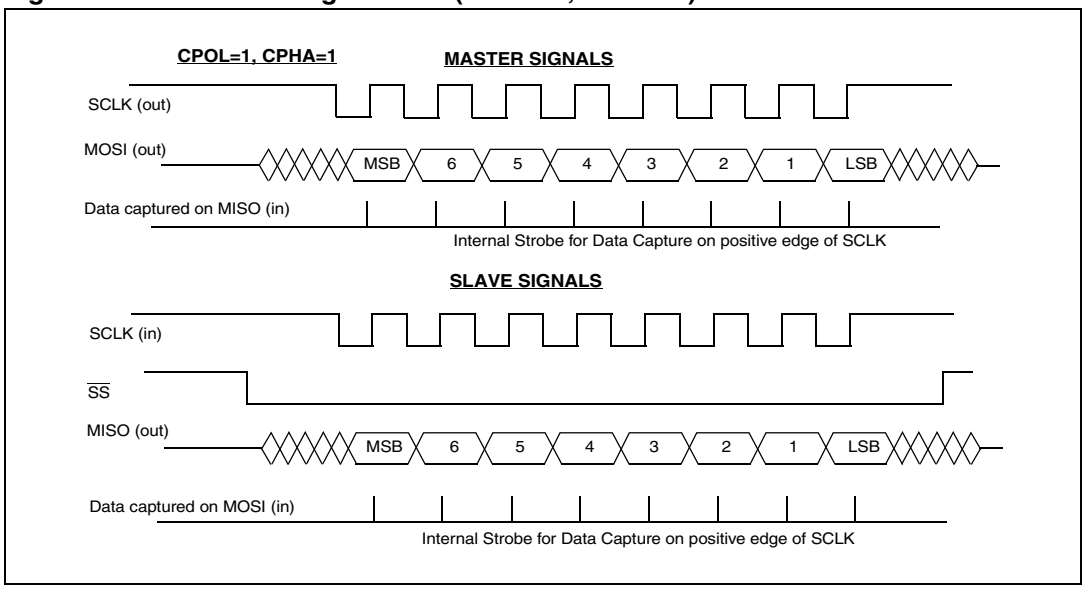


Figure 66. BSPI Clocking Scheme (CPOL=1, CPHA=1)



10.5 Transmit FIFO

The transmit FIFO consists of a 10 by 16 bit register bank which can operate in 8/16 bit modes as configured by the word length (WL[1:0]) control bits of BSPIn_CSR1. Data is left justified indicating that only the most significant portion of the word is transmitted if using 8 bit mode. After a transmission is completed the next data word is loaded from the transmit FIFO.

The user can set the depth of the FIFO from the default one location up to a maximum of ten locations. This can be set dynamically but will only take effect after the completion of the current transmission. Status flags report if the FIFO is full (TFF), the FIFO is not empty (TFNE), the FIFO is empty (TFE) and the transmit buffer has under flown (TUFL). The transmit interrupt enable (TIE[1:0]) control bits of BSPIn_CSR2 determine the source of the transmit interrupt. If the interrupt source is enabled then an active high interrupt will be asserted to the processor.

If the TUFL flag is asserted then a subsequent write to the transmit FIFO will clear the flag. If interrupts are enabled then the interrupt will be de-asserted. The TFF and TFNE flags are updated at the end of the processor write cycle and at the end of each transmission.

Note: Data should be written in the FIFO only if the macro is enabled (see BSPI System Enable bit of BSPI Control Register). If one data word is written in the transmit FIFO before enabling the BSPI, no data is transmitted.

10.6 Receive FIFO

The BSPI Receive FIFO is a 10 word by 16-bit FIFO used to buffer the data words received from the BSPI bus.

The FIFO can operate in 8-bit and 16-bit modes as configured by the WL[1:0] bits of the Control/Status Register 1 BSPIn_CSR1 . Irrelevant of the word depth in the FIFO, if operating in 8-bit mode the data will occupy the Most Significant Byte of each location of the

FIFO (data is left justified). The receive FIFO enable bits RFE[3:0] declare how many words deep the FIFO is for all transfers. The FIFO defaults to one word deep. Whenever there is at least one block of data in the FIFO the RFNE bit is set in the Control/Status register 2 BSPIn_CSR2, i.e there is data in at least one location. The RFF flag does not get set until all locations of the FIFO contain data, i.e. RFF is set when the depth of FIFO is filled and nothing has been read out.

If the FIFO is one word deep then the RFNE and RFF flags are set once data is written to it. When the data is read then both flags are cleared. A write to and a read from the FIFO can happen independent of each once RFF is not set, if RFF is set a read must occur before the next write or an overflow(ROFL) will occur.

10.7 Start-up status

If the BSPI is to operate in Master mode, the MSTR bit must be set high and then the BSPI must be enabled. The TFE flag will be set, signalling that the Transmit FIFO is empty, if the TIE is set, a TFE interrupt will be generated. The data to be transferred must be written to the Transmit Data Register BSPIn_TXR, the TFE interrupt will be cleared and then the BSPI clock will be generated according to the value of the BSPIn_CLK register. The Transfer of data then begins. A second TFE interrupt occurs so that the peripheral has a full data transfer time to request the data before the next transfer is to begin.

If the BSPI is to operate in slave mode, once again the device must be enabled. The SS line must only be asserted low after the SCLK from the master is stable. The TFE flag will be set signalling that the Transmit Data register is empty and will be cleared by a write to the Transmit Data register BSPIn_TXR. The second TFE interrupt occurs to request data for the following transfer.

10.8 Clocking problems and clearing of the shift-register

Should a problem arise on the clock which results in a misalignment of data in the shift register of the BSPI, it may be cleared by disabling the BSPI enable. This has the effect of setting the TFE which requests data to be written to the Transmit Register for the next transfer. Clearing the BSPI enable will also reset the counter of bits received. The next block of data received will be written to the next location in the FIFO continuing on from the last good transfer, if the FIFO was just one word deep it will be written to the only location available.

10.9 Interrupt control

The BSPI generates one interrupt based upon the status bits monitoring the transmit and receive logic. The interrupt is acknowledged or cleared by subsequent read or write operations which remove the error or status update condition. It is the responsibility of the programmer to ascertain the source of the interrupt and then remove the error condition or alter the state of the BSPI. In the case of multiple errors the interrupt will remain active until all interrupt sources have been cleared.

For example, in the case of TFE, whenever the last word has been transferred to the transmit buffer, the TFE flag is asserted. If interrupts are enabled then an interrupt will be asserted to the processor. To clear the interrupt the user must write at least one data word into the FIFO, or disable the interrupts if this condition is valid.

10.10 Register description

10.10.1 BSPI control/status register 1 (BSPIn_CSR1)

Address Offset: 08h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFE[3:0]				WL[1:0]		CPHA	CPOL	BEIE	res.	res.	REIE	RIE[1:0]		MSTR	BSPE
rw	rw	rw	rw	rw		rw	rw	rw	-	-	rw	rw		rw	rw

Bits 15:12	**RFE[3:0]**: *Receive FIFO Enable* The receive FIFO can be programmed to operate with a word depth up to 10. The receive FIFO enable bits declare how many words deep the FIFO is for all transfers. The FIFO defaults to one word deep, i.e. similar to a single data register. The table below shows how the FIFO is controlled. 0000: 1st word enabled 0001: 1st & 2nd words enabled 0010: 1-3 words enabled 0011: 1-4 words enabled 0100: 1-5 words enabled 0101: 1-6 words enabled 0110: 1-7 words enabled 0111: 1-8 words enabled 1000: 1-9 words enabled 1001: 1-10 words enabled 1010: Default: 1st word enabled 1011: Default: 1st word enabled 1100: Default: 1st word enabled 1101: Default: 1st word enabled 1110: Default: 1st word enabled 1111: Default: 1st word enabled
Bits 11:10	**WL[1:0]**: *Word Length* These two bits configure the word length operation of the Receive FIFO and transmit data registers as shown below: 00: 8-bit word length 01: 16-bit word length 10: Reserved 11: Reserved
Bit 9	**CPHA**: *Clock Phase Select.* Used with the CPOL bit to define the master-slave clock relationship. When CPHA=0, as soon as the SS goes low the first data sample is captured on the first edge of SCLK. When CPHA=1, the data is captured on the second edge.
Bit 8	**CPOL**: *Clock Polarity Select.* When this bit is cleared and data is not being transferred, a stable low value is present on the SCLK pin. If the bit is set the SCLK pin will idle high. This bit is used with the CPHA bit to define the master-slave clock relationship. 0: Active high clocks selected; SCLK idles low. 1: Active low clocks selected; SCLK idles high.

Bit 7	**BEIE**: *Bus Error Interrupt Enable.* When this bit is set to a 1, an interrupt will be asserted to the processor whenever a Bus Error condition occurs.
Bits 6:5	Reserved, must be kept at reset value (0).
Bit 4	**REIE**: *Receive Error Interrupt Enable.* When this bit is set to a 1 and the Receiver Overflow error condition occurs, a Receive Error Interrupt will be asserted to the processor.
Bits 3:2	**RIE[1:0]**: *BSPI Receive Interrupt Enable bits.* These bits are interrupt enable bits which configure when the processor will be interrupted on received data. The following configurations are possible 00: disabled 01:Receive FIFO Not Empty 10: Reserved 11: Receive FIFO Full
Bit 1	**MSTR**: *Master/Slave Select.* 0: BSPI is configured as a slave 1: BSPI is configured as a master
Bit 0	**BSPE**: *BSPI System Enable.* 0: BSPI system is disabled 1: BSPI system is enabled

Note: The peripheral should be enabled before selecting the interrupts to avoid spurious interrupt requests.

10.10.2 BSPI control/status register 2 (BSPIn_CSR2)

Address Offset: 0Ch
Reset value: 0040h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TIE[1:0]		TFE[3:0]				TFNE	TFF	TUFL	TFE	ROFL	RFF	RFNE	BERR	res.	DFIFO
rw		rw				r	r	r	r	r	r	r	r	-	w

Bits 15:14	**TIE[1:0]**: *BSPI Transmit Interrupt Enable.* These bits control the source of the transmit interrupt. 00: Disabled 01: Interrupt on Transmit FIFO Empty enabled 10: Interrupt on Transmit underflow enabled 11: Interrupt on Transmit FIFO Full enabled
Bits 13:10	**TFE[3:0]**: *Transmit FIFO Enable.* These bits control the depth of the transmit FIFO. The table below indicates all valid settings. 0000: 1st word enabled 0001: 1st & 2nd words enabled 0010: 1-3 words enabled 0011: 1-4 words enabled 0100: 1-5 words enabled 0101: 1-6 words enabled 0110: 1-7 words enabled 0111: 1-8 words enabled 1000: 1-9 words enabled 1001: 1-10 words enabled 1010: Default: 1st word enabled 1011: Default: 1st word enabled 1100: Default: 1st word enabled 1101: Default: 1st word enabled 1110: Default: 1st word enabled 1111: Default: 1st word enabled
Bit 9	**TFNE**: *Transmit FIFO Not Empty.* This bit is set whenever the FIFO contains at least one data word.
Bit 8	**TFF**: *Transmit FIFO Full.* TFF is set whenever the number of words written to the transmit FIFO is equal to the number of FIFO locations enabled by TFE[3:0]. The flag is set immediately after the data write is complete.

Bit 7	**TUFL**: *Transmit Underflow.* This status bit gets set if the TFE bit is set and, by the time the Transmit Data Register contents are to be transferred to the shift register for the next transmission, the processor has not yet put the data for transmission into the Transmit Data Register. TUFL is set on the first edge of the clock when CPHA = 1 and when CPHA = 0 on the assertion of $\overline{SS}$. If TIE[1:0] bits are set to “10” then, when TUFL gets set an interrupt will be asserted to the processor. **Note:** From an application point of view, it is important to be aware that the first word available after an underflow event has occurred should be ignored, as this data was loaded into the shift register before the underflow condition was flagged.
Bit 6	**TFE**: *Transmit FIFO Empty.* This bit gets set whenever the Transmit FIFO has transferred its last data word to the transmit buffer. If interrupts are enabled then an interrupt will be asserted whenever the last word has been transferred to the transmit buffer.
Bit 5	**ROFL**: *Receiver Overflow.* This bit gets set if the Receive FIFO is full and has not been read by the processor by the time another received word arrives. If the REIE bit is set then, when this bit gets set an interrupt will be asserted to the processor. This bit is cleared when a read takes place of the CSR register and the FIFO.
Bit 4	**RFF**: *Receive FIFO Full.* This status bit indicates that the number of FIFO locations, as defined by the RFE[3:0] bits, are all full, i.e. if the FIFO is 4 deep then all data has been received to all four locations. If the RIE[1:0] bits are configured as ‘11’ then, when this status bit gets set, an interrupt will be asserted to the processor. This bit is cleared when at least one data word has been read.
Bit 3	**RFNE**: *Receive FIFO Not Empty.* This status bit indicates that there is data in the Receive FIFO. It is set whenever there is at least one block of data in the FIFO i.e. for 8-bit mode 8 bits and for 16-bit mode 16 bits. If the RIE[1:0] bits are configured to ‘01’ then whenever this bit gets set an interrupt will be asserted to the processor. This bit is cleared when all valid data has been read out of the FIFO.
Bit 2	**BERR**: *Bus Error.* This status bit indicates that a Bus Error condition has occurred, i.e. that more than one device has acted as a Master simultaneously on the BSPI bus. A Bus Error condition is defined as a condition where the Slave Select line goes active low when the module is configured as a Master. This indicates contention in that more than one node on the BSPI bus is attempting to function as a Master.
Bit 1	Reserved, must be kept at reset value (0).
Bit 0	**DFIFO**: *Disable for the FIFO.* When this bit is enabled, the FIFO pointers are all reset to zero, the RFE bits are set to zero and therefore the BSPI is set to one location. The data within the FIFO is lost. This bit is reset to zero after a clock cycle.

10.10.3 BSPI master clock divider register (BSPIn_CLK)

Address Offset: 10h
Reset value: 0006h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved								DIV[7:0]							
-								rw							

Bits 15:8	Reserved, must be kept at reset value (0).
Bits 7:0	**DIV[7:0]**: *Divide factor bits.* These bits are used to control the frequency of the BSPI serial clock with relation to the APB1 clock. In master mode this number must be an even number greater than five, i.e. six is the lowest divide factor. In slave mode this number must be an even number greater than seven, i.e. eight is the lowest divide factor. These bits must be set before the BSPE or MSTR bits, i.e. before the BSPI is configured into master mode.

10.10.4 BSPI transmit register (BSPIn_TXR)

Address Offset: 04h

Reset value: n/a

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16-bit Transmission	TX[15:0]															
8-bit Transmission	TX[7:0]								N/U							
	w															

Bits 15:0	**TX[15:0]**: *Transmit data.* This register is used to write data for transmission into the BSPI. If the FIFO is enabled then data written to this register will be transferred to the FIFO before transmission. If the FIFO is disabled then the register contents are transferred directly to the shift register for transmission. In sixteen bit mode all of the register bits are used. In eight bit mode only the upper eight bits of the register are used. In both case the data is left justified,.i.e Bit[15] = MSB, Bit[0] / Bit[8] = LSB depending on the operating mode.

10.10.5 BSPI receive register (BSPIn_RXR)

Address Offset: 00h

Reset value: 0000h

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16-bit Transmission	RX[15:0]															
8-bit Transmission	RX[7:0]								N/U							
	r															

Bits 15:0	**RX[15:0]**: *Received data.* This register contains the data received from the BSPI bus. If the FIFO is disabled then the data from the shift register is placed into the receive register directly. If the FIFO is enabled then the received data is transferred into the FIFO. In sixteen bit mode all the register bits are utilised. In eight bit transmission mode only the upper eight bits of the register are used. The data is left justified in the register in both transmission modes,.i.e Bit[15] = MSB, Bit[0] / Bit[8] = LSB depending on the operating mode.

10.11 BSPI register map

A summary overview of the BSPI registers is given in the following table.

Table 36. BSPI register map

Addr. Offset	Reg. Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00h	BSPIn_RXR	RX[15:0] (*)															
04h	BSPIn_TXR	TX[15:0](*)															
08h	BSPIn_CSR1	RFE[3:0]				WL[1:0]		CPHA	CPOL	BEIE	res.	res.	REIE	RIE[1:0]		MSTR	BSPE
0Ch	BSPIn_CSR2	TIE[1:0]		TFE[3:0]				TFNE	TFF	TUFL	TFE	ROFL	RFF	RFNE	BERR	res.	DFIFO
10h	BSPIn_CLK	Unused								DIV[7:0]							

Note ** Data is left justified depending on transmission mode, BIT[15] = MSB*

See *Table 2 on page 13* for base addresses.

11 UART

11.1 Introduction

A UART interface, provides serial communication between the STR71x and other microcontrollers, microprocessors or external peripherals.

A UART supports full-duplex asynchronous communication. Eight or nine bit data transfer, parity generation, and the number of stop bits are programmable. Parity, framing, and overrun error detection are provided to increase the reliability of data transfers. Transmission and reception of data can simply be double-buffered, or 16-deep fifos may be used. For multiprocessor communications, a mechanism to distinguish the address from the data bytes is included. Testing is supported by a loop-back option. A 16-bit baud rate generator provides the UART with a separate serial clock signal.

11.2 Main features

- Full-duplex asynchronous communication
- Two internal FIFOs (16 words deep) for transmit and receive data
- 16-bit baud rate generator
- Data frames both 8 and 9 bit long
- Parity bit (even or odd) and stop bit

11.3 Functional description

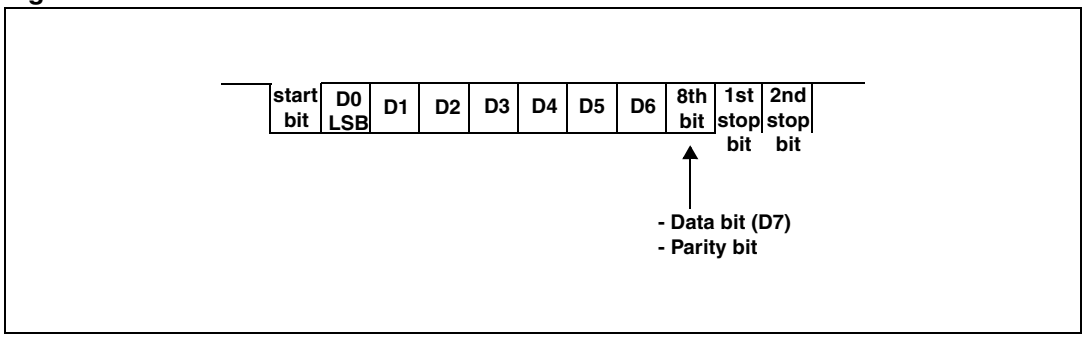
The UART supports full-duplex asynchronous communication, where both the transmitter and the receiver use the same data frame format and the same baud rate. Data is transmitted on the **TXD** pin and received on the **RXD** pin. Data frames

Eight bit data frames (see *Figure 67*) either consist of:

- eight data bits **D0-7** (by setting the **Mode** bit field to 001);
- seven data bits **D0-6** plus an automatically generated parity bit (by setting the **Mode** bit field to 011).

Parity may be odd or even, depending on the **ParityOdd** bit in the **UARTn_CR** register. An even parity bit will be set, if the modulo-2-sum of the seven data bits is 1. An odd parity bit will be cleared in this case.

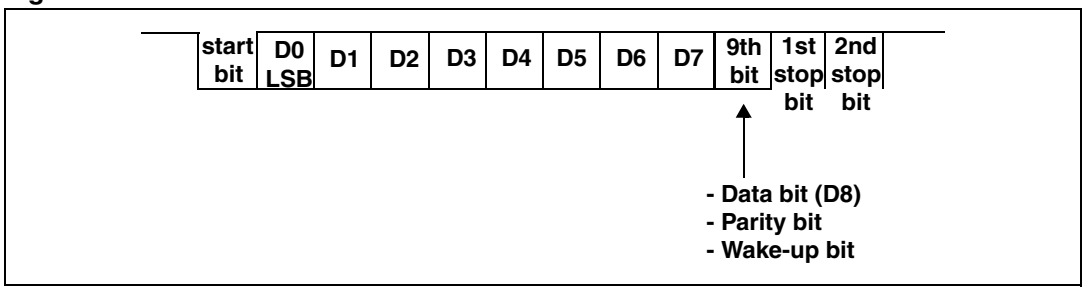
Figure 67. 8-bit data frames



Nine bit data frames (see *Figure 68 on page 199*) either consist of:

- nine data bits **D0-8** (by setting the **Mode** bit field to 100)
- eight data bits **D0-7** plus an automatically generated parity bit (by setting the **Mode** bit field to 111)
- eight data bits **D0-7** plus a wake-up bit (by setting the **Mode** bit field to 101)

Figure 68. 9-bit data frames



Parity may be odd or even, depending on the **ParityOdd** bit in the **UARTn_CR** register. An even parity bit will be set, if the modulo-2-sum of the eight data bits is 1. An odd parity bit will be cleared in this case.

In wake-up mode, received frames are only transferred to the receive buffer register if the ninth bit (the wake-up bit) is 1. If this bit is 0, no receive interrupt request will be activated and no data will be transferred.

This feature may be used to control communication in multi-processor systems. When the master processor wants to transmit a block of data to one of several slaves, it first sends out an address byte which identifies the target slave. An address byte differs from a data byte in that the additional ninth bit is a 1 for an address byte and a 0 for a data byte, so no slave will be interrupted by a data byte. An address byte will interrupt all slaves (operating in 8-bit data + wake-up bit mode), so each slave can examine the 8 least significant bits (LSBs) of the received character (the address). The addressed slave will switch to 9-bit data mode, which enables it to receive the data bytes that will be coming (with the wake-up bit cleared). The slaves that are not being addressed remain in 8-bit data + wake-up bit mode, ignoring the following data bytes.

11.3.1 Transmission

Values to be transmitted are written to the transmit fifo, TxFIFO, by writing to **UARTn_TxBUFR**. The TxFIFO is implemented as a 16 deep array of 9 bit vectors.

If the fifos are enabled (the **UARTn_CR(FifoEnable)** is set), the TxFIFO is considered full **(UARTn_SR(TxFull)** is set) when it contains 16 characters. Further writes to **UARTn_TxBUFR** in this situation will fail to overwrite the most recent entry in the TxFIFO. If the fifos are disabled, the TxFIFO is considered full **(UARTn_SR(TxFull)** is set) when it contains 1 character, and a write to **UARTn_TxBUFR** in this situation will overwrite the contents.

If the fifos are enabled, **UARTn_SR(TxHalfEmpty)** is set when the TxFIFO contains 8 or fewer characters. If the fifos are disabled, it's set when the TxFIFO is empty.

Writing anything to **UARTn_TxRSTR** empties the TxFIFO.

Values are shifted out of the bottom of the TxFIFO into a 9-bit txshift register in order to be transmitted. If the transmitter is idle (the txshift register is empty) and something is written to

the **UARTn_TxBUFR** so that the TxFIFO becomes non-empty, the txshift register is immediately loaded from the TxFIFO and transmission of the data in the txshift register begins at the next baud rate tick.

At the time the transmitter is just about to transmit the stop bits, then if the TxFIFO is non-empty, the txshift register will be immediately loaded from the TxFIFO, and transmission of this new data will begin as soon as the current stop bit period is over (i.e. the next start bit will be transmitted immediately following the current stop bit period). Thus back-to-back transmission of data can take place. If instead the TxFIFO is empty at this point, then the txshift register will become empty. **UARTn_SR(TxEmpty)** indicates whether the txshift register is empty.

After changing the FifoEnable bit, it is important to reset the FIFO to empty (by writing to the **UARTn_TxRSTR** register), since the state of the fifo pointer may be garbage.

The loop-back option (selected by the **UARTn_CR(LoopBack)** bit) internally connects the output of the transmitter shift register to the input of the receiver shift register. This may be used to test serial communication routines at an early stage without having to provide an external network.

11.3.2 Reception

Reception is initiated by a falling edge on the data input pin **(RXD)**, provided that the **UARTn_CR(Run)** and **UARTn_CR(RxEnable)** bits are set. The **RXD** pin is sampled at 16 times the rate of the selected baud rate. A majority decision of the first, second and third samples of the start bit determines the effective bit value. This avoids erroneous results that may be caused by noise.

If the detected value is not a 0 when the start bit is sampled, the receive circuit is reset and waits for the next falling edge transition at the **RXD** pin. If the start bit is valid, the receive circuit continues sampling and shifts the incoming data frame into the receive shift register. For subsequent data and parity bits, the majority decision of the seventh, eighth and ninth samples in each bit time is used to determine the effective bit value.

Note: If reception is initiated when the data input pin (RXD) is being stretched at ‘0’, a frame error is reported since the reception stage samples the initial value as a falling edge.

For 0.5 stop bits, the majority decision of the third, fourth, and fifth samples during the stop bit is used to determine the effective stop bit value.

For 1 and 2 stop bits, the majority decision of the seventh, eighth, and ninth samples during the stop bits is used to determine the effective stop bit values.

For 1.5 stop bits, the majority decision of the fifteenth, sixteenth, and seventeenth samples during the stop bits is used to determine the effective stop bit value.

The effective values received on the **RXD** pin are shifted into a 10-bit rxshift register.

The receive fifo, RxFIFO, is implemented as a 16 deep array of 10-bit vectors (each 9 down to 0). If the RxFIFO is empty, **UARTn_SR(RxBufNotEmpty)** is set to ‘0’. If the RxFIFO is not empty, a read from **UARTn_RxBUFR** will get the oldest entry in the RxFIFO. If fifos are disabled, the RxFIFO is considered full when it contains one character. **UARTn_SR(RxHalfFull)** is set when the RxFIFO contains more than 8 characters. Writing anything to **UARTn_RxRSTR** empties the RxFIFO.

As soon as the effective value of the last stop bit has been determined, the content of the rxshift register is transferred to the RxFIFO (except in wake-up mode, in which case this

happens only if the wake-up bit, bit8, is a ‘1’). The receive circuit then waits for the next start bit (falling edge transition) at the **RXD** pin.

UARTn_SR(OverrunError) is set when the RxFIFO is full and a character is loaded from the rxshift register into the RxFIFO. It is cleared when the **UARTn_RxBUFR** register is read.

The most significant bit of each RxFIFO entry (RxFIFO[x][9]) records whether or not there was a frame error when that entry was received (i.e. one of the effective stop bit values was ’0’). **UARTn_SR(FrameError)** is set when at least one of the valid entries in the RxFIFO has its MSB set.

If the mode is one where a parity bit is expected, then the bit RxFIFO[x][8] (if 8 bit data + parity mode is selected) or the bit RxFIFO[x][7] (if 7 bit data + parity mode is selected) records whether there was a parity error when that entry was received.

Note: It does not contain the parity bit that was received. **UARTn_SR(ParityError)** is set when at least one of the valid entries in the RxFIFO has bit 8 set (if 8 bit data + parity mode is selected) or bit 7 set (if 7 bit data + parity mode is selected).

After changing the fifoenable bit, it is important to reset the fifo to empty (by writing to the **UARTn_RxRSTR** register), since the state of the fifo pointers may be garbage.

Reception is stopped by clearing the **UARTn_CR(RxEnable)** bit. A currently received frame is completed including the generation of the receive status flags. Start bits that follow this frame will not be recognized.

11.3.3 Timeout mechanism

The UART contains an 8-bit timeout counter. This reloads from **UARTn_TOR** whenever one or more of the following is true

- **UARTn_RxBBUFR** is read
- The UART starts to receive a character
- **UARTn_TOR** is written to

If none of these conditions hold, the counter decrements towards 0 at every baud rate tick.

UARTn_SR(TimeoutNotEmpty) is ’1’ exactly whenever the RxFIFO is not empty and the timeout counter is zero.

UARTn_SR(TimeoutIdle) is ‘1’ exactly whenever the RxFIFO is empty and the timeout counter is zero.

The effect of this is that whenever the RxFIFO has got something in it, the timeout counter will decrement until something happens to the RxFIFO. If nothing happens, and the timeout counter reaches zero, the **UARTn_SR(TimeoutNotEmpty)** flag will be set.

When the software has emptied the RxFIFO, the timeout counter will reset and start decrementing. If no more characters arrive, when the counter reaches zero the **UARTn_SR(TimeoutIdle)** flag will be set.

11.3.4 Baud rate generation

The baud rate generator provides a clock at 16 times the baud rate, called the oversampling clock. This clock only ticks if **UARTn_CR(Run)** is set to’1’. Setting this bit to 0 will immediately freeze the state of the UART’s transmitter and receiver. This should only be done when the UART is idle.

The baud rate and the required reload value for a given baud rate can be determined by the following formulae:

Baudrate = PCLK1 / (16 * <UART_BaudRate>)

<UART_BaudRate> = PCLK1 / (16 x Baudrate)

where: <UART_BaudRate> represents the content of the **UARTn_BR** register, taken as unsigned 16-bit integer, and PCLK1 is the clock frequency of the APB1 system peripherals.

Table 37 and *Table 38* list various commonly used baud rates together with the required reload values and the deviation errors for two different PCLK1 clock frequencies (16 and 20MHz respectively).

Table 37. Baud rates with PCLK1 = 16 MHz

Baud rate	Reload value (exact)	Reload value (integer)	Reload value (hex)	Deviation error
625K	1.6	2	0002	20%
38.4K	26.042	26	001A	0.160%
19.2K	52.083	52	0034	0.160%
9600	104.167	104	0068	0.160%
4800	208.333	208	00D0	0.160%
2400	416.667	417	01A1	0.080%
1200	833.333	833	0341	0.040%
600	1666.667	1667	0683	0.020%
300	3333.333	3333	0D05	0.010%
75	13333.333	13333	3415	0.003%

Table 38. Baud rates with PCLK1 = 20 MHz

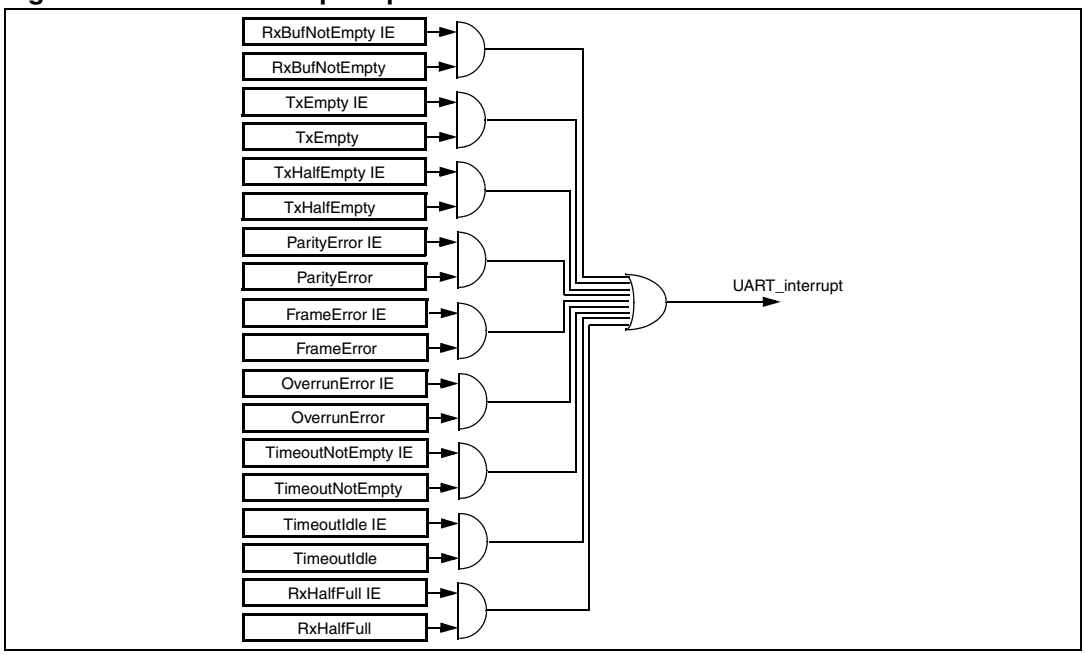
Baud rate	Reload value (exact)	Reload value (integer)	Reload value (hex)	Deviation error
625K	2	2	0002	0%
38.4K	32.552	33	0021	1.358%
19.2K	65.104	65	0041	0.160%
9600	130.208	130	0082	0.160%
4800	260.417	260	0104	0.160%
2400	520.833	521	0209	0.032%
1200	1041.667	1042	0412	0.032%
600	2083.333	2083	0823	0.016%
300	4166.667	4167	1047	0.008%
75	16666.667	16667	411B	0.002%

11.3.5 Interrupt control

The UART has a single interrupt request line, called **UARTn_interrupt**. The status bits in the **UARTn_SR** register determine the cause of the interrupt. **UARTn_interrupt** will go high when a status bit is 1 (high) and the corresponding bit in the **UARTn_IER** register is 1 (see *Figure 69*).

Note: The **UARTn_Status** register is read only. The UART_Status bits can only be cleared by operating on the FIFOs. The RxFIFO and TxFIFO can be reset by writing to the **UARTn_RxReset** and **UARTn_TxReset** registers.

Figure 69. UART interrupt request



11.3.6 Using the UART interrupts when FIFOs are disabled

When fifos are disabled, the UART provides three interrupt requests to control data exchange via the serial channel:

- **TxHalfEmpty** is activated when data is moved from **UARTn_TxBUFR** to the txshift register.
- **TxEmpty** is activated before the stop bit is transmitted.
- **RxBufNotEmpty** is activated when the received frame is moved to **UARTn_RxBUFR**.

For single transfers it is sufficient to use the transmitter interrupt **(TxEmpty)**, which indicates that the previously loaded data has been transmitted, except for the stop bit.

For multiple back-to-back transfers using **TxEmpty** would leave just one stop bit time for the handler to respond to the interrupt and initiate another transmission. Using the transmit buffer interrupt **(TxHalfEmpty)** to reload transmit data allows the time to transmit a complete frame for the service routine, as **UARTn_TxBUFR** may be reloaded while the previous data is still being transmitted.

TxHalfEmpty is an early trigger for the reload routine, while **TxEmpty** indicates the completed transmission of the data field of the frame. Therefore, software using handshake

should rely on **TxEmpty** at the end of a data block to make sure that all data has really been transmitted.

11.3.7 Using the UART interrupts when FIFOs are enabled

To transmit a large number of characters back to back, the driver routine would write 16 characters to **UARTn_TxBUFR**, then every time a **TxHalfEmpty** interrupt fired, it would write 8 more. When it had nothing more to send, a **TxEmpty** interrupt would tell it when everything has been transmitted.

When receiving, the driver could use **RxBufNotEmpty** to interrupt every time a character came in. Alternatively, if data is coming in back-to-back, it could use **RxHalfFull** to interrupt it when there were more than 8 characters in the RxFIFO to read. It would have as long as it takes to receive 8 characters to respond to this interrupt before data would overrun. If less than eight character streamed in, and no more were received for at least a timeout period, the driver could be woken up by one of the two timeout interrupts, **TimeoutNotEmpty** or **TimeoutIdle**.

11.3.8 SmartCard mode specific operation

To conform to the ISO SmartCard specification the following modes are supported in the UART SmartCard mode.

When the SmartCard mode bit is set to 0, normal UART operation occurs.
When the SmartCard mode bit is set to 1, the following operation occurs:

- Transmission of data from the transmit shift register is guaranteed to be delayed by a minimum of 1/2 baud clock. In normal operation a full transmit shift register will start shifting on the next baud clock edge. In SmartCard mode this transmission is further delayed by a guaranteed 1/2 baud clock.
- If a parity error is detected during reception of a frame programmed with a 1/2 stop bit period, the transmit line is pulled low for a baud clock period after the completion of the receive frame, i.e. at the end of the 1/2 stop bit period. This is to indicate to the SmartCard that the data transmitted to UART1 has not been correctly received.
- The assertion of the TxEmpty flag can be delayed by programming the **UART1_GTR** register. In normal operation, TxEmpty is asserted when the transmit shift register is empty and no further transmit requests are outstanding.
  In SmartCard mode an empty transmit shift register triggers the guardtime counter to count up to the programmed value in the **UART1_GTR** register. TxEmpty is forced low during this time. When the guardtime counter reaches the programmed value TxEmpty is asserted high.
  The de-assertion of TxEmpty is unaffected by SmartCard mode.

The receiver enable bit is reset after a character has been received. This avoids the receiver detecting another start bit in the case of the smartcard driving the RXD line low until the UART driver software has dealt with the previous character.

11.4 Register description

11.4.1 UART baudrate register (UARTn_BR)

Address Offset: 00h
Reset value: 0001h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BaudRate[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The **UARTn_BR** register is the dual-function baud rate generator/reload register.

A read from this register returns the content of the timer, writing to it updates the reload register.

An auto-reload of the timer with the content of the reload register is performed each time the **UARTn_BR** register is written to. However, if the **Run** bit of the **UARTn_CR** register is 0 at the time the write operation to the **UARTn_BR** register is performed, the timer will not be reloaded until the first PCLK1 clock cycle after the **Run** bit is 1.

Bits 15:0	**BaudRate[15:0]** *UART Baudrate* Write function: 16-bit reload value Read function: 16-bit count value

11.4.2 UART TxBuffer register (UARTn_TxBUFR)

Address Offset: 04h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							TX[8]	TX[7]	TX[6]	TX[5]	TX[4]	TX[3]	TX[2]	TX[1]	TX[0]
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Writing to the transmit buffer register starts data transmission.

Bits 15:9	Reserved, must be kept at reset value (0).
Bit 8	**TX[8]**: *Transmit buffer data D8.* Transmit buffer data D8, or parity bit, or wake-up bit or undefined - dependent on the operating mode (the setting of the **Mode** field in **UARTn_CR** register). **Note:** If the **Mode** field selects an 8 bit frame then this bit should be written as 0. **Note:** If the **Mode** field selects a frame with parity bit, then the TX[8] bit will contain the parity bit (automatically generated by the UART). Writing ‘0’ or ‘1’ in this bit will have no effect on the transmitted frame.
Bit 7	**TX[7]**: *Transmit buffer data D7.* Transmit buffer data D7 or parity bit - dependent on the operating mode (the setting of the **Mode** field in **UARTn_CR** register). **Note:** If the **Mode** field selects a frame with parity bit, then the TX[7] bit will contain the parity bit (automatically generated by the UART). Writing ‘0’ or ‘1’ in this bit will have no effect on the transmitted frame.
Bits 6:0	**TX[6:0]**: *Transmit buffer data D[6:0]*

11.4.3 UART RxBuffer register (UARTn_RxBUFR)

Address Offset: 08h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						RX[9]	RX[8]	RX[7]	RX[6]	RX[5]	RX[4]	RX[3]	RX[2]	RX[1]	RX[0]
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

The received data and, if provided by the selected operating mode, the received parity bit can be read from the receive buffer register.

Bits 15:10	Reserved, always read as 0.
Bit 9	**RX[9]**: *Frame error.* If set, it indicates a frame error occurred on data stored in RX[8:0] (i.e. one of the effective stop bit values was '0' when the data was received).
Bit 8	**RX[8]**: *Receive buffer data D8.* Receive buffer data D8, or parity error, or wake-up bit - dependent on the operating mode (the setting of the **Mode** field in the **UARTn_CR** register). **Note:** If the **Mode** field selects a 7- or 8-bit frame then this bit is undefined. Software should ignore this bit when reading 7- or 8-bit frames.
Bit 7	**RX[7]**: *Receive buffer data D7.* Receive buffer data D7, or parity error - dependent on the operating mode (the setting of the **Mode** field in the **UARTn_CR** register).
Bits 6:0	**RX[6:0]**: *Receive buffer data D[6:0].*

11.4.4 UART control register (UARTn_CR)

Address Offset: 0Ch
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved					Fifo Enable	SC-Enable	Rx Enable	Run	Loop Back	ParityOdd	Stop Bits		Mode		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

This register controls the operating mode of the UART and contains control bits for mode and error check selection, and status flags for error identification.

*Note: 1 Programming the mode control field **(Mode)** to one of the reserved combinations may result in unpredictable behavior.*

*2 Serial data transmission or reception is only possible when the baud rate generator run bit **(Run)** is set to 1. When the **Run** bit is set to 0, **TXD** will be 1. Setting the **Run** bit to 0 will immediately freeze the state of the transmitter and receiver. This should only be done when the UART is idle.*

Bits 15:11	Reserved, always read as 0.
Bit 10	**FifoEnable**: *FIFO Enable* 0: FIFO mode disabled 1: FIFO mode enabled
Bit 9	**SCEnable -** *Reserved to SmartCard: Mode Enable* 0: SmartCard mode disabled 1: SmartCard mode enabled Note: If SmartCard mode is not used: Must be kept at 0.
Bit 8	**RxEnable:** *Receiver Enable* 0: Receiver disabled 1: Receiver enabled
Bit 7	**Run:** *Baudrate generator Run bit* 0: Baud rate generator disabled (UART inactive) 1: Baud rate generator enabled
Bit 6	**LoopBack:** *LoopBack mode enable* 0: Standard transmit/receive mode 1: Loopback mode enabled **Note:** This bit may be modified only when the UART is inactive.
Bit 5	**ParityOdd:** *Parity selection* 0: Even parity (parity bit set on odd number of ‘1’s in data) 1: Odd parity (parity bit set on even number of ‘1’s in data)
Bits 4:3	**Stop Bits:** *Number of stop bits selection* These bits select the number of stop bits 00: 0.5 stop bits 01: 1 stop bit 10: 1.5 stop bits 11: 2 stop bits
Bits 2:0	**Mode:** *UART Mode control* 000: reserved 001: 8 bit data 010: reserved 011: 7 bit data + parity 100: 9 bit data 101: 8 bit data + wake up bit 110: reserved 111: 8 bit data + parity

11.4.5 UART IntEnable register (UARTn_IER)

Address Offset: 10h
Reset value 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED							Rx Half FullIE	Time out IdleIE	Timeout Not Empty IE	Overrun Error IE	Frame Error IE	Parity Error IE	TxHalf Empty IE	Tx Empty IE	RxBuf NotEmpty IE
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The **UARTn_IE** register enables the interrupt sources.

Interrupts will occur when a status bit in the **UARTn_SR** register is 1, and the corresponding bit in the **UARTn_IER** register is 1.

Bits 15:9	Reserved, always read as 0.
Bit 8	**RxHalfFullIE**: *Receiver buffer Half Full Interrupt Enable* 0: interrupt disabled. 1: interrupt enabled.
Bit 7	**TimeoutIdleIE**: *Timeout Idle Interrupt Enable* 0: Interrupt disabled. 1: Interrupt enabled.
Bit 6	**TimeoutNotEmptyIE**: *Timeout Not Empty Interrupt Enable* 0: Interrupt disabled. 1: Interrupt enabled.
Bit 5	**OverrunErrorIE**: *Overrun Error Interrupt Enable* 0: Interrupt disabled. 1: Interrupt enabled.
Bit 4	**FrameErrorIE**: *Framing Error Interrupt Enable* 0: Interrupt disabled. 1: Interrupt enabled.
Bit 3	**ParityErrorIE:** *Parity Error Interrupt Enable* 0: Interrupt disabled. 1: Interrupt enabled.
Bit 2	**TxHalfEmptyIE:** *Transmitter buffer Half Empty Interrupt Enable* 0: Interrupt disabled. 1: Interrupt enabled.
Bit 1	**TxEmptyIE:** *Transmitter Empty Interrupt Enable* 0: Interrupt disabled. 1: Interrupt enabled.
Bit 0	**RxBufNotEmptyIE:** *Receiver Buffer Not Empty Interrupt Enable* 0: Interrupt disabled. 1: Interrupt enabled.

11.4.6 UART status register (UARTn_SR)

Address Offset: 14h
Reset value: 0006h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						Tx Full	Rx Half Full	Timeout Idle	Timeout Not Empty	Overrun Error	Frame Error	Parity Error	Tx Half Empty	Tx Empty	Rx BufNotEmpty
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

The **UARTn_SR** register indicates the cause of an interrupt.

Bits 15:10	Reserved, always read as 0.
Bit 9	**TxFull:** *TxFIFO Full* Set when the TxFIFO contains 16 characters.
Bit 8	**RxHalfFull:** *RxFIFO Half Full* Set when the RxFIFO contains more than 8 characters.
Bit 7	**TimeoutIdle:** *Timeout Idle* Set when there is a timeout and the RxFIFO is empty
Bit 6	**TimeoutNotEmpty**: *TimeoutNotEmpty* Set when there is a timeout and the RxFIFO is not empty
Bit 5	**OverrunError:** *Overrun Error* Set when data is received and the RxFIFO is full.
Bit 4	**FrameError:** *Frame Error* Set when the RxFIFO contains something received with a frame error
Bit 3	**ParityError:** *Parity Error* Set when the RxFIFO contains something received with a parity error
Bit 2	**TxHalfEmpty:** *TxFIFO Half Empty* Set when TxFIFO at least half empty
Bit 1	***TxEmpty*:** *TxFIFO Empty* Set when transmit shift register is empty
Bit 0	**RxBufNotEmpty:** *Rx Buffer not Empty* Set when RxFIFO not empty (RxFIFO contains at least one entry)

11.4.7 UART guardtime register (UARTn_GTR)

Address Offset: 18h
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								UART_GuardTime							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

The **UARTn_GTR** register enables the user to define a programmable number of baud clocks to delay the assertion of **TxEmpty**.

Bits 15:8	Reserved, always read as 0.
Bits 7:0	**UART_GuardTime:** *Guard time value* Number of baud clocks to delay assertion of TxEmpty.

11.4.8 UART timeout register (UARTn_TOR)

Address Offset: 1Ch
Reset value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								UART_Timeout							
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

This register is to have a timeout system to be sure that not too much time passes between two successive received characters.

Bits 15:8	Reserved, always read as 0.
Bits 7:0	**UART_Timeout:** *Timeout.* Timeout period in baud rate ticks.

11.4.9 UART TxReset register (UARTn_TxRSTR)

Address Offset: 20h
Reset value: Reserved

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

A write to this register empties the TxFIFO.

11.4.10 UART RxReset register (UARTn_RxRSTR)

Address Offset: 24h
Reset Value: Reserved

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

A write to this register empties the RxFIFO.

11.5 UART register map

The following table summarizes the registers implemented in the UART.

See *Table 2 on page 13* for base addresses.

Table 39. UART peripheral register map

Addr. Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	UARTn_BR	UART_BaudRate															
4	UARTn_ TxBUFR	Reserved							UART_TxBuffer								
8	UARTn_ RxBUFR	Reserved						UART_RxBuffer									
C	UARTn_ CR	Reserved					FifoEnable	Reserved	RxEnable	Run	LoopBack	ParityOdd	Stop Bits		Mode		
10	UARTn_ IER	Reserved							UART_IntEnable								
14	UARTn_ SR	Reserved						UART_Status									
18	UARTn_ GTR	Reserved								UART_GuardTime							
1C	UARTn_ TOR	Reserved								UART_Timeout							
20	UARTn_ TxRSTR	UART_TxReset															
24	UARTn_ RxRSTR	UART_RxReset															

12 SmartCard interface (SC)

12.1 Introduction

The SmartCard Interface an extension of UART1, for the description of the UART registers, pls refer to *Section 11 on page 198*. The SmartCard interface is designed to support asynchronous protocol SmartCards as defined in the ISO7816-3 standard. UART1 configured as eight data bits plus parity, 0.5 or 1.5 stop bits, with SmartCard mode enabled provides the UART function of the SmartCard interface. A 16 bit counter, the SmartCard clock generator, divides down the PCLK1 clock to provide the clock to the SmartCard. GPIO bits in conjunction with software are used to provide the rest of the functions required to interface to the SmartCard. The inverse signalling convention as defined in ISO7816-3, inverted data and MSB first, is handled in software.

12.2 External interface

The signals required by the SmartCard are given in *Table 40*:

Table 40. SmartCard pins

Pin	Function
SCClk	Clock for SmartCard
I/O	Input or output serial data. Open drain drive at both ends.
RST	Reset to card
Vcc	Supply voltage
Vpp	Programming voltage

The signals provided by the STR71x are given in *Table 41*:

Table 41. SmartCard interface pins

Pin	Function	In/Out	Function
Port 0.12 (true open drain 5V-tolerant)	ScClk	out, open drain for 5 V cards	Clock for SmartCard.
Port0.10	ScDataOut	out, open drain driver	Serial data output. Open drain drive.
	ScDataIn	in	Serial data input.
Any GPIO port	ScRST	out, open drain	Reset to card.
	ScCmdVcc	out	Supply voltage enable/disable.
	ScCmdVpp	out	Programming voltage enable/disable.
	ScDetect	in	SmartCard detect.

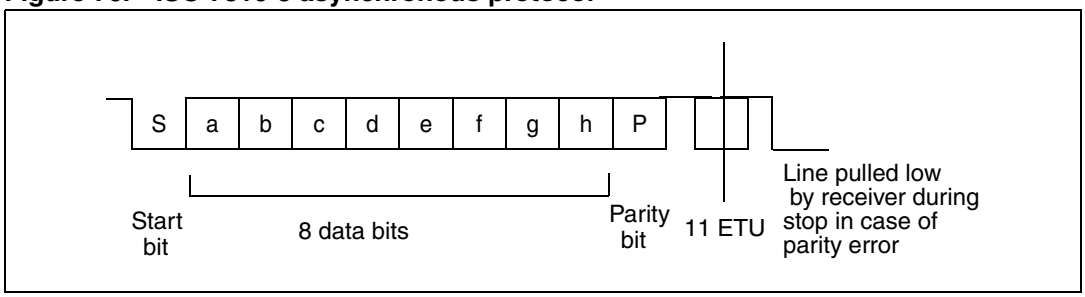
The ScRST, ScCmdVpp, ScCmdVcc, and ScDetect signals are provided by GPIO bits of the IO ports under software control. Programming the GPIO bits of the port for alternate function modes connects the UART TXD data signal to the ScDataOut pin with the correct driver type and the clock generator to the ScClk pin. Details of the GPIO bit assignments for the Alternate Function pins can be found in the STR71x pinout table.

Note: The STR71x I/Os are compatible with 3V smartcards. In the case of 5V cards, the STR71x I/Os can correctly drive a 5V input to the smartcard. The only problem could come from the SCDATA line, because it is bi-directional and the 5V logic levels (VIH/VIL) will not match. This is why only in this case an open drain buffer has to be used. The pull-up resistor (external), must be connected to VCC (5V or 3.3V depending on the type of smartcard).

12.3 Protocol

The ISO standard defines the bit times for the asynchronous protocol in terms of a time unit called an ETU which is related to the clock frequency input to the card. One bit time is of length one ETU. The UART transmitter output and receiver input need to be connected together externally. For the transmission of data from the STR71x to the SmartCard, the UART will need to be set up in SmartCard mode.

Figure 70. ISO 7816-3 asynchronous protocol



Note: The STR71xx is able to detect, via hardware, a parity error on a data byte received from the Card, however a parity error detected on a data byte received from the Reader has to be handled by the software.

12.4 SmartCard clock generator

The SmartCard clock generator provides a clock signal to the connected SmartCard. The SmartCard uses this clock to derive the baud rate clock for the serial I/O between the SmartCard and another UART. The clock is also used for the CPU in the card, if present. Operation of the Smart-Card interface requires that the clock rate to the card is adjusted while the CPU in the card is running code so that the baud rate can be changed or the performance of the card can be increased. The protocols that govern the negotiation of these clock rates and the altering of the clock rate are detailed in ISO7816-3 standard. The clock is used as the CPU clock for the SmartCard therefore updates to the clock rate must be synchronized to the clock (Clk) to the SmartCard, i.e. the clock high or low pulse widths must not be shorter than either the old or new programmed value. The clock generator clock source is the PCLK1 clock. Two registers control the period of the clock and the running of the clock.

Note: The clock generator is independent of the UART Baud rate.

12.5 Register description

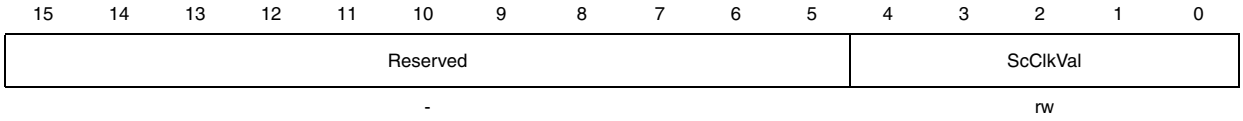
The SmartCard can be programmed via registers which are mapped into the STR71x address space. The base addresses for the SmartCard registers are given in the Memory Map chapter.

Note: During reset all of the registers are reset to 0.

12.5.1 SmartCard clock prescaler value (SC_CLKVAL)

Address Offset: 40h

Reset value: 00h



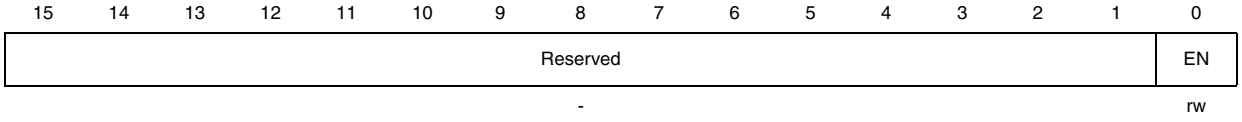
The SC_CLKVAL register determines the SmartCard clock frequency. The value given in the register is multiplied by 2 to give the division factor of the input clock frequency.

Bits 15:5	Reserved, must be kept at reset value (0).
Bits 4:0	**SCCLKVAL [4:0]** *Source Clock Divider* These bits determine the source clock divider value. This value multiplied by 2 gives the clock division factor: 00000: Reserved - DO NOT PROGRAM THIS VALUE 00001: Divides the source clock frequency by 2 00010:: Divides the source clock frequency by 4

12.5.2 SmartCard clock control register (SC_CLKCON)

Address Offset: 44h

Reset value: 00h



The SC_CLKCON register controls the source of the clock and determines whether the SmartCard clock output is enabled. The programmable divider and the output are reset when the enable bit is set to 0.

Bits 15:1	Reserved, must be kept at reset value (0).
Bit 0	**EN** *SmartCard clock generator enable bit.* 0: stop clock, set output low and reset divider 1: enable clock generator

12.6 Register map

Table 42. SmartCard Interface Register Map

Addr. Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
40h	SC_ClkVal	Reserved											ScClkVal				
44h	SC_ClkCon	Reserved															EN

See *Table 2 on page 13* for the base address

13 USB full speed device interface (USB)

13.1 Introduction

The USB Peripheral implements an interface between a full-speed USB 2.0 bus and the APB bus.

USB suspend/resume are supported which allows to stop the device clocks for low power consumption.

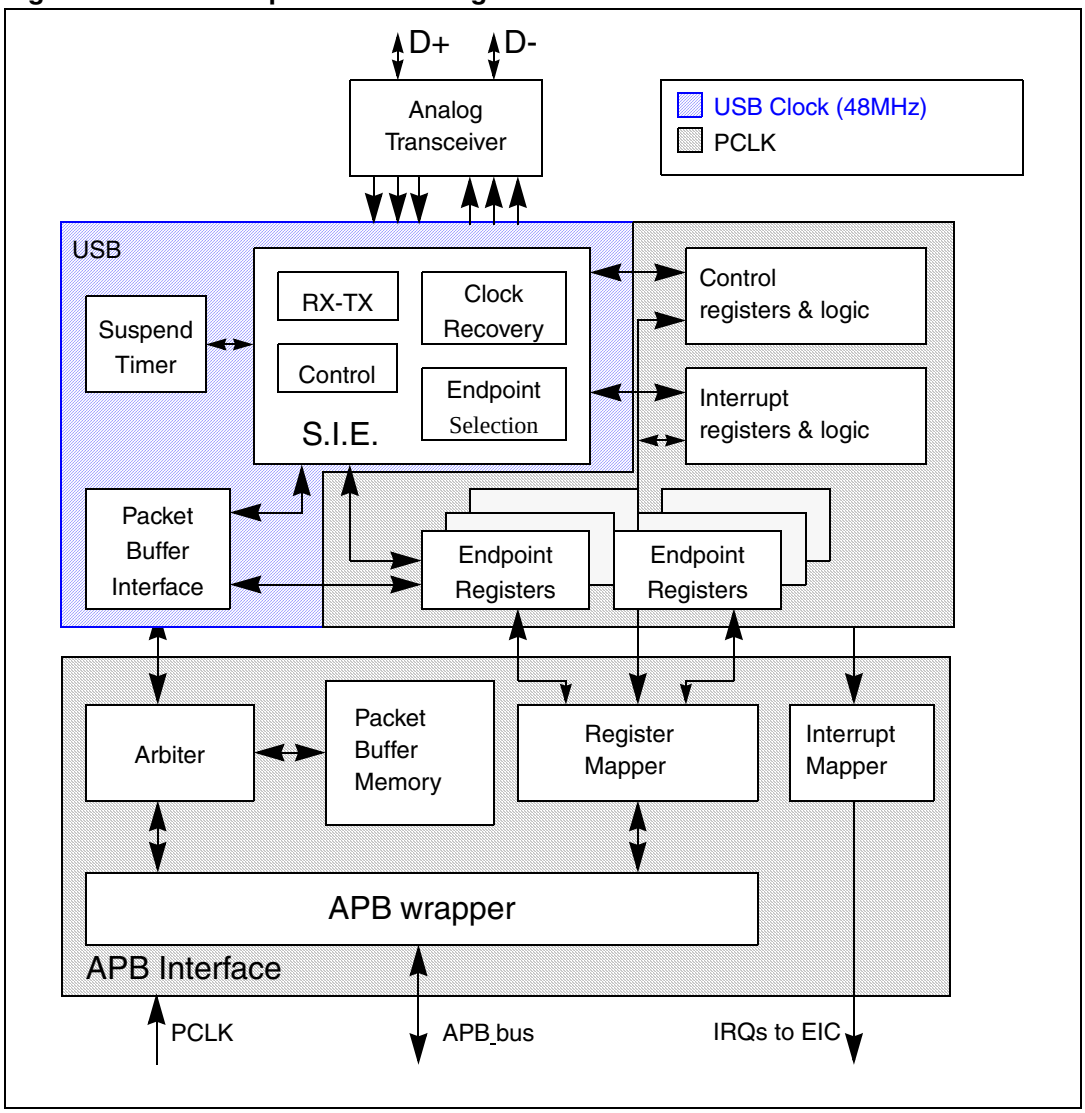
13.2 Main features

- USB specification version 2.0 Full speed compliant.
- Configurable number of endpoints from 1 to 16.
- Cyclic Redundancy Check (CRC) generation/checking, Non-Return-to-Zero Inverted (NRZI) encoding/decoding and bit-stuffing.
- Isochronous transfers support.
- Double-buffered bulk endpoint support.
- USB Suspend/Resume operations.
- Frame locked clock pulse generation.

13.3 Block diagram

Figure 71 shows the block diagram of the USB Peripheral.

Figure 71. USB Peripheral block diagram



13.4 Functional description

The USB Peripheral provides an USB compliant connection between the host PC and the function implemented by the microcontroller. Data transfer between the host PC and the system memory occurs through a dedicated packet buffer memory accessed directly by the USB Peripheral. The size of this dedicated buffer memory must be according to the number of endpoints used and the maximum packet size. This dedicated memory is sized to 512 Byte and up to 16 mono-directional/single-buffered endpoints can be used. The USB Peripheral interfaces with the USB host, detecting token packets, handling data transmission/reception, and processing handshake packets as required by the USB standard. Transaction formatting is performed by the hardware, including CRC generation and checking.

Each endpoint is associated with a buffer description block indicating where the endpoint related memory area is located, how large it is or how many bytes must be transmitted.

When a token for a valid function/endpoint pair is recognized by the USB Peripheral, the related data transfer (if required and if the endpoint is configured) takes place. The data buffered by the USB Peripheral is loaded in an internal 16 bit register and memory access to the dedicated buffer is performed. When all the data has been transferred, if needed, the proper handshake packet over the USB is generated or expected according to the direction of the transfer.

At the end of the transaction, an endpoint-specific interrupt is generated, reading status registers and/or using different interrupt response routines. The microcontroller can determine:

- which endpoint has to be served
- which type of transaction took place, if errors occurred (bit stuffing, format, CRC, protocol, missing ACK, over/underrun, etc).

Two interrupt lines are generated by the USB Peripheral : one IRQ collecting high priority endpoint interrupts (isochronous and double-buffered bulk) and another IRQ collecting all other interrupt sources (check the IRQ interrupt vector table for detailed interrupt source mapping).

Special support is offered to Isochronous transfers and high throughput bulk transfers, implementing a double buffer usage, which allows to always have an available buffer for the USB Peripheral while the microcontroller uses the other one.

The unit can be placed in low-power mode (SUSPEND mode), by writing in the control register, whenever required. At this time, all static power dissipation is avoided, and the USB clock can be slowed down or stopped. The detection of activity at the USB inputs, while in low-power mode, wakes the device up asynchronously. A special interrupt source can be connected directly to a wake-up line to allow the system to immediately restart the normal clock generation and/or support direct clock start/stop.

13.4.1 Description of USB blocks

The USB Peripheral implements all the features related to USB interfacing, which include the following blocks:

- Serial Interface Engine (SIE): The functions of this block include: synchronization pattern recognition, bit-stuffing, CRC generation and checking, PID verification/generation, and handshake evaluation. It must interface with the USB transceivers and uses the virtual buffers provided by the packet buffer interface for local data storage,. This unit also generates signals according to USB Peripheral events, such as Start of Frame (SOF), USB_Reset, Data errors etc. and to Endpoint related events like end of transmission or correct reception of a packet; these signals are then used to generate interrupts.
- Suspend Timer: This block generates the frame locked clock pulse for any external device requiring Start-of-Frame synchronization and it detects a global suspend (from the host) when no traffic has been received for 3 mS.
- Packet Buffer Interface: This block manages the local memory implementing a set of buffers in a flexible way, both for transmission and reception. It can choose the proper buffer according to requests coming from the SIE and locate them in the memory addresses pointed by the Endpoint registers. It increments the address after each exchanged word until the end of packet, keeping track of the number of exchanged bytes and preventing the buffer to overrun the maximum capacity.
- Endpoint-Related Registers: Each endpoint has an associated register containing the endpoint type and its current status. For mono-directional/single-buffer endpoints, a

single register can be used to implement two distinct endpoints. The number of registers is 16, allowing up to 16 double-buffer endpoints or up to 16 mono-directional/single-buffer ones in any combination. For example the USB Peripheral can be programmed to have 4 doublebuffer endpoints and 8 single-buffer/mono-directional endpoints.

- Control Registers: These are the registers containing information about the status of the whole USB Peripheral and used to force some USB events, such as resume and power-down.
- Interrupt Registers: These contain the Interrupt masks and a record of the events. They can be used to inquire an interrupt reason, the interrupt status or to clear the status of a pending interrupt.

The USB Peripheral is connected to the APB bus through an APB interface, containing the following blocks:

- Packet Memory: This is the local memory that physically contains the Packet Buffers. It can be used by the Packet Buffer interface, which creates the data structure and can be accessed directly by the application software. The size of the Packet Memory is 512 Bytes, structured as 256 words by 16 bits.
- Arbiter: This block accepts memory requests coming from the APB bus and from the USB interface. It resolves the conflicts by giving priority to APB accesses, while always reserving half of the memory bandwidth to complete all USB transfers. This time-duplex scheme implements a virtual dual-port RAM that allows memory access, while an USB transaction is happening. Multi-word APB transfers of any length are also allowed by this scheme.
- Register Mapper: This block collects the various byte-wide and bit-wide registers of the USB Peripheral in a structured 16-bit wide word set addressed by the APB.
- Interrupt Mapper: This block is used to select how the possible USB events can generate interrupts and map them to IRQ lines of the EIC.
- APB Wrapper: This provides an interface to the APB for the memory and register. It also maps the whole USB Peripheral in the APB address space.

13.5 Programming considerations

In the following sections, the expected interactions between the USB Peripheral and the application program are described, in order to ease application software development.

13.5.1 Generic USB device programming

This part describes the main tasks required of the application software in order to obtain USB compliant behaviour. The actions related to the most general USB events are taken into account and paragraphs are dedicated to the special cases of double-buffered endpoints and Isochronous transfers. Apart from system reset, action is always initiated by the USB Peripheral, driven by one of the USB events described below.

13.5.2 System and power-on reset

Upon system and power-on reset, the first operation the application software should perform is to provide all required clock signals to the USB Peripheral and subsequently de-assert its reset signal so to be able to access its registers. The whole initialization sequence is hereafter described.

As a first step application software needs to activate register macrocell clock and de-assert macrocell specific reset signal using related control bits provided by device clock management logic.

After that the analog part of the device related to the USB transceiver must be switched on using the PDWN bit in CNTR register which requires a special handling. This bit is intended to switch on the internal voltage references supplying the port transceiver . Since this circuit has a defined start-up time ($t_{STARTUP}$) of about 1 µs, during which the behaviour of USB transceiver is not defined, it is necessary to wait this time, after having set the PDWN bit in CNTR register, then the reset condition on the USB part can be removed (clearing of FRES bit in CNTR register) and the ISTR register can be cleared, removing any spurious pending interrupt, before enabling any other macrocell operation.

As a last step the USB specific 48 MHz clock needs to be activated, using the related control bits provided by device clock management logic.

At system reset, the microcontroller must initialize all required registers and the packet buffer description table, to make the USB Peripheral able to properly generate interrupts and data transfers. All registers not specific to any endpoint must be initialized according to the needs of application software (choice of enabled interrupts, chosen address of packet buffers, etc.). Then the process continues as for the USB reset case (see further paragraph).

USB reset (RESET interrupt)

When this event occurs, the USB Peripheral is put in the same conditions it is left by the system reset after the initialization described in the previous paragraph: communication is disabled in all endpoint registers (the USB Peripheral will not respond to any packet). As a response to the USB reset event, the USB function must be enabled, having as USB address 0, implementing only the default control endpoint (endpoint address is 0 too). This is accomplished by setting the Enable Function (EF) bit of the USB_DADDR register and initializing the EP0R register and its related packet buffers accordingly. During USB enumeration process, the host assigns a unique address to this device, which must be written in the ADD[6:0] bits of the USB_DADDR register, and configures any other necessary endpoint.
When a RESET interrupt is received, the application software is responsible to enable again the default endpoint of USB function 0 within 10mS from the end of reset sequence which triggered the interrupt.

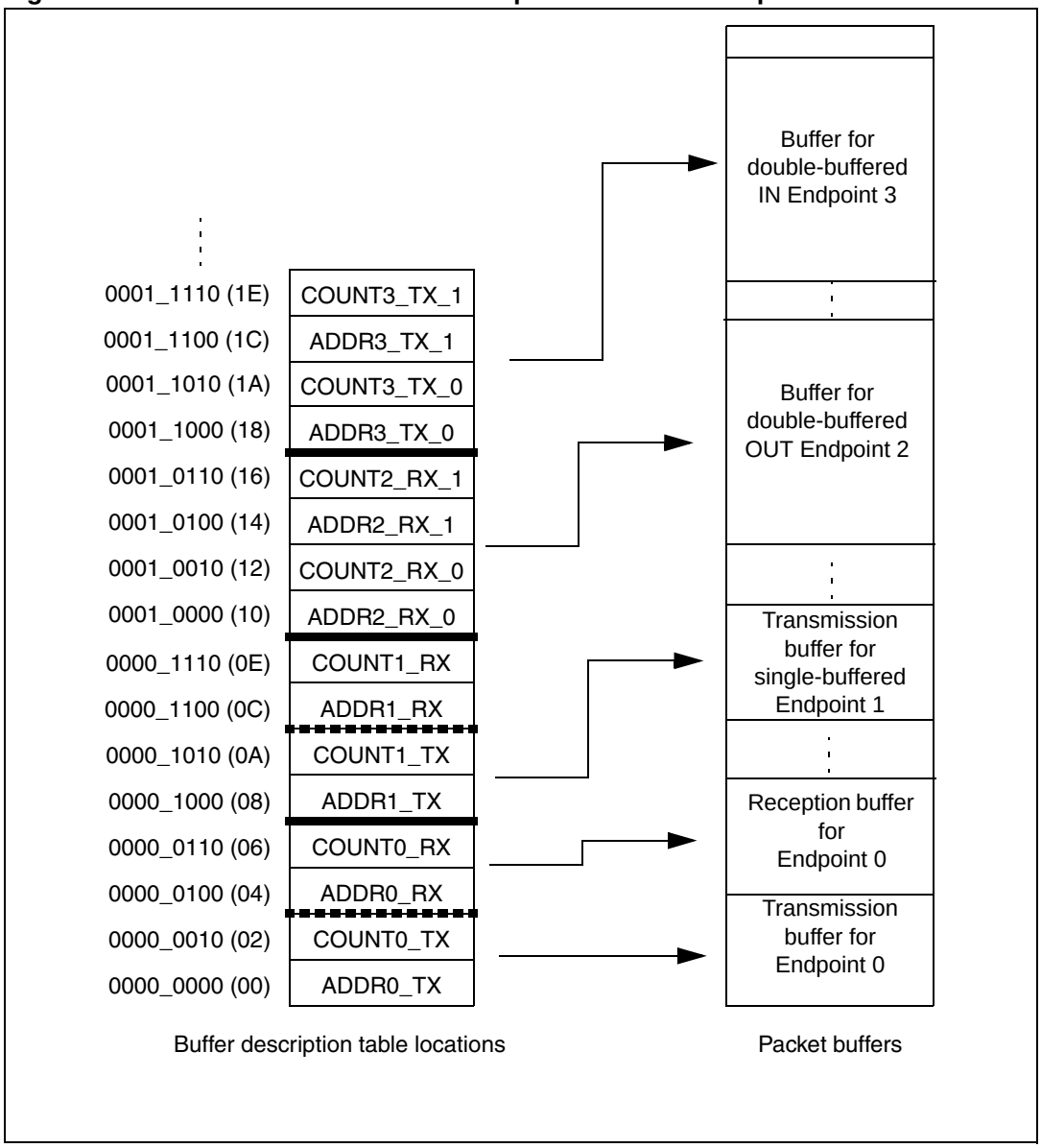
Structure and usage of packet buffers

Each bidirectional endpoint may receive or transmit data from/to the host. The received data is stored in a dedicated memory buffer reserved for that endpoint, while another memory buffer contains the data to be transmitted by the endpoint. Access to this memory is performed by the packet buffer interface block, which delivers a memory access request and waits for its acknowledgement. Since the packet buffer memory has to be accessed by the microcontroller also, an arbitration logic takes care of the access conflicts, using half APB cycle for microcontroller access and the remaining half for the USB Peripheral access. In this way, both the agents can operate as if the packet memory is a dual-port RAM, without being aware of any conflict even when the microcontroller is performing back-to-back accesses. The USB Peripheral logic uses a dedicated clock. The frequency of this dedicated clock is fixed by the requirements of the USB standard at 48 MHz, and this can be different from the clock used for the interface to the APB bus. Different clock configurations are possible where the APB clock frequency can be higher or lower than the USB Peripheral one.

Note: Due to USB data rate and packet memory interface requirements, the APB clock frequency must be greater than 8 MHz to avoid data overrun/underrun problems.

Each endpoint is associated with two packet buffers (usually one for transmission and the other one for reception). The size of the buffer can be upto 512 words each. Buffers can be placed anywhere inside the packet memory because their location and size is specified in a buffer description table, which is also located in the packet memory at the address indicated by the USB_BTABLE register. Each table entry is associated to an endpoint register and it is composed of four 16-bit words so that table start address must always be aligned to an 8-byte boundary (the lowest three bits of USB_BTABLE register are always “000”). Buffer descriptor table entries are described in the *Section 13.6.3: Buffer descriptor table*. If an endpoint is unidirectional and it is neither an Isochronous nor a double-buffered bulk, only one packet buffer is required (the one related to the supported transfer direction). Other table locations related to unsupported transfer directions or unused endpoints, are available to the user. isochronous and double-buffered bulk endpoints have special handling of packet buffers (Refer to *Section 13.5.4: Isochronous transfers* and *Section 13.5.3: Double-buffered endpoints* respectively). The relationship between buffer description table entries and packet buffer areas is depicted in *Figure 72*.

Figure 72. Packet buffer areas with examples of buffer description table locations



Each packet buffer is used either during reception or transmission starting from the bottom. The USB Peripheral will never change the contents of memory locations adjacent to the allocated memory buffers; if a packet bigger than the allocated buffer length is received (buffer overrun condition) the data will be copied to the memory only up to the last available location.

Endpoint initialization

The first step to initialize an endpoint is to write appropriate values to the ADDRn_TX/ADDRn_RX registers so that the USB Peripheral finds the data to be transmitted already available and the data to be received can be buffered. The EP_TYPE bits in the USB_EPnR register must be set according to the endpoint type, eventually using the EP_KIND bit to enable any special required feature. On the transmit side, the endpoint must be enabled using the STAT_TX bits in the USB_EPnR register and COUNTn_TX must be initialized. For reception, STAT_RX bits must be set to enable reception and COUNTn_RX must be written with the allocated buffer size using the BL_SIZE and NUM_BLOCK fields. Unidirectional endpoints, except Isochronous and double-buffered bulk endpoints, need to initialize only bits and registers related to the supported direction. Once the transmission and/or reception are enabled, register USB_EPnR and locations ADDRn_TX/ADDRn_RX, COUNTn_TX/COUNTn_RX (respectively), should not be modified by the application software, as the hardware can change their value on the fly. When the data transfer operation is completed, notified by a CTR interrupt event, they can be accessed again to re-enable a new operation.

IN packets (data transmission)

When receiving an IN token packet, if the received address matches a configured and valid endpoint one, the USB Peripheral accesses the contents of ADDRn_TX and COUNTn_TX locations inside buffer descriptor table entry related to the addressed endpoint. The content of these locations is stored in its internal 16 bit registers ADDR and COUNT (not accessible by software). The packet memory is accessed again to read the first word to be transmitted (Refer to *Structure and usage of packet buffers*) and starts sending a DATA0 or DATA1 PID according to USB_EPnR bit DTOG_TX. When the PID is completed, the first byte from the word, read from buffer memory, is loaded into the output shift register to be transmitted on the USB bus. After the last data byte is transmitted, the computed CRC is sent. If the addressed endpoint is not valid, a NAK or STALL handshake packet is sent instead of the data packet, according to STAT_TX bits in the USB_EPnR register.

The ADDR internal register is used as a pointer to the current buffer memory location while COUNT is used to count the number of remaining bytes to be transmitted. Each word read from the packet buffer memory is transmitted over the USB bus starting from the least significant byte. Transmission buffer memory is read starting from the address pointed by ADDRn_TX for COUNTn_TX/2 words. If a transmitted packet is composed of an odd number of bytes, only the lower half of the last word accessed will be used.

On receiving the ACK receipt by the host, the USB_EPnR register is updated in the following way: DTOG_TX bit is toggled, the endpoint is made invalid by setting STAT_TX=10 (NAK) and bit CTR_TX is set. The application software must first identify the endpoint, which is requesting microcontroller attention by examining the EP_ID and DIR bits in the USB_ISTR register. Servicing of the CTR_TX event starts clearing the interrupt bit; the application software then prepares another buffer full of data to be sent, updates the COUNTn_TX table location with the number of byte to be transmitted during the next transfer, and finally sets STAT_TX to ‘11’ (VALID) to re-enable transmissions. While the STAT_TX bits are equal to ‘10’ (NAK), any IN request addressed to that endpoint is NAKed, indicating a flow control condition: the USB host will retry the transaction until it succeeds. It is mandatory to execute the sequence of operations in the above mentioned order to avoid losing the notification of a second IN transaction addressed to the same endpoint immediately following the one which triggered the CTR interrupt.

OUT and SETUP packets (data reception)

These two tokens are handled by the USB Peripheral more or less in the same way; the differences in the handling of SETUP packets are detailed in the following paragraph about control transfers. When receiving an OUT/SETUP PID, if the address matches a valid endpoint, the USB Peripheral accesses the contents of the ADDRn_RX and COUNTn_RX locations inside the buffer descriptor table entry related to the addressed endpoint. The content of the ADDRn_RX is stored directly in its internal register ADDR. While COUNT is now reset and the values of BL_SIZE and NUM_BLOCK bit fields, which are read within COUNTn_RX content are used to initialize BUF_COUNT, an internal 16 bit counter, which is used to check the buffer overrun condition (all these internal registers are not accessible by software). Data bytes subsequently received by the USB Peripheral are packed in words (the first byte received is stored as least significant byte) and then transferred to the packet buffer starting from the address contained in the internal ADDR register while BUF_COUNT is decremented and COUNT is incremented at each byte transfer. When the end of DATA packet is detected, the correctness of the received CRC is tested and only if no errors occurred during the reception, an ACK handshake packet is sent back to the transmitting host. In case of wrong CRC or other kinds of errors (bit-stuff violations, frame errors, etc.), data bytes are anyways copied in the packet memory buffer, at least until the error detection point, but ACK packet is not sent and the ERR bit in USB_ISTR register is set. However, there is usually no software action required in this case: the USB Peripheral recovers from reception errors and remains ready for the next transaction to come. If the addressed endpoint is not valid, a NAK or STALL handshake packet is sent instead of the ACK, according to bits STAT_RX in the USB_EPnR register and no data is written in the reception memory buffers.

Reception memory buffer locations are written starting from the address contained in the ADDRn_RX for a number of bytes corresponding to the received data packet length, CRC included (i.e. data payload length + 2), or up to the last allocated memory location, as defined by BL_SIZE and NUM_BLOCK, whichever comes first. In this way, the USB Peripheral never writes beyond the end of the allocated reception memory buffer area. If the length of the data packet payload (actual number of bytes used by the application) is greater than the allocated buffer, the USB Peripheral detects a buffer overrun condition. in this case, a STALL handshake is sent instead of the usual ACK to notify the problem to the host, no interrupt is generated and the transaction is considered failed.

When the transaction is completed correctly, by sending the ACK handshake packet, the internal COUNT register is copied back in the COUNTn_RX location inside the buffer description table entry, leaving unaffected BL_SIZE and NUM_BLOCK fields, which normally do not require to be re-written, and the USB_EPnR register is updated in the following way: DTOG_RX bit is toggled, the endpoint is made invalid by setting STAT_RX = ‘10’ (NAK) and bit CTR_RX is set. If the transaction has failed due to errors or buffer overrun condition, none of the previously listed actions take place. The application software must first identify the endpoint, which is requesting microcontroller attention by examining the EP_ID and DIR bits in the USB_ISTR register. The CTR_RX event is serviced by first determining the transaction type (SETUP bit in the USB_EPnR register); the application software must clear the interrupt flag bit and get the number of received bytes reading the COUNTn_RX location inside the buffer description table entry related to the endpoint being processed. After the received data is processed, the application software should set the STAT_RX bits to ‘11’ (Valid) in the USB_EPnR, enabling further transactions. While the STAT_RX bits are equal to ‘10’ (NAK), any OUT request addressed to that endpoint is NAKed, indicating a flow control condition: the USB host will retry the transaction until it succeeds. It is mandatory to execute the sequence of operations in the above mentioned

order to avoid losing the notification of a second OUT transaction addressed to the same endpoint following immediately the one which triggered the CTR interrupt.

Control transfers

Control transfers are made of a SETUP transaction, followed by zero or more data stages, all of the same direction, followed by a status stage (a zero-byte transfer in the opposite direction). SETUP transactions are handled by control endpoints only and are very similar to OUT ones (data reception) except that the values of DTOG_TX and DTOG_RX bits of the addressed endpoint registers are set to 1 and 0 respectively, to initialize the control transfer, and both STAT_TX and STAT_RX are set to '10' (NAK) to let software decide if subsequent transactions must be IN or OUT depending on the SETUP contents. A control endpoint must check SETUP bit in the USB_EPnR register at each CTR_RX event to distinguish normal OUT transactions from SETUP ones. A USB device can determine the number and direction of data stages by interpreting the data transferred in the SETUP stage, and is required to STALL the transaction in the case of errors. To do so, at all data stages before the last, the unused direction should be set to STALL, so that, if the host reverses the transfer direction too soon, it gets a STALL as a status stage. While enabling the last data stage, the opposite direction should be set to NAK, so that, if the host reverses the transfer direction (to perform the status stage) immediately, it is kept waiting for the completion of the control operation. If the control operation completes successfully, the software will change NAK to VALID, otherwise to STALL. At the same time, if the status stage will be an OUT, the STATUS_OUT (EP_KIND in the USB_EPnR register) bit should be set, so that an error is generated if a status transaction is performed with not-zero data. When the status transaction is serviced, the application clears the STATUS_OUT bit and sets STAT_RX to VALID (to accept a new command) and STAT_TX to NAK (to delay a possible status stage immediately following the next setup).

Since the USB specification states that a SETUP packet cannot be answered with a handshake different from ACK, eventually aborting a previously issued command to start the new one, the USB logic doesn't allow a control endpoint to answer with a NAK or STALL packet to a SETUP token received from the host.

When the STAT_RX bits are set to '01' (STALL) or '10' (NAK) and a SETUP token is received, the USB accepts the data, performing the required data transfers and sends back an ACK handshake. If that endpoint has a previously issued CTR_RX request not yet acknowledged by the application (i.e. CTR_RX bit is still set from a previously completed reception), the USB discards the SETUP transaction and does not answer with any handshake packet regardless of its state, simulating a reception error and forcing the host to send the SETUP token again. This is done to avoid losing the notification of a SETUP transaction addressed to the same endpoint immediately following the transaction, which triggered the CTR_RX interrupt.

13.5.3 Double-buffered endpoints

All different endpoint types defined by the USB standard represent different traffic models, and describe the typical requirements of different kind of data transfer operations. When large portions of data are to be transferred between the host PC and the USB function, the bulk endpoint type is the most suited model. This is because the host schedules bulk transactions so as to fill all the available bandwidth in the frame, maximizing the actual transfer rate as long as the USB function is ready to handle a bulk transaction addressed to it. If the USB function is still busy with the previous transaction when the next one arrives, it will answer with a NAK handshake and the host PC will issue the same transaction again until the USB function is ready to handle it, reducing the actual transfer rate due to the bandwidth occupied by re-transmissions. For this reason, a dedicated feature called 'double-buffering' can be used with bulk endpoints.

When 'double-buffering' is activated, data toggle sequencing is used to select, which buffer is to be used by the USB Peripheral to perform the required data transfers, using both 'transmission' and 'reception' packet memory areas to manage buffer swapping on each successful transaction in order to always have a complete buffer to be used by the application, while the USB Peripheral fills the other one. For example, during an OUT transaction directed to a 'reception' double-buffered bulk endpoint, while one buffer is being filled with new data coming from the USB host, the other one is available for the microcontroller software usage (the same would happen with a 'transmission' double-buffered bulk endpoint and an IN transaction).

Since the swapped buffer management requires the usage of all 4 buffer description table locations hosting the address pointer and the length of the allocated memory buffers, the USB_EPnR registers used to implement double-buffered bulk endpoints are forced to be used as uni-directional ones. Therefore, only one STAT bit pair must be set at a value different from '00' (Disabled): STAT_RX if the double-buffered bulk endpoint is enabled for reception, STAT_TX if the double-buffered bulk endpoint is enabled for transmission. In case it is required to have double-buffered bulk endpoints enabled both for reception and transmission, two USB_EPnR registers must be used.

To exploit the double-buffering feature and reach the highest possible transfer rate, the endpoint flow control structure, described in previous chapters, has to be modified, in order to switch the endpoint status to NAK only when a buffer conflict occurs between the USB Peripheral and application software, instead of doing it at the end of each successful transaction. The memory buffer which is currently being used by the USB Peripheral is defined by the DTOG bit related to the endpoint direction: DTOG_RX (bit 14 of USB_EPnR register) for 'reception' double-buffered bulk endpoints or DTOG_TX (bit 6 of USB_EPnR register) for 'transmission' double-buffered bulk endpoints. To implement the new flow control scheme, the USB Peripheral should know which packet buffer is currently in use by the application software, so to be aware of any conflict. Since in the USB_EPnR register, there are two DTOG bits but only one is used by USB Peripheral for data and buffer sequencing (due to the uni-directional constraint required by double-buffering feature) the other one can be used by the application software to show which buffer it is currently using. This new buffer flag is called SW_BUF. In the following table the correspondence between USB_EPnR register bits and DTOG/SW_BUF definition is explained, for the cases of 'transmission' and 'reception' double-buffered bulk endpoints.

Table 43. Double-buffering buffer flag definition

Buffer flag	'Transmission' endpoint	'Reception' endpoint
DTOG	DTOG_TX (USB_EPnR bit 6)	DTOG_RX (USB_EPnR bit 14)
SW_BUF	USB_EPnR bit 14	USB_EPnR bit 6

The memory buffer which is currently being used by the USB Peripheral is defined by DTOG buffer flag, while the buffer currently in use by application software is identified by SW_BUF buffer flag. The relationship between the buffer flag value and the used packet buffer is the same in both cases, and it is listed in the following table.

Table 44. Double-buffering memory buffers usage

Endpoint Type	DTOG or SW_BUF bit value	Packet buffer used by USB Peripheral (DTOG) or application software (SW_BUF)
IN	0	ADDRn_TX_0 / COUNTn_TX_0 buffer description table locations.
	1	ADDRn_TX_1 / COUNTn_TX_1 buffer description table locations.
OUT	0	ADDRn_RX_0 / COUNTn_RX_0 buffer description table locations.
	1	ADDRn_RX_1 / COUNTn_RX_1 buffer description table locations.

Double-buffering feature for a bulk endpoint is activated by:

- writing EP_TYPE bit field at '00' in its USB_EPnR register, to define the endpoint as a bulk, and
- setting EP_KIND bit at '1' (DBL_BUF), in the same register.

The application software is responsible for DTOG and SW_BUF bits initialization according to the first buffer to be used; this has to be done considering the special toggle-only property that these two bits have. The end of the first transaction occurring after having set DBL_BUF, triggers the special flow control of double-buffered bulk endpoints, which is used for all other transactions addressed to this endpoint until DBL_BUF remain set. At the end of each transaction the CTR_RX or CTR_TX bit of the addressed endpoint USB_EPnR register is set, depending on the enabled direction. At the same time, the affected DTOG bit in the USB_EPnR register is hardware toggled making the USB Peripheral buffer swapping completely software independent. Unlike common transactions, and the first one after DBL_BUF setting, STAT bit pair is not affected by the transaction termination and its value remains '11' (Valid). However, as the token packet of a new transaction is received, the actual endpoint status will be masked as '10' (NAK) when a buffer conflict between the USB Peripheral and the application software is detected (this condition is identified by DTOG and SW_BUF having the same value). The application software responds to the CTR event notification by clearing the interrupt flag and starting any required handling of the completed transaction. When the application packet buffer usage is over, the software toggles the SW_BUF bit, writing '1' to it, to notify the USB Peripheral about the availability of that buffer. In this way, the number of NAKed transactions is limited only by the application elaboration time of a transaction data: if the elaboration time is shorter than the time required to complete a transaction on the USB bus, no re-transmissions due to flow control will take place and the actual transfer rate will be limited only by the host PC.

The application software can always override the special flow control implemented for double-buffered bulk endpoints, writing an explicit status different from '11' (Valid) into the STAT bit pair of the related USB_EPnR register. In this case, the USB Peripheral will always use the programmed endpoint status, regardless of the buffer usage condition.

13.5.4 Isochronous transfers

The USB standard supports full speed peripherals requiring a fixed and accurate data production/consume frequency, defining this kind of traffic as 'Isochronous'. Typical examples of this data are: audio samples, compressed video streams, and in general any sort of sampled data having strict requirements for the accuracy of delivered frequency. When an endpoint is defined to be 'isochronous' during the enumeration phase, the host allocates in the frame the required bandwidth and delivers exactly one IN or OUT packet each frame, depending on endpoint direction. To limit the bandwidth requirements, no re-transmission of failed transactions is possible for Isochronous traffic; this leads to the fact that an isochronous transaction does not have a handshake phase and no ACK packet is expected or sent after the data packet. For the same reason, Isochronous transfers do not support data toggle sequencing and always use DATA0 PID to start any data packet.

The Isochronous behaviour for an endpoint is selected by setting the EP_TYPE bits at '10' in its USB_EPnR register; since there is no handshake phase the only legal values for the STAT_RX/STAT_TX bit pairs are '00' (Disabled) and '11' (Valid), any other value will produce results not compliant to USB standard. Isochronous endpoints implement double-buffering to ease application software development, using both 'transmission' and 'reception' packet memory areas to manage buffer swapping on each successful transaction in order to have always a complete buffer to be used by the application, while the USB Peripheral fills the other.

The memory buffer which is currently used by the USB Peripheral is defined by the DTOG bit related to the endpoint direction (DTOG_RX for 'reception' isochronous endpoints, DTOG_TX for 'transmission' isochronous endpoints, both in the related USB_EPnR register) according to *Table 45*.

Table 45. Isochronous memory buffers usage

Endpoint Type	DTOG bit value	DMA buffer used by USB Peripheral	DMA buffer used by application software
IN	0	ADDRn_TX_0 / COUNTn_TX_0 buffer description table locations.	ADDRn_TX_1 / COUNTn_TX_1 buffer description table locations.
	1	ADDRn_TX_1 / COUNTn_TX_1 buffer description table locations.	ADDRn_TX_0 / COUNTn_TX_0 buffer description table locations.
OUT	0	ADDRn_RX_0 / COUNTn_RX_0 buffer description table locations.	ADDRn_RX_1 / COUNTn_RX_1 buffer description table locations.
	1	ADDRn_RX_1 / COUNTn_RX_1 buffer description table locations.	ADDRn_RX_0 / COUNTn_RX_0 buffer description table locations.

As it happens with double-buffered bulk endpoints, the USB_EPnR registers used to implement Isochronous endpoints are forced to be used as uni-directional ones. In case it is

required to have Isochronous endpoints enabled both for reception and transmission, two USB_EPnR registers must be used.

The application software is responsible for the DTOG bit initialization according to the first buffer to be used; this has to be done considering the special toggle-only property that these two bits have. At the end of each transaction, the CTR_RX or CTR_TX bit of the addressed endpoint USB_EPnR register is set, depending on the enabled direction. At the same time, the affected DTOG bit in the USB_EPnR register is hardware toggled making buffer swapping completely software independent. STAT bit pair is not affected by transaction completion; since no flow control is possible for Isochronous transfers due to the lack of handshake phase, the endpoint remains always '11' (Valid). CRC errors or buffer-overrun conditions occurring during Isochronous OUT transfers are anyway considered as correct transactions and they always trigger an CTR_RX event. However, CRC errors will anyway set the ERR bit in the USB_ISTR register to notify the software of the possible data corruption.

13.5.5 Suspend/Resume events

The USB standard defines a special peripheral state, called SUSPEND, in which the average current drawn from the USB bus must not be greater than 500 µA. This requirement is of fundamental importance for bus-powered devices, while self-powered devices are not required to comply to this strict power consumption constraint. In suspend mode, the host PC sends the notification to not send any traffic on the USB bus for more than 3mS: since a SOF packet must be sent every mS during normal operations, the USB Peripheral detects the lack of 3 consecutive SOF packets as a suspend request from the host PC and set the SUSP bit to '1' in USB_ISTR register, causing an interrupt if enabled. Once the device is suspended, its normal operation can be restored by a so called RESUME sequence, which can be started from the host PC or directly from the peripheral itself, but it is always terminated by the host PC. The suspended USB Peripheral must be anyway able to detect a RESET sequence, reacting to this event as a normal USB reset event.

The actual procedure used to suspend the USB peripheral is device dependent since according to the device composition, different actions may be required to reduce the total consumption.

A brief description of a typical suspend procedure is provided below, focused on the USB-related aspects of the application software routine responding to the SUSP notification of the USB Peripheral:

1. Set the FSUSP bit in the USB_CNTR register to 1. This action activates the suspend mode within the USB Peripheral. As soon as the suspend mode is activated, the check on SOF reception is disabled to avoid any further SUSP interrupts being issued while the USB is suspended.
2. Remove or reduce any static power consumption in blocks different from the USB Peripheral.
3. Set LP_MODE bit in USB_CNTR register to 1 to remove static power consumption in the analog USB transceivers but keeping them able to detect resume activity.
4. Optionally turn off external oscillator and device PLL to stop any activity inside the device.

When an USB event occurs while the device is in SUSPEND mode, the RESUME procedure must be invoked to restore nominal clocks and regain normal USB behaviour. Particular care must be taken to insure that this process does not take more than 10mS when the wakening event is an USB reset sequence (See "Universal Serial Bus Specification" for

more details). The start of a resume or reset sequence, while the USB Peripheral is suspended, clears the LP_MODE bit in USB_CNTR register asynchronously. Even if this event can trigger an WKUP interrupt if enabled, the use of an interrupt response routine must be carefully evaluated because of the long latency due to system clock restart; to have the shorter latency before re-activating the nominal clock it is suggested to put the resume procedure just after the end of the suspend one, so its code is immediately executed as soon as the system clock restarts. To prevent ESD discharges or any other kind of noise from waking-up the system (the exit from suspend mode is an asynchronous event), a suitable analog filter on data line status is activated during suspend; the filter width is about 70ns.

The following is a list of actions a resume procedure should address:

1. Optionally turn on external oscillator and/or device PLL.
2. Clear FSUSP bit of USB_CNTR register.
3. If the resume triggering event has to be identified, bits RXDP and RXDM in the USB_FNR register can be used according to *Table 46*, which also lists the intended software action in all the cases. If required, the end of resume or reset sequence can be detected monitoring the status of the above mentioned bits by checking when they reach the “10” configuration, which represent the Idle bus state; moreover at the end of a reset sequence the RESET bit in USB_ISTR register is set to 1, issuing an interrupt if enabled, which should be handled as usual.

Table 46. Resume event detection

[RXDP,RXDM] Status	Wake-up event	Required resume software action
“00”	Root reset	None
“10”	None (noise on bus)	Go back in Suspend mode
“01”	Root resume	None
“11”	Not Allowed (noise on bus)	Go back in Suspend mode

A device may require to exit from suspend mode as an answer to particular events not directly related to the USB protocol (e.g. a mouse movement wakes up the whole system). In this case, the resume sequence can be started by setting the RESUME bit in the USB_CNTR register to ‘1’ and resetting it to 0 after an interval between 1mS and 15mS (this interval can be timed using ESOF interrupts, occurring with a 1mS period when the system clock is running at nominal frequency). Once the RESUME bit is clear, the resume sequence will be completed by the host PC and its end can be monitored again using the RXDP and RXDM bits in the USB_FNR register.

Note: The RESUME bit must be anyway used only after the USB Peripheral has been put in suspend mode, setting the FSUSP bit in USB_CNTR register to 1.

13.6 Register description

The USB Peripheral registers can be divided into the following groups:

- Common Registers: Interrupt and Control registers
- Endpoint Registers: Endpoint configuration and status
- Buffer Descriptor Table: Location of packet memory used to locate data buffers

All register addresses are expressed as offsets with respect to the USB Peripheral registers base address 0xC000 8800, except the buffer descriptor table locations, which starts at the address specified by the USB_BTABLE register. Due to the common limitation of APB bridges on word addressability, all register addresses are aligned to 32-bit word boundaries although they are 16-bit wide. The same address alignment is used to access packet buffer memory locations, which are located starting from 0xC000 8000. In this section, the following abbreviations are used:

Read/write (rw)	The software can read and write to these bits.
Read-only (r)	The software can only read these bits.
Write-only (w)	The software can only write to these bits.
Read-clear (rc)	The software can only read or clear this bit.
Toggle (t)	The software can only toggle this bit by writing ‘1’. Writing ‘0’ has no effect.

13.6.1 Common registers

These registers affect the general behaviour of the USB Peripheral defining operating mode, interrupt handling, device address and giving access to the current frame number updated by the host PC.

USB control register (USB_CNTR)

Address Offset: 40h

Reset Value: 0000 0000 0000 0011 (0003h)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CTRM	DOVRM	ERRM	WKUPM	SUSPM	RESETM	SOFM	ESOFM	Reserved			RESUME	FSUSP	LP MODE	PDWN	FRES
rw	rw	rw	rw	rw	rw	rw	rw	-	-	-	rw	rw	rw	rw	rw

Bit 15	**CTRM:** *Correct Transfer Interrupt Mask* 0: Correct Transfer (CTR) Interrupt disabled. 1: CTR Interrupt enabled, an interrupt request is generated when the corresponding bit in the USB_ISTR register is set.
Bit 14	**DOVRM:** *DMA over / underrun Interrupt Mask* 0: DOVR Interrupt disabled. 1: DOVR Interrupt enabled, an interrupt request is generated when the corresponding bit in the USB_ISTR register is set.

Bit 13	**ERRM:** *Error Interrupt Mask* 0: ERR Interrupt disabled. 1: ERR Interrupt enabled, an interrupt request is generated when the corresponding bit in the USB_ISTR register is set.
Bit 12	**WKUPM:** *Wake-up Interrupt Mask* 0: WKUP Interrupt disabled. 1: WKUP Interrupt enabled, an interrupt request is generated when the corresponding bit in the USB_ISTR register is set.
Bit 11	**SUSPM:** *Suspend mode Interrupt Mask* 0: Suspend Mode Request (SUSP) Interrupt disabled. 1: SUSP Interrupt enabled, an interrupt request is generated when the corresponding bit in the USB_ISTR register is set.
Bit 10	**RESETM:** *USB Reset Interrupt Mask* 0: RESET Interrupt disabled. 1: RESET Interrupt enabled, an interrupt request is generated when the corresponding bit in the USB_ISTR register is set.
Bit 9	**SOFM:** *Start Of Frame Interrupt Mask* 0: SOF Interrupt disabled. 1: SOF Interrupt enabled, an interrupt request is generated when the corresponding bit in the USB_ISTR register is set.
Bit 8	**ESOFM:** *Expected Start Of Frame Interrupt Mask* 0: Expected Start of Frame (ESOF) Interrupt disabled. 1: ESOF Interrupt enabled, an interrupt request is generated when the corresponding bit in the USB_ISTR register is set.
Bits 7:5	Reserved, forced by hardware to 0.
Bit 4	**RESUME:** *Resume request* The microcontroller can set this bit to send a Resume signal to the host. It must be activated, according to USB specifications, for no less than 1mS and no more than 15mS after which the Host PC is ready to drive the resume sequence up to its end.
Bit 3	**FSUSP:** *Force suspend* Software must set this bit when the SUSP interrupt is received, which is issued when no traffic is received by the USB Peripheral for 3 mS. 0: No effect. 1: Enter suspend mode. Clocks and static power dissipation in the analog transceiver are left unaffected. If suspend power consumption is a requirement (bus-powered device), the application software should set the LP_MODE bit after FSUSP as explained below.
Bit 2	**LP_MODE:** *Low-power mode* This mode is used when the suspend-mode power constraints require that all static power dissipation is avoided, except the one required to supply the external pull-up resistor. This condition should be entered when the application is ready to stop all system clocks, or reduce their frequency in order to meet the power consumption requirements of the USB suspend condition. The USB activity during the suspend mode (WKUP event) asynchronously resets this bit (it can also be reset by software). 0: No Low Power Mode. 1: Enter Low Power mode.

Bit 1	**PDWN:** *Power down* This bit is used to completely switch off all USB-related analog parts if it is required to completely disable the USB Peripheral for any reason. When this bit is set, the USB Peripheral is disconnected from the transceivers and it cannot be used. 0: Exit Power Down. 1: Enter Power down mode.
Bit 0	**FRES:** *Force USB Reset* 0: Clear USB reset. 1: Force a reset of the USB Peripheral, exactly like a RESET signalling on the USB. The USB Peripheral is held in RESET state until software clears this bit. A “USB-RESET” interrupt is generated, if enabled.

USB interrupt status register (USB_ISTR)

Address Offset: 44h

Reset Value: 0000 0000 0000 0000 (0000h)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CTR	DOVR	ERR	WKUP	SUSP	RESET	SOF	ESOF	Reserved			DIR	EP_ID[3:0]			
r	rc	rc	rc	rc	rc	rc	rc	-	-	-	r	r	r	r	r

This register contains the status of all the interrupt sources allowing application software to determine, which events caused an interrupt request.

The upper part of this register contains single bits, each of them representing a specific event. These bits are set by the hardware when the related event occurs; if the corresponding bit in the USB_CNTR register is set, a generic interrupt request is generated. The interrupt routine, examining each bit, will perform all necessary actions, and finally it will clear the serviced bits. If any of them is not cleared, the interrupt is considered to be still pending, and the interrupt line will be kept high again. If several bits are set simultaneously, only a single interrupt will be generated.

Endpoint transaction completion can be handled in a different way to reduce interrupt response latency. The CTR bit is set by the hardware as soon as an endpoint successfully completes a transaction, generating a generic interrupt request if the corresponding bit in USB_CNTR is set. An endpoint dedicated interrupt condition is activated independently from the CTRM bit in the USB_CNTR register. Both interrupt conditions remain active until software clears the pending bit in the corresponding USB_EPnR register (the CTR bit is actually a read only bit). The USB Peripheral has two interrupt request lines:

- Higher priority USB IRQ: The pending requests for endpoints, which have transactions with a higher priority (isochronous and double-buffered bulk) and they cannot be masked.
- Lower priority USB IRQ: All other interrupt conditions, which can either be non-maskable pending requests related to the lower priority transactions and all other maskable events flagged by the USB_ISTR high bytes.

For endpoint-related interrupts, the software can use the Direction of Transaction (DIR) and EP_ID read-only bits to identify, which endpoint made the last interrupt request and called the corresponding interrupt service routine.

The user can choose the relative priority of simultaneously pending USB_ISTR events by specifying the order in which software checks USB_ISTR bits in an interrupt service routine.

Only the bits related to events, which are serviced, are cleared. At the end of the service routine, another interrupt will be requested, to service the remaining conditions.

To avoid spurious clearing of some bits, it is recommended to clear them with a load instruction where all bits which must not be altered are written with 1, and all bits to be cleared are written with ‘0’ (these bits can only be cleared by software). Read-modify-write cycles should be avoided because between the read and the write operations another bit could be set by the hardware and the next write will clear it before the microprocessor has the time to serve the event.

The following describes each bit in detail:

Bit 15	**CTR:** *Correct Transfer* This bit is set by the hardware to indicate that an endpoint has successfully completed a transaction; using DIR and EP_ID bits software can determine which endpoint requested the interrupt. This bit is read-only.
Bit 14	**DOVR:** *DMA over / underrun* This bit is set if the microcontroller has not been able to respond in time to an USB memory request. The USB Peripheral handles this event in the following way: During reception an ACK handshake packet is not sent, during transmission a bit-stuff error is forced on the transmitted stream; in both cases the host will retry the transaction. The DOVR interrupt should never occur during normal operations. Since the failed transaction is retried by the host, the application software has the chance to speed-up device operations during this interrupt handling, to be ready for the next transaction retry; however this does not happen during Isochronous transfers (no isochronous transaction is anyway retried) leading to a loss of data in this case. This bit is read/write but only ‘0’ can be written and writing ‘1’ has no effect.
Bit 13	**ERR:** *Error* This flag is set whenever one of the errors listed below has occurred: NANS: No ANSwer. The timeout for a host response has expired. CRC: Cyclic Redundancy Check error. One of the received CRCs, either in the token or in the data, was wrong. BST: Bit Stuffing error. A bit stuffing error was detected anywhere in the PID, data, and/or CRC. FVIO: Framing format Violation. A non-standard frame was received (EOP not in the right place, wrong token sequence, etc.). The USB software can usually ignore errors, since the USB Peripheral and the PC host manage retransmission in case of errors in a fully transparent way. This interrupt can be useful during the software development phase, or to monitor the quality of transmission over the USB bus, to flag possible problems to the user (e.g. loose connector, too noisy environment, broken conductor in the USB cable and so on). This bit is read/write but only ‘0’ can be written and writing ‘1’ has no effect.
Bit 12	**WKUP:** *Wake up* This bit is set to 1 by the hardware when, during suspend mode, activity is detected that wakes up the USB Peripheral. This event asynchronously clears the LP_MODE bit in the CTLR register and activates the USB_WAKEUP line, which can be used to notify the rest of the device (e.g. wake-up unit) about the start of the resume process. This bit is read/write but only ‘0’ can be written and writing ‘1’ has no effect.

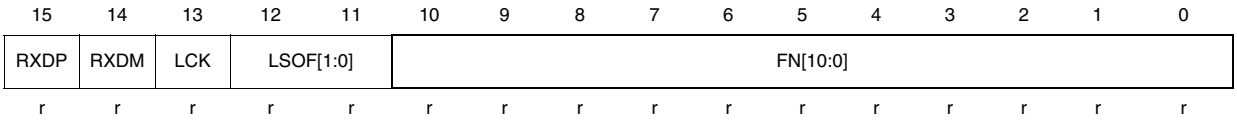
Bit 11	**SUSP:** *Suspend mode request* This bit is set by the hardware when no traffic has been received for 3mS, indicating a suspend mode request from the USB bus. The suspend condition check is enabled immediately after any USB reset and it is disabled by the hardware when the suspend mode is active (FSUSP=1) until the end of resume sequence. This bit is read/write but only '0' can be written and writing '1' has no effect.
Bit 10	**RESET:** *USB RESET request* Set when the USB Peripheral detects an active USB RESET signal at its inputs. The USB Peripheral, in response to a RESET, just resets its internal protocol state machine, generating an interrupt if RESETM enable bit in the USB_CNTR register is set. Reception and transmission are disabled until the RESET bit is cleared. All configuration registers do not reset: the microcontroller must explicitly clear these registers (this is to ensure that the RESET interrupt can be safely delivered, and any transaction immediately followed by a RESET can be completed). The function address and endpoint registers are reset by an USB reset event. This bit is read/write but only '0' can be written and writing '1' has no effect.
Bit 9	**SOF:** *Start Of Frame* This bit signals the beginning of a new USB frame and it is set when a SOF packet arrives through the USB bus. The interrupt service routine may monitor the SOF events to have a 1mS synchronization event to the USB host and to safely read the USB_FNR register which is updated at the SOF packet reception (this could be useful for isochronous applications). This bit is read/write but only '0' can be written and writing '1' has no effect.
Bit 8	**ESOF:** *Expected Start Of Frame* This bit is set by the hardware when an SOF packet is expected but not received. The host sends an SOF packet each mS, but if the hub does not receive it properly, the Suspend Timer issues this interrupt. If three consecutive ESOF interrupts are generated (i.e. three SOF packets are lost) without any traffic occurring in between, a SUSP interrupt is generated. This bit is set even when the missing SOF packets occur while the Suspend Timer is not yet locked. This bit is read/write but only '0' can be written and writing '1' has no effect.
Bits 7:5	Reserved, forced by hardware to 0.

Bit 4	**DIR:** *Direction of transaction*. This bit is written by the hardware according to the direction of the successful transaction, which generated the interrupt request. If DIR bit=0, CTR_TX bit is set in the USB_EPnR register related to the interrupting endpoint. The interrupting transaction is of IN type (data transmitted by the USB Peripheral to the host PC). If DIR bit=1, CTR_RX bit or both CTR_TX/CTR_RX are set in the USB_EPnR register related to the interrupting endpoint. The interrupting transaction is of OUT type (data received by the USB Peripheral from the host PC) or two pending transactions are waiting to be processed. This information can be used by the application software to access the USB_EPnR bits related to the triggering transaction since it represents the direction having the interrupt pending. This bit is read-only.
Bits 3:0	**EP_ID[3:0]:** *Endpoint Identifier*. These bits are written by the hardware according to the endpoint number, which generated the interrupt request. If several endpoint transactions are pending, the hardware writes the endpoint identifier related to the endpoint having the highest priority defined in the following way: Two endpoint sets are defined, in order of priority: Isochronous and double-buffered bulk endpoints are considered first and then the other endpoints are examined. If more than one endpoint from the same set is requesting an interrupt, the EP_ID bits in USB_ISTR register are assigned according to the lowest requesting endpoint register, EP0R having the highest priority followed by EP1R and so on. The application software can assign a register to each endpoint according to this priority scheme, so as to order the concurring endpoint requests in a suitable way. These bits are read only.

USB frame number register (USB_FNR)

Address Offset: 48h

Reset Value: 0000 0xxx xxxx xxxx (0xxxh)



Bit 15	**RXDP:** *Receive Data + Line Status* This bit can be used to observe the status of received data plus upstream port data line. It can be used during end-of-suspend routines to help determining the wake-up event.
Bit 14	**RXDM:** *Receive Data - Line Status* This bit can be used to observe the status of received data minus upstream port data line. It can be used during end-of-suspend routines to help determining the wake-up event.
Bit 13	**LCK:** *Locked* This bit is set by the hardware when at least two consecutive SOF packets have been received after the end of an USB reset condition or after the end of an USB resume sequence. Once locked, the frame timer remains in this state until an USB reset or USB suspend event occurs.
Bits12:11	**LSOF[1:0]:** *Lost SOF* These bits are written by the hardware when an ESOF interrupt is generated, counting the number of consecutive SOF packets lost. At the reception of an SOF packet, these bits are cleared.
Bits 10:0	**FN[10:0]:** *Frame Number* This bit field contains the 11-bits frame number contained in the last received SOF packet. The frame number is incremented for every frame sent by the host and it is useful for Isochronous transfers. This bit field is updated on the generation of an SOF interrupt.

USB device address (USB_DADDR)

Address Offset: 4Ch

Reset Value: 0000 0000 0000 0000 (0000h)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								EF	ADD[6:0]						
-	-	-	-	-	-	-	-	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:8	Reserved, forced by hardware to 0.
Bit 7	**EF:** *Enable Function* This bit is set by the software to enable the USB device. The address of this device is contained in the following ADD[6:0] bits. If this bit is at '0' no transactions are handled, irrespective of the settings of USB_EPnR registers.
Bits 6:0	**ADD[6:0]:** *Device Address* These bits contain the USB function address assigned by the host PC during the enumeration process. Both this field and the Endpoint Address (EA) field in the associated USB_EPnR register must match with the information contained in a USB token in order to handle a transaction to the required endpoint.

Buffer table address (USB_BTABLE)

Address Offset: 50h

Reset Value: 0000 0000 0000 0000 (0000h)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BTABLE[15:3]													Reserved		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	-	-	-

Bits 15:3	**BTABLE[15:3]:** *Buffer Table*. These bits contain the start address of the buffer allocation table inside the dedicated packet memory. This table describes each endpoint buffer location and size and it must be aligned to an 8 byte boundary (the 3 least significant bits are always '0'). At the beginning of every transaction addressed to this device, the USP peripheral reads the element of this table related to the addressed endpoint, to get its buffer start location and the buffer size (Refer to *Structure and usage of packet buffers*).
Bits 2:0	Reserved, forced by hardware to 0.

13.6.2 Endpoint-specific registers

The number of these registers varies according to the number of endpoints that the USB Peripheral is designed to handle. The USB Peripheral supports up to 16 bi-directional endpoints. Each USB device must support a control endpoint whose address (EA bits) must be set to 0. The USB Peripheral behaves in an undefined way if multiple endpoints are enabled having the same endpoint number value. For each endpoint, an USB_EPnR register is available to store the endpoint specific information.

USB endpoint n register (USB_EPnR), n=[0..7]

Address Offset: 00h to 3Ch

Reset value: 0000 0000 0000 0000b (0000h)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CTR_ RX	DTOG _RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_ KIND	CTR TX	DTOG_ TX	STAT TX[1:0]		EA[3:0]			
r-c	t	t	t	r	rw	rw	rw	r-c	t	t	t	rw	rw	rw	rw

They are also reset when an USB reset is received from the USB bus or forced through bit FRES in the CTLR register, except the CTR_RX and CTR_TX bits, which are kept unchanged to avoid missing a correct packet notification immediately followed by an USB reset event. Each endpoint has its USB_EPnR register where *n* is the endpoint identifier.

Read-modify-write cycles on these registers should be avoided because between the read and the write operations some bits could be set by the hardware and the next write would modify them before the CPU has the time to detect the change. For this purpose, all bits affected by this problem have an ‘invariant’ value that must be used whenever their modification is not required. It is recommended to modify these registers with a load instruction where all the bits, which can be modified only by the hardware, are written with their ‘invariant’ value.

Bit 15	**CTR_RX:** *Correct Transfer for reception* This bit is set by the hardware when an OUT/SETUP transaction is successfully completed on this endpoint; the software can only clear this bit. If the CTRM bit in USB_CNTR register is set accordingly, a generic interrupt condition is generated together with the endpoint related interrupt condition, which is always activated. The type of occurred transaction, OUT or SETUP, can be determined from the SETUP bit described below. A transaction ended with a NAK or STALL handshake does not set this bit, since no data is actually transferred, as in the case of protocol errors or data toggle mismatches. This bit is read/write but only '0' can be written, writing 1 has no effect.
Bit 14	**DTOG_RX:** *Data Toggle, for reception transfers* If the endpoint is not Isochronous, this bit contains the expected value of the data toggle bit (0=DATA0, 1=DATA1) for the next data packet to be received. Hardware toggles this bit, when the ACK handshake is sent to the USB host, following a data packet reception having a matching data PID value; if the endpoint is defined as a control one, hardware clears this bit at the reception of a SETUP PID addressed to this endpoint. If the endpoint is using the double-buffering feature this bit is used to support packet buffer swapping too (Refer to *Section 13.5.3: Double-buffered endpoints*). If the endpoint is Isochronous, this bit is used only to support packet buffer swapping since no data toggling is used for this sort of endpoints and only DATA0 packet are transmitted (Refer to *Section 13.5.4: Isochronous transfers*). Hardware toggles this bit just after the end of data packet reception, since no handshake is used for isochronous transfers. This bit can also be toggled by the software to initialize its value (mandatory when the endpoint is not a control one) or to force specific data toggle/packet buffer usage. When the application software writes '0', the value of DTOG_RX remains unchanged, while writing '1' makes the bit value toggle. This bit is read/write but it can be only toggled by writing 1.
Bits 13:12	**STAT_RX [1:0]:** *Status bits, for reception transfers* These bits contain information about the endpoint status, which are listed in *Table 47: Reception status encoding on page 242*.These bits can be toggled by software to initialize their value. When the application software writes '0', the value remains unchanged, while writing '1' makes the bit value toggle. Hardware sets the STAT_RX bits to NAK when a correct transfer has occurred (CTR_RX=1) corresponding to a OUT or SETUP (control only) transaction addressed to this endpoint, so the software has the time to elaborate the received data before it acknowledge a new transaction Double-buffered bulk endpoints implement a special transaction flow control, which control the status based upon buffer availability condition (Refer to *Section 13.5.3: Double-buffered endpoints*). If the endpoint is defined as Isochronous, its status can be only "VALID" or "DISABLED", so that the hardware cannot change the status of the endpoint after a successful transaction. If the software sets the STAT_RX bits to 'STALL' or 'NAK' for an Isochronous endpoint, the USB Peripheral behaviour is not defined. These bits are read/write but they can be only toggled by writing '1'.
Bit 11	**SETUP:** *Setup transaction completed* This bit is read-only and it is set by the hardware when the last completed transaction is a SETUP. This bit changes its value only for control endpoints. It must be examined, in the case of a successful receive transaction (CTR_RX event), to determine the type of transaction occurred. To protect the interrupt service routine from the changes in SETUP bits due to next incoming tokens, this bit is kept frozen while CTR_RX bit is at 1; its state changes when CTR_RX is at 0. This bit is read-only.

Bits 10:9	**EP_TYPE[1:0]:** *Endpoint type* These bits configure the behaviour of this endpoint as described in *Table 48: Endpoint type encoding on page 243*. Endpoint 0 must always be a control endpoint and each USB function must have at least one control endpoint which has address 0, but there may be other control endpoints if required. Only control endpoints handle SETUP transactions, which are ignored by endpoints of other kinds. SETUP transactions cannot be answered with NAK or STALL. If a control endpoint is defined as NAK, the USB Peripheral will not answer, simulating a receive error, in the receive direction when a SETUP transaction is received. If the control endpoint is defined as STALL in the receive direction, then the SETUP packet will be accepted anyway, transferring data and issuing the CTR interrupt. The reception of OUT transactions is handled in the normal way, even if the endpoint is a control one. Bulk and interrupt endpoints have very similar behaviour and they differ only in the special feature available using the EP_KIND configuration bit. The usage of Isochronous endpoints is explained in *Section 13.5.4: Isochronous transfers*
Bit 8	**EP_KIND:** *Endpoint Kind* The meaning of this bit depends on the endpoint type configured by the EP_TYPE bits. *Table 49* summarizes the different meanings. DBL_BUF: This bit is set by the software to enable the double-buffering feature for this bulk endpoint. The usage of double-buffered bulk endpoints is explained in *Section 13.5.3: Double-buffered endpoints*. STATUS_OUT: This bit is set by the software to indicate that a status out transaction is expected: in this case all OUT transactions containing more than zero data bytes are answered ‘STALL’ instead of ‘ACK’. This bit may be used to improve the robustness of the application to protocol errors during control transfers and its usage is intended for control endpoints only. When STATUS_OUT is reset, OUT transactions can have any number of bytes, as required.
Bit 7	**CTR_TX:** *Correct Transfer for transmission* This bit is set by the hardware when an IN transaction is successfully completed on this endpoint; the software can only clear this bit. If the CTRM bit in the USB_CNTR register is set accordingly, a generic interrupt condition is generated together with the endpoint related interrupt condition, which is always activated. A transaction ended with a NAK or STALL handshake does not set this bit, since no data is actually transferred, as in the case of protocol errors or data toggle mismatches. This bit is read/write but only ‘0’ can be written.

Bit 6	**DTOG_TX:** *Data Toggle, for transmission transfers* If the endpoint is non-isochronous, this bit contains the required value of the data toggle bit (0=DATA0, 1=DATA1) for the next data packet to be transmitted. Hardware toggles this bit when the ACK handshake is received from the USB host, following a data packet transmission. If the endpoint is defined as a control one, hardware sets this bit to 1 at the reception of a SETUP PID addressed to this endpoint. If the endpoint is using the double buffer feature, this bit is used to support packet buffer swapping too (Refer to *Section 13.5.3: Double-buffered endpoints*) If the endpoint is Isochronous, this bit is used to support packet buffer swapping since no data toggling is used for this sort of endpoints and only DATA0 packet are transmitted (Refer to *Section 13.5.4: Isochronous transfers*). Hardware toggles this bit just after the end of data packet transmission, since no handshake is used for Isochronous transfers. This bit can also be toggled by the software to initialize its value (mandatory when the endpoint is not a control one) or to force a specific data toggle/packet buffer usage. When the application software writes '0', the value of DTOG_TX remains unchanged, while writing '1' makes the bit value toggle. This bit is read/write but it can only be toggled by writing 1.
Bit 5:4	**STAT_TX [1:0]:** *Status bits, for transmission transfers* These bits contain the information about the endpoint status, listed in *Table 50*. These bits can be toggled by the software to initialize their value. When the application software writes '0', the value remains unchanged, while writing '1' makes the bit value toggle. Hardware sets the STAT_TX bits to NAK, when a correct transfer has occurred (CTR_TX=1) corresponding to a IN or SETUP (control only) transaction addressed to this endpoint. It then waits for the software to prepare the next set of data to be transmitted. Double-buffered bulk endpoints implement a special transaction flow control, which controls the status based on buffer availability condition (Refer to *Section 13.5.3: Double-buffered endpoints*). If the endpoint is defined as Isochronous, its status can only be "VALID" or "DISABLED". Therefore, the hardware cannot change the status of the endpoint after a successful transaction. If the software sets the STAT_TX bits to 'STALL' or 'NAK' for an Isochronous endpoint, the USB Peripheral behaviour is not defined. These bits are read/write but they can be only toggled by writing '1'.
Bit 3:0	**EA[3:0]:** *Endpoint Address.* Software must write in this field the 4-bit address used to identify the transactions directed to this endpoint. A value must be written before enabling the corresponding endpoint.

Table 47. Reception status encoding

STAT_RX[1:0]	Meaning
00	**DISABLED:** all reception requests addressed to this endpoint are ignored.
01	**STALL**: the endpoint is stalled and all reception requests result in a STALL handshake.
10	**NAK**: the endpoint is naked and all reception requests result in a NAK handshake.
11	**VALID**: this endpoint is enabled for reception.

Table 48. Endpoint type encoding

EP_TYPE[1:0]	Meaning
00	BULK
01	CONTROL
10	ISO
11	INTERRUPT

Table 49. Endpoint kind meaning

EP_TYPE[1:0]		EP_KIND Meaning
00	BULK	DBL_BUF
01	CONTROL	STATUS_OUT
10	ISO	Not used
11	INTERRUPT	Not used

Table 50. Transmission status encoding

STAT_TX[1:0]	Meaning
00	**DISABLED:** all transmission requests addressed to this endpoint are ignored.
01	**STALL**: the endpoint is stalled and all transmission requests result in a STALL handshake.
10	**NAK**: the endpoint is naked and all transmission requests result in a NAK handshake.
11	**VALID**: this endpoint is enabled for transmission.

13.6.3 Buffer descriptor table

Although this table is located inside packet buffer memory, its entries can be considered as additional registers used to configure the location and size of packet buffers used to exchange data between USB macrocell and the STR71x. Due to the common APB bridge limitation on word addressability, all packet memory locations are accessed by the APB using 32-bit aligned addresses, instead of the actual memory location addresses utilized by the USB Peripheral for the USB_BTABLE register and buffer description table locations. In the following pages two location addresses are reported: the one to be used by application software while accessing the packet memory, and the local one relative to USB Peripheral access. To obtain the correct STR71x memory address value to be used in the application software while accessing the packet memory, the actual memory location address must be multiplied by two. The first packet memory location is located at 0xC000 8000.

The buffer description table entry associated with the USB_EPnR registers is described below. A thorough explanation of packet buffers and buffer descriptor table usage can be found in the *Structure and usage of packet buffers*.

Transmission buffer address n (USB_ADDRn_TX)

Address Offset: [USB_BTABLE] + n*16

USB local Address: [USB_BTABLE] + n*8

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDRn_TX[15:1]															-
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	-

Bits 15:1	**ADDRn_TX[15:1]:** *Transmission Buffer Address* These bits point to the starting address of the packet buffer containing data to be transmitted by the endpoint associated with the USB_EPnR register at the next IN token addressed to it.
Bit 0	Must always be written as ‘0’ since packet memory is word-wide and all packet buffers must be word-aligned.

Transmission byte count n (USB_COUNTn_TX)

Address Offset: [USB_BTABLE] + n*16 + 4

USB local Address: [USB_BTABLE] + n*8 + 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-						COUNTn_TX[9:0]									
-	-	-	-	-	-	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:10	These bits are not used since packet size is limited by USB specifications to 1023 bytes. Their value is not considered by the USB Peripheral.
Bits 9:0	**COUNTn_TX[9:0]:** *Transmission Byte Count* These bits contain the number of bytes to be transmitted by the endpoint associated with the USB_EPnR register at the next IN token addressed to it.

Note: Double-buffered and Isochronous IN Endpoints have two USB_COUNTn_TX registers: named USB_COUNTn_TX_1 and USB_COUNTn_TX_0 with the following content

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
-						COUNTn_TX_1[9:0]									
-	-	-	-	-	-	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-						COUNTn_TX_0[9:0]									
-	-	-	-	-	-	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Reception buffer address n (USB_ADDRn_RX)

Address Offset: [USB_BTABLE] + n*16 + 8

USB local Address: [USB_BTABLE] + n*8 + 4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDRn_RX[15:1]															-
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	-

Bits 15:1	**ADDRn_RX[15:1]:** *Reception Buffer Address* These bits point to the starting address of the packet buffer, which will contain the data received by the endpoint associated with the USB_EPnR register at the next OUT/SETUP token addressed to it.
Bit 0	This bit must always be written as ‘0’ since packet memory is word-wide and all packet buffers must be word-aligned.

Reception byte count n (USB_COUNTn_RX)

Address Offset: [USB_BTABLE] + n*16 + 12

USB local Address: [USB_BTABLE] + n*8 + 6

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BLSIZE	NUM_BLOCK[4:0]					COUNTn_RX[9:0]									
rw	rw	rw	rw	rw	rw	r	r	r	r	r	r	r	r	r	r

This table location is used to store two different values, both required during packet reception. The most significant bits contains the definition of allocated buffer size, to allow buffer overflow detection, while the least significant part of this location is written back by the USB Peripheral at the end of reception to give the actual number of received bytes. Due to the restrictions on the number of available bits, buffer size is represented using the number of allocated memory blocks, where block size can be selected to choose the trade-off between fine-granularity/small-buffer and coarse-granularity/large-buffer. The size of allocated buffer is a part of the endpoint descriptor and it is normally defined during the enumeration process according to its maxPacketSize parameter value (See “Universal Serial Bus Specification”).

Bit 15	**BL_SIZE:** *BLock SIZE.* This bit selects the size of memory block used to define the allocated buffer area. – If BL_SIZE=0, the memory block is 2 `byte` large, which is the minimum block allowed in a word-wide memory. With this block size the allocated buffer size ranges from 2 to 62 `bytes`. – If BL_SIZE=1, the memory block is 32 `byte` large, which allows to reach the maximum packet length defined by USB specifications. With this block size the allocated buffer size ranges from 32 to 1024 bytes, which is the longest packet size allowed by USB standard specifications.
Bits 14:10	**NUM_BLOCK[4:0]:** *Number of blocks.* These bits define the number of memory blocks allocated to this packet buffer. The actual amount of allocated memory depends on the BL_SIZE value as illustrated in *Table 51*.
Bits 9:0	**COUNTn_RX[9:0]:** *Reception Byte Count* These bits contain the number of bytes received by the endpoint associated with the USB_EPnR register during the last OUT/SETUP transaction addressed to it.

Note: Double-buffered and Isochronous OUT Endpoints have two USB_COUNTn_RX registers: named USB_COUNTn_RX_1 and USB_COUNTn_RX_0 with the following content

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BLSIZE_1	NUM_BLOCK_1[4:0]					COUNTn_RX_1[9:0]									
rw	rw	rw	rw	rw	rw	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BLSIZE_0	NUM_BLOCK_0[4:0]					COUNTn_RX_0[9:0]									
rw	rw	rw	rw	rw	rw	r	r	r	r	r	r	r	r	r	r

Table 51. Definition of allocated buffer memory

Value of NUM_BLOCK[4:0]	Memory allocated when BL_SIZE=0	Memory allocated when BL_SIZE=1
0 (‘00000’)	Not allowed	32 bytes
1 (‘00001’)	2 bytes	64 bytes
2 (‘00010’)	4 bytes	96 bytes
3 (‘00011’)	6 bytes	128 bytes
...	...	...
15 (‘01111’)	30 bytes	512 bytes
16 (‘10000’)	32 bytes	544 bytes
17 (‘10001’)	34 bytes	576 bytes
18 (‘10010’)	36 bytes	608 bytes
...	...	...
30 (‘11110’)	60 bytes	992 bytes
31 (‘11111’)	62 bytes	1024 bytes

13.7 USB register map

To be able to find the correct offset for each register, *Table 52* shows the mapping of all USB Peripheral registers.

Table 52. USB Peripheral register page mapping

Offset	Register	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	USB_EP0R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x04	USB_EP1R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x08	USB_EP2R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x0C	USB_EP3R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x10	USB_EP4R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x14	USB_EP5R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x18	USB_EP6R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x1C	USB_EP7R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x20	USB_EP8R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x24	USB_EP9R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x28	USB_EP10R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x2C	USB_EP11R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x30	USB_EP12R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			

Table 52. USB Peripheral register page mapping

Offset	Register	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x34	USB_EP13R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x38	USB_EP14R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x3C	USB_EP15R	CTR_RX	DTOG_RX	STAT RX[1:0]		SETUP	EP TYPE[1:0]		EP_KIND	CTR TX	DTOG_TX	STAT TX[1:0]		EA[3:0]			
0x40	USB_CNTR	CTRM	DOVRM	ERRM	WKUPM	SUSPM	RESETM	SOFM	ESOFM	Reserved			RESUME	FSUSP	LP MODE	PDWN	FRES
0x44	USB_ISTR	CTR	DOVR	ERR	WKUP	SUSP	RESET	SOF	ESOF	Reserved			DIR	EP_ID[3:0]			
0x48	USB_FNR	RXDP	RXDM	LCK	LSOF[1:0]		FN[10:0]										
0x4C	USB_DADDR	Reserved								EF	ADD[6:0]						
0x50	USB_BTABLE	BTABLE[15:3]													Reserved		

Refer to *Table 2* for the base address.

14 A/D converter (ADC)

14.1 Introduction

The ADC is used to measure signal strength and other slowly-changing signals. Four input channels are supported, with conversion rates of up to 1 kHz per channel.

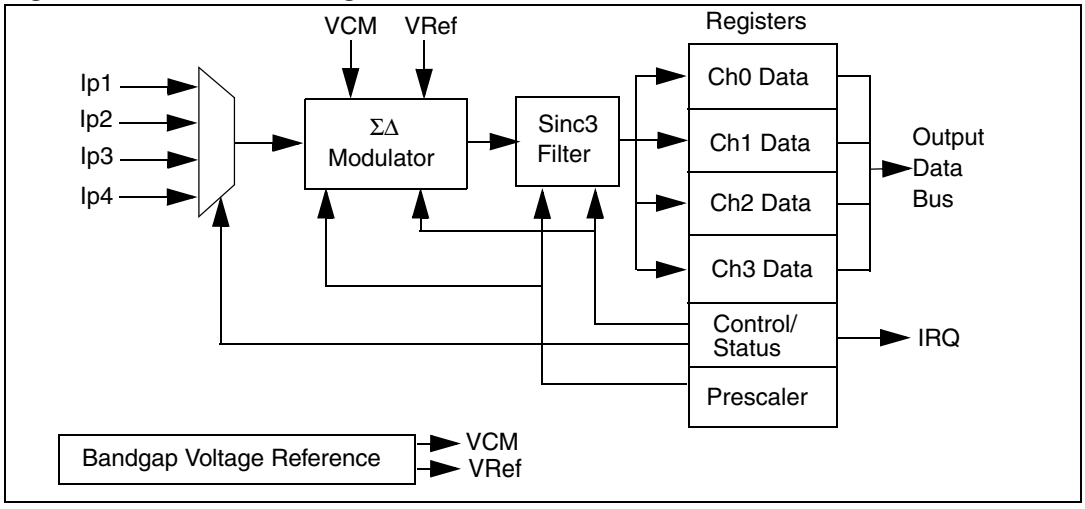
14.2 Main features

- 12-bit resolution
- 0 to 2.5V range
- 4 input channels
- Round-Robin or single channel mode
- Maskable interrupt
- Programmable prescaler

14.3 Functional description

The ADC consists of a four-channel single-bit Sigma-Delta modulator with a 511-tap Sinc^3 digital filter and a bandgap voltage reference. A block diagram of the converter is shown below in *Figure 73*. The analog part of the converter is enabled by the ADC_EN bit in the PCU_BOOTCR register.

Figure 73. ADC Block Diagram



14.3.1 Normal (Round-Robin) ADC operation

In its normal mode of operation, the converter samples each input channel for 512 cycles of the oversampling clock. In the first clock cycle, the Sigma-Delta modulator is reset and the digital filter cleared. The remaining 1-bit samples from the modulator are filtered by the Sinc^3 filter and a 16 bit output sample supplied to the relevant data register after 511 clock cycles, the period over which the Sinc^3 filter has filled up and settled down. The channel

select will then switch to the next input channel, the reset will again be asserted on the first clock cycle, and the filter will again fill up over 511 cycles to produce a sample. This process will be repeated for each of the channels continually in a round-robin fashion.

14.3.2 Single-channel operation

When sampling a single channel, that channel alone will be selected as input to the analog signal to the sigma-delta modulator. The functionality of the converter will remain the same as above in that the converter will be reset every 512 cycles, once a valid sample is produced. However, to maintain the same output frequency of the converter, only one of these samples will be taken out of every four, i.e. a valid sample for the channel will be produced every 2048 clock cycles, as before.

Note: In order to speed-up the ADC conversion of an input signal, you can use the Round-Robin mode and connect the input signal to the four ADC input channels. Refer to AN1798.

14.3.3 Clock timing

The Σ–Δ modulator must run at a clock frequency (f_{Mod}, oversampling rate) not greater than 2.1MHz for the ADC. Converter logic is clocked by PCLK2. Double clocked synchronization for data crossing clock boundaries avoids any metastability issue. Based on the following equations, it is up to the user to correctly program the prescaler in order to generate the correct oversampling frequency based on the PCLK2 frequency:

f_S = Sampling Frequency of the input signal

f_{Mod} = Σ–Δ Modulator Sampling Rate

$f_S = f_{Mod} / \{512 * 4\}$

$f_{Mod} = f_{PCLK2} / \{Prescaling\ Factor\}$

Example: If f_{PCLK2} is 16 MHz, and the desired input signal sampling frequency f_s is 500 Hz, the Σ–Δ modulator must run at 1.024 MHz and the prescaler factor must be set to 0x8 (in the ADC_CPR register) to get a prescaling of 16.

Note: If the prescaler is set to generate a sampling frequency higher than specified, conversion performance and accuracy is not guaranteed

14.3.4 Gain and offset errors

An on-chip bandgap reference generates a 1.22V reference. This is used to generate two voltages used by the modulator — V_{CM} & V_{Ref}. V_{CM} is designed to be 1.25V, the midpoint of the converter's voltage range and V_{Ref} is the feedback reference, 1.85V. As the bandgap reference is not trimmed, absolute values of V_{CM} & V_{Ref} could be inaccurate by up to ± 5%. This will lead to gain and offset errors in the converter which can be calibrated out if necessary by software.

To calibrate the converter, it is necessary to input the minimum and maximum inputs supported — 0V and 2.5V. The digital output for a 0V input is the offset.

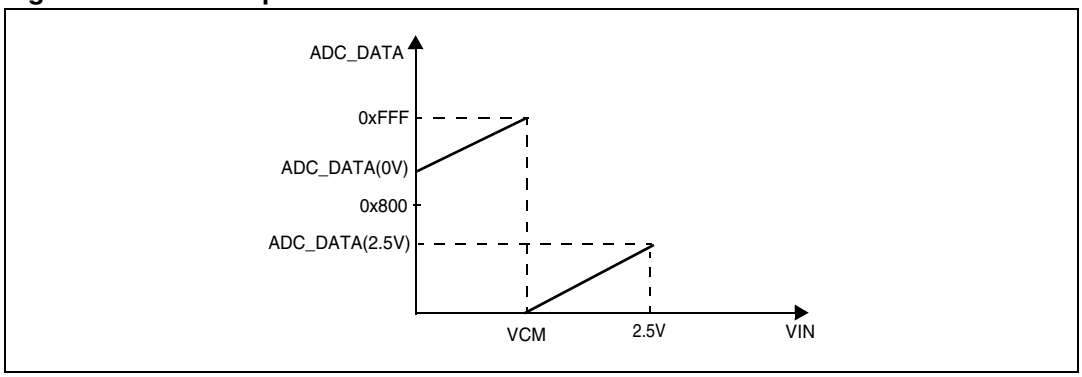
14.3.5 ADC output coding

The Σ–Δ converter produces a digital sample of each analog input channel every 512 oversampling clocks. The digital samples in output from the $Sinc^3$ digital filter are stored in the four ADC_DATA[n] registers as 16-bit samples of which only the first 12 MS bits are significant. The converted value stored in ADC_DATA[n] is a signed two's complent value

and proportional to the difference (V_{IN}-V_{CM}), being ideally 0 if the input voltage were V_{IN} = V_{CM}.

The folowing figure gives the ADC output (coded on 12 bits) versus the input voltage:

Figure 74. ADC output



ADC_DATA(0V) and ADC_DATA(2.5V) are respectiveley the conversion results for 0V and 2.5V. These two values must be determined in order to calculate the ADC gain using the following formula:

G = [0xFFF - ADC_DATA(0V) + ADC_DATA(2.5V)] / 2.5

*Note: The analog input voltage should not exceed twice the Center Voltage of the Σ-Δ Modulator (2 * V_{CM}) otherwise converter performances cannot be guaranteed Also the V_{CM} Voltage has an accuracy of +/- 5% which imposes a calibration of the converter.*

14.3.6 Power saving features

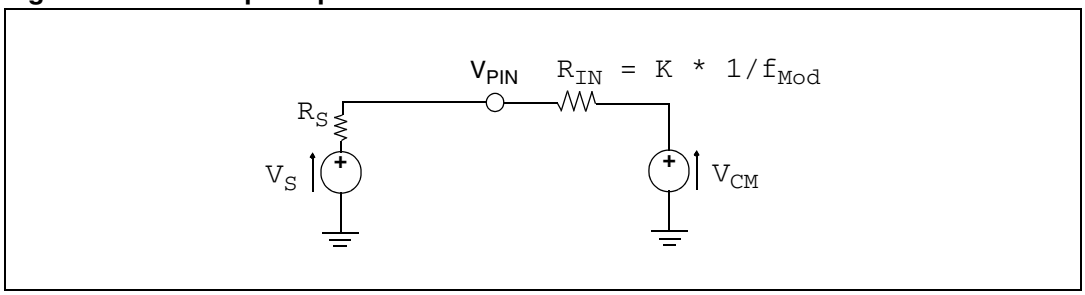
The analog circuitry of the ADC block is switched on when bit “ADC_EN” in the PCU_BOOTCR register is set. By default, this bit is cleared during reset and the power consumption via AV_{DD} / AV_{SS} -pins is minimized. It is recommended to disable the ADC by software before entering low power modes, if it was previously used.

Note: Disabling the ADC with “ADC_EN” only switches, the analog section of the Sigma-Delta Converter off. If ADC is not used, the digital section may be stopped as well by means of “Bit 7” of the APB2_CKDIS register. This disables the clock for the ADC.

14.3.7 ADC input equivalent circuit

The input equivalent circuit, due to the switching at f_{Mod} rate of the input capacitance where the charge taken from the input signal to be measured is stored, can be represented as shown in *Figure 75*.

Figure 75. ADC input equivalent circuit



where V_S is the voltage under measurement, R_S is the output resistance of the source, V_{PIN} the voltage that will actually be converted and R_{IN} the input equivalent resistance of the ADC. R_{IN} is inversely proportional to the modulator oversampling clock. The constant K depends on the operating mode of the converter and it is equal to:

- 750 [kΩ][MHz] +/- 20% - single channel conversion mode
- 3000 [kΩ][MHz] +/- 20% - round robin conversion mode.

14.4 Register description

14.4.1 ADC control/status register (ADC_CSR)

Address Offset: 20h
Reset Value: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved		OR	Res	IE[3:0]				Res	AXT	A[1:0]		DA[3:0]			
		rw		rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

This register controls the operating mode of the ADC, sets the interrupt enables, contains status flags for the availability of data and error flags in the event of data being overwritten before being read.

Bits 15:14	Reserved, must be kept at reset value (0).
Bit 13	**OR:** *OverRun* This read-write bit is used to notify application software that data on one of the channels has been overwritten before being read. 0: Normal operation. No overrun has occurred. 1: Overrun event occurred. This bit is set by hardware as soon as an overrun condition is detected and must be cleared by software by explicitly writing it to “0”.
Bit 12	Reserved, must be kept at reset value (0).
Bits 11:8	**IE[3:0]:** *Interrupt Enable* This set of bits allows to enable interrupt requests independently for each of the ADC channels, where bit IE[n] corresponds to ADC channel n. 0: Channel [n] interrupt disabled. 1: Channel [n] interrupt enabled.
Bit 7	Reserved, must be kept at reset value (0).

Bit 6	**AXT:** *Addressing eXTernal enable* This bit allows to enable the single-channel operation, configuring the ADC to convert repeatedly the channel identified by A[1:0] bits of this register. 0: Round-robin addressing enabled. 1: Single-channel addressing enabled.
Bits 5:4	**A[1:0]:** *Channel Address* These bits select the external channel to be sampled when external addressing is enabled.
Bits 3:0	**DA[3:0]:** *Data Available* This read-write set of bits allows to determine on which channel data register a new sample is ready to be read, with bit DA[n] corresponds to ADC channel n. 0: No sample available on the corresponding channel. 1: New sample on the corresponding channel is available. This bit is set by hardware as soon as a new sample is available and must be cleared by software by explicitly writing it to “0”. Writing it to “1” has no effect. These bits also act as interrupt flags for the corresponding channels.

14.4.2 ADC clock prescaler register (ADC_CPR)

Address Offset: 30h

Reset Value: 05h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				Presc[11:0]											
				w	w	w	w	w	w	w	w	w	w	w	w

Bits 15:12	Reserved, must be kept at reset value (0).
Bits 11:0	**PRESC[11:0]:** *Prescaler value* The 12-bit binary value specified on the clock prescaler register determines the factor by which the ADC input clock will be divided down in order to produce the oversampling clock of the sigma-delta modulator, the actual factor being twice the PRESC register value. The value placed in this register must subsequently generate an oversampling clock frequency not greater than 2.1 MHz from the PCLK2 clock applied to the ADC. These bits can only be written by software, any read operation on them returns 0h. 00h: Not available 01h: Not available 02h: $f_{ADC}=f_{PCLK2}/4$ 03h: $f_{ADC}=f_{PCLK2}/6$ 04h: $f_{ADC}=f_{PCLK2}/8$ 05h: $f_{ADC}=f_{PCLK2}/10$ ..... 0FFEh: $f_{ADC}=f_{PCLK2}/8188$ 0FFFh: $f_{ADC}=f_{PCLK2}/8190$

14.4.3 ADC data register n, n = 0...3 (ADC_DATA[n])

Address Offset: 0x00h *(channel 0)*
Address Offset: 0x08h *(channel 1)*
Address Offset: 0x10h *(channel 2)*
Address Offset: 0x18h *(channel 3)*
Reset Values: 0000h

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data[11:0]												not relevant			
r	r	r	r	r	r	r	r	r	r	r	r				

Four data registers, one for each of the analog input channels, are available. The 12 most significant bits will contain the result of the conversion, while the 4 least significant bits of each register should be ignored. The data registers will be filled in numerical sequence in the round-robin channel mode. In single channel mode, only the selected channel will be updated.

Bits 15:4	**DATA[11:0]:** *DATA sample* This read-only register contains the last sampled value on the corresponding channel.

14.5 ADC register map

Table 53. ADC Register Map

Addr. Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00	ADC_DATA0	DATA0[11:0]												N/U			
08	ADC_DATA1	DATA1[11:0]												N/U			
10	ADC_DATA2	DATA2[11:0]												N/U			
18	ADC_DATA3	DATA3[11:0]												N/U			
20	ADC_CSR	-	-	OR	-	IE[3:0]				-	AXT	A[1:0]		DA3	DA2	DA1	DA0
30	ADC_CPR	-									PRE[6:0]						

See *Table 3 on page 14* for the base address

15 APB bridge registers

The on-chip peripherals are addressable via two APB bridges, APB1 and APB2.

Each bridge has two 32-bit registers. Under software control, each peripheral can be individually reset using the SWRES register. The PCLK signal to each peripheral (except the Watchdog) also can be enabled/disabled individually using the CKDIS register. The CKDIS register is also used to enable/disable the signal on the CKOUT pin. The clock output from this pin is the PCLK2. The frequency, as programmed through the PRCCU (APBDIV register, see *Section 15*)

Note: The APB Bridge registers MUST be accessed with 32-bit aligned operations (i.e. no byte/half word cycles are allowed).

15.1 APB clock disable register (APBn_CKDIS)

Address Offset: 0x10h

Reset value: 0x0000 0000h

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved															

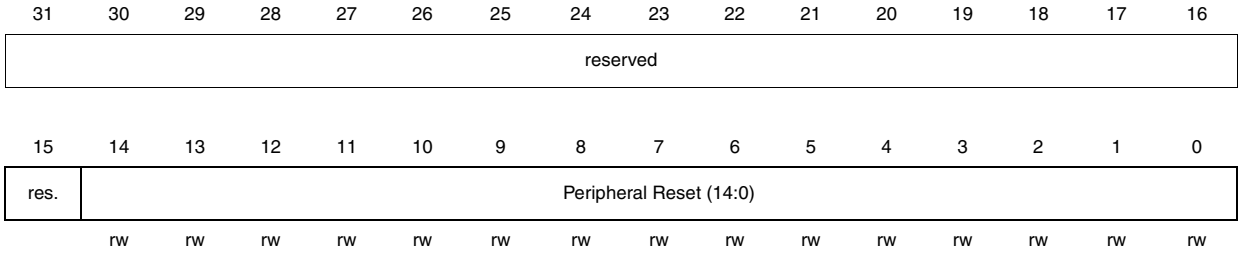
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res.	Peripheral Clock Disable (14:0)														
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:15	Reserved, must be kept at reset value (0).
Bits 14:0	**Peripheral Clock Disable [14:0]** Each of these bits control the clock gating of the corresponding APB1 or APB2 peripherals listed in *Table 54* and *Table 55*. 0: Peripheral clock enabled 1: Peripheral clock disabled

15.2 APB Software Reset Register (APBn_SWRES)

Address Offset: 0x14h

Reset value: 0x0000 0000h



Bits 31:15	Reserved, must be kept at reset value (0).
Bits 14:0	**Peripheral Reset [14:0]** Each of these bits controls the activation of the Reset to the corresponding APB1 or APB2 peripherals listed in *Table 54* and *Table 55*. 0: The peripheral is reset by the system-wide Reset 1: The peripheral is kept under reset independently from the System-wide Reset.

Table 54. APB1 peripherals

Bit No	Peripheral
31:14	Reserved, must be kept at reset value
13	HDLC
12:11	Reserved, must be kept at reset value
10	BSPI1
9	BSPI0
8	CAN
7	USB
6	UART3
5	UART2
4	UART1 + SMARTCARD
3	UART0
2	Reserved
1	I2C1
0	I2C0

Table 55. APB2 peripherals

Bit No	Peripheral
31:15	Reserved, must be kept at reset value
14	EIC
13	Reserved, must be kept at reset value

Table 55. APB2 peripherals

Bit No	Peripheral
12	RTC
11	TIMER3
10	TIMER2
9	TIMER1
8	TIMER0
7	CKOUT
6	ADC
5	Reserved, must be kept at reset value
4	IOPORT2
3	IOPORT1
2	IOPORT0
1	Reserved, must be kept at reset value
0	XTI

15.3 APB register map

Table 56. APBn Register Map

Addr. Offset	Register Name	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
00h	Reserved	Reserved															
04h	Reserved	Reserved															
08h	Reserved	Reserved															
0Ch	Reserved	Reserved															
10h	APBn_CKDIS	res.	Peripheral clock disable[14:0]														
14h	APBn_SWRES	res.	Peripheral reset [14:0]														

See *Table 2 on page 13* and *Table 3 on page 14* for base addresses

16 JTAG interface

16.1 Overview

STR71x is built around an ARM7TDMI core whose debug interface is Joint Test Action Group (JTAG) based. ARM7TDMI contains hardware extensions for advanced debugging features. The debug extensions allow the core to be stopped either on a given instruction fetch (breakpoint) or data access (watchpoint) or asynchronously by a debug-request. When this happens, ARM7TDMI® is said to be in *debug* state. At this point, the core's internal state and the system's external state may be examined. Once examination is complete, the core and the system may be restored and program execution resumed.

ARM7TDMI is forced into debug state either by a request on one of the external interface signals or by an internal functional unit known as In-circuit Emulation Unit (ICE). Once in debug state, the core isolates itself from the memory system. The core can then be examined while all other system activity continues as normal.

ARM7TDMI's internal state is examined via a JTAG-style serial interface, which allows instructions to be serially inserted into the core's pipeline without using the data bus. Thus, when in debug state, a store-multiple (STM) could be inserted into the instruction pipeline and this would dump the contents of the ARM7TDMI® registers. This data can be serially shifted out without affecting the rest of the system.

16.2 Debug system

The ARM7TDMI® is only one component of a more complex debug system. This is typically composed of three parts: the debug host, the protocol converted and the ARM7TDMI® core.

16.2.1 The debug host

The debug host is typically a computer running a software debugger. The debug host allows the user to issue high level commands such as "set breakpoint at location XX" or "examine the contents of memory from address 0x0 to 0x100".

16.2.2 The protocol converter

The debug host will be connected to the ARM7TDMI development system via an interface (RS232, for example). The messages broadcast over this link must be converted to the interface signals of the core. This function is performed by the protocol converter.

16.2.3 ARM7TDMI

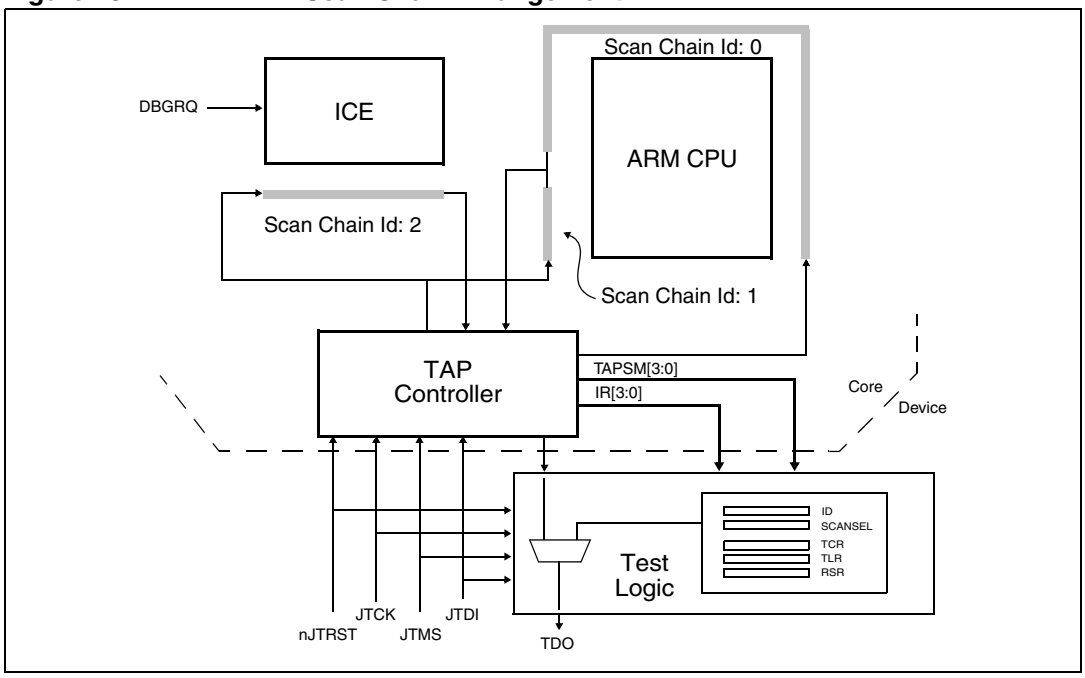
The ARM7TDMI is the lowest level of the system. Its debug extensions allow the user to stall the core from program execution, examine its internal state and the state of the memory system and then resume the program execution.

16.3 ARM7TDMI debug interface

The details of the ARM7TDMI® hardware extensions for advanced debugging are shown in *Figure 76*. The main blocks are:

- The CPU core, with hardware support for debug.
- The In-Circuit Emulation unit (ICE), which,through a set of registers and comparators, generates debug exceptions (breakpoints or watchpoints).
- The Test Access Port (TAP) controller.
- The Scan Chains 0, 1 and 2 which surround ARM CPU and ICE.
- The device Test Logic.

Figure 76. ARM7TDMI Scan Chain Arrangement



16.3.1 Physical interface signals

According to IEEE 1149.1 standard (JTAG), the physical interface to the TAP controller is based on five signals. Beyond their usage, the standard also gives a set of indications about their reset status and the usage of pull-up resistors. These signals are:

- **nJTRST:** Test Reset. Active low reset signal for the TAP controller finite state machine. This pin has to be held low at power-on in such a way so as to produce an initialization (reset) of the controller. When out of reset, the pin must be pulled up. When the JTAG interface is not in use, it may be held in its reset status, by grounding the nJTRST pin.
- **JTDI:** Test Data Input. To be pulled up by an external resistor.
- **JTMS:** Test Mode Select. To be pulled up by an external resistor. It must be high during '0' to '1' transition of nJTRST.
- **JTCK:** Test Clock. This clock is used to advance the TAP controller finite state machine. The TAP state machine maintains its state indefinitely when JTCK is held low. Optionally, similar behaviour is also obtained by holding the pin TCK high. The

standard does not impose any pull-up or pull-down resistor. A floating input is not recommended, to avoid any static power consumption.

- **JTDO:** Test Data Output. This is the output from the boundary scan logic. This output is in high impedance when not in use.

According to the standard, all interface signals should have an external pull-up or pull-down resistor as follows:

- nJTRST Pull-up
- JTDI Pull-up
- JTMS Pull-up
- JTCK Pull-down or Pull up
- JTDO Floating or Pull-up/down (no static consumption anyway)

In STR71x, none of the previous resistors are implemented internally. Therefore, they need to be implemented on the application board. In case, the JTAG interface is not used all the interface pins can be grounded indefinitely. This will not introduce any additional power consumption since there are no internal pull-up resistors. The following additional signals, not part of the IEEE 1149.1 standard interface, are also made externally available to allow external user logic to request debug events:

- DBGRQ. When high, the system requests the ARM7TDMI to unconditionally enter the debug state. This pin must be kept LOW when emulation features are not enabled.

16.3.2 JTAG ID code

STR71x devices have two JTAG ID codes:

- JTAG standard ID code (accessible with standard JTAG instruction = 0b 1110):
  - 0x3F0F0F0
- Device ID code (accessible with extended JTAG instruction = 0b 0001):
  - 0xv221C041 (where v bits are reserved and to be ignored)

17 Revision history

Table 57. Revision history

Date	Revision	Description of Changes
02-Jul-2007	1	Created new document RM0002 to replace UM0084 and restart revision numbering. Changes compared to the last revision of UM0084 (Rev 8 dated 8-Nov-2006): Modified EMI *Figure 3 on page 21* through *Figure 5 on page 23* Updated *Section 2.6.2: Clock flag register (RCCU_CFR)*, bits 14:11 read/clear write 1. Modified *Section 3.1.3: Alternate function I/O (AF) on page 58* Added *Section 5.3.4: RTC flag assertion on page 95* Updated BSPI *Section 10.7: Start-up status* Corrected description of *UART guardtime register (UARTn_GTR) on page 210* Added USB startup time in *Section 13.5.2 on page 219* Added *Table 54: APB1 peripherals* and *Table 55: APB2 peripherals*

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 any time, 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 AN AUTHORIZED ST REPRESENTATIVE, 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.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com