NUC501 IP Programming Guide

V1.00

Publication Release Date: Feb. 2009

Table of Contents

Table of Figures

1. Introduction

The NUC501 is an ARM7TDMI-based MCU, specifically designed to offer low-cost and high performance for various applications, like interactive toys, edutainment robots, and home appliances. It integrates the 32-bit RISC CPU with 32KB high-speed SRAM, crypto engine with OTP key, boot ROM, LDO regulator, ADC, DAC, I2C, SPI, USB2.0 FS Device, & GPIO into a cost-affordable while feature-rich micro-controller.

Owing to the simplicity of the NUC501 architecture that boots SpiMemory1 into the high-speed SRAM for program execution, the total system BOM is reduced to its minimum. Unlike usual ARM-based MCU products, the NUC501 operates without the use of SDRAM, which is usually the source of complexity, higher power consumption, and cost.

The ARM7TDMI runs up to 108MHz on the high-speed SRAM to offer enough horsepower for many MIPS-hungry tasks, while the remaining MIPS is still able to serve the need of application program. For those applications, like cartridge games, that require large code storage and variation of game play scenarios, the patented Extensible XIP Addressing on SpiMemory gives the flexibility whenever program execution speed is not a critical concern.

To protect the code against illegal pirating, the NUC501 provides a crypto engine that works with internal OTP2 key to encrypt the data stored at external SpiMemory when the design-in is finished. Without the knowledge of the OTP key, others can’t decrypt the data even by means of ICE debugging.

The NUC501 is designed with special care to minimize the power consumption while allowing for the flexibility to reach for high performance. It includes the clock gating, variable frequency control for individual IP’s, and bus control to reduce signal toggle. Besides, the NUC501 can be further operated under different power-saving modes: idle, power down with RTC active, and power down mode.

With so many practical peripherals integrated around the high-performance ARM7 CPU, the NUC501 is suitable for such applications as Interactive toys, edutainment robots, and home appliances. Whenever MIPS-hungry task meets cost-effective demand, you’ll find the NUC501 truly useful to satisfy the requirement.

1.1. Block Diagram

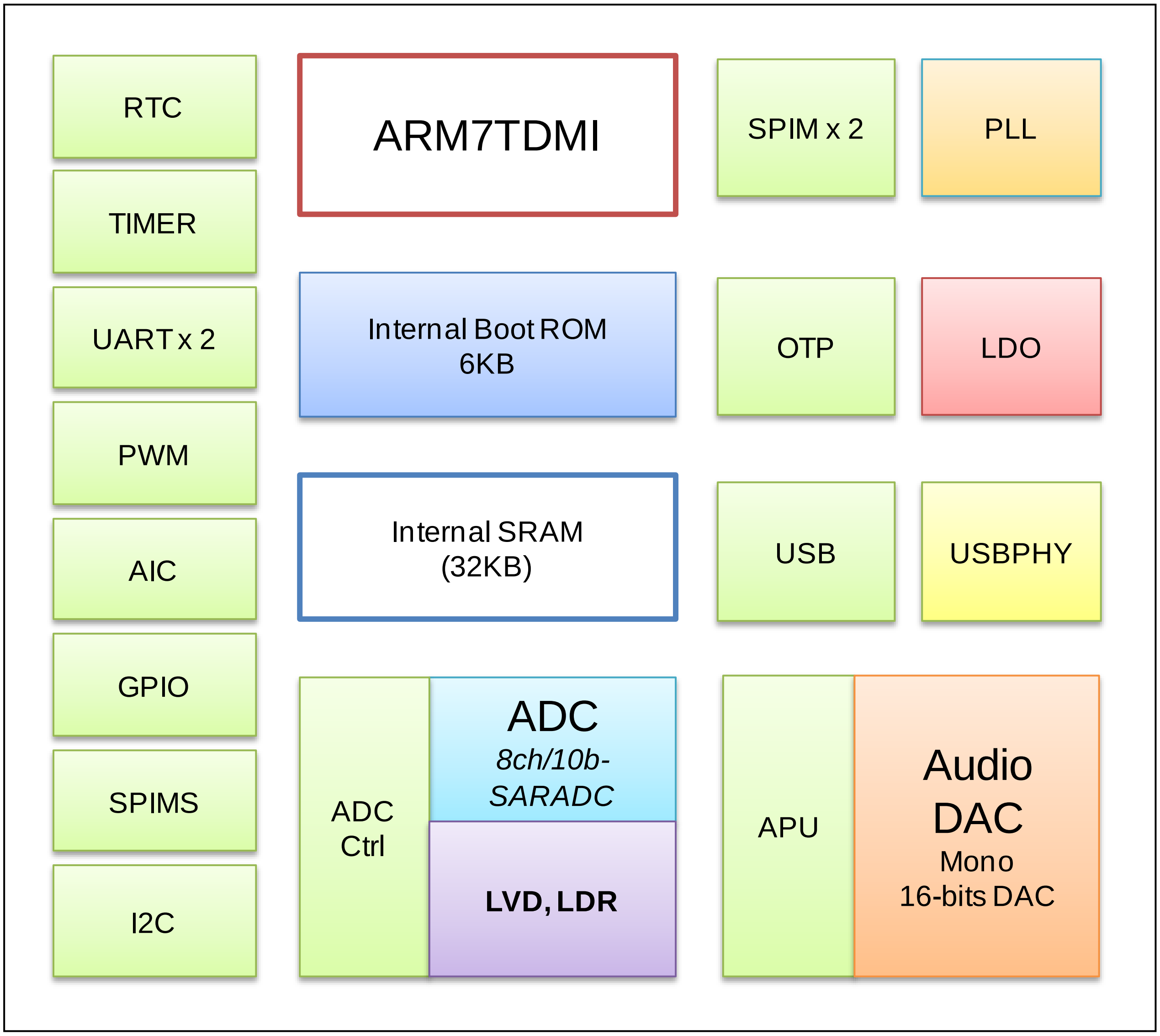


Figure 1-1 NUC501 Functional Block Diagram

On the following chapters, programming note of each chapter will be described in detailed.

- Chapter 2 : System Manager
- Chapter 3 : Advanced Interrupt Controller
- Chapter 4 : SPI Synchronous Serial Interface Controller
- Chapter 5 : Analog to Digital Converter
- Chapter 6 : Analog Processing Unit
- Chapter 7 : I^2C Synchronous Serial Interface Controller
- Chapter 8 : General Purpose I/O
- Chapter 9 : Pulse Width Modulation
- Chapter 10: Real Time Clock
- Chapter 11 : Serial Peripheral Interface Controller (SPI Master/Slave)

- Chapter 12 : Timer and WDT
- Chapter 13 : UART
- Chapter 14 : USB

2. System Manager (SYS)

2.1. Overview

The following functions are included in system manager section

- System memory map
- Bus arbitration algorithm
- Clock controller
- SRAM bank mapping
- System suspend

2.2. System Memory Map

NUC501 provides a 4G-byte address space for programmers. The memory locations assigned to each on-chip modules are shown in following table. The detailed register and memory addressing and programming will be described in the following sections for individual on-chip modules. NUC501 only supports little-endian data format.

Address Space	Token	Modules
Memory Space		
0x0000_0000 – 0x0000_7FFF	IBR_BA	Internal Boot ROM (IBR) Memory Space (IBR_remap = 0)
0x0000_0000 – 0x1FFF_FFFF	SRAM_BA	SRAM Memory Space (IBR_remap = 1)
0x2000_0000 – 0x3FFF_FFFF	SRAM_BA	SRAM Memory Space (IBR_remap = 0)
0x4000_0000 – 0x4FFF_FFFF		SPI Flash/ROM Memory Space (SPIM0)
0x6000_0000 – 0x6000_7FFF	IBR_BA	Internal Boot ROM (IBR) Memory Space (IBR_remap = 1)
AHB Modules Space		
0xB100_0000 – 0xB100_01FF	GCR_BA	Global Control Registers
0xB100_0200 – 0xB100_02FF	CLK_BA	Clock Control Registers
0xB100_4000 – 0xB100_4FFF	SRAMCTL_BA	SRAM Control Registers
0xB100_7000 – 0xB100_7FFF	SPIM0_BA	SPIM0 Control Register
0xB100_8000 – 0xB100_8FFF	APU_BA	Audio Process Unit (APU) Controller Registers

0xB100_9000 – 0xB100_9FFF	USB_BA	USB Device Controller Registers
APB Modules Space		
0xB800_1000 – 0xB800_1FFF	ADC_BA	Analog-Digital-Converter (ADC) Controller Registers
0xB800_2000 – 0xB800_2FFF	AIC_BA	Interrupt Controller Registers
0xB800_3000 – 0xB800_3FFF	GPIO_BA	GPIO Controller Registers
0xB800_4000 – 0xB800_4FFF	I2C_BA	I2C Interface Control Registers
0xB800_7000 – 0xB800_7FFF	PWM_BA	PWM Controller Registers
0xB800_8000 – 0xB800_8FFF	RTC_BA	Real Time Clock (RTC) Control Register
0xB800_A000 – 0xB800_AFFF	SPIMS_BA	SPI master/slave function Controller Registers
0xB800_B000 – 0xB800_BFFF	TIMER_BA	Timer Control Registers
0xB800_C000 – 0xB800_CFFF	UART_BA	UART Control Registers

Table 1 : Memory Map

2.3. AHB Bus Arbitration

The internal bus of NUC501 chip is an AHB-compliant Bus and supports to connect with the standard AHB master or slave. NUC501's AHB arbiter provides a choice of two arbitration algorithms for simultaneous requests. These two arbitration algorithms are the d-priority mode and the round-robin-priority (rotate) mode. The selection of modes and types is determined on the **PRTMOD0** control register in the Arbitration Control Register.

AHB bus arbiter also provides a mechanism for the maximum burst length for each AHB bus transfer. The maximum burst length is 16, and when the current AHB data transfer count is equal to the maximum burst length, the access of current AHB bus owner will be broken.

Register	Address	R/W	Description	Default Value
AHB_CTRL	GCR_BA+0x20	R/W	AHB Control Register	0x0000_0000

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved							
7	6	5	4	3	2	1	0
Reserved		**IPACT**	**IPEN**	**Reserved**			**PRTMOD0**

2.4. Priority Mode

2.4.1. Fixed Priority Mode

Fixed priority mode is selected if **PRTMOD** = 0. The order of priorities on the AHB mastership among the on-chip master modules, listed in following table. If two or more master modules request to access AHB bus at the same time, the higher priority request will get the permission to access AHB bus.

Priority Sequence	AHB Bus Priority PRTMOD[0] = 0
1 (Lowest)	ARM7TDMI
2	SPIM0
3 (Highest)	APU

The SPI flash controller normally has the lowest priority under the fixed priority mode. NUC501 provides a mechanism to raise the priority of CPU request to the highest. If the IPEN bit (bit-4 of *AHB Control Register*) is set to 1, the **IPACT** bit (bit-5 of *AHB Control Register*) will be automatically set to 1 while an unmasked external NFIQ or NIRQ occurs. Under this circumstance, the ARM core will become the highest priority to access AHB bus.
The programmer can recover the original priority order by directly writing “1” to clear the **IPACT** bit. For example, this can be done that at the end of an interrupt service routine. Note that **IPACT** only can be automatically set to 1 by an external interrupt when **IPEN** = 1. It will not take effect for a programmer to directly write 1 to **IPACT** to raise ARM core’s AHB priority.

2.4.2. Round Robin Priority Mode

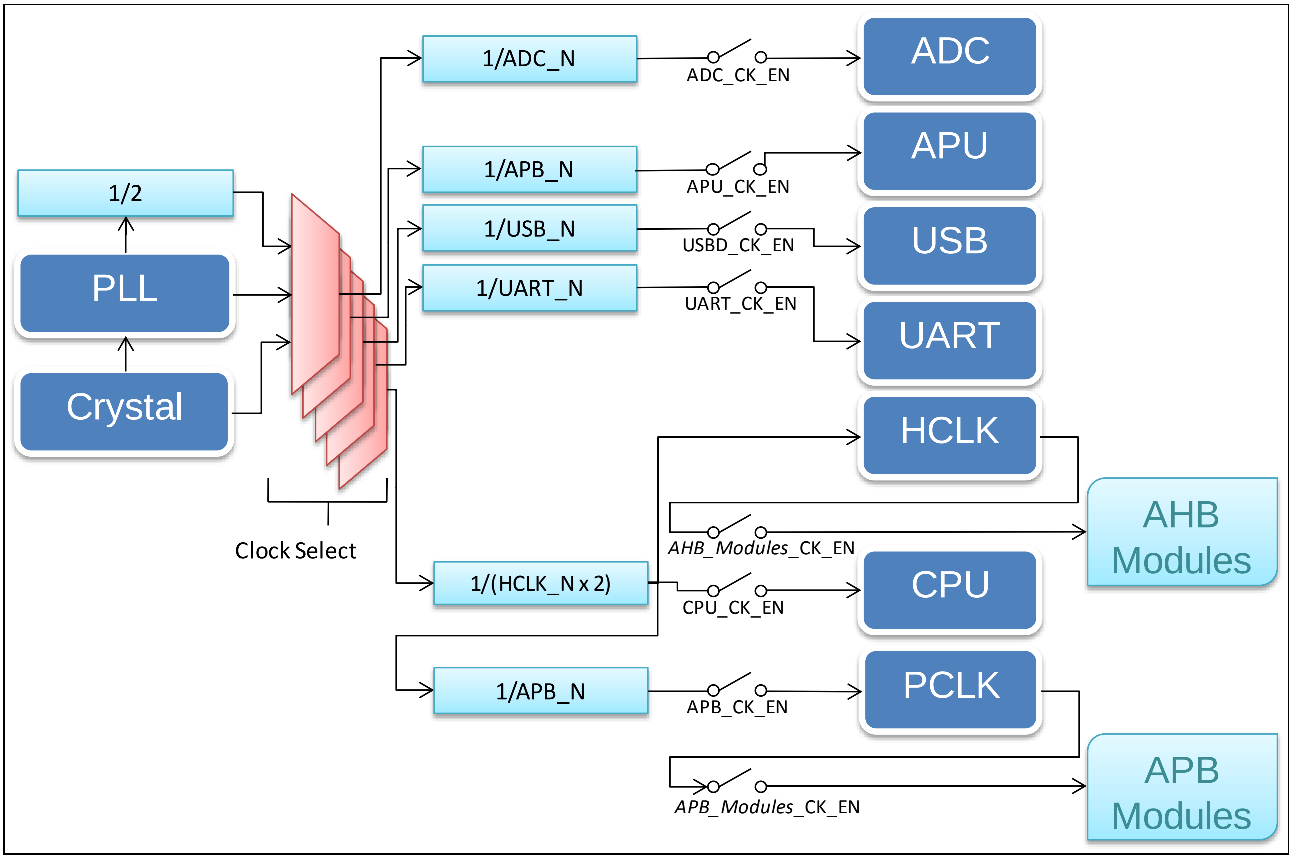
Round-robin priority mode is selected if PRTMOD = 1. The AHB bus arbiter uses a round robin arbitration scheme for every master module to gain the bus ownership in turn. That is the requestor having the highest priority becomes the lowest-priority requestor after it has been granted access.

2.5. Clock Controller

The clock controller generates the clocks for the whole chip, it include all AMBA interface modules and all peripheral clocks, the USB, UART, APU and so on. There is one PLL modules in this chip, and the PLL clock source is from the external crystal input.

The clock controller implements the power control function, include the individually clock on or off control register, clock source select and the divided number from clock source. These functions minimize the extra power consumption and the chip run on the just condition. On the power down mode the controller turn off the crystal oscillator to minimize the chip power consumption.

The clock HCLK is the source for all the AMBA modules. The HCLK is the operating clock for the SRAM and it is divided by two from one of the sources, Crystal, PLL, PLL/2 and the crystal 32 KHz, the HCLK is used for the AMBA AHB BUS clock. The ARM7 CPU uses the same frequency as the HCLK. The APB clock is divided from the HCLK too.



MPLL Control Register （MPLLCON）

The MPLL reference clock input is directly from the external clock input, and the other PLL control inputs are connected to bits of the registers.

Register	Address	R/W	Description	Reset Value
MPLLCON	CLK_BA + 20	R/W	**MPLL Control Register**	0x0001_4035

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved					**OE**	**BP**	**PD**
15	14	13	12	11	10	9	8
OUT_DV		***IN_DV***					***FB_DV***
7	6	5	4	3	2	1	0
FB_DV							

Output Clock Frequency Setting:

FOUT = FIN * NF/NR*1/NO

Constrain:

- 3.2MHz < FIN <150MHz
- 800KHz < FIN/NR < 8MHz
- 200MHz < FCO = FIN*NF/NR < 500MHz
- 250MHz < FCO is preferred

Where

FOUT	Output Clock Frequency
FIN	Input (Reference) Clock Frequency
NR	Input Divider (2 x (IN_DV + 2))
NF	Feedback Divider (2 x (FB_DV + 2))
NO	OUT_DV = “00” : NO = 1 OUT_DV = “01” : NO = 2 OUT_DV = “10” : NO = 2 OUT_DV = “11” : NO = 4

AHB Devices Clock Enable Control Register （AHBCLK）

These register bits are used to enable/disable clock for AMBA clock, AHB engine and peripheral

Register	Address	R/W	Description	Reset Value
AHBCLK	CLK_BA + 04	R/W	**AHB Devices Clock Enable Control Register**	0x0000_0083

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

Reserved							
15	14	13	12	11	10	9	8
Reserved							APU_CK_EN
7	6	5	4	3	2	1	0
SPIM_CK_EN	USBD_CK_EN	Reserved				APB_CK_EN	CPU_CK_EN

APB Devices Clock Enable Control Register （APBCLK）

These register bits are used to enable/disable clock for APB engine and peripheral.

Register	Address	R/W	Description	Reset Value
APBCLK	CLK_BA + 08	R/W	APB Devices Clock Enable Control Register	0x0000_0007

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
Reserved						ADC_CK_EN	SPIMS_CK_EN
7	6	5	4	3	2	1	0
Reserved	I2C_CK_EN	PWM_CK_EN	UART1_CK_EN	UART0_CK_EN	RTC_CK_EN	WD_CK_EN	TIMER_CK_EN

Clock Source Select Control Register （CLKSEL）

Before clock switch the related clock sources (pre-select and new-select) must be turn on.

Register	Address	R/W	Description	Reset Value
CLKSEL	CLK_BA + 10	R/W	Clock Source Select Control Register	0x0000_0000

31	30	29	28	27	26	25	24
Reserved							
23	22	21	20	19	18	17	16
Reserved							
15	14	13	12	11	10	9	8
ADC_S		Reserved					

7	6	5	4	3	2	1	0
UART_S		**APU_S**		**USB_S**		**HCLK_S**	

Clock Divider Register1 (CLKDIV1)

Register	Address	R/W	Description	Reset Value
CLKDIV1	CLK_BA_+ 18	R/W	**Clock Divider Number Register**	0x0000_0000

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

2.6. SRAM Controller

The SRAM controller is design for program code and data storage. It's an AHB slave and SRAM size is up to 32KB. This 32KB memory is separated into 16 memory block and the size of each memory block is 2KB. Each memory block could be randomly mapped to any 2KB space of 0x0000_0000 ~ 0x1FFF_FFFF of system memory by modifying the control register. Each 2KB memory block could also be disabled individually by modifying control register.
In default, these 16*2KB memory blocks are all enabled and mapped to 0x0000_0000 ~ 0x0000_7FFF sequentially. There are 2 features list as following

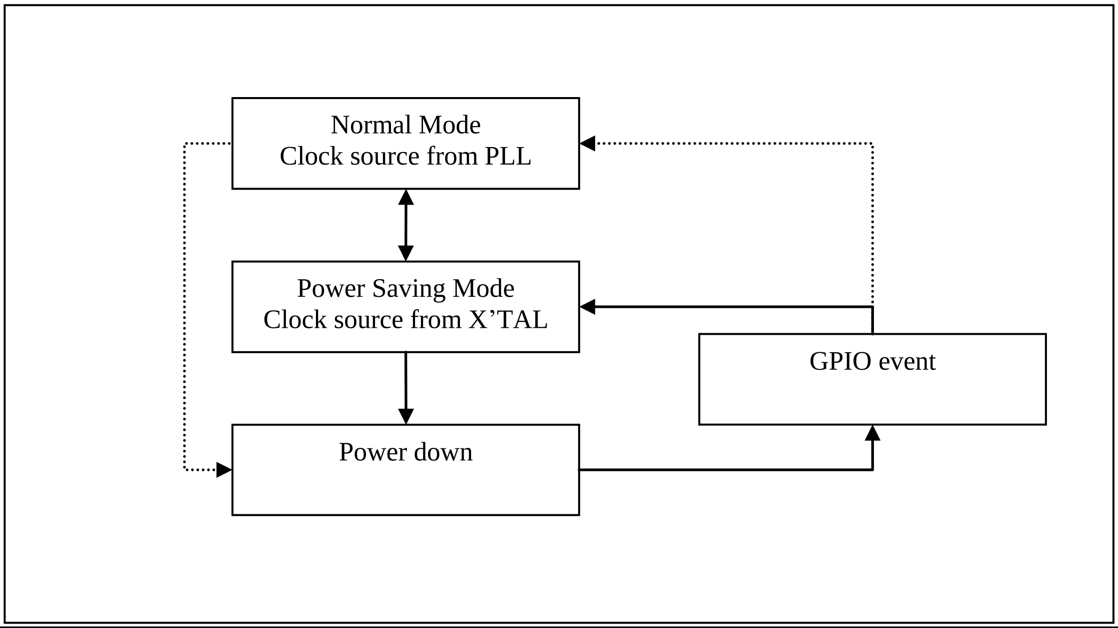
- Support maximum SRAM size is 32KB that cascade 16 banks SRAM.
- Support random memory address mapping in 2KB space of 0x0000_0000 ~ 0x1FFF_FFFF of system memory.

2.7. Power Manager Mode

The NUC501 is designed with special care to minimize the power consumption while allowing for the flexibility to reach for high performance. It includes the clock gating, variable frequency control for individual IP's, and bus control to reduce signal toggle. Besides, the NUC501 can be further operated under different power-saving modes: idle, power down with RTC active, and power down mode. The following figure is the control sequence to enter power down mode or wake up from GPIO. Due to NUC501 only has SRAM, system can enter power down mode directly without switching to external clock. **PWRCON[XTAL_EN]** = 0, system enter power down mode. If system is in power down mode, a GPIO event can wake up the system. However, the system clock may be unstable**. PWRCON[Pre-Scale]** sets time between wake-up to system receiving the stable clock.

Register	Address	R/W	Description	Reset Value
PWRCON	CLK_BA + 00	R/W	**System Power Down Control Register**	0x00FF_FF03

31	30	29	28	27	26	25	24
Reserved							
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Pre-Scale[15:8]							
15	14	13	12	11	10	9	8
Pre-Scale[7:0]							
7	6	5	4	3	2	1	0
Reserved				**INT_EN**	**INTSTS**	**XIN_CTL**	**XTAL_EN**



3. Advanced Interrupt Controller (AIC)

3.1. Overview

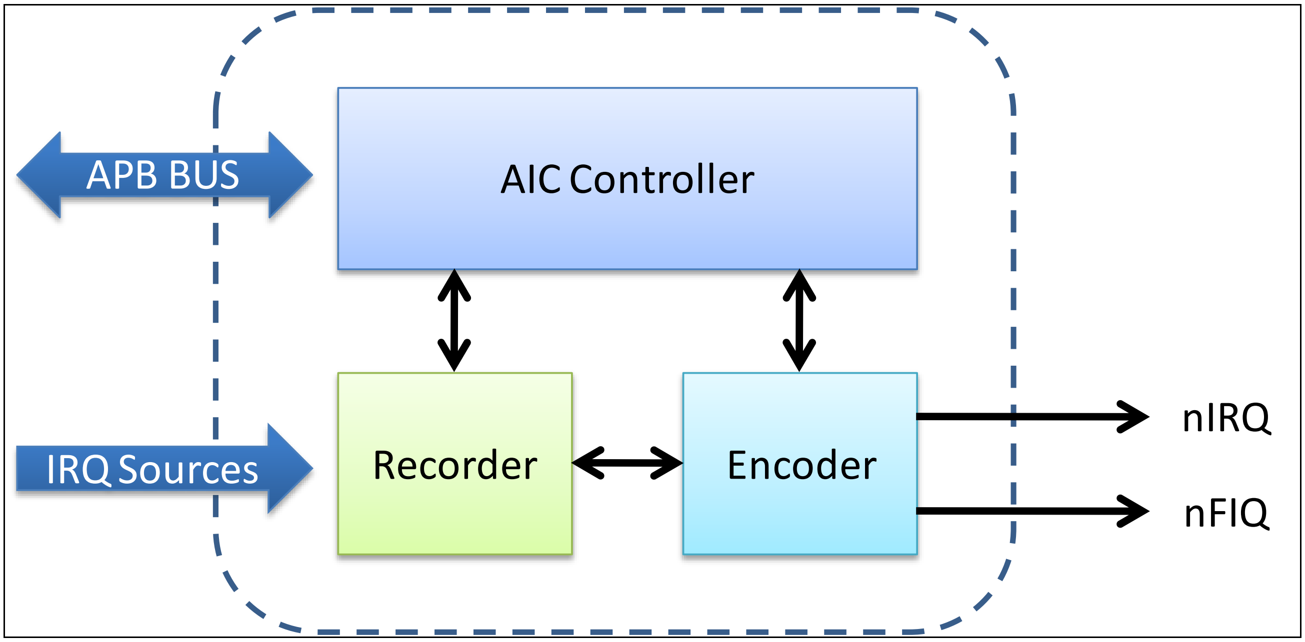
An *interrupt* temporarily changes the sequence of program execution to react to a particular event such as power failure, watchdog timer timeout, transmit/receive request from Serial Interface (UART or SPI) Controller, and so on. The ARM processor provides two modes of interrupt, the **Fast Interrupt (FIQ)** mode for critical session and the **Interrupt (IRQ)** mode for general purpose. The IRQ exception is occurred when the nIRQ input is asserted. Similarly, the FIQ exception is occurred when the nFIQ input is asserted. The FIQ has privilege over the IRQ and can preempt an ongoing IRQ. It is possible to ignore the FIQ and the IRQ by setting the F and I bits in the **current program status register (CPSR)**.

The NUC501 incorporates the **advanced interrupt controller (AIC)** that is capable of dealing with the interrupt requests from a total of 32 different sources. Currently, 31 interrupt sources are defined. Each interrupt source is uniquely assigned to an *interrupt channel*.

The advanced interrupt controller includes the following features:

- AMBA APB bus interface
- External interrupts can be programmed as either edge-triggered or level-sensitive
- External interrupts can be programmed as either low-active or high-active
- Has flags to reflect the status of each interrupt source
- Individual mask for each interrupt source
- Proprietary 8-level interrupt scheme to ease the burden from the interrupt
- Priority methodology is adopted to allow for interrupt daisy-chaining
- Automatically masking out the lower priority interrupt during interrupt nesting
- Automatically clearing the interrupt flag when the external interrupt source is programmed to be edge-triggered.

3.2. Block Diagram



3.3. Interrupt Source

Channel	Name	SCR	Source	Reset (default) level
1	WDT_INT	SCR1[15:8]	Watch Dog Timer Interrupt	Low
2	**Reserved**	**Reserved**	**Reserved**	Low
3	INT_GPIO0	SCR1[31:24]	GPIO Interrupt0	Low
4	INT_GPIO1	SCR2[7:0]	GPIO Interrupt1	Low
5	INT_GPIO2	SCR2[15:8]	GPIO Interrupt2	Low
6	INT_GPIO3	SCR2[23:16]	GPIO Interrupt3	Low
7	INT_APU	SCR2[31:24]	Audio Processing Unit Interrupt	Low

8	**Reserved**	**Reserved**	**Reserved**	Low
9	**Reserved**	**Reserved**	**Reserved**	Low
10	INT_ADC	SCR3[23:16]	AD Converter Interrupt	Low
11	INT_RTC	SCR3[31:24]	RTC Interrupt	Low
12	INT_UART0	SCR4[7:0]	UART-0 Interrupt	Low
13	INT_UART1	SCR4[15:8]	UART-1 Interrupt	Low
14	INT_TMR1	SCR4[23:16]	Timer-1 Interrupt	Low
15	INT_TMR0	SCR4[31:24]	Timer-0 Interrupt	Low
16	**Reserved**	**Reserved**	**Reserved**	Low
17	**Reserved**	**Reserved**	**Reserved**	Low
18	**Reserved**	**Reserved**	**Reserved**	Low
19	INT_USB	SCR5[31:24]	USB Device Interrupt(Notes)	Low
20	**Reserved**	**Reserved**	**Reserved**	Low
21	**Reserved**	**Reserved**	**Reserved**	Low
22	INT_PWM0	SCR6[23:16]	PWM Interrupt0	Low
23	INT_PWM1	SCR6[31:24]	PWM Interrupt1	Low
24	INT_PWM2	SCR7[7:0]	PWM Interrupt2	Low
25	INT_PWM3	SCR7[15:8]	PWM Interrupt3	Low
26	INT_I2C	SCR7[23:16]	I2C Interface Interrupt	Low
27	INT_SPIMS	SCR7[31:24]	SPI (Master/Slave) Serial Interface Interrupt	Low
28	**Reserved**	**Reserved**	**Reserved**	Low
29	INT_PWR	SCR8[15:8]	System Wake-Up Interrupt	Low
30	INT_SPI_ROM	SCR8[23:16]	SPI ROM Interrupt	Low
31	**Reserved**	**Reserved**	**Reserved**	Low

3.4. Registers

Register		R/W	Description	Reset Value
Base Address		0xB800_2000		
AIC_SCR1	AIC_BA+000	R/W	Source Control Register 1	0x4747_4747
AIC_SCR2	AIC_BA+004	R/W	Source Control Register 2	0x4747_4747
AIC_SCR3	AIC_BA+008	R/W	Source Control Register 3	0x4747_4747
AIC_SCR4	AIC_BA+00C	R/W	Source Control Register 4	0x4747_4747
AIC_SCR5	AIC_BA+010	R/W	Source Control Register 5	0x4747_4747
AIC_SCR6	AIC_BA+014	R/W	Source Control Register 6	0x4747_4747
AIC_SCR7	AIC_BA+018	R/W	Source Control Register 7	0x4747_4747
AIC_SCR8	AIC_BA+01C	R/W	Source Control Register 8	0x4747_4747

AIC_IRSR	AIC_BA+100	R	Interrupt Raw Status Register	0x0000_0000
AIC_IASR	AIC_BA+104	R	Interrupt Active Status Register	0x0000_0000
AIC_ISR	AIC_BA+108	R	Interrupt Status Register	0x0000_0000
AIC_IPER	AIC_BA+10C	R	Interrupt Priority Encoding Register	0x0000_0000
AIC_ISNR	AIC_BA+110	R	Interrupt Source Number Register	0x0000_0000
AIC_IMR	AIC_BA+114	R	Interrupt Mask Register	0x0000_0000
AIC_OISR	AIC_BA+118	R	Output Interrupt Status Register	0x0000_0000
Reserved	**Reserved**		**Reserved**	Undefined
AIC_MECR	AIC_BA+120	W	Mask Enable Command Register	Undefined
AIC_MDCR	AIC_BA+124	W	Mask Disable Command Register	Undefined
AIC_SSCR	AIC_BA+128	W	Source Set Command Register	Undefined
AIC_SCCR	AIC_BA+12C	W	Source Clear Command Register	Undefined
AIC_EOSCR	AIC_BA+130	W	End of Service Command Register	Undefined
AIC_TEST	AIC_BA+134	W/R	ICE/Debug mode Register	0x0000_0000

3.5. Function Description

3.5.1. Interrupt Channel, Priority and Source Type

An 8-level priority encoder controls the nIRQ line. Each interrupt source belongs to priority group between of 0 to 7. Group 0 has the highest priority and group 7 the lowest. When more than one unmasked interrupt channels are active at a time, the interrupt with the highest priority is serviced first. If all active interrupts have equal priority, the interrupt with the lowest interrupt source number is serviced first.
It means:

- Level 0 > Level 1 > Level 2 > Level 3 > Level 4 > Level 5 > Level 6 > Level 7. The interrupt level was determined **AIC_SCRXX[PRIORITY]**
- Channel 1 > Channel 2 > Channel 3 >…> Channel 30 > Channel 31 if all interrupts at the same level. Interrupt channel 1, channel 2… channel31 maps to AIC_SCR1, AIC_SCR2… AIC_SCR31 respectively.
- Level 0 is FIQ interrupt. Other levels interrupt are IRQ interrupt.
- Interrupt channel 0 was reserved.

SRCTYPE [7:6]: Interrupt Source Type
Whether an interrupt source is considered active or not by the AIC is subject to the settings of this field. Interrupt sources other than nIRQ0, nIRQ1, nIRQ2, nIRQ3, should be configured as level sensitive during normal operation unless in the testing situation.

SRCTYPE [7:6]		Interrupt Source Type
0	0	Low-level Sensitive
0	1	High-level Sensitive
1	0	Negative-edge Triggered
1	1	Positive-edge Triggered

AIC Source Control Registers (AIC_SCR1 ~ AIC_SCR31)

Register	Address	R/W	Description	Reset Value
AIC_SCR1	AIC_BA+0x004	R/W	Source Control Register 1	0x4747_4747
AIC_SCR2	AIC_BA+0x008	R/W	Source Control Register 2	0x4747_4747
• • •	• • •	• • •	• • •	• • •
AIC_SCR8	AIC_BA+0x01C	R/W	Source Control Register 31	0x4747_4747

31	30	29	28	27	26	25	24
SRCTYPE (Channel 4n+3)		**RESERVED**			**PRIORITY** (Channel 4n+3)		
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
SRCTYPE (Channel 4n+2)		**RESERVED**			**PRIORITY** (Channel 4n+2)		
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
SRCTYPE (Channel 4n+1)		**RESERVED**			**PRIORITY** (Channel 4n+1)		

7	6	5	4	3	2	1	0
SRCTYPE (Channel 4n)		**RESERVED**			**PRIORITY** (Channel 4n)		

There are 4 channels in one control register where n =0 to 7. The interrupt source table reference section 3.3 Interrupt source.
The current priority level is defined as the priority level of the interrupt with the highest priority at the time the register AIC_IPER is read. In the case when a higher priority unmasked interrupt occurs while an interrupt already exits, there are two possible outcomes depending on whether the AIC_IPER has been read.

- If the processor has already read the AIC_IPER and caused the NIRQ line to be de-asserted, then the NIRQ line is reasserted. When the processor has enabled nested interrupts and reads the AIC_IPER again, it reads the new, higher priority interrupt vector. At the same time, the current priority level is updated to the higher priority.

If the AIC_IPER has not been read after the NIRQ line has been asserted, then the processor will read the new higher priority interrupt vector in the AIC_IPER register and the current priority level is updated.
When the End of Service Command Register (AIC_EOSCR) is written, the current interrupt level is updated with the last stored interrupt level from the stack (if any). Therefore, at the end of a higher priority interrupt, the AIC returns to the previous state corresponding to the preceding lower priority interrupt which had been interrupted.

3.5.2. Fake Interrupt

When the AIC asserts the nIRQ line, the processor enters interrupt mode and the interrupt handler reads the AIC_IPER, it may happen that AIC de-asserts the nIRQ line after the processor has taken into account the nIRQ assertion and before the read of the AIC_IPER.

This behavior is called a fake interrupt.

The AIC is able to detect these fake interrupts and returns all zero when AIC_IPER is read. The same mechanism of fake interrupt occurs if the processor reads the AIC_IPER (application software or ICE) when there is no pending-interrupt. The current priority level is not updated in this situation. Hence, the AIC_EOSCR shouldn’t be written.

3.5.3. Interrupt Handling

When the NIRQ line is asserted, the interrupt handler must read the AIC_IPER as soon as possible. This can de-assert the NIRQ request to the processor and clears the interrupt if it is programmed to be edge triggered. This allows the AIC to assert the NIRQ line again when a higher priority unmasked interrupt occurs.
The AIC_EOSCR (End of Service Command Register) must be written at the end of the interrupt service routine. This permits pending interrupts to be serviced.

3.5.4. Interrupt Masking

The AIC provides a set of registers to mask individual interrupt channel. The **Mask Enable Command Register (AIC_MECR)** is used to enable interrupt. Write 1 to any bit of AIC_MECR will enable the corresponding interrupt channel. Oppositely, the **Mask Disable Command Register (AIC_MDCR)** is used to disable the interrupt. Write 1 to any bit of AIC_MDCR will disable the corresponding interrupt channel. Write 0 to a bit of AIC_MECR or AIC_MDCR has no effect. Therefore, the device driver can arbitrarily change these two registers without keeping their original values. If it's necessary, the device driver can read the **Interrupt Mask Register (AIC_IMR)** to know whether the interrupt channel is enabled or disabled. If the interrupt channel is enabled, its corresponding bit is read as 1, otherwise 0.

3.5.5. Interrupt Clearing and Setting

For the interrupt channels that are edge-triggered, the device driver must clear AIC status to de-assert the interrupt request. To clear AIC status, the device driver may write **Source Clear Command Register (AIC_SCCR)**. Write 1 to any bit of AIC_SCCR will clear the corresponding interrupt. As soon as the device's interrupt status was cleared, the AIC de-asserts the interrupt request.

The register **Source Set Command Register (AIC_SSCR)** is used to active an interrupt channel when it is programmed to edge-triggered. Write 1 to any bit of AIC_SSCR will set the corresponding interrupt. This feature is useful in auto-testing or software debugging.

3.5.6. ICE/Debug Mode

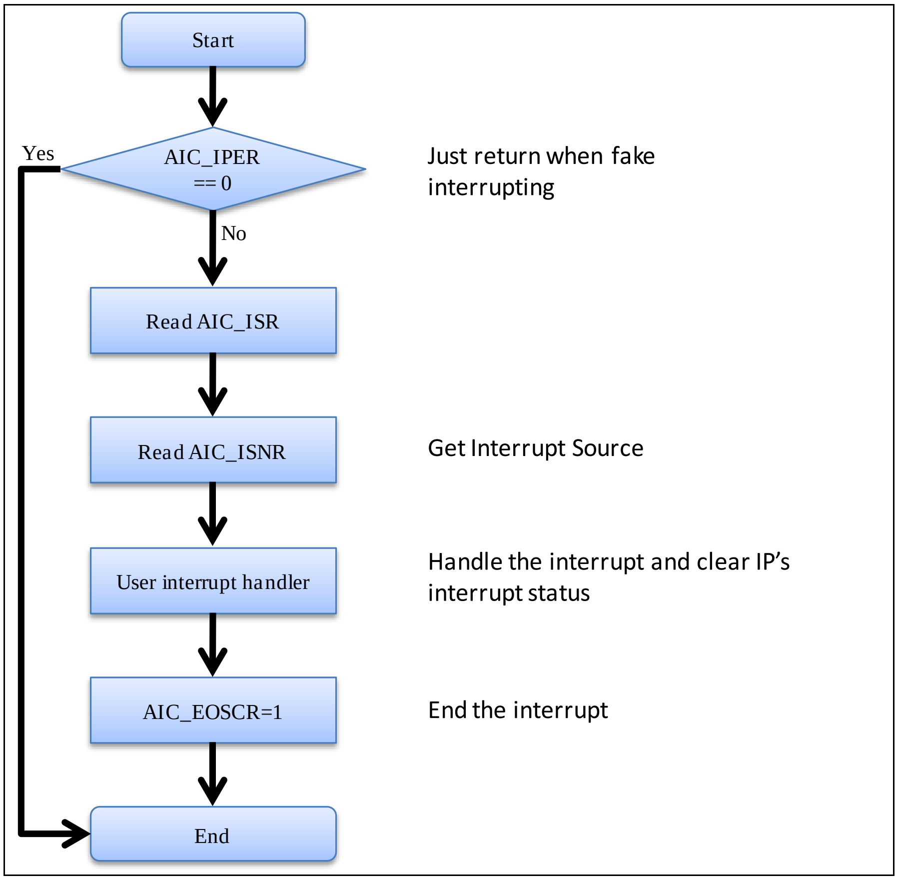
This mode allows reading of the AIC_IPER without performing the associated automatic operations. This is necessary when working with a debug system. When an ICE or debug monitor reads the AIC user interface, the AIC_IPER can be read. This has the following consequences in normal mode:
If there is no enabled pending interrupt, the fake vector will be returned.
If an enabled interrupt with a higher priority than the current one is pending, it will be stacked.
In the second case, an End-of-Service command would be necessary to restore the state of the AIC. This operation is generally not performed by the debug system. Therefore, the debug system would become strongly intrusive, and could cause the application to enter an undesired state.
This can be avoided by using ICE/Debug Mode. When this mode is enabled, the AIC performs interrupt stacking only when a write access is performed on the AIC_IPER. Hence, the interrupt service routine must write to the AIC_IPER (any value) just after reading it. When AIC_IPER is written, the new status of AIC, including the value of interrupt source number register (AIC_ISNR), is updated with the value that is kept at previous reading of AIC_IPER, the debug system must not write to the AIC_IPER as this would cause undesirable effects.
The following table shows the main steps of an interrupt and the order in which they are performed according to the mode:

Action	Normal Mode	ICE/Debug Mode
Calculate active interrupt	Read AIC_IPER	Read AIC_IPER
Determine and return the vector of the active interrupt	Read AIC_IPER	Read AIC_IPER

Push on internal stack the current priority level	Read AIC_IPER	Write AIC_IPER
Acknowledge the interrupt (Note 1)	Read AIC_IPER	Write AIC_IPER
No effect (Note 2)	Read AIC_IPER	

Notes:
NIRQ de-assertion and automatic interrupt clearing if the source is programmed as level sensitive.
Note that software which has been written and debugged using this mode will run correctly in normal mode without modification. However, in normal mode writing to AIC_IPER has no effect and can be removed to optimize the code.

3.5.7. FIQ/IRQ Handler Control Sequence



4. SPI Synchronous Serial Interface Controller

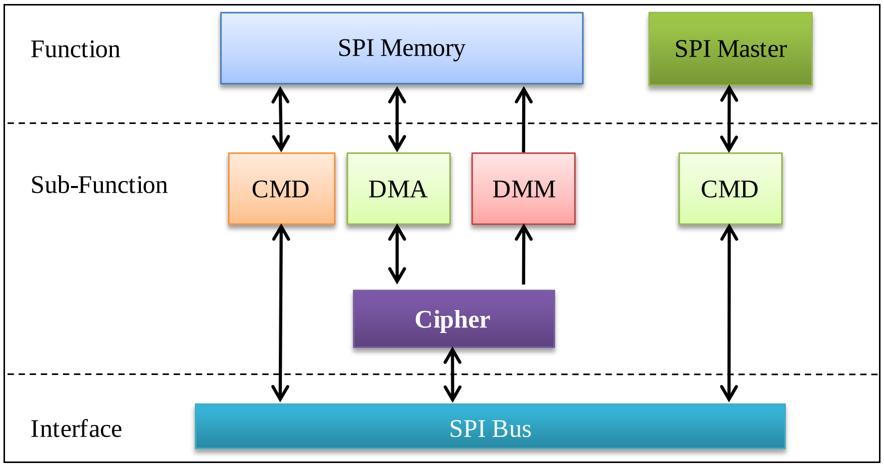
4.1. Overview

The SPI Synchronous Serial Interface performs a serial-to-parallel conversion on data characters received from the peripheral, and a parallel-to-serial conversion on data characters received from CPU. This interface can drive up to 2 external peripherals and is seen as the master. It can generate an interrupt signal when data transfer is finished and can be cleared by writing 1 to the interrupt flag. The active level of device/slave select signal can be chosen to low active or high active, which depends on the peripheral it's connected. Writing a divisor into DIVIDER register can program the frequency of serial clock output. This master core contains four 32-bit transmit/receive buffers, and can provide burst mode operation. The maximum bits can be transmitted/received is 32 bits, and can transmit/receive data up to four times successive.

There are two chip select pins exists in SPIM. These chip select pins are dedicated to SPI memory and SPI master function that are supported by SPIM engine. These functions are also named as SPIM0 and SPIM1. SPI memory and SPI master map to chip select pin 0 and chip select pin1 respectively. SPI memory has dedicated pin for chip select. SPI master must use one GPIO to emulate chip select. However, only one function can work in the same times.

SPI memory supports 3 sub functions. The first is command-read and command-write. The second is DMA-read and DMA-write. The third is direction memory mapping mode. The SPI master only supports command-read and command-write.

The chip also supports cipher function to encrypt and decrypt the ROM code through DMA and DMM sub-function. User can use it without any extra software effect. The detail for cipher should not be described in the document.



There are 4 main functions that supported in SPIM engine. They are

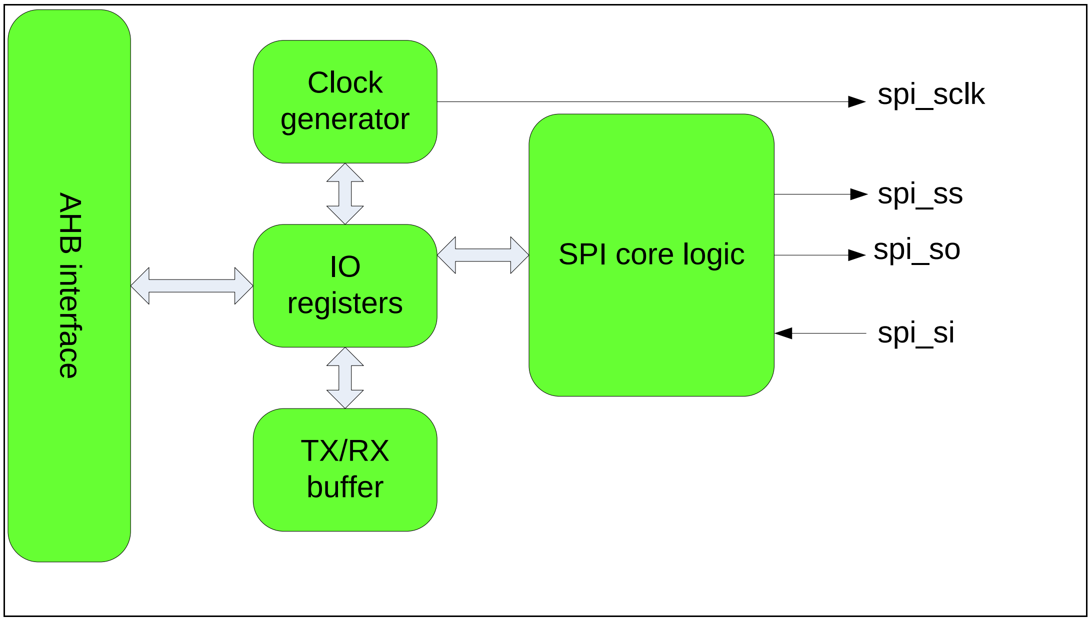
- Command mode – Programming/Reading SPI flash through command mode. However, the cipher, encryption and decryption, is not supported in the mode.
- DMA mode – Programming/Reading SPI flash through DMA mode. The cipher is supported in the mode.
- DMM mode – Reading SPI flash through DMM mode. CPU can fetch code in the mode. The cipher function is also supported in the mode.
- Cipher – The function performances the encryption or decryption based on the key1, key2, NUC501 IBR programming guide.

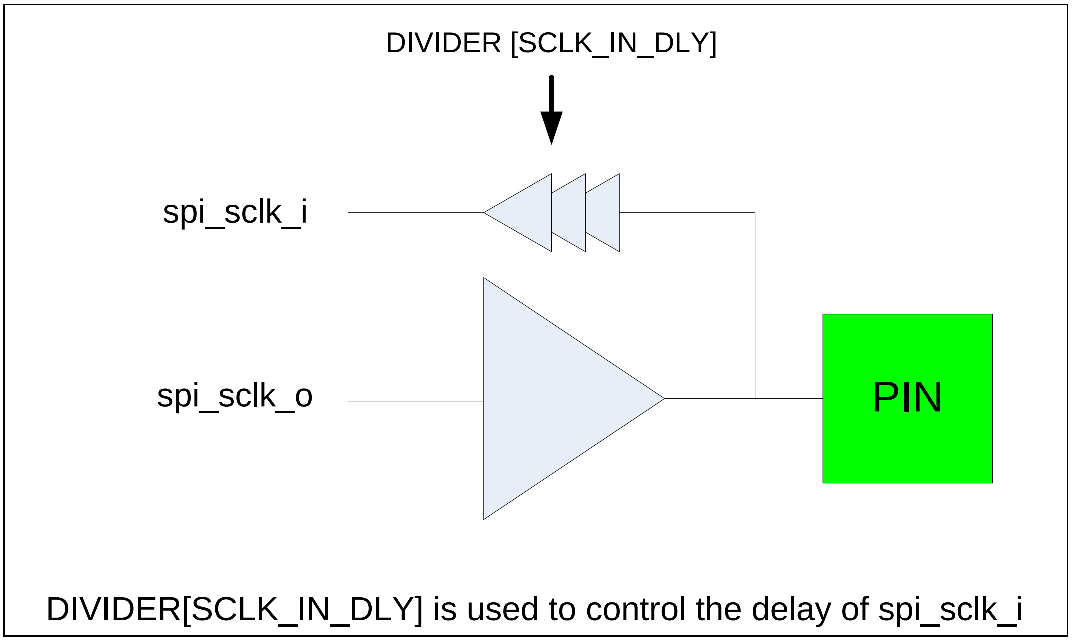
4.2. Block Diagram

The block diagram of SPI Serial Interface controller is shown as following:

Pin descriptions:

spi_sclk:	SPI serial clock output pin
spi_ss:	SPI slave/device select signal output
spi_so:	SPI serial data output pin (to slave device)
spi_si:	SPI serial data input pin (from slave device)





4.3. Registers

R: read only, **W**: write only, **R/W**: both read and write, **C**: Only value 0 can be written

Register	Address	R/W/C	Description	Reset Value
Base Address: 0xB100_7000				
CNTRL	SPI_BA + 0x00	R/W	Control and Status Register	0x0000_0004
DIVIDER	SPI_BA + 0x04	R/W	Clock Divider Register	0x0000_0000
SSR	SPI_BA + 0x08	R/W	Slave Select Register	0x0000_0000
Reserved	SPI_BA + 0x0C	N/A	Reserved	0xFFFF_FFFF
Rx0	SPI_BA + 0x10	R	Data Receive Register 0	0x0000_0000
Rx1	SPI_BA + 0x14	R	Data Receive Register 1	0x0000_0000
Rx2	SPI_BA + 0x18	R	Data Receive Register 2	0x0000_0000
Rx3	SPI_BA + 0x1C	R	Data Receive Register 3	0x0000_0000
Tx0	SPI_BA + 0x20	R/W	Data Transmit Register 0	0x0000_0000
Tx1	SPI_BA + 0x24	R/W	Data Transmit Register 1	0x0000_0000
Tx2	SPI_BA + 0x28	R/W	Data Transmit Register 2	0x0000_0000
Tx3	SPI_BA + 0x2C	R/W	Data Transmit Register 3	0x0000_0000
AHB_ADDR	SPI_BA + 0x30	R/W	AHB memory address	0x0000_0000
CODE_LEN	SPI_BA + 0x34	R/W	Boot code length	0x0000_0000
Reserved	SPI_BA + 0x38	N/A	Reserved	0xFFFF_FFFF

Reserved	SPI_BA + 0x3C	N/A	Reserved	0xFFFF_FFFF
SPI_ADDR	SPI_BA + 0x40	N/A	SPI Flash Start Address	0x0000_0000
OTP_CNTRL	SPI_BA + 0x44	N/A	OTP Control Register	0xFFFF_FFFF
OTP_PROG	SPI_BA + 0x48	N/A	OTP Program Register	0xFFFF_FFFF
Reserved	SPI_BA + 0x4C	N/A	Reserved	0xFFFF_FFFF
OTP_DISABLE	SPI_BA + 0x50	N/A	OTP Security Register	0xFFFF_FFFF

NOTE1: When software programs CNTRL, the GO_BUSY bit should be written last.

4.4. Function Description

4.4.1. Command mode

If users want to access a device with following specifications:

- Data bit latches on positive edge of serial clock
- Data bit drives on negative edge of serial clock
- Data is transferred with the MSB first
- Only one byte transmits/receives in a transfer
- Chip select signal is active low

However, the mode does not support cipher function. If you want to use cipher, please use DMA mode to access the SPI flash.

You should do following actions basically (you should refer to the specification of device for the detailed steps):

- Write a divisor into **DIVIDER** to determine the frequency of serial clock.
- Write in **SSR**, set **SSR[ASS]** = 0, **SSR[SS_LVL] = 0** and **SSR[0]** or **SSR[1]** to 1 to activate the device users want to access. To set or clear the **SSR[0]** and **SSR[1]** depends on the active level of chip select that you want to access.

When transmit (write) data to device:

- Write the data you want to transmit into **Tx0[7:0] / TX[31:0]**.
- Write the **CNTRL[Tx_NUM]** and **CNTRL**[**Tx_BIT_LEN]** for the transfer length.
- **CNTRL[Tx_NEG]** = 1 for negative edge to transmit data.
- Set **CNTRL[GO_BUSY]** = 1 to drive data and clock out.

When receive (read) data from device:

- Write **CNTRL**, Rx_NEG = 0, Tx_NEG = 1, Tx_BIT_LEN = 0x08, Tx_NUM = 0x0, LSB = 0, SLEEP = 0x0 (or 0x1 or 0x2 depend on the speed of SPI) and GO_BUSY = 1 to start the transfer. Waiting for interrupt (if IE = 1) or polling the GO_BUSY bit until it turns to 0.
- Read out the received data from Rx0.
- Go to point 3 to continue data transfer or set **SSR[0]** or **SSR[1]** to 0 to inactivate the device.

4.4.2. DMA mode

If you want to access SPI flash with cipher function. You can use DMA mode to access SPI flash.
DMA read mode:

- Set the target memory address in **AHB_ADDR** register.
- Set the boot code length which read from step 1 into CO**DE_LEN** register
- Set the SPI start address in **SPI_ADDR** register.
- Set **SSR** register to select spi slave. (no support ASS in dma mode)
- Set the READ command (0x03) and 3-Byte SPI Start Address into **Tx0**, **Tx1**, **Tx2**, **Tx3**. (It is same as **SPI_ADDR**)
- Set **SPI_CNTRL = 0x1a0345**.for control information.
- Wait code read finish. Wait INT.
- Set **SSR** register to un-select spi slave. (no support ASS in dma mode)

For other read mode:

- Fast read (0x0b), set read command (0x0b) into Tx0, & **CNTRL = 0x0b1a0b45**.
- Fast dual read (0x3b), set read command (0x3b) into Tx0, & **CNTRL = 0x3b1a0b45**.

DMA write mode: (Be sure the SPI flash is blank. To erase it if the SPI flash is not blank through command mode)

- Send Write Enable command to SPI flash
- Set the source memory address in A**HB_ADDR**
- Set the code length into **CODE_LEN** register
- Set the spi start address in **SPI_ADDR**
- Set SSR register to select spi slave. (no support ASS in dma mode)
- Set the Page Program command (0x02) and 3-Byte SPI Start Address into **Tx0**, **Tx1**, **Tx2**, **Tx3**. (It is same as **SPI_ADDR**)
- Set **CNTRL = 0x160345** for control information.
- Wait code write finish. Wait INT
- Set **SSR** register to un-select spi slave. (no support ASS in dma mode)
- Check the BUSY status in SPI Flash

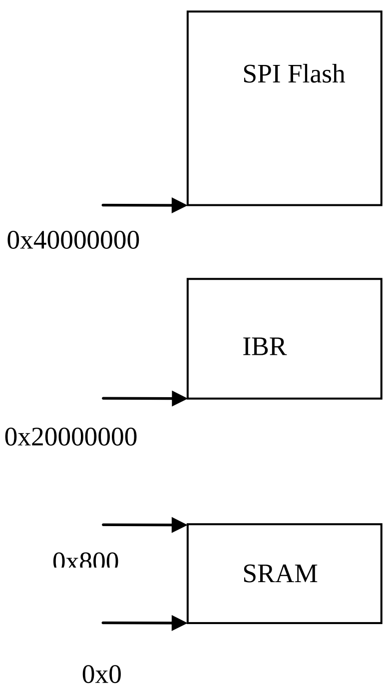
4.4.3. DMM mode

If you want to run code or read SPI flash without any extra effect. You can set the SPIM to DMM mode. The DMM mode will do serial to parallel conversion automatically. You can access it as ROM. However, the speed of access is more slowly than ROM because the hardware must transform the serial to parallel conversion.

- AHB master function (**CNTRL[DIS_M]** high), disable flash data read (**CNTRL[F_DRD]** low), set sleep interval to 1 (**CNTRL[SLEEP]** = 4'h1) and set SPI flash read command(**CNTRL[SPI_MODE]** 0x03
- Standard Read: Set **CNTRL** = 0x0332_1344 , Fast Read: Set **CNTRL** = 0x0b32_1344, Fast dual Read: Set **CNTRL** = 0x3b32_1344
- If the SPI clock speed up to 72MHz. Fine tuning the following register bits
    - Set the **CNTRL[SLEEP]** = 4'h2
    - **Divider[SCLK_IN_DLY]** = 0x07. **Divider[IDLE_CNT] = 0xF**.

4.4.4. Fetch code from SPI memory

As power on, internal boot ROM (IBR) is default map to 0. IBR will copy first 16K bytes ROM code that stored in SPI flash to RAM. Programmer must initial the SPIM to DMM mode after boot from IBR if CPU will fetch the code from SPI flash. The memory map after booted from IBR lists as following figure.



Init.s

```
        …                                       ;Init stack

        LDR r0, SPIM_PIN_REG      ;Set Pad function for SPIM0
        LDR r1, SPIM_PIN_CNT
        STR r1, [r0]
        LDR r0, SPIM_MODE_REG ;Set SPIM0 to DMM mode
        LDR r1, SPIM_MOD_CNT
        STR r1, [r0]
        LDR r0, SPIM_DIV_REG      ;Set Divider of SPIM
        LDR r1, SPIM_DIV_CNT
        STR r1, [r0]
        LDR r0, SPIM_ACS_REG      ;Set Auto chip select
        LDR r1, SPIM_ACS_CNT
        STR r1, [r0]
        LDR r15, ROMADDR          ;Long jump to flash address

SPIM_PIN_REG        DCD         0xb1000034
SPIM_PIN_CNT        DCD         0x00000140
SPIM_MODE_REG       DCD         0xb1007000
SPIM_MODE_CNT       DCD         0x0b322344
SPIM_DIV_REG        DCD         0xb1000034
SPIM_DIV_CNT        DCD         0x00000140
SPIM_ACS_REG        DCD         0xb1007000
SPIM_ACS_CNT        DCD         0x0b322344
ROMADDR             DCD         ROM_CODE_ADDR

ROM_CODE_ADDR
        B                   __main
```

And scatter loading descriptor file may architecture the program code as following scheme. The scatter loading descriptor file defines: one load region (FLASH) and four execution regions (FLASH, 32bitRAM, HEAP and STACK). The entire program is placed in FLASH which resides at 0x40000000. On power on, IBR maps to address 0 and it will copy the first 16K bytes of FLASH to RAM. Then IBR will remap the SRAM to 0 then execute a CPU reset. CPU will fetch the aliased copy code to initialize the SPIM to DMM mode. After the initial phase, the initialization code in the C library copies the RO and RW execution regions from their load address to their execution address before create any zero-initialized areas. Detail reference the document-ARM Developer Suite Developer Guide.

```
FLASH 0x40000000
{
        FLASH 0x40000000
        {
              init.o (Init, +First)
              * (+RO)
        }
        32bitRAM 0x0000
        {
                vectors.o (Vect, +First)
                * (+RW,+ZI)
        }
        HEAP +0 UNINIT
        {
                heap.o (+ZI)
        }
        STACK 0x8000 UNINIT
        {
                stack.o (+ZI)
        }
}
```

4.4.5. Application limitations

There are many limitations for SPIM0 and SPIM1 work together. These limitations are

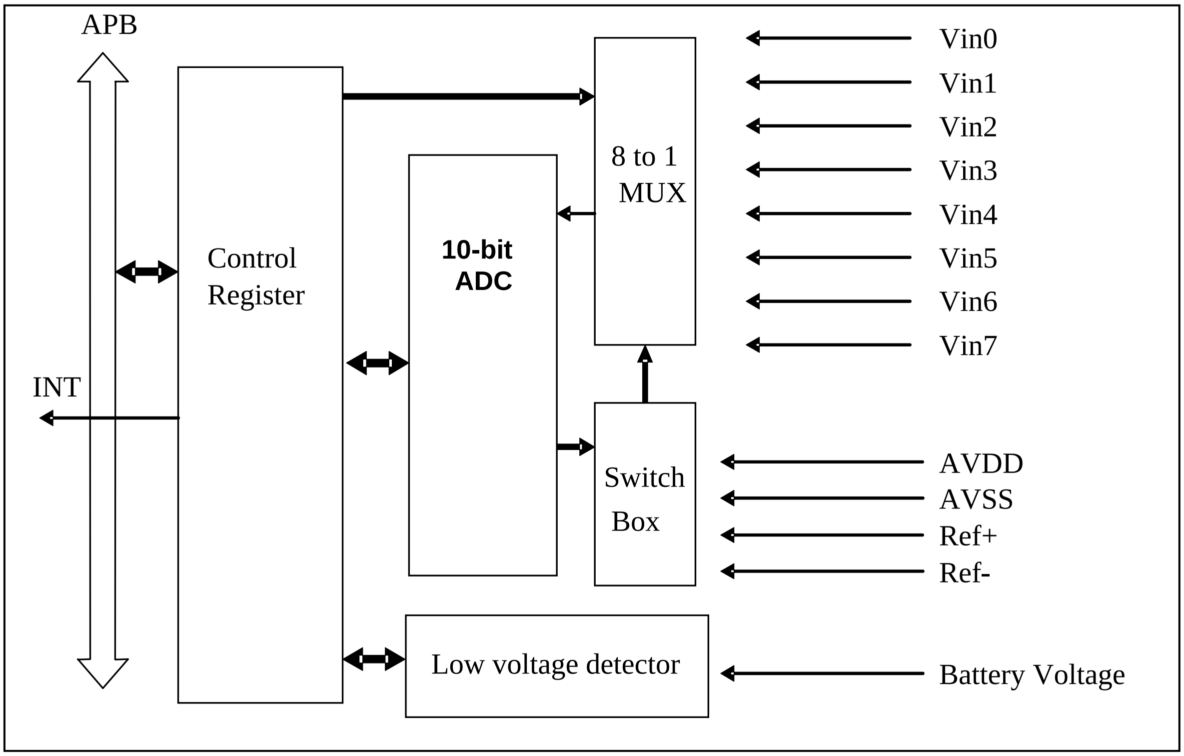
- SPIM0 and SPIM1 can not work in the same time due to the limitations of hardware. They share the same registers and hardware IP. So the SPI memory and SPI master must work time-sharing.
- If programmer wants to programming SPIM0 or SPIM1. Programmer must set the program code in the RAM area.
- The SPI master should be useless if fetch code from the SPI memory. It means programmer can not access device through SPI master unless run fetch code from RAM.

5. Analog to Digital Converter (ADC)

5.1. Overview

The ADC module is 10 bit analog to digital converter, it contains successive 8 channel analog input for conversion, the touch screen interface for 4/5/8-wire analog resistive touch screen, 4-level voltage detector. The ADC needs around 34 cycles to convert one sample, while the maximum clock of ADC is 17 MHz, so maximum conversion rate is 500K (if one cycle per one clock, then 25M/34 = 500K) sample/sec, reality the conversion rate about 300K to guarantee digital data correcting (experimental value).

5.2. Block Diagram



5.3. Registers

R: read only, **W**: write only, **R/W**: both read and write, **C**: Only value 0 can be written

Register	Address	R/W	Description	Reset Value
ADC_BA = 0xB800_1000				
ADC_CON	ADC_BA+0x000	R/W	ADC control register	0x0000_0000
ADC_DLY	ADC_BA+0x008	R/W	ADC delay register	0x0000_0000
LV_CON	ADC_BA+0x014	R/W	Low Voltage Detector Control register	0x0000_0000
LV_STS	ADC_BA+0x018	R/W	Low Voltage Detector Status register	0x0000_0000
AUDIO_CON	ADC_BA+0x01C	R/W	Audio control register	0x0000_0000
AUDIO_BUF0	ADC_BA+0x020	R/W	Audio data register 0	0x0000_0000
AUDIO_BUF1	ADC_BA+0x024	R/W	Audio data register 1	0x0000_0000
AUDIO_BUF2	ADC_BA+0x028	R/W	Audio data register 2	0x0000_0000
AUDIO_BUF3	ADC_BA+0x02C	R/W	Audio data register 3	0x0000_0000

5.4. Function Description

5.4.1. ADC normal mode operation

The normal conversion mode operates for general purpose ADC. The ADC registers control the 8 to 1 MUX to select an analog input channel. Specifically, both AIN0 and AIN1 are dedicated for audio recording and will be introduced in the next section.

Before converting the ADC data, the ADC engine clock must be given first. The ADC engine clock is given by

$$Freq_{ADC\_EngineClock} = \frac{Freq_{PLL}}{(ADC\_N + 1)} \qquad (5.4.1.1)$$

where $Freq_{PLL}$ is the PLL output frequency and ADC_N is the value of bits[24 16] of Clock Divider Register 1, **CLKDIV1**. As for the conversion rate, it is determined by

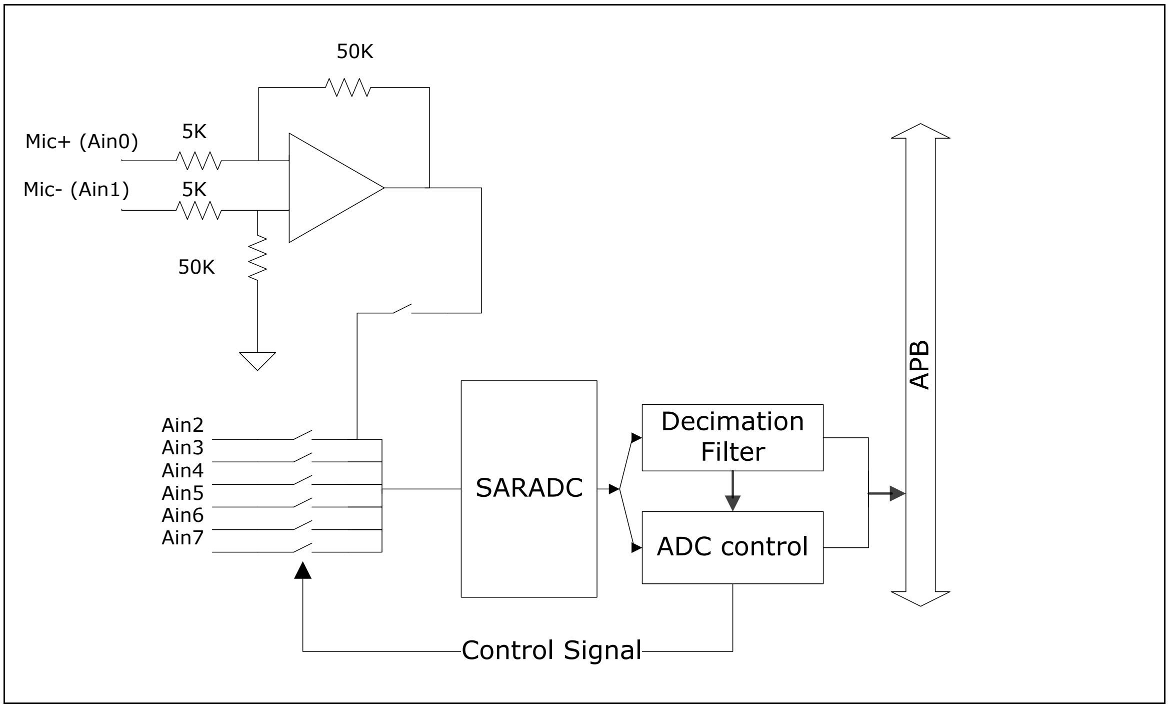
$$Freq_{ADC\_ConvertRate} = \frac{Freq_{ADC\_EngineClock}}{\{[round(ADC\_DIV/2)+1]*2\}*34} \qquad (5.4.1.2)$$

where *ADC_DIV* must be except 0 or 1. When *ADC_DIV* is equal to 0 or 1, $Freq_{ADC\_ConvertRate} = Freq_{ADC\_EngineClock} / 34$.

Additionally, the conversion rate must be determined first, we can get only one converted data when to enable ADC conversion. That is, the conversion is periodic. The procedure is described as follows,

1. Enable ADC engine clock
   - outp32(APBCLK, inp32(APBCLK) | ADC_CK_EN);
2. Reset ADC IP
   - outp32(IPRST, inp32(IPRST) | IPRST_ADC_RST); // Reset ADC IP
   - outp32(IPRST, inp32(IPRST) & ~IPRST_ADC_RST);
   - outp32(ADC_CON, ADC_RST); // ADC software reset
   - outp32(ADC_CON, (inp32(ADC_CON)&~ADC_RST));
3. Given ADC engine clock by Eq. (5.4.1.1)
4. Given conversion rate by Eq. (5.4.1.2)
5. Determine the converted channel
6. Enable to convert ADC
   - outp32(ADC_CON, inp32(ADC_CON) | ADC_CON_ADC_EN);
7. Check if conversion to be finished
   - while((inp32(ADC_CON)&ADC_INT)==0);
8. Get the converted data
9. Clear ADC_INT flag and repeat steps 6 – 9

5.4.2. Audio recording



The audio recording path can convert the analog data to digital one by means of the ADC hardware. When the ADC is switched to audio recording mode, other ADC data-conversion function can’t be operated. The audio sampling rate is determined by

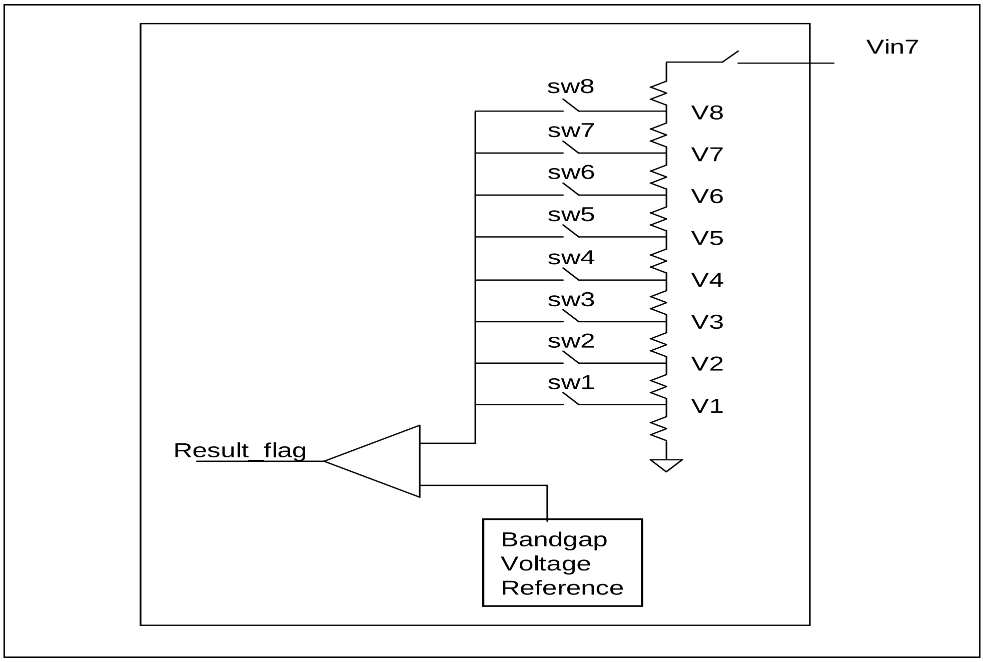
$$Freq_{SamplingRa} = \frac{Freq_{ADC\_EngineClk}}{1280} \quad (4.4.2.1)$$

The procedure for audio recording is described as follows.

1. Enable ADC engine clock
   - outp32(APBCLK, inp32(APBCLK) | ADC_CK_EN);
2. Reset ADC IP
   - outp32(IPRST, inp32(IPRST) | IPRST_ADC_RST); // Reset ADC IP
   - outp32(IPRST, inp32(IPRST) & ~IPRST_ADC_RST);
   - outp32(ADC_CON, ADC_RST); // ADC software reset
   - outp32(ADC_CON, (inp32(ADC_CON)&~ADC_RST));
3. Given ADC engine clock by Eq. (5.4.1.1)
4. Given sampling rate by Eq. (5.4.2.1)
5. Given AGC settings if to be enabled
   - Set period time, attack time, recovery time and hold time in **LV_STS** register
6. Given recording volume
   - Set bits[8:3] of **AUDIO_CON** register
7. Select recording mode
   - Mode_00 ➔ *AUD_INT* interrupt bit is set when one recorded sample is finished.
   - Mode_01 ➔ *AUD_INT* interrupt bit is set when two recorded samples are finished.
   - Mode_10 ➔ *AUD_INT* interrupt bit is set when four recorded samples are finished.
   - Mode_11 ➔ *AUD_INT* interrupt bit is set when eight recorded samples are finished.
8. Start audio recording
   - Set bit[1] of **AUDIO_CON** register
9. Wait recorded data has been finished
   - Wait *AUD_INT* interrupt bit to be set
10. Read the recorded data from **AUDIO_BUF_0/1/2/3** buffer registers
    - Mode_00 ➔ one sample from bits[15:0] of **AUDIO_BUF_0**
    - Mode_01 ➔ two samples from bits[31:16] and bits[15:0] of **AUDIO_BUF_0**, individually
    - Mode_10 ➔ four samples from **AUDIO_BUF_0** and **AUDIO_BUF_1**, individually
    - Mode_11 ➔ four samples from **AUDIO_BUF_0, AUDIO_BUF_1, AUDIO_BUF_2** and **AUDIO_BUF_3**, individually
11. Clear *AUD_INT* interrupt flag and repeat steps 9 -11

5.4.3. Low voltage detection

The architecture of the voltage detector is shown as in the following figure. By controlling the switch, sw1 ~ sw8, the ADC can do the voltage detection from V1 to V8. The voltage will not be influenced by the change of supply voltage or temperature.



The low voltage detection source is only from Vin7 pin. The procedure of detection is described as follows.

1. Enable ADC engine clock
   - outp32(APBCLK, inp32(APBCLK) | ADC_CK_EN);
2. Reset ADC IP
   - outp32(IPRST, inp32(IPRST) | IPRST_ADC_RST); // Reset ADC IP
   - outp32(IPRST, inp32(IPRST) & ~IPRST_ADC_RST);
   - outp32(ADC_CON, ADC_RST); // ADC software reset
   - outp32(ADC_CON, (inp32(ADC_CON)&~ADC_RST));
3. Enable LVR detection
   - Enable bit [3] of **LV_CON** and set bits[2:0] of **LV_CON** to select the switch settings.
   - Wait *LVD_INT* interrupt flag to be set.
   - When the input voltage is lower than the target voltage, *LVD_INT* interrupt flag, bit-19 of **ADC_CON** will be set.

6. Analog Processing Unit (APU)

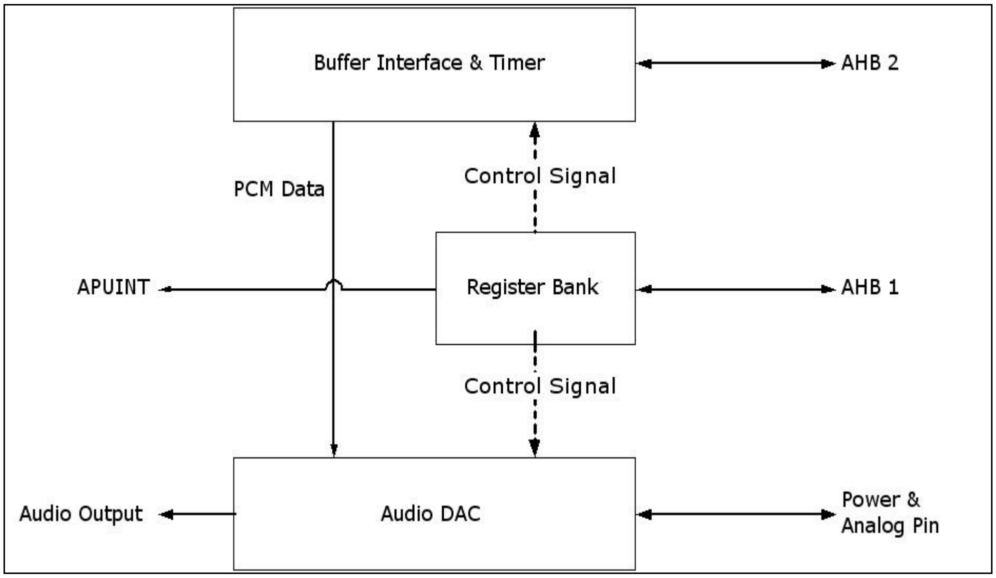
6.1. Overview

The main purpose of Audio Processing Unit (APU) is used to playback the audio data (PCM format) which CPU decoded and stored in global RAM. The APU built in a monophonic DAC with 16-bit resolution per channel which supports speakerphone output and monophonic output for headphone. The APU is composed of an AHB Master and built in FIFO and timer.

Features

- Monophonic Digital to Analog Converter with 16-bit resolution
- Supports speakerphone output and monophonic output for headphone
- Read Audio PCM data from global RAM
- Built in FIFO with length 16Bytes * 2
- Built in 10-band equalizer
- Built in timer to generate conversion trigger signal automatically

6.2. Block Diagram



6.3. Registers

R: read only, **W**: write only, **R/W**: both read and write, **C**: Only value 0 can be written

Register	Address	R/W	Description	Reset Value
APU_BA = 0xB100_8000				
APUCON	APU_BA + 0x00	R/W	APU Control Register	0x0000_0000
PARCON	APU_BA + 0x04	R/W	Parameter Control Register	0x0000_0001
PDCON	APU_BA + 0x08	R/W	Power Down Control Register	0x0001_0000
APUINT	APU_BA + 0x0C	R/W	APU Interrupt Register	0x0000_0000
RAMBSAD	APU_BA + 0x10	R/W	RAM Base Address Register	0x0000_0000
THAD1	APU_BA + 0x14	R/W	Threshold 1 Address Register	0x0000_0000
THAD2	APU_BA + 0x18	R/W	Threshold 2 Address Register	0x0000_0000
CURAD	APU_BA + 0x1C	R	Current Access RAM Address Register	0x0000_0000
EQGAIN0	APU_BA + 0x20	R/W	Equalizer Band Gain Register 0	0x7777_7777
EQGAIN1	APU_BA + 0x24	R/W	Equalizer Band Gain Register 1	0x000D_0077
APURAMBIST	APU_BA + 0x2C	R/W	APU ram BIST control register	0x0000_0000

6.4. Function Description

6.4.1. Sampling rate control

The APU sampling rate is determined by the APU engine clock. The APU engine clock is given by

$$Freq_{APU\_EngineClk} = \frac{Freq_{PLL}}{(APU\_N + 1)} \qquad (6.4.1.1)$$

where $Freq_{PLL}$ is the PLL output frequency and APU_N is the value of bits[15 8] of Clock Divider Register 0, **CLKDIV0**. As for the sampling rate, it is determined by

$$Freq_{APU\_SamplingRate} = \frac{Freq_{APU\_EngineClk}}{128} \qquad (6.4.1.2)$$

6.4.2. Threshold and DAC control

Threshold Control

The main purpose of APU is used to playback the audio data which CPU decoded and stored in global RAM. Before to enable APU, both RAM base and two RAM threshold address must be set first. The APU engine will get the RAM data which between the RAM base and higher threshold address and play to the 16-bit DAC repeatedly.

For example,

RAM base address = 0x1000
Threshold_0 address = 0x1400
Threshold_1 address = 0x1800

The APU engine will play the audio data between 0x1000and 0x1800 repeatedly. Additionally, while the APU internal counter encounters Threshold_0 or Threshold_1, the corresponding interrupt flag will be set. Therefore, the user can update the old audio data after the interrupt flag to be set.

DAC Control

The DAC control register, ANA_PD bit of PDCON register, determine if to power down the APU DAC output or not.

6.4.3. Equalizer control

Three register controls the equalizer function, including bit-24 of PARCON, EQGain0 and EQGain1. First, we must give the desired values to EQGain0 and EQGain1 registers, and then enable equalizer function by setting bit-24 of PARCON register.

6.4.4. APU example

The procedure for APU playback is described as follows.

1. Enable APU engine clock
   - outp32(AHBCLK, inp32(AHBCLK)|APU_CK_EN);
2. Reset APU engine
   - outp32(IPRST, inp32(IPRST)|APU_RST); // Reset APU
   - outp32(IPRST, inp32(IPRST)&~APU_RST); // APU normal operation
   - outp32(APUCON, inp32(APUCON) | APURST);
   - outp32(APUCON, inp32(APUCON) & ~APURST);
3. Given APU engine clock by Eq. (6.4.1.1)

4. Given sampling rate by Eq. (6.4.1.2)
5. Given the base address
   - outp32(RAMBSAD, BaseAddress);
6. 6. Given TH1 address
   - outp32(THAD1, TH1Address);
7. 7. Given TH2 address
   - outp32(THAD2, TH2Address);
8. Enable TH1 and TH2 interrupts if they are necessary
9. Set EQ parameters and enable EQ if they are necessary
10. Start APU audio playback
11. Wait TH1 and TH2 interrupt flags, *T1INTS* and *T2INTs*, and update the audio data in the buffer
12. Clear *T1INTS* and *T2INTs* interrupt flags
13. Repeat step 11 – 12

7. I2C Synchronous Serial Interface Controller

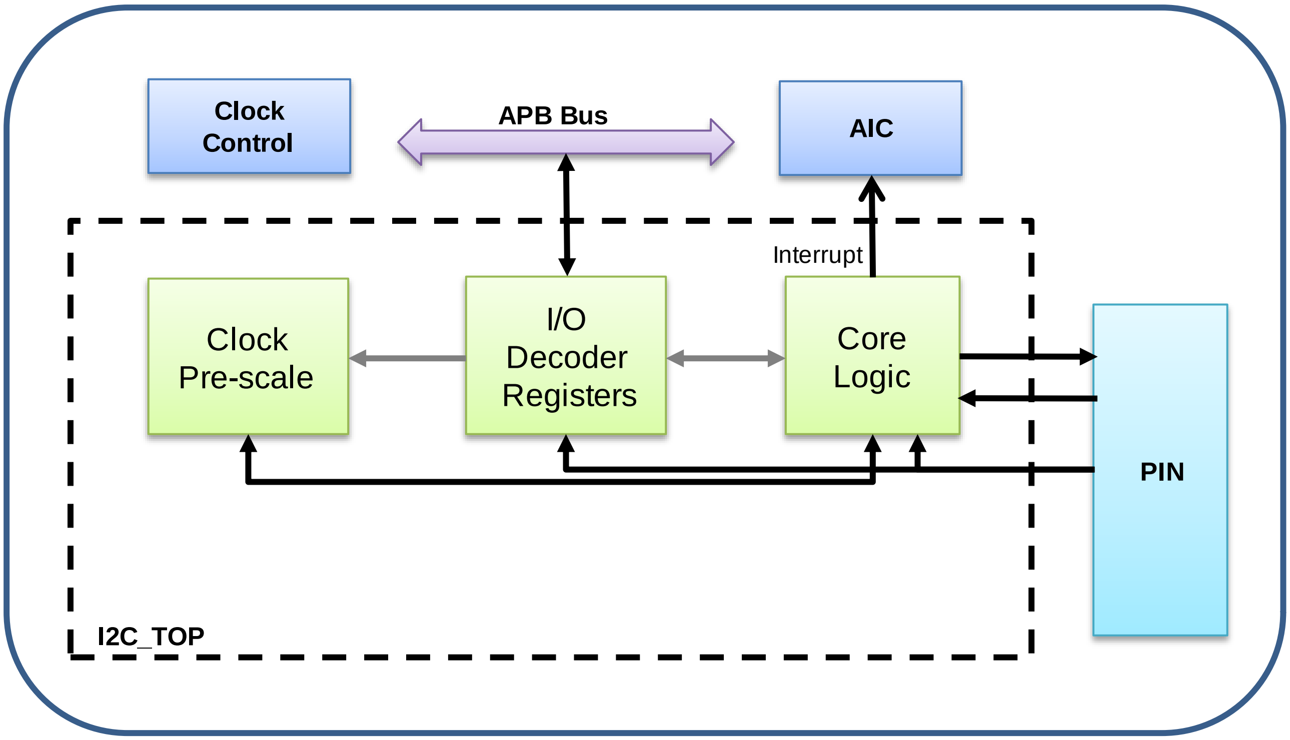
7.1. Overview

I2C is a two-wire, bi-directional serial bus that provides a simple and efficient method of data exchange between devices. The I2C standard is a true multi-master bus including collision detection and arbitration that prevents data corruption if two or more masters attempt to control the bus simultaneously.
Serial, 8-bit oriented bi-directional data transfers can be made up to 100 k-bit/s in Standard-mode, up to 400 k-bit/s in the Fast-mode, or up to 3.4 M-bit/s in the High-speed mode. Only 100kbps and 400kbps modes are supported directly. For High-speed mode, special IOs are needed. If these IOs are available and used, then High-speed mode is also supported.
Data is transferred between a Master and a Slave synchronously to SCL on the SDA line on a **byte-by-byte** basis. Each data byte is 8 bits long. There is one SCL clock pulse for each data bit with the **MSB being transmitted first**. An acknowledge bit follows each transferred byte. Each bit is sampled during the high period of SCL; therefore, the SDA line may be changed only during the low period of SCL and must be held stable during the high period of SCL. A transition on the SDA line while SCL is high is interpreted as a command (START or STOP).

The I2C Master core includes the following features:

- AMBA APB interface compatible
- Compatible with Philips I2C standard, support master mode
- Multi Master Operation
- Clock stretching and wait state generation
- Provide multi-byte transmit operation, up to 4 bytes can be transmitted in a single transfer
- Software programmable acknowledge bit
- Arbitration lost interrupt, with automatic transfer cancellation
- Start/Stop/Repeated Start/Acknowledge generation
- Start/Stop/Repeated Start detection
- Bus busy detection
- Supports 7 bit addressing mode
- Fully static synchronous design with one clock domain
- Software mode I2C

7.2. Block Diagram



7.3. Registers

R: read only, **W**: write only, **R/W**: both read and write
Base Address: 0xB800_4000

Register	Offset	R/W/C	Description	Reset Value
I2C_BA = 0xB800_4000				
CSR	I2C_BA+0x00	R/W	Control and Status Register	0x0000_0000
DIVIDER	I2C_BA+0x04	R/W	Clock Pre-scale Register	0x0000_0000
CMDR	I2C_BA+0x08	R/W	Command Register	0x0000_0000
SWR	I2C_BA+0x0C	R/W	Software Mode Control Register	0x0000_003F
RxR	I2C_BA+0x10	R	Data Receive Register	0x0000_0000
TxR	I2C_BA+0x14	R/W	Data Transmit Register	0x0000_0000

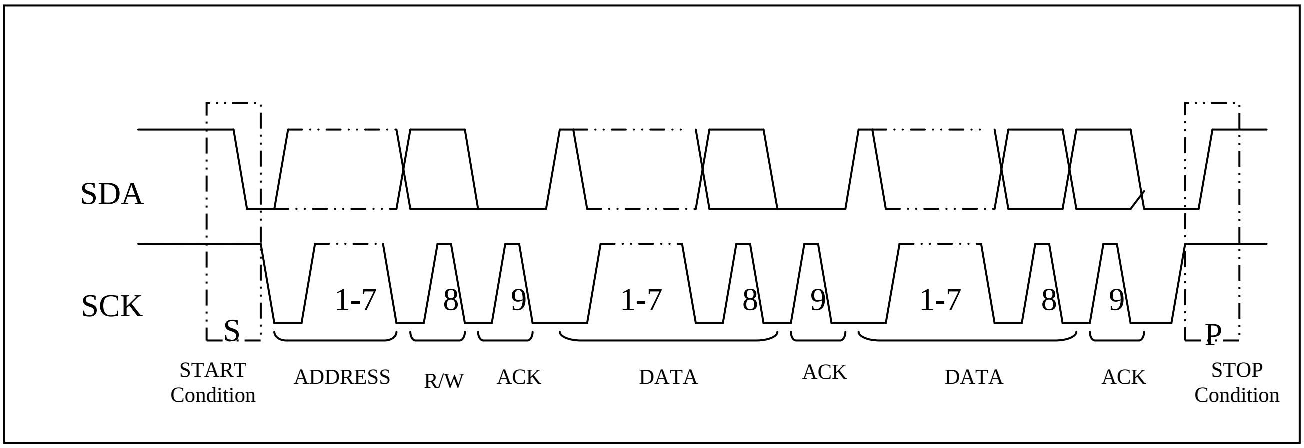
NOTE: The reset value of SWR is 0x3F only when SCR, SDR and SER are connected to pull high resistor.

7.4. Functional Descriptions

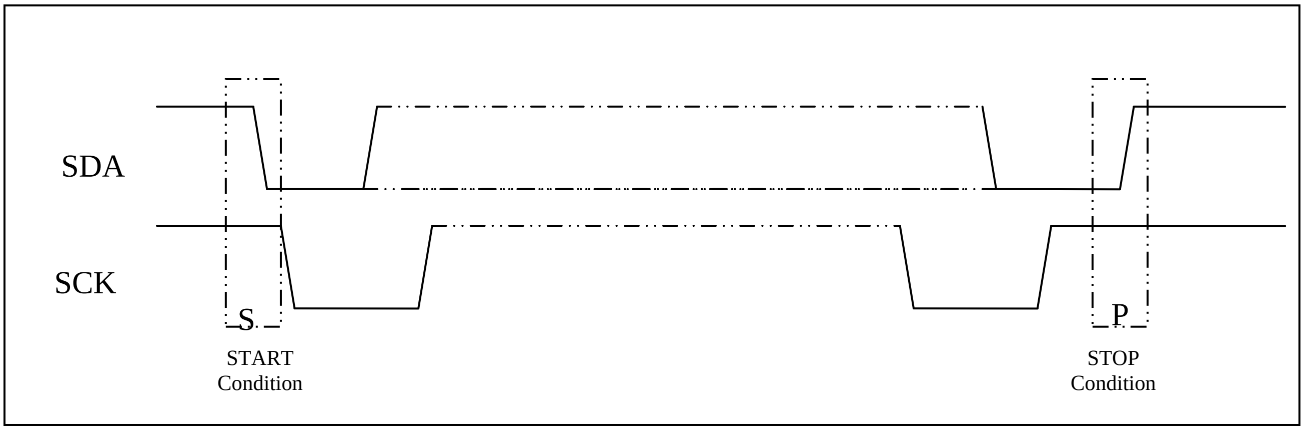
7.4.1. Limitation

- **Byte** basis transfer for hardware I2C.

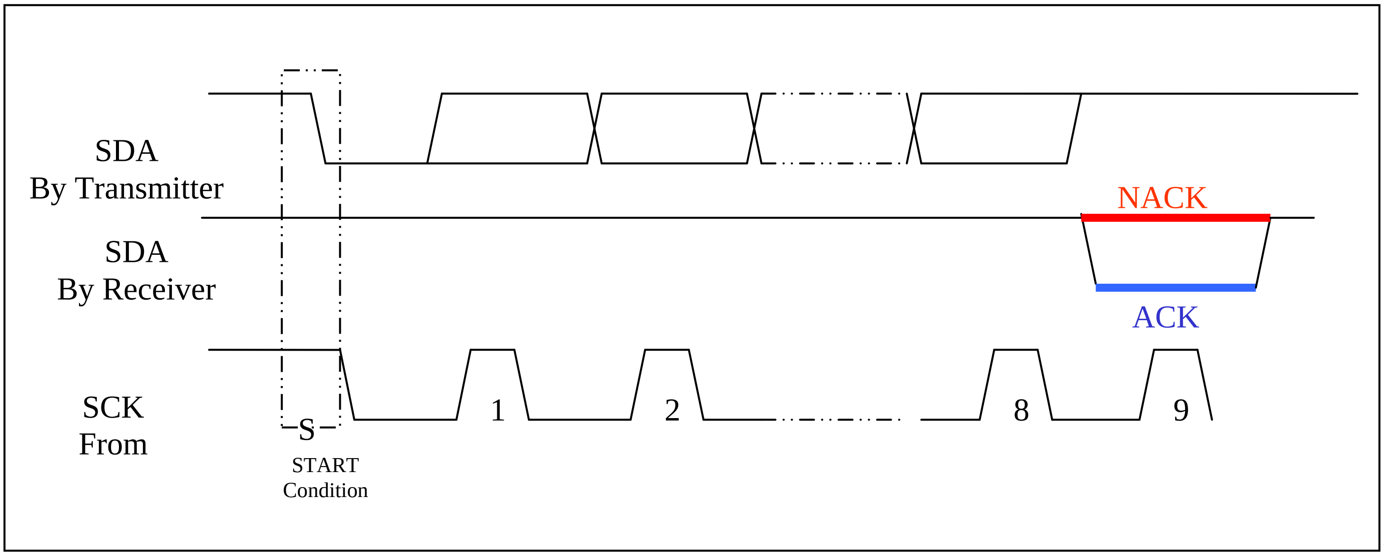
7.4.2. A complete data transfer



7.4.3. START and STOP condition

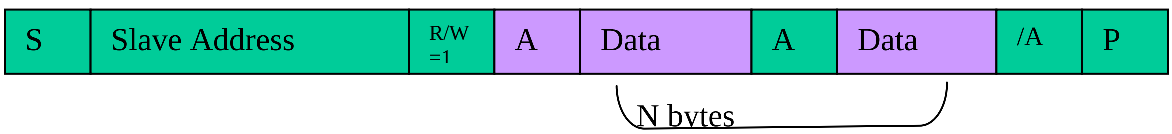


7.4.4. Acknowledge



7.4.5. Mater read and write

A master-transmitter addresses a slave receiver with 7-bit address.
The transfer direction is not changed



A master reads a slave immediately after the first byte.

From master to

From slave to

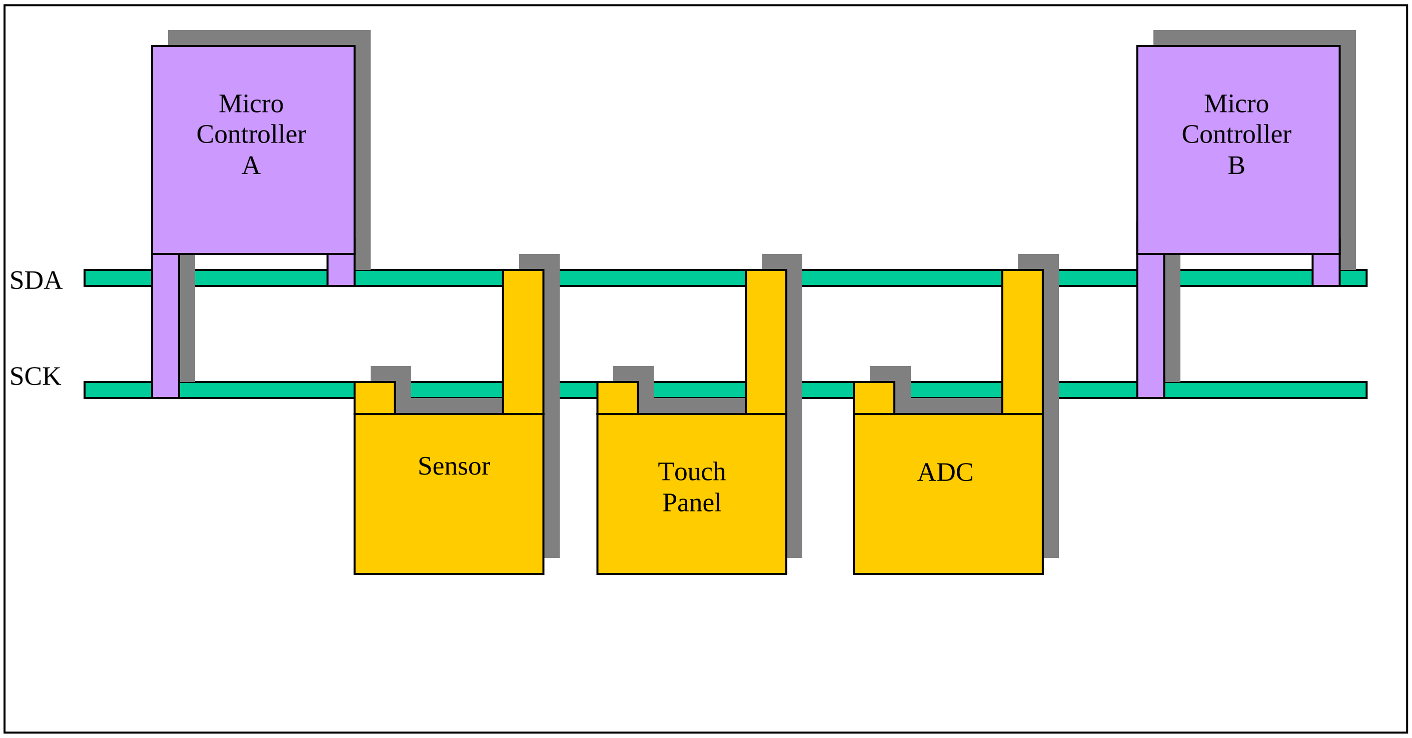
A = Acknowledge (SDA

/A = Not acknowledge (SDA

S = Start

P = Stop

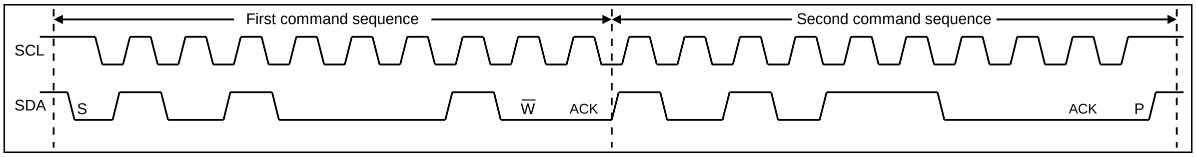
7.4.6. Example of an I²C-bus configuration using two micro-controllers



7.4.7. Hardware I²C

Control sequence.

14. Write a value into DIVIDER to determine the frequency of serial clock.
15. Set Tx_NUM = 0x1 and set I2C_EN = 1 to enable I²C core.
16. Write 0xA2 (address + write bit) to Transmit Register (TxR [15:8]) and 0xAC to TxR [7:0].
17. Set START bit and WRITE bit. — Wait for interrupt or I2C_TIP flag to negate —
18. Read I2C_RxACK bit from CSR Register, it should be ‘0’.
19. Set Tx_NUM = 0x0.
20. Set STOP bit.



Note 1: It has fixed serial clock specify in the between master and slave. The work frequency specify in register-**DIVIDER.** Source clock of Hardware I2C is **APB** clock. The formula of **DIVIDER** lists as following.

DIVIDER = APB / (5* Frequency of SCK) - 1;

Note 2: Transfer number specify in register **CSR [Tx_NUM].**

TX_NUM	Meaning
0x0	Only one byte is left for transmission.
0x1	Two bytes are left to for transmission.
0x2	Three bytes are left for transmission.
0x3	Four bytes are left for transmission.

Note 3: Transfer temporary buffer. There are 4 bytes temporary buffer for Hardware I2 transfer data.

Case 1: Only data A was transferred

TX_NUM=0

TXR[3]	D
TXR[2]	C
TXR[1]	B
TXR[0]	A

Case 2: Data B was transferred first then data A.

TX_NUM=0

TXR[3]	D
TXR[2]	C
TXR[1]	B
TXR[0]	A

Case 3: Transfer Data C first then data B. Data A was transferred last.

TX_NUM=0

TXR[3]	D
TXR[2]	C
TXR[1]	B
TXR[0]	A

Case 4: Transferred sequence is Data D, C, B then A.

TX_NUM=0

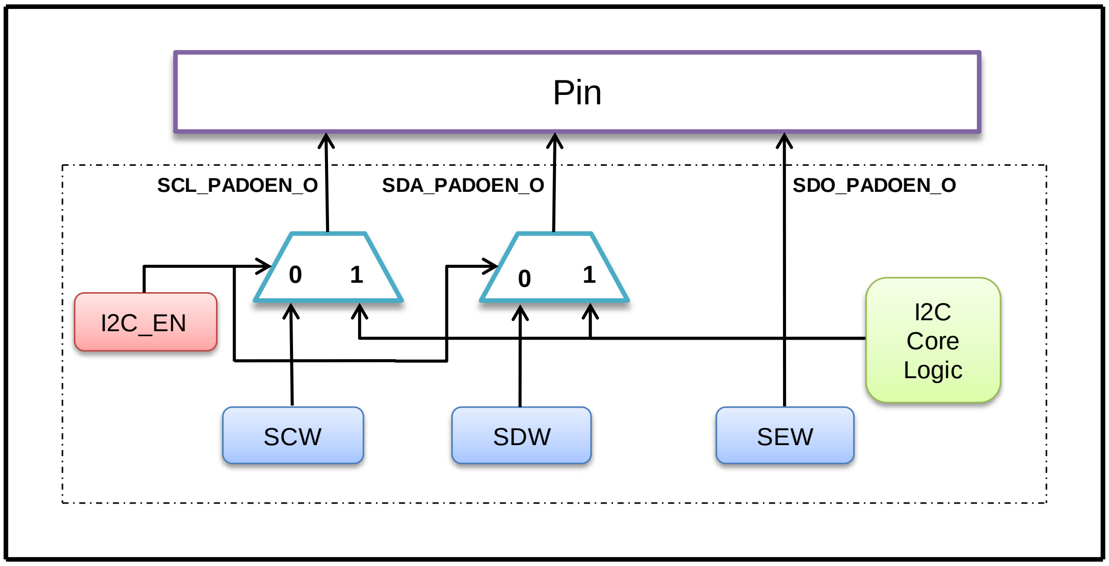
TXR[3]	D
TXR[2]	C
TXR[1]	B
TXR[0]	A

Note 4: Command.

Programmer can set register CMDR to generate the STAR, ACK, WRITE or READ and STOP phase. Please reference register CMDR.

7.4.8. Software I^{2}C

The software I^{2}C function contains 3 registers for software to control the output enable of pad actually. The implementation of software I^{2}C is shown as bellow. Software I2C works as I2C_EN bit set to 0. You can toggle these bits to emulation the I2C protocol.

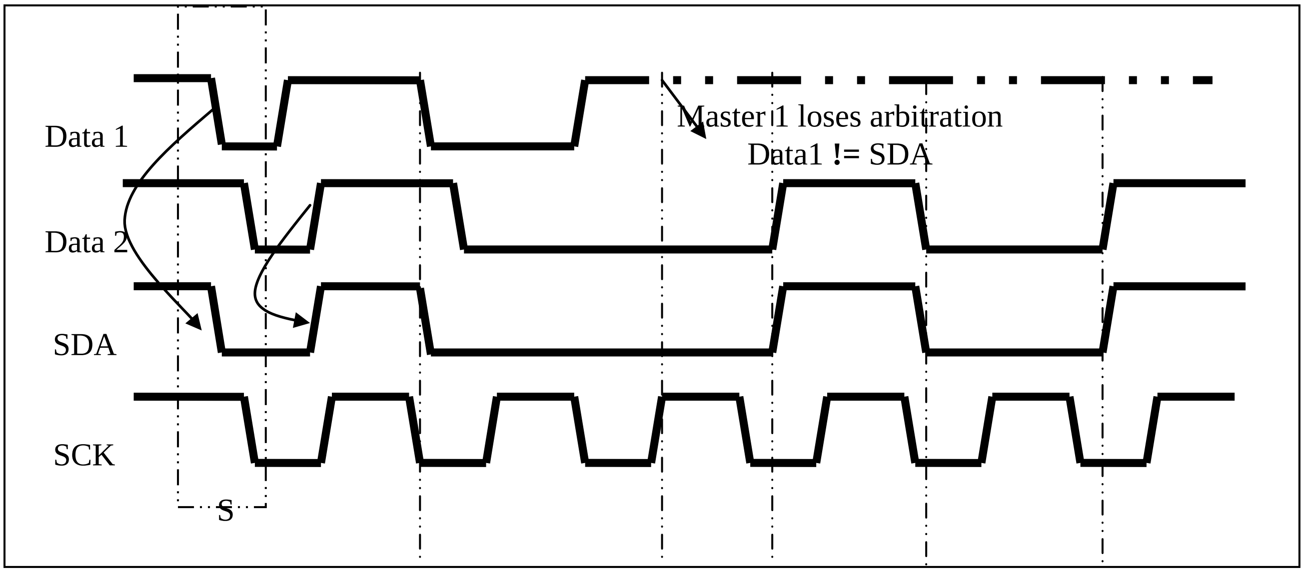


SCW	Serial clock
SDW	Serial data
SEW	Serial enable output

SCR	Serial clock pin status
SDR	Serial data pin status
SER	Serial enable output pin status

7.4.9. Arbitration

A master may start a transfer only if the bus is free. Two or more masters may generate a START condition within the minimum hold time.
CSR [I2C_AL]: Indicate the arbitration lose if the bit is equal to 1.



7.5. Relative registers definition

Control and Status Register (CSR)

Register	Offset	R/W/C	Description	Reset Value
CSR	0x00	R/W	Control and Status Register	0x0000_0000

31	30	29	28	27	26	25	24
Reserved							
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Reserved							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Reserved				**I2C_RxACK**	**I2C_BUSY**	**I2C_AL**	**I2C_TIP**
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Reserved		**Tx_NUM**		**Reserved**	**IF**	**IE**	**I2C_EN**

Bits	Descriptions	
[31:12]	**Reserved**	**Reserved**
[11]	**I2C_RxACK**	**Received Acknowledge From Slave (Read only)** This flag represents acknowledge from the addressed slave. • 0 = Acknowledge received (ACK).

		• 1 = Not acknowledge received (NACK).
[10]	**I2C_BUSY**	**I^{2}C Bus Busy (Read only)** • 0 = After STOP signal detected. • 1 = After START signal detected.
[9]	**I2C_AL**	**Arbitration Lost (Read only)** This bit is set when the I^{2}C core lost arbitration. Arbitration is lost when: • A STOP signal is detected, but no requested. • The master drives SDA high, but SDA is low.
[8]	**I2C_TIP**	**Transfer In Progress (Read only)** • 0 = Transfer complete. • 1 = Transferring data. **NOTE:** When a transfer is in progress, you will not allow writing to any register of the I^{2}C master core except SWR.
[7:6]	**Reserved**	**Reserved**
[5:4]	**Tx_NUM**	**Transmit Byte Counts** These two bits represent how many bytes are remained to transmit. When a byte has been transmitted, the Tx_NUM will decrease 1 until all bytes are transmitted (Tx_NUM = 0x0) or NACK received from slave. Then the interrupt signal will assert if IE was set. 0x0 = Only one byte is left for transmission. 0x1 = Two bytes are left to for transmission. 0x2 = Three bytes are left for transmission. 0x3 = Four bytes are left for transmission. **NOTE:** When NACK received, Tx_NUM will not decrease.
[3]	**Reserved**	**Reserved**

[2]	**IF**	**Interrupt Flag** The Interrupt Flag is set when: • Transfer has been completed. • Transfer has not been completed, but slave responded NACK (in multi-byte transmit mode). • Arbitration is lost. **NOTE:** This bit is read only, but can be cleared by writing 1 to this bit.
[1]	**IE**	**Interrupt Enable** • 0 = **Disable** I^2C Interrupt. • 1 = **Enable** I^2C Interrupt.
[0]	**I2C_EN**	**I^2C Core Enable** • 0 = **Disable** I^2C core, serial bus outputs are controlled by SDW/SCW. • 1 = **Enable** I^2C core, serial bus outputs are controlled by I^2C core.

Command Register (CMDR)

Register	Offset	R/W/C	Description	Reset Value
CMDR	0x08	R/W	Command Register	0x0000_000x

31	30	29	28	27	26	25	24
Reserved							
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Reserved							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Reserved							

7	6	5	4	3	2	1	0
Reserved	Reserved	Reserved	START	STOP	READ	WRITE	ACK

NOTE: Software can write this register only when I2C_EN = 1.

Bits	Descriptions	
[31:5]	**Reserved**	**Reserved**
[4]	**START**	**Generate Start Condition** Generate (repeated) start condition on I^2C bus.
[3]	**STOP**	**Generate Stop Condition** Generate stop condition on I^2C bus.
[2]	**READ**	**Read Data From Slave** Retrieve data from slave.
[1]	**WRITE**	**Write Data To Slave** Transmit data to slave.
[0]	**ACK**	**Send Acknowledge To Slave** When I^2C behaves as a receiver, sent ACK (ACK = ‘0’) or NACK (ACK = ‘1’) to slave.

NOTE: The START, STOP, READ and WRITE bits are cleared automatically while transfer finished. READ and WRITE cannot be set concurrently.

8. General Purpose I/O (GPIO)

8.1. Overview

26 pins for 48-pins package and 37 pins for 64-pins package and COB of General Purpose I/O are shared with special feature functions.
Supported Features of these I/O are: input or output facilities, pull-up resistors.
All these general purpose I/O functions are achieved by software programming setting. And the following figures illustrate the control mechanism to achieve the GPIO functions.

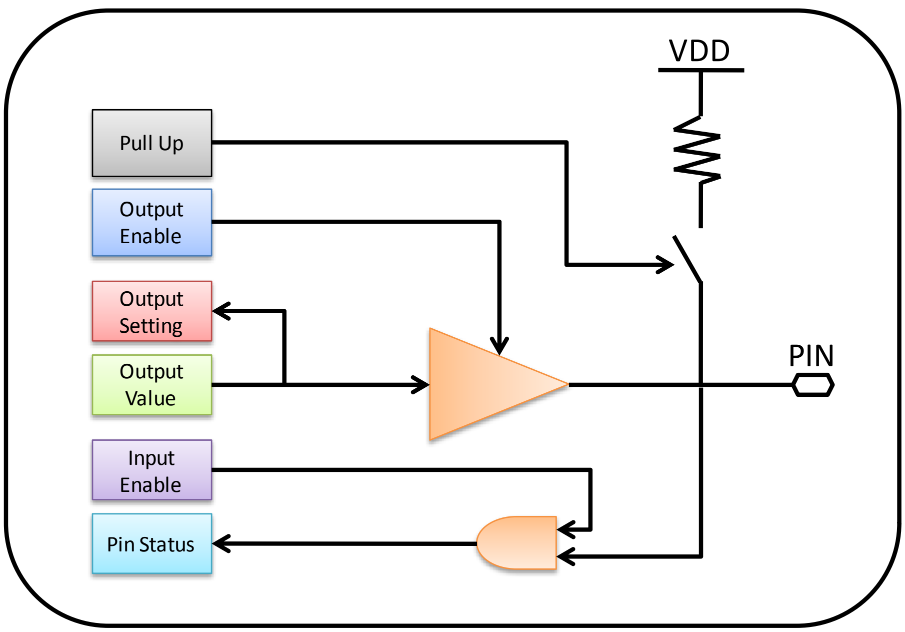


Figure 8-1 Type I GPIO: Input/Output Port with Program Controlled Weakly Pull-High

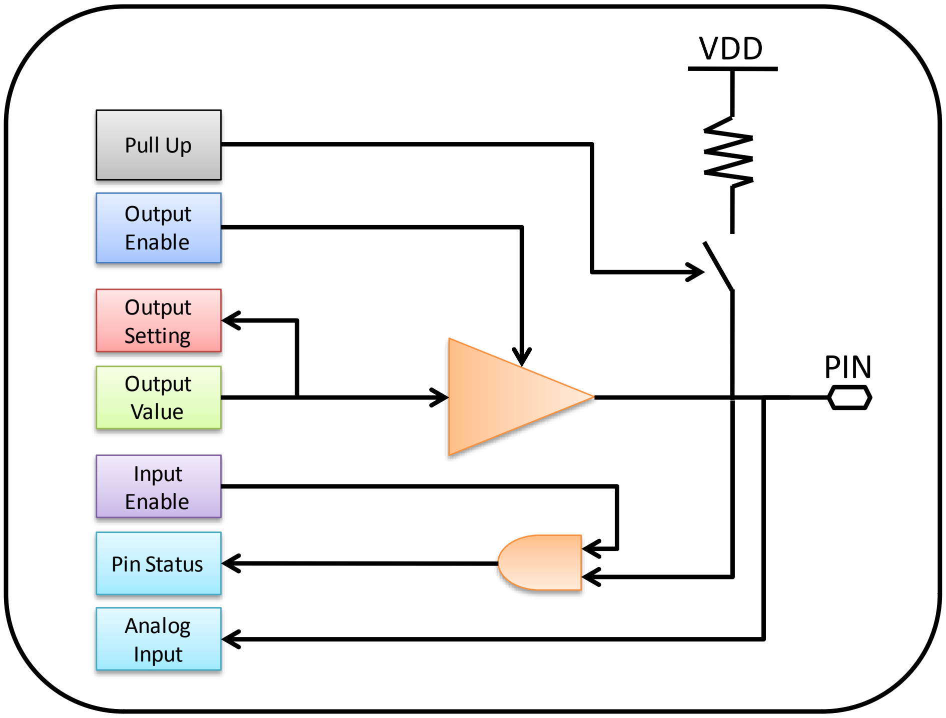


Figure 8-2 Type II GPIO: Input/Output Port with Schmitt-Trigger Input

8.2. Block Diagram

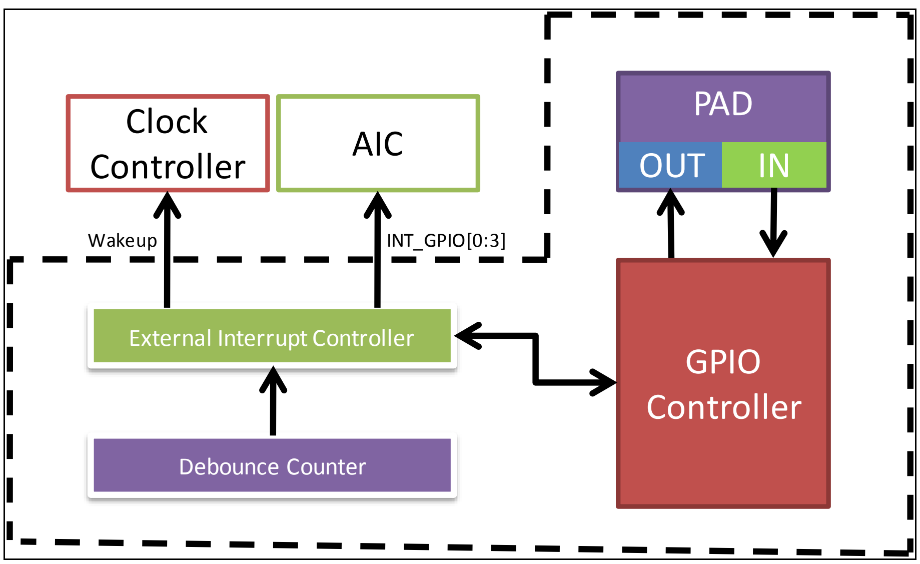


Figure 8-3 GPIO Block Diagram

8.3. Registers

R : Read only, **W** : Write only, **R/W** : Both read and write, **C** : Only value 0 can be written

Register	Address	R/W	Description	Reset Value
GP_BA = 0xB800_3000				
GPIOA_OMD	GP_BA+0x00	R/W	GPIO Port A Bit Output Mode Enable	0x0000_0000
GPIOA_PUEN	GP_BA+0x04	R/W	GPIO Port A Bit Pull-up Resistor Enable	0x0000_0000
GPIOA_DOUT	GP_BA+0x08	R/W	GPIO Port A Data Output Value	0x0000_0000
GPIOA_PIN	GP_BA+0x0C	R	GPIO Port A Pin Value	0xXXXX_XXXX
GPIOB_OMD	GP_BA+0x10	R/W	GPIO Port B Bit Output Mode Enable	0x0000_0000
GPIOB_PUEN	GP_BA+0x14	R/W	GPIO Port B Bit Pull-up Resistor Enable	0x0000_0000
GPIOB_DOUT	GP_BA+0x18	R/W	GPIO Port B Data Output Value	0x0000_0000
GPIOB_PIN	GP_BA+0x1C	R	GPIO Port B Pin Value	0xXXXX_XXXX
GPIOC_OMD	GP_BA+0x20	R/W	GPIO Port C Bit Output Mode Enable	0x0000_0000
GPIOC_PUEN	GP_BA+0x24	R/W	GPIO Port C Bit Pull-up Resistor Enable	0x0000_0000
GPIOC_DOUT	GP_BA+0x28	R/W	GPIO Port C Data Output Value	0x0000_0000
GPIOC_PIN	GP_BA+0x2C	R	GPIO Port C Pin Value	0xXXXX_XXXX
DBNCECON	GP_BA+0x70	R/W	External Interrupt De-bounce Control	0x0000_0000
IRQSRCGPA	GP_BA+0x80	R/W	GPIO Port A IRQ Source Grouping	0x0000_0000
IRQSRCGPB	GP_BA+0x84	R/W	GPIO Port B IRQ Source Grouping	0x5555_5555
IRQSRCGPC	GP_BA+0x88	R/W	GPIO Port C IRQ Source Grouping	0xAAAA_AAAA
IRQENGPA	GP_BA+0x90	R/W	GPIO Port A Interrupt Enable	0x0000_0000
IRQENGPB	GP_BA+0x94	R/W	GPIO Port B Interrupt Enable	0x0000_0000
IRQENGPC	GP_BA+0x98	R/W	GPIO Port C Interrupt Enable	0x0000_0000
IRQLHSEL	GP_BA+0xA0	R/W	Interrupt Latch Trigger Selection Register	0x0000_0000
IRQLHGPA	GP_BA+0xA4	R	GPIO Port A Interrupt Latch Value	0x0000_0000
IRQLHGPB	GP_BA+0xA8	R	GPIO Port B Interrupt Latch Value	0x0000_0000
IRQLHGPC	GP_BA+0xAC	R	GPIO Port C Interrupt Latch Value	0x0000_0000
IRQTGSRC0	GP_BA+0xB4	R/C	IRQ0~3 Interrupt Trigger Source Indicator from GPIO Port A and GPIO Port B	0x0000_0000
IRQTGSRC1	GP_BA+0xB8	R/C	IRQ0~3 Interrupt Trigger Source Indicator from GPIO Port C	0x0000_0000

8.4. Functional Description

8.4.1. Pin description

Pin name	Type	Description
GPA[0] ~ GPA[15]	Input/Output	GPIO Port A (16bit)
GPB[0] ~ GPB[9]	Input/Output	GPIO Port B (10bit)
GPC[0] ~ GPC[10]	Input/Output	GPIO Port C (11bit) (Only valid with 64 pin)

8.4.2. PAD Function Setting

The GPIO input/output and multiple functions are configured by setting PAD Control Register (PAD_REG0, PAD_REG1, and PAD_REG2). Programmers should not change the value of whole register except the corresponding field of the register. A sample code configures GPIOB[4:1] as UART0 I/O is given below.

```
int value;
// Read PAD_REG1 register value
value = inp32(PAD_REG1);
// Select UART0 as multi-function
value |= 0x100;
// Save the setting to PAD_REG1 register
outp32(PAD_REG1, value);
```

8.4.3. GPIO Output Mode

Before the system uses the GPIO pin as output pin, programmers need to configure GPIO Port x Bit Output Enable Control Register. These registers decide the direction of GPIOs. Set the GPIOx_OMD[n] value as 1 (output mode).
After the above steps, user can change the GPIO pin output value (high or low) by writing 1 or 0 to GPIO Port x Data Output Value register (GPIOx_DOUT). Programmers should not change the value of whole register except the corresponding field of the register.

A sample code sets GPIOB[0] as GPIO output, then change the output between high and low is given below.

```
// Set GPIOB[0] as output mode by GPIOB_OMD register
outp32(GPIOB_OMD, (inp32(GPIOB_OMD) & ~0x0001) | 0x1);
// Set GPIOB[0] output 1 by GPIOB_DOUT register
outp32(GPIOB_DOUT, inp32(GPIOB_DOUT) | 0x0001);
// Set GPIOB[0] output 0 by GPIOB_DOUT register
outp32(GPIOB_DOUT, inp32(GPIOB_DOUT) & ~0x0001);
```

8.4.4. GPIO Input Mode

Before the system uses the GPIO pin as input pin, programmers need to configure GPIO Port x Bit Output Enable Control Register. These registers decide the direction of GPIOs. Set the GPIOx_OMD [n] value as 0 (input mode).
After the above steps, user can get the GPIO pin status (high or low) by reading GPIO Port x Pin Value Register (GPIOx_PIN). By the way, user can enable/disable the GPIO pin as pull up by configure GPIO Port x Bit Pull-up Register Enable

A sample code that sets GPIOB[0] as GPIO input, then reads its status is given below.

```
UINT32 status;
// Set GPIOB[0] as input mode by GPIOB_OMD register
outp32(GPIOB_OMD, (inp32(GPIOB_OMD) & ~0x0001)
// Read status from GPIOB_PIN register
status = inp32(GPIOB_PIN);
if(status & 0x1)
    printf("GPIOB[0] input value is High.");
else
    printf("GPIOB[0] input value is Low.");
```

8.4.5. GPIO Interrupt

Only the first set of GPIO supports interrupt mechanism. The usage of GPIO interrupt is described as following steps.

1. Set the GPIOx_OMD [n] value as 0 (input mode).
2. Set the IRQSRCGPx to select the interrupt source group.
3. Set IRQENGPx to enable input falling/rising edge to trigger one of the interrupt sources.
4. Set IRQLHSEL to active IRQx interrupt to latch the input value of GPIOA/GPIOB/GPIOC.

Besides the above steps, programmers also need to handle AIC for system interrupt entry.

A sample code to install a call back function GpioIsr in GPIO interrupt is as follows.

```
// Set GPIOA[0] as input mode by GPIOA_OMD register
outp32(GPIOA_OMD, inp32(GPIOA_OMD) & ~0x0001);
// Set GPIOA[0] pin as one of interrupt sources to IRQ1 by IRQSRCGPA register
outp32(IRQSRCGPA, inp32(IRQSRCGPA) | 0x0001);
```

```
// Enable GPIOA[0] input falling and rising edge interrupt by IRQSRCGPA register
outp32(IRQENGPA, inp32(IRQENGPA) | PA0ENF | PA0ENR);
// Set Interrupt latch trigger by IRQLHSEL register
outp32(IRQLHSEL, inp32(IRQLHSEL) | IRQ1LHE);

/* Install ISR */
...
/* enable CPSR I bit */
...
```

9. Pulse Width Modulation (PWM)

9.1. Overview

The NUC501 have 4 channels PWM-timers. The 4 channels PWM-timers has 2 prescaler, 2 clock divider, 4 clock selectors, 4 16-bit counters, 4 16-bit comparators, 2 Dead-Zone generator. They are all driven by system clock. Each channel can be used as a timer and issue interrupt independently.
Each two channels PWM-timers share the same prescaler(channel0-1 share prescalar0 and channel2-3 share prescalar1). Clock divider provides each channel with 5 clock sources (1, 1/2, 1/4, 1/8, 1/16). Each channel receives its own clock signal from clock divider which receives clock from 8-bit prescaler. The 16-bit counter in each channel receive clock signal from clock selector and can be used to handle one PWM period. The 16-bit comparator compares number in counter with threshold number in register loaded previously to generate PWM duty cycle.

The NUC501 have 4 channels PWM-timers and each PWM-timer includes a capture channel. The Capture 0 and PWM 0 share a timer that included in PWM 0; and the Capture 1 and PWM 1 share another timer, and etc. Therefore user must setup the PWM-timer before turn on Capture feature. After enabling capture feature, the capture always latched PWM-counter to CRLR when input channel has a rising transition and latched PWM-counter to CFLR when input channel has a falling transition. Capture channel 0 interrupt is programmable by setting CCR0[1] (Rising latch Interrupt enable) and CCR0[2] (Falling latch Interrupt enable) to decide the condition of interrupt occur. Capture channel 1 has the same feature by setting CCR0[17] and CCR0[18]. And capture channel 2 & 3 has the same feature by setting CCR1[1],CCR1[2] and CCR1[17], CCR1[18] respectively. Whenever Capture issues Interrupt 0/1/2/3, the PWM counter 0/1/2/3 will be reload at this moment.

There are only four interrupts from PWM to advanced interrupt controller (AIC). PWM 0 and Capture 0 share the same interrupt channel, PWM1 and Capture 1 share the same interrupt and so on. Therefore, PWM function and Capture function in the same channel cannot be used at the same time.

The PWM features are :

- Two 8-bit prescalers and Two clock dividers
- Four clock selectors
- Four 16-bit counters and four 16-bit comparators
- Two Dead-Zone generator
- Capture function

9.2. Block Diagram

The following figure describes the architecture of PWM in one group. (channel0&1 are in one group and channel2&3 are in another group)

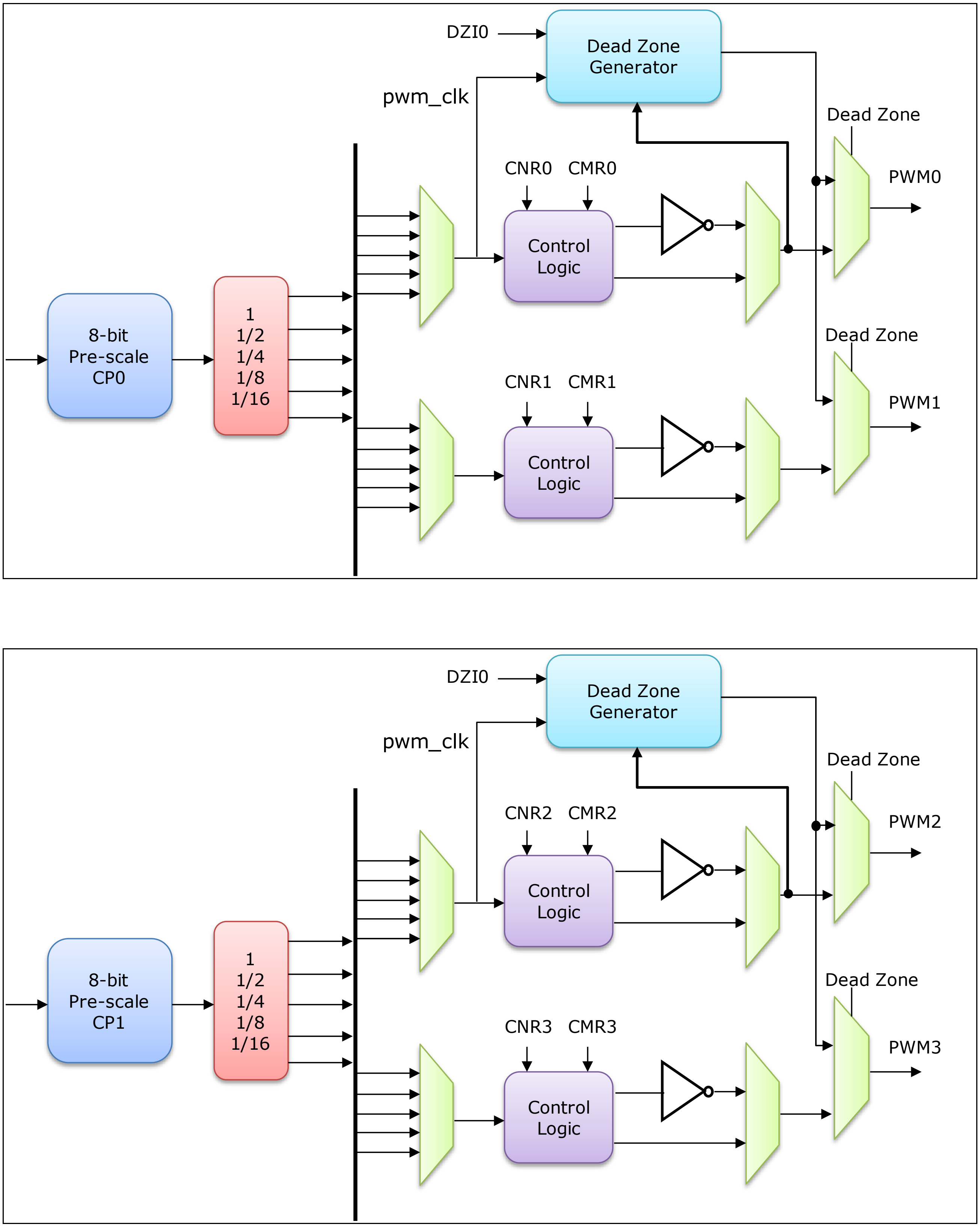


Figure 9-1 PWM Architecture Diagram

9.3. Registers

R: read only, W: write only, R/W: both read and write, C: Only value 0 can be written

Register	Address	R/W	Description	Reset Value
PWM_BA = 0xB800_7000				
PPR	PWM_BA+0x000	R/W	PWM Pre-scale Register	0x0000_0000
CSR	PWM_BA+0x004	R/W	PWM Clock Select Register	0x0000_0000
PCR	PWM_BA+0x008	R/W	PWM Control Register	0x0000_0000
CNR0	PWM_BA+0x00C	R/W	PWM Counter Register 0	0x0000_0000
CMR0	PWM_BA+0x010	R/W	PWM Comparator Register 0	0x0000_0000
PDR0	PWM_BA+0x014	R	PWM Data Register 0	0x0000_0000
CNR1	PWM_BA+0x018	R/W	PWM Counter Register 1	0x0000_0000
CMR1	PWM_BA+0x01C	R/W	PWM Comparator Register 1	0x0000_0000
PDR1	PWM_BA+0x020	R	PWM Data Register 1	0x0000_0000
CNR2	PWM_BA+0x024	R/W	PWM Counter Register 2	0x0000_0000
CMR2	PWM_BA+0x028	R/W	PWM Comparator Register 2	0x0000_0000
PDR2	PWM_BA+0x02C	R	PWM Data Register 2	0x0000_0000
CNR3	PWM_BA+0x030	R/W	PWM Counter Register 3	0x0000_0000
CMR3	PWM_BA+0x034	R/W	PWM Comparator Register 3	0x0000_0000
PDR3	PWM_BA+0x038	R	PWM Data Register 3	0x0000_0000
PIER	PWM_BA+0x040	R/W	PWM Interrupt Enable Register	0x0000_0000
PIIR	PWM_BA+0x044	R/C	PWM Interrupt Indication Register	0x0000_0000
CCR0	PWM_BA+0x050	R/W	Capture Control Register 0	0x0000_0000
CCR1	PWM_BA+0x054	R/W	Capture Control Register 1	0x0000_0000
CRLR0	PWM_BA+0x058	R/W	Capture Rising Latch Register (Channel 0)	0x0000_0000
CFLR0	PWM_BA+0x05C	R/W	Capture Falling Latch Register (Channel 0)	0x0000_0000
CRLR1	PWM_BA+0x060	R/W	Capture Rising Latch Register (Channel 1)	0x0000_0000
CFLR1	PWM_BA+0x064	R/W	Capture Falling Latch Register (Channel 1)	0x0000_0000
CRLR2	PWM_BA+0x068	R/W	Capture Rising Latch Register (Channel 2)	0x0000_0000
CFLR2	PWM_BA+0x06C	R/W	Capture Falling Latch Register (Channel 2)	0x0000_0000
CRLR3	PWM_BA+0x070	R/W	Capture Rising Latch Register (Channel 3)	0x0000_0000
CFLR3	PWM_BA+0x074	R/W	Capture Falling Latch Register (Channel 3)	0x0000_0000
CAPENR	PWM_BA+0x078	R/W	Capture Input Enable Register	0x0000_0000
POE	PWM_BA+0x07C	R/W	PWM Output Enable	0x0000_0000

9.4. Functional Description

9.4.1. PWM Timer / Capture Channel

Here is brief description to tell the difference between Timer and Capture

1. PWM timer function can be used to be a general counter (No waveform output) or to create a specified frequency waveform (Waveform output).
2. Capture function can get the input signal information. It gets the PWM internal counter value when input signal is rising or falling. Then, user can use the APB clock and the captured values to obtain the input signal information. Therefore, the corresponding PWM timer needs to be enabled before using capture function.
3. The difference of register configuration between PWM timer and capture function
   - A. Pin function (PAD_REG0)
     - i. PWM timer : Select one or several pin to be the output pin(s)
     - ii. Capture Select only one pin to be the input pin
   - B. Only Capture function to configure the Capture function registers (CCR0/CCR1)
   - C. PWM function I/O Enable
     - i. PWM timer : PWM Output Enable Register (POE)
     - ii. Capture : Capture Input Enable Register (CAPENR)

9.4.2. PWM Timer

9.4.2.1. Prescaler and clock selector

The PWM has two groups (two channels in each group) of timers. The clock input of the group is according to the PWM Prescaler Register (**PPR**) value. The PWM prescaler divided the clock input by PPR+1 before it is fed to the counter. Please notice that when the PPR value equals zero, the prescaler output clock will stop. Furthermore, according to the PWM Clock Select Register (**CSR**) value, the clock input of PWM timer channel can be divided by 1,2,4,8 and 16.

Consider following examples, which explain the PWM timer period.

$$\text{period} = \frac{1}{(APBCLK) \div (PPR+1) \div CSR}$$

When the PCLK = 60 MHz, the maximum and minimum PWM timer counting period is described as follows.

Maximum period: PPR = 255 (since the length of PPR is 8bit) and CSR = 16

$$\text{period}_{\max} = \frac{1}{(60Mhz) \div (255+1) \div 16} = 68.266us$$

Minimum period: PCLK = 60 MHz, PPR=1 and CSR=1

$$\text{period}_{\min} = \frac{1}{(60Mhz) \div (1+1) \div 1} = 0.0333us$$

The maximum and minimum interval between two interrupts are according to the $period_{max}$, $period_{min}$ and PWM Counter Register(**CNRx**) length. The maximum interval between two interrupts is (65535)*(68.266us) since the length of CNR is 16bit. Please notice that the above calculation is based on the APBCLK = 60MHz. Therefore, all of the values need to be recalculated when the APBCLK is not equal to 60 MHz.

9.4.2.2. Basic Timer Operation

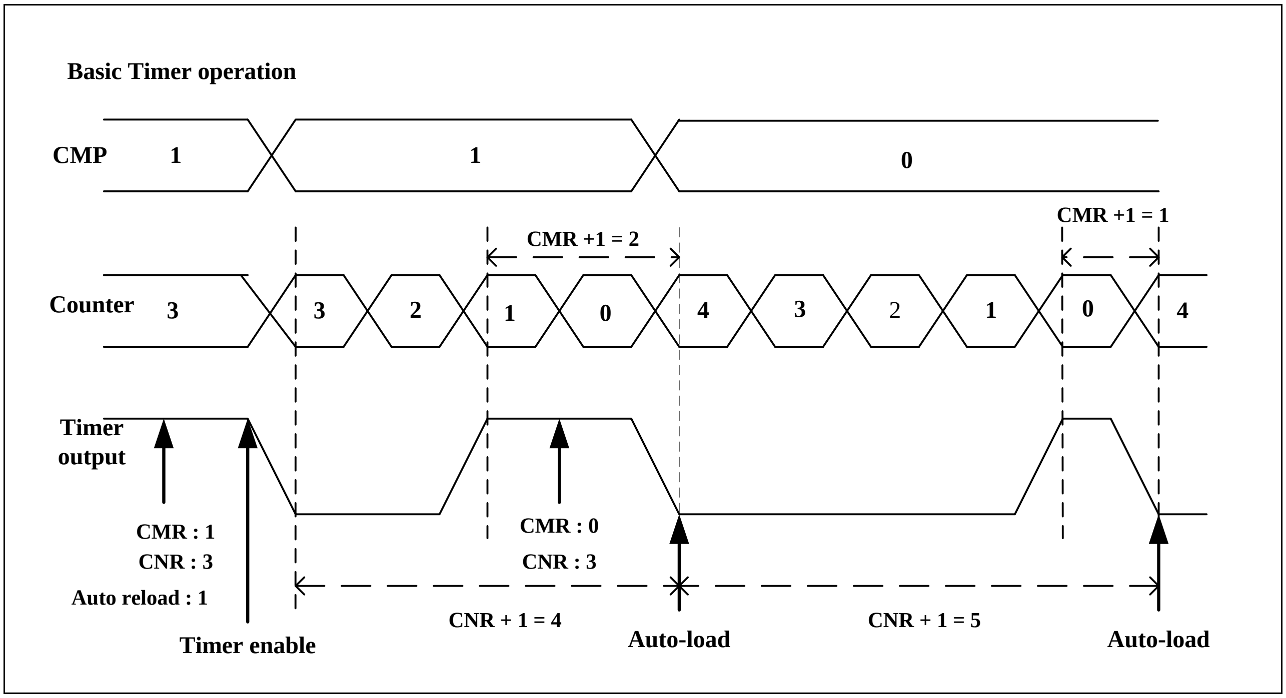


Figure 9-2 Basic Timer Operation Timing

9.4.2.3. PWM Double Buffering and Automatic Reload

NUC501 PWM Timers have a double buffering function, enabling the reload value changed for next timer operation without stopping current timer operation. Although new timer value is set, current timer operation still operate successfully.

The counter value can be written into CNR0~3 and current counter value can be read from PDR0~3.

The auto-reload operation copies loaded value from CNR0~3 to down-counter when down-counter reaches zero. If CNR0~3 are set as zero, counter will be halt when counter count to zero. If auto-reload bit is set as zero, counter will be stopped immediately.

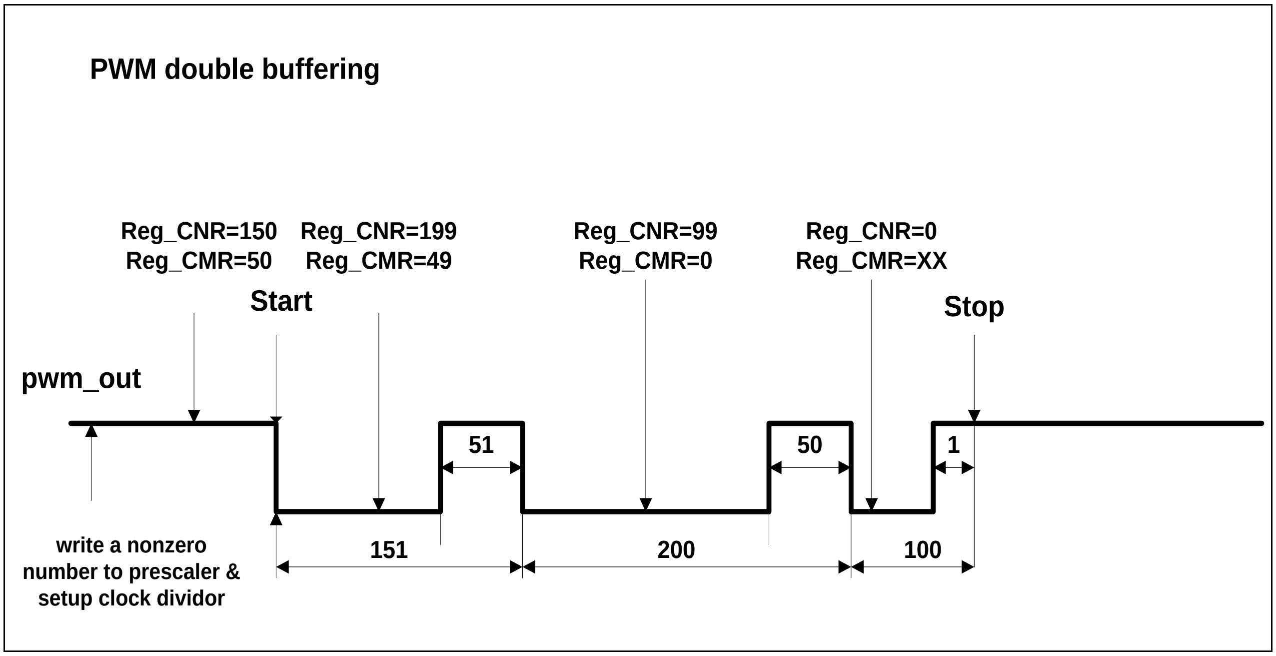


Figure 9-3 PWM Double Buffering Illustration

9.4.2.4. Modulate Duty Ratio

The double buffering function allows CMR written at any point in current cycle. The loaded value will take effect from next cycle.

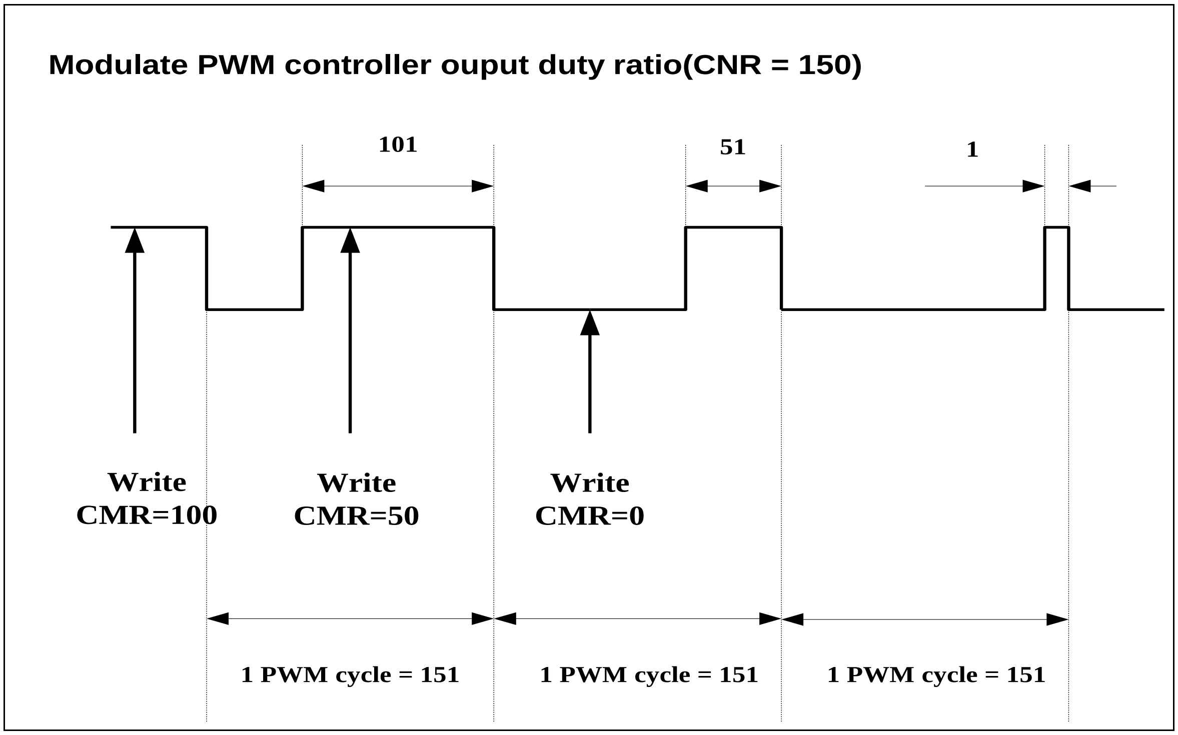


Figure 9-4 PWM Controller Output Duty Ratio

9.4.2.5. Dead-Zone Generator

NUC501 PWM is implemented with Dead Zone generator. They are built for power device protection. This function enables generation of a programmable time gap at the rising of PWM output waveform. User can program PPR [31:24] and PPR [23:16] to determine the two Dead Zone interval respectively.

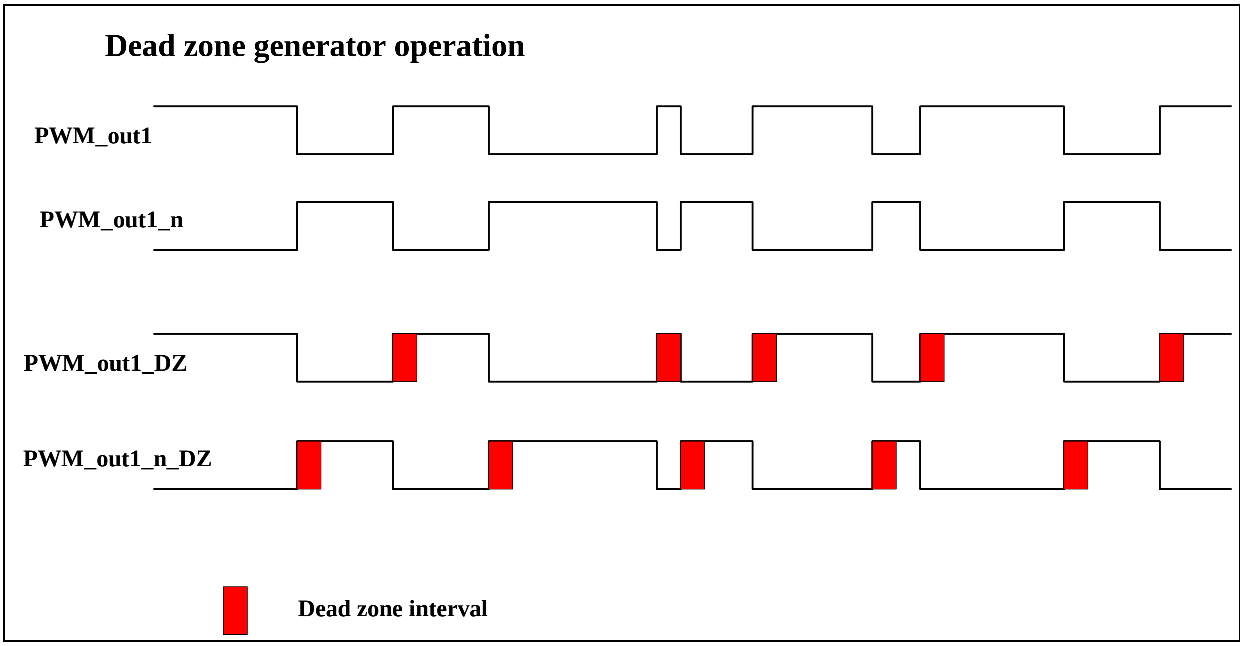


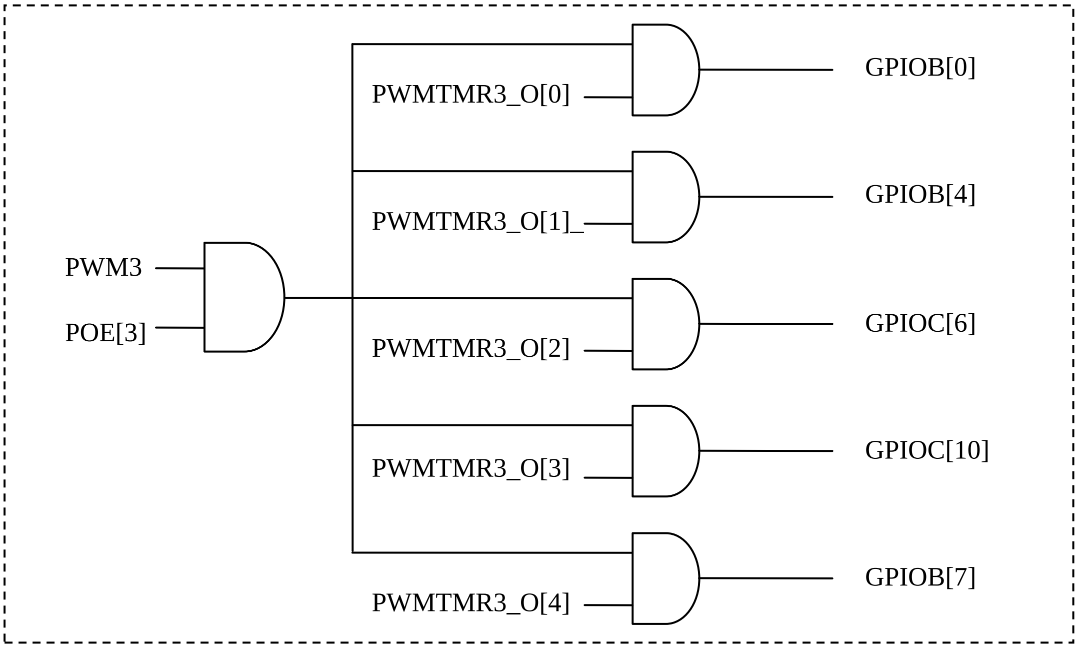
Figure 9-5 Dead Zone Generation Operation

9.4.2.6. PWM Timer Start Procedure

1. Pin function setting

   Each PWM channel has several output pins. User can configure the PAD Control Register (PAD_REG0) to choose the pin for PWM timer output pins. (PWM timer can output to several pins) and enable the output function in the PWM Output Enable Register (**POE**).

   The following figure shows the relationship between waveform output pins and I/O enable configuration. Only when the corresponding POE bit is enabled, the waveform can output from the specified pins. The waveform can output to several pins (controlled by PWMTMRx_O) concurrently.



PAD_REG0:

Bits	Descriptions	
[28:24]	**PWM_TMR3_O**	PWM Timer 3 output pin selection 1 = output enable 0 = output disable [24] = PWM Timer 3 channel 0 output to GPIOB[0] [25] = PWM Timer 3 channel 1 output to GPIOB[4] [26] = PWM Timer 3 channel 2 output to GPIOC[6] [27] = PWM Timer 3 channel 3 output to GPIOC[10] [28] = PWM Timer 3 channel 4 output to GPIOB[7]
[20:16]	**PWM_TMR2_O**	PWM Timer 2 output pin selection 1 = output enable 0 = output disable [16] = PWM Timer 2 channel 0 output to GPIOA[15] [17] = PWM Timer 2 channel 1 output to GPIOB[3] [18] = PWM Timer 2 channel 2 output to GPIOC[5] [19] = PWM Timer 2 channel 3 output to GPIOC[9] [20] = PWM Timer 2 channel 4 output to GPIOB[6]

[12:8]	**PWM_TMR1_O**	PWM Timer 1 output pin selection 1 = output enable 0 = output disable [8] = PWM Timer 1 channel 0 output to GPIOA[13] [9] = PWM Timer 1 channel 1 output to GPIOB[2] [10] = PWM Timer 1 channel 2 output to GPIOC[4] [11] = PWM Timer 1 channel 3 output to GPIOC[8] [12] = PWM Timer 1 channel 4 output to GPIOB[9]
[4:0]	**PWM_TMR0**	PWM Timer 0 output pin selection 1 = output enable 0 = output disable [0] = PWM Timer 0 channel 0 output to GPIOA[12] [1] = PWM Timer 0 channel 1 output to GPIOB[1] [2] = PWM Timer 0 channel 2 output to GPIOC[3] [3] = PWM Timer 0 channel 3 output to GPIOC[7] [4] = PWM Timer 0 channel 4 output to GPIOB[8]

2. Setup clock selector (CSR)
3. Setup prescaler & dead zone interval (PPR)
4. Setup inverter on/off, dead zone generator on/off, toggle mode /one-shot mode, and pwm timer off. (PCR)
5. Setup comparator register (CMR)
6. Setup counter register (CNR)
7. Setup interrupt enable register (PIER)
8. Enable pwm timer (PCR)

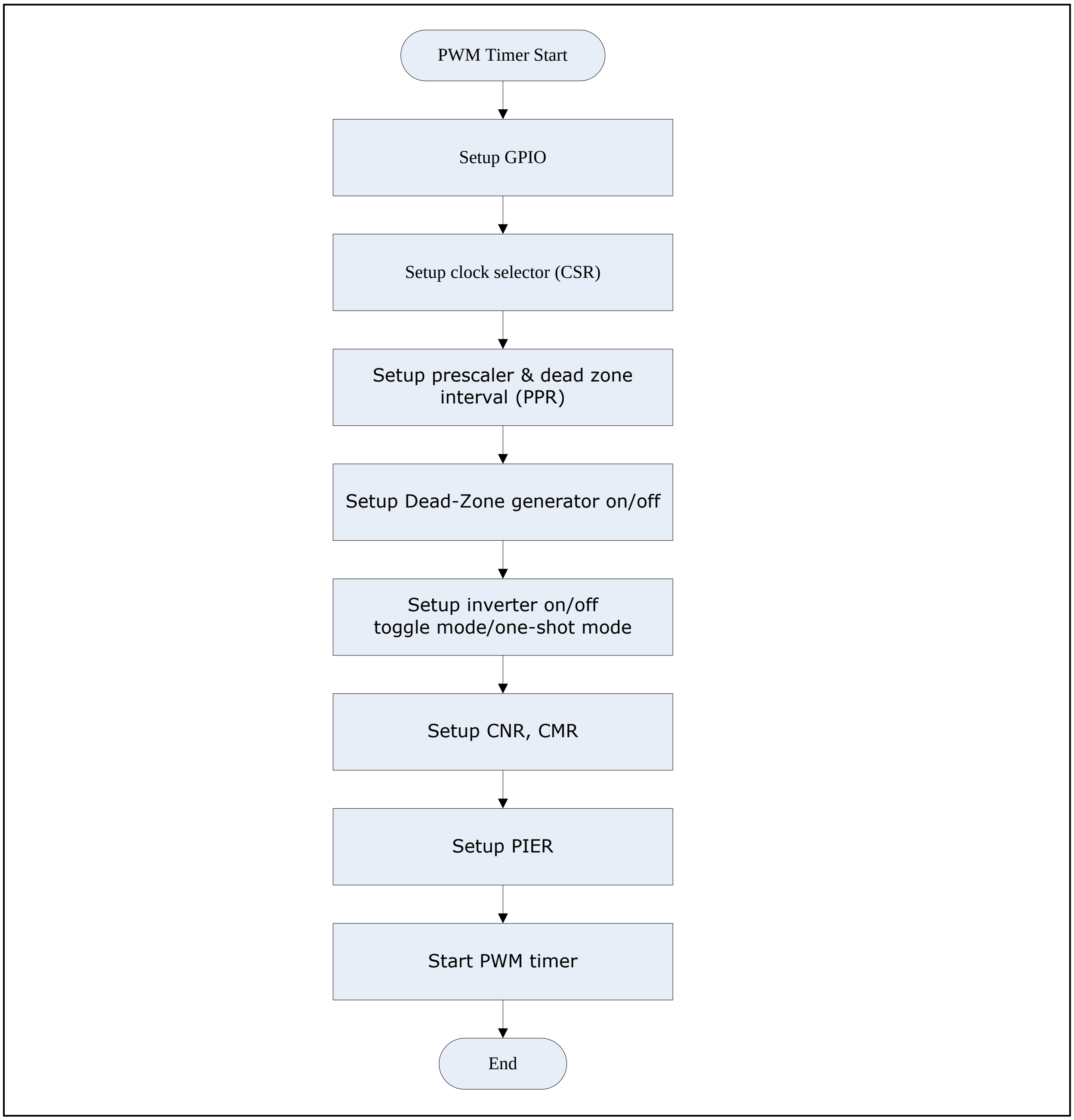


Figure 9-6 PWM Timer Start Procedure

9.4.2.7. PWM Timer Stop Procedure

Method 1:
Set 16-bit down counter (CNR) as 0, and monitor PDR. When PDR reaches to 0, disable pwm timer (PCR). ***(Recommended)***

Method 2:
Set 16-bit down counter (CNR) as 0. When interrupt request happen, disable pwm timer (PCR). ***(Recommended)***

Method 3:
Disable pwm timer directly (PCR). ***(Not recommended)***

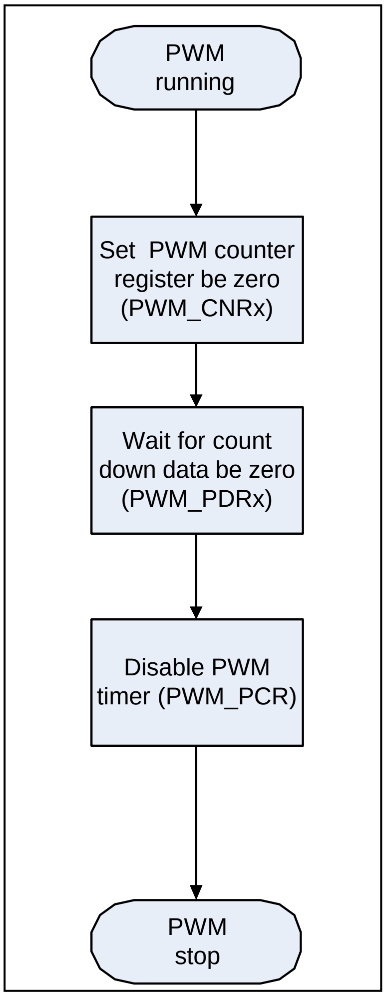


Figure 9-7 PWM Timer Stop flow chart (method 1)

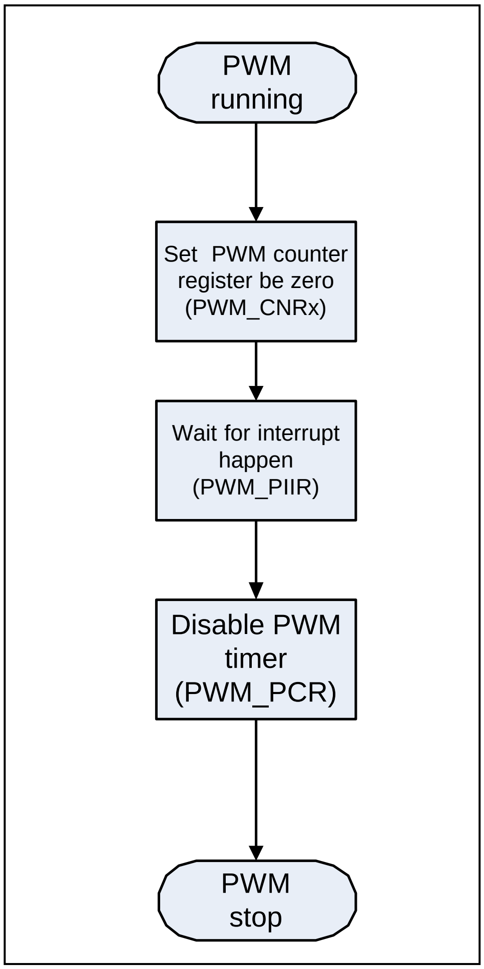


Figure 9-8 PWM Timer Stop flow chart (method 2)

9.4.3. Capture

9.4.3.1. Capture Description

Capture function can get the input signal information. It gets the PWM internal counter value when input signal is rising or falling. Then, user can use the APB clock and the captured values to obtain the input signal information. Therefore, the corresponding timer needs to be enabled before using capture function.

Here is some note for capture register:

1. CIIRx, FL&IEx, and RL&IEx
   - A. FL&IEx (Falling interrupt enable)
   - B. RL&IEx (Rising interrupt enable)
   - C. CIIR (Interrupt flag) : When a rising/falling transition and the rising/falling interrupt is enabled, this bit is 1. Write “0” to clear.
   - D. The rising & falling interrupt can be enabled concurrently. User can tell the interrupt type by the falling/.rising transition dirty bit.
2. CFLRDx and CRLRD

A. CFLRDx (Falling transition dirty bit)
B. CRLRDx (Rising transition dirty bit)
C. When input channel has a rising/falling transition, CRLRDx/CFLRDx is updated to “1” (No matter the rising/falling interrupt is enabled or not)
D. The bit is not updated to “0” when it has a falling transition. It needs to be clear by user.
E. Write “0” to clear

3. Interrupt and reload behavior
   A. The corresponding timer reloads when next capture interrupt occur when the falling interrupt is enabled and the Capture interrupt is clear.

9.4.3.2. Capture Start Procedure

1. Pin function setting.
   Each PWM channel has several input pins. User can configure the PAD Control Register (PAD_REG0) to choose the pin for PWM Capture input pin. (Capture only can input from one pin) and enable the input function in the Capture Input Enable Register (**CAPENR**).

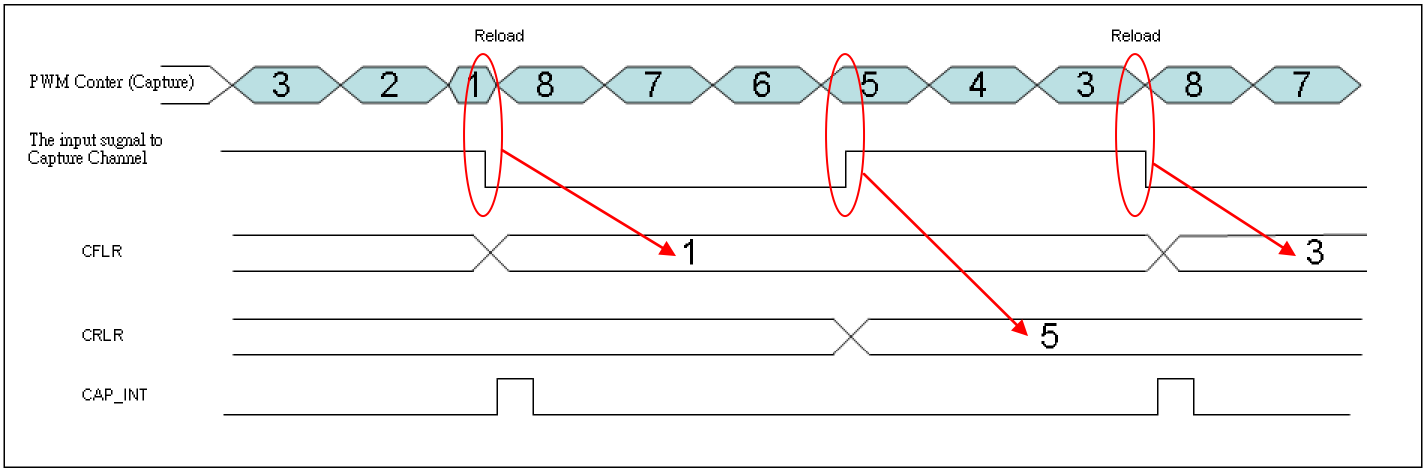
PAD_REG0:

Bits	Descriptions	
[31:29]	**PWM_TMR3_I**	PWM Timer 3 input pin selection 000 = PWM Timer 3 input from GPIOB[0] 001 = PWM Timer 3 input from GPIOB[4] 010 = PWM Timer 3 input from GPIOC[6] 011 = PWM Timer 3 input from GPIOC[10] 100 = PWM Timer 3 input from GPIOB[7] Others is unacceptable
[23:21]	**PWM_TMR2_I**	PWM Timer 2 input pin selection 000 = PWM Timer 2 input from GPIOA[15] 001 = PWM Timer 2 input from GPIOB[3] 010 = PWM Timer 2 input from GPIOC[5] 011 = PWM Timer 2 input from GPIOC[9] 100 = PWM Timer 2 input from GPIOB[6] Others is unacceptable
[15:13]	**PWM_TMR1_I**	PWM Timer 1 input pin selection 000 = PWM Timer 1 input from GPIOA[13] 001 = PWM Timer 1 input from GPIOB[2] 010 = PWM Timer 1 input from GPIOC[4] 011 = PWM Timer 1 input from GPIOC[8] 100 = PWM Timer 1 input from GPIOB[9] Others is unacceptable

[7:5]	**PWM_TMR0_I**	PWM Timer 0 input pin selection 000 = PWM Timer 0 input from GPIOA[12] 001 = PWM Timer 0 input from GPIOB[1] 010 = PWM Timer 0 input from GPIOC[3] 011 = PWM Timer 0 input from GPIOC[7] 100 = PWM Timer 0 input from GPIOB[8]

2. Enable the corresponding timer
   - Setup clock selector (CSR)
   - Setup prescaler & dead zone interval (PPR)
   - Setup inverter on/off, dead zone generator on/off, toggle mode /one-shot mode, and pwm timer off. (PCR)
   - Setup comparator register (CMR)
   - Setup counter register (CNR)
   - Enable pwm timer (PCR)
3. Setup capture register (CCR0/CCR1)
   - Clear dirty bit (CRLRDx/CFLRDx)
   - Clear interrupt flag(CIIRx)
   - Enable/Disable Inverter function (INVx)
   - Enable /Disable the interrupt (FL&IEx/ RL&IEx)
4. Enable pwm capture (CAPCHxEN bit)

9.4.3.3. Capture Basic Timer Operation



At this case, the CNR is 8 for capture channel (CAPCHxEN = 1):

1. When set falling interrupt enable, the pwm counter will be reload at time of interrupt occur.
2. The channel low pulse width is (CNR – CRLR).
3. The channel high pulse width is (CRLR - CFLR).
4. The channel cycle time is (CNR – CFLR).

10. Real Time Clock (RTC)

10.1. Overview

Real Time Clock (RTC) block can be operated by independent power supply while the system power is off. The RTC uses a 32.768 KHz external crystal. The RTC can transmit data to CPU with BCD values. The data includes the time by (second, minute and hour), the date by (day, month and year). In addition, to achieve better frequency accuracy, the RTC counter can be adjusted by software.

Features:

- Time counter (second, minute, hour) and calendar counter (day, month, year).
- Alarm register (second, minute, hour, day, month, year).
- 12-hour or 24-hour mode is selectable.
- Recognize leap year automatically
- Day of the week counter
- Frequency compensate register(FCR)
- Beside FCR, all clock and alarm data expressed in BCD code
- Support tick time interrupt
- Support wake up function.

10.2. Block Diagram

The following figure describes the architecture of real time clock

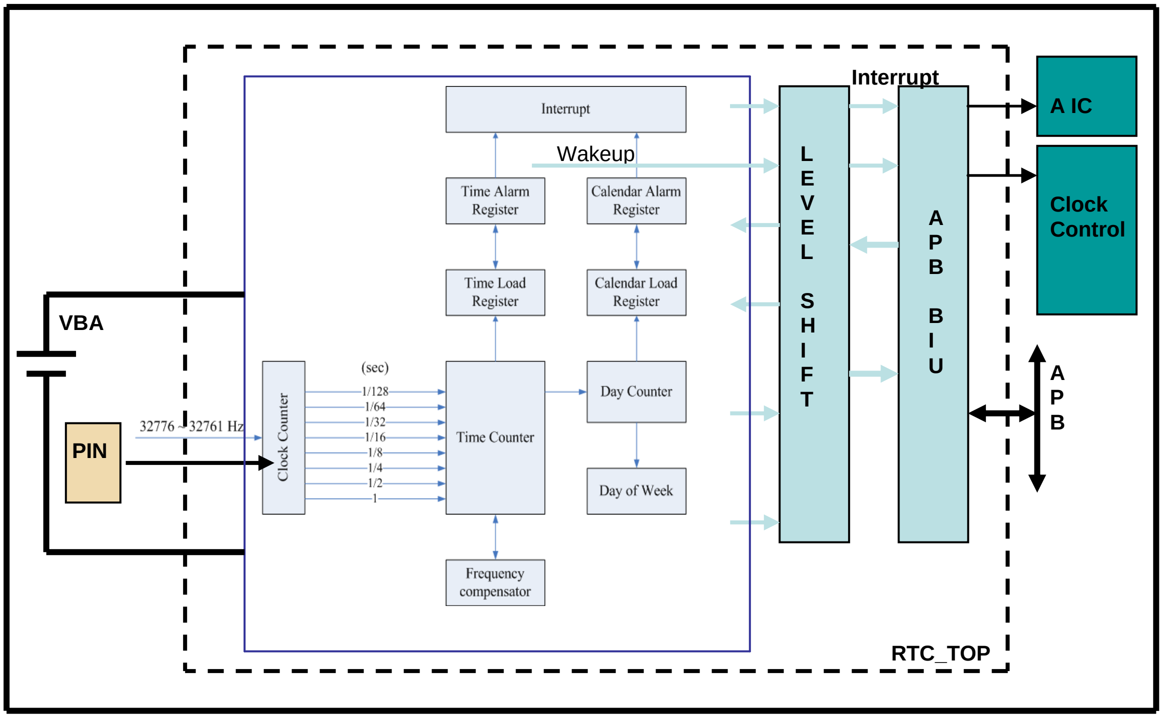


Figure 10-1 RTC Architecture Diagram

10.3. Registers

Register	Address	R/W	Description	Reset Value
RTC_BA = 0xB800_8000				
INIR	RTC_BA+0x000	R/W	RTC Initiation Register	0x0000_0000
AER	RTC_BA+0x004	R/W	RTC Access Enable Register	0x0000_0000
FCR	RTC_BA+0x008	R/W	RTC Frequency Compensation Register	0x0000_0700
TLR	RTC_BA+0x00C	R/W	Time Loading Register	0x0000_0000
CLR	RTC_BA+0x010	R/W	Calendar Loading Register	0x0005_0101
TSSR	RTC_BA+0x014	R/W	Time Scale Selection Register	0x0000_0001
DWR	RTC_BA+0x018	R/W	Day of the Week Register	0x0000_0006
TAR	RTC_BA+0x01C	R/W	Time Alarm Register	0x0000_0000
CAR	RTC_BA+0x020	R/W	Calendar Alarm Register	0x0000_0000
LIR	RTC_BA+0x024	R	Leap year Indicator Register	0x0000_0000
RIER	RTC_BA+0x028	R/W	RTC Interrupt Enable Register	0x0000_0000
RIIR	RTC_BA+0x02C	R/C	RTC Interrupt Indicator Register	0x0000_0000
TTR	RTC_BA+0x030	R/W	RTC Time Tick Register	0x0000_0000

10.4. Functional Description

10.4.1. Initialization

When RTC block is power on, programmer has to write a number (**0xa5eb1357**) to register **INIR** to reset all logic. **INIR** act as hardware reset circuit. Once **INIR** has been set as **0xa5eb1357,** there is no action for RTC if any value be programmed into **INIR** register.

10.4.2. RTC Read/Write Enable

Register **AER** bit 15~0 is for RTC read/write password. It is used to avoid signal interference from system during system power off. **AER** bit 15~0 has to be set as **0xa965** after system power on. Once it is set, it will take effect 512 RTC clocks later (about **15ms**). Programmer can read **AER** bit 16 to find out whether RTC register can be accessed.

10.4.3. Frequency Compensation

The RTC **FCR** allows software control digital compensation of a 32.768 KHz crystal oscillator. User can utilize a frequency counter to measure RTC clock in one of GPIO pin during manufacture, and store the value in Flash memory for retrieval when the product is first power on. The equation fro FCR please see the section 10.4.8.

10.4.4. Time and Calendar counter

TLR and **CLR** are used to load the time and calendar. **TAR** and **CAR** are used for alarm. They are all represented by BCD.

10.4.5. Day of the week counter

Count from Sunday to Saturday.

10.4.6. Time tick interrupt

RTC block use a counter to calibrate the time tick count value. When the value in counter reaches zero, RTC will issue an interrupt.

10.4.7. RTC register property

When system power is off but RTC power is on, data stored in RTC registers will not lost except **RIER** and **RIIR**. Because of clock difference between RTC clock and system clock, when user write new data to any one of the registers, the register will not be updated until 2 RTC clocks later (60us). Hence programmer should consider about access sequence between **TSSR**, **TAR** and **TLR**.
In addition, user must be aware that RTC block does not check whether loaded data is out of bounds or not. RTC does not check rationality between **DWR** and **CLR** either.

10.4.8. Application Note

- **TAR**, **CAR**, **TLR** and **CLR** are all BCD counter, but **FCR** is not a BCD counter.
- Programmer has to make sure that the loaded values are reasonable, for example, Load **CLR** as 201a (year), 13 (month), 00 (day), or **CLR** does not match with **DWR**, etc.
- Reset state :

Register	Reset State
AER	0(RTC read/write disable)
CLR	05, 1, 1 (2005-1-1)
TLR	00 hr: 00 min: 00 sec
CAR	00/00/00
TAR	00:00:00
TSSR	1 (24 hr mode)
DWR	6 (Saturday)
RIER	0
RIIR	0
LIR	0
TTR	0

- FCR Calibration :
  - (a) FCR integer : look up the below table.

Integer part of detected value	FCR[11:8]	Integer part of detected value	FCR[11:8]
32776	1111	32768	0111
32775	1110	32767	0110
32774	1101	32766	0101
32773	1100	32765	0100
32772	1011	32764	0011

32771	1010	32763	0010
32770	1001	32762	0001
32769	1000	32761	0000

- (b) **FCR** Calibration :

  Example 1,

  Frequency counter measurement : 32773.65Hz (> 32768 Hz)
  Integer part : 32773 => 0x8005
  FCR_int = 0x05 – 0x01 + 0x08 = 0x0c
  Fraction part : 0.65 X 60 = 39 => 0x27
  FCR_fra = 0x27

  Example 2,

  Frequency counter measurement : 32765.27Hz (≦ 32768 Hz)
  Integer part : 32765 => 0x7ffd
  FCR_int = 0x0d – 0x01 – 0x08 = 0x04
  Fraction part : 0.27 x 60 = 16.2 => 0x10
  FCR_fra = 0x10

- In **TLR** and **TAR**, only 2 BCD digits are used to express "year". We assume 2 BCD digits of XY denote 20XY , but not 19XY or 21XY.

10.5. Programming Note

Be sure to write RTC access password (0xa965) to **AER** to enable RTC registers write before you write RTC register and each access time is about 15 ms.

- Set Calendar and Time
  1. When RTC is power on, programmer has to write a number 0xa5eb1357 to **INIR** to reset all logic RTC
  2. Read register **INIR**[0] if it equals to 1 means RTC is at normal active state.
  3. Write RTC access password (0xa965) to **AER** to enable RTC register write.
  4. Read register **AER**[16], RTC is read/write enable if it's equal to 1.
  5. Set register **TSSR**[0] to select 12-hour or 24-hour time scale mode.
  6. Set year, month and day to register **CLR**
  7. Set day of week to register **DWR**
  8. Set hour, minute and second to register **TLR**
  9. Write 0x0 to **AER** means disable RTC access enable/disable password

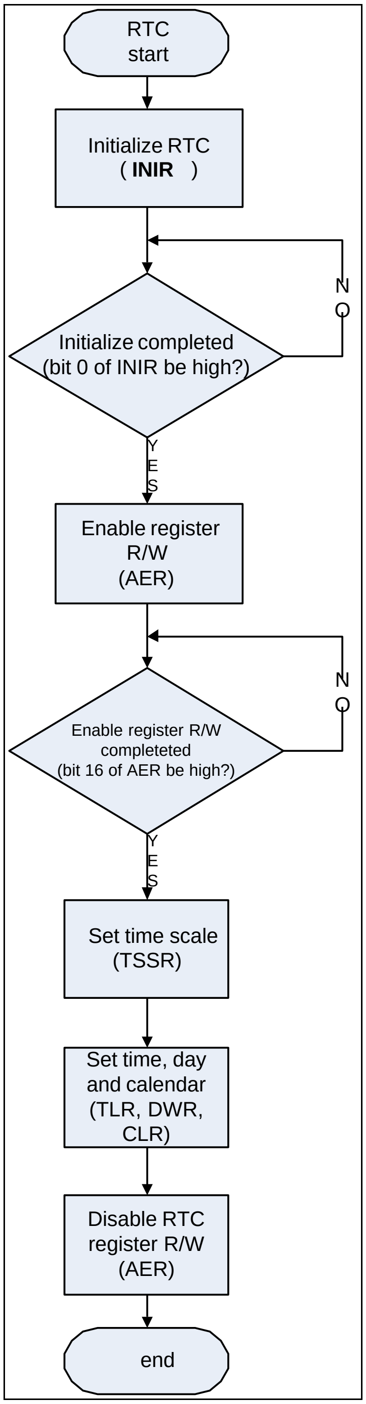


Figure 10-2 RTC Set Calendar and Time flow chart

- Set Calendar and Time Alarm
  - Set and prepare the ISR of RTC alarm
  - Set time and calendar same as above step 1-8
  - Set alarm year, month and day to register CAR
  - Set alarm hour, minute and second to register TAR

- Set “1” to RIER[0] for alarm interrupt enable
- Write 0x0 to AER means disable RTC access enable/disable password

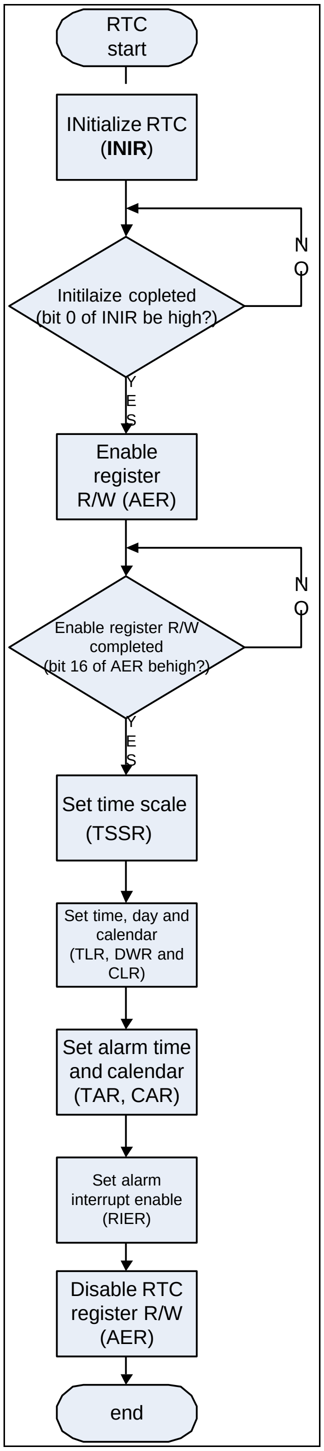


Figure 10-3 RTC Set Calendar and Time Alarm flow chart

- Set tick interrupt

- Set and prepare the ISR of RTC tick interrupt
- When RTC is power on, programmer has to write a number 0xa5eb1357 to **INIR** to reset all logic RTC
- Read register **INIR**[0] if it equals to 1 means RTC is at normal active state.
- Write RTC access password (0xa965) to **AER** to enable RTC register write.
- Read register **AER**[16], RTC is read/write enable if it's equal to 1.
- Set the **TTR** for tick interrupt happen time interval per second
- Set "1" to **RIER**[1] for tick interrupt enable
- Write 0x0 to **AER** means disable RTC access enable/disable password

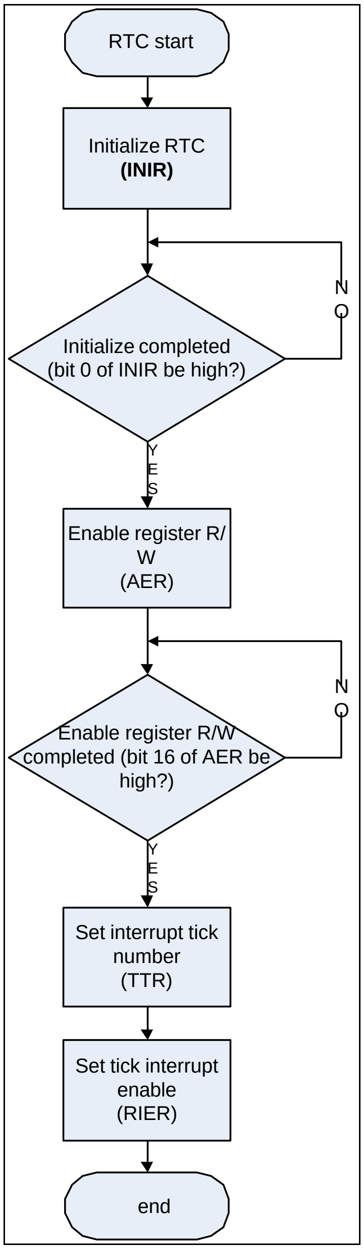


Figure 10-4 RTC Set tick interrupt flow chart

11. Serial Peripheral Interface Controller (SPI Master/Slave)

11.1. Overview

11.1.1. SPI Serial Interface Controller (Master/Slave)

The SPI controller performs a serial-to-parallel conversion on data characters received from the peripheral, and a parallel-to-serial conversion on data characters received from CPU. This controller can drive up to 2 external peripherals, but is time-shared and can not operate simultaneously. It also can be driven as the slave device when the CNTRL[18], SLAVE bit, be set.
It can generate an interrupt signal when data transfer is finished and can be cleared by writing 1 to the interrupt flag. The active level of slave select signal can be chosen to low active or high active on SSR[SS_LVL] bit, which depends on the peripheral it's connected. Writing a divisor into DIVIDER register can program the frequency of serial clock output. This controller contains four 32-bit transmit/receive buffers, and can provide burst mode operation. It supports variable length transfer and the maximum transmitted/received length can be up to 128 bits.

The SPI Master/Slave Core includes the following features:

- AMBA APB interface compatible
- Support SPI master/slave mode
- Full duplex synchronous serial data transfer
- Variable length of transfer word up to 32 bits
- Provide burst mode operation, transmit/receive can be executed up to four times in one transfer
- MSB or LSB first data transfer
- Rx and Tx on both rising or falling edge of serial clock independently
- 2 slave/device select lines when it is as the master mode, and 1 slave/device select line when it is as the slave mode
- Fully static synchronous design with one clock domain
- Only Support the external master device that the frequency of its serial clock output is less 1/4 than the SPI Core clock input (PCLK) and its slave select output is edge-active trigger.

11.2. Block Diagram

11.2.1. SPI Block Diagram (Master/Slave)

The block diagram of SPI Serial Interface controller is shown as following.

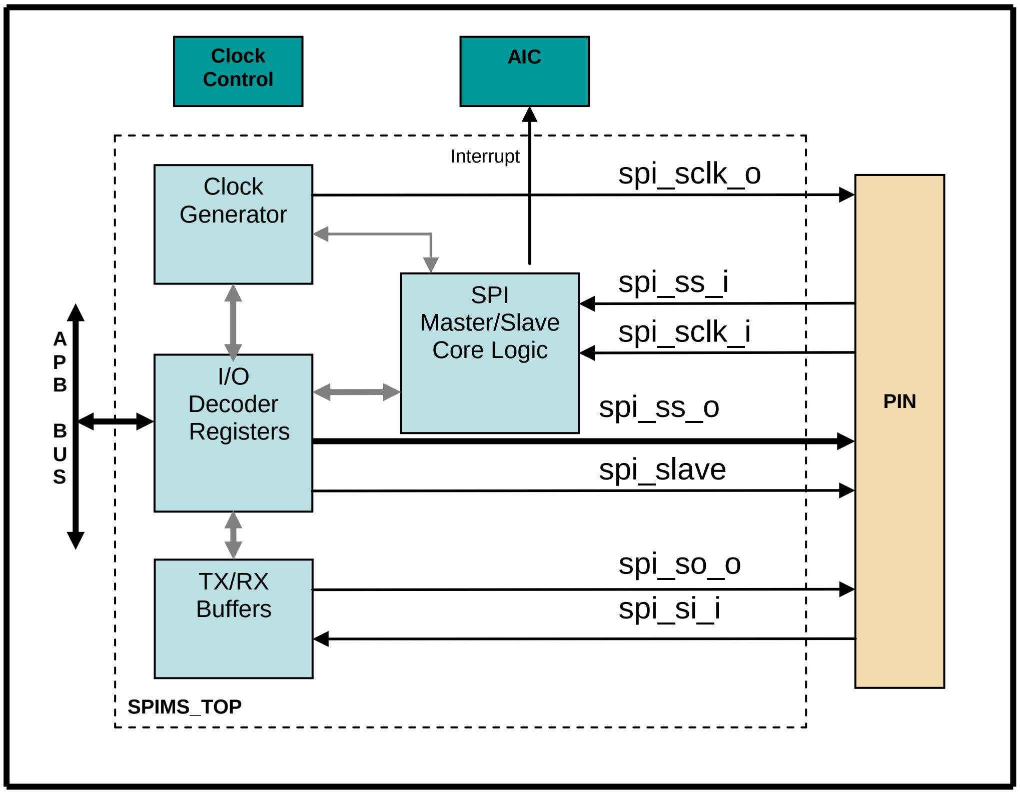


Figure 11-1 SPIMS Block Diagram(Master/Slave)

Pin descriptions:

spi_sclk_o: SPI master serial clock output pin.

spi_int_o: SPI interrupt signal output.

spi_ss_o[1:0]: SPI two slave/device select signals output.

spi_so_o: SPI serial data output pin (to slave device in master mode or to master device in slave mode).

spi_si_i: SPI serial data input pin (from slave device in master mode or from master device in salve mode).

spi_sclk_i: SPI slave serial clock input pin.

spi_ss_i: V SPI slave slave/device select signal input (edge-active trigger).

11.2.2. SPI Timing Diagram (Master/Slave)

The timing diagrams of SPI Master/Slave are shown as following.

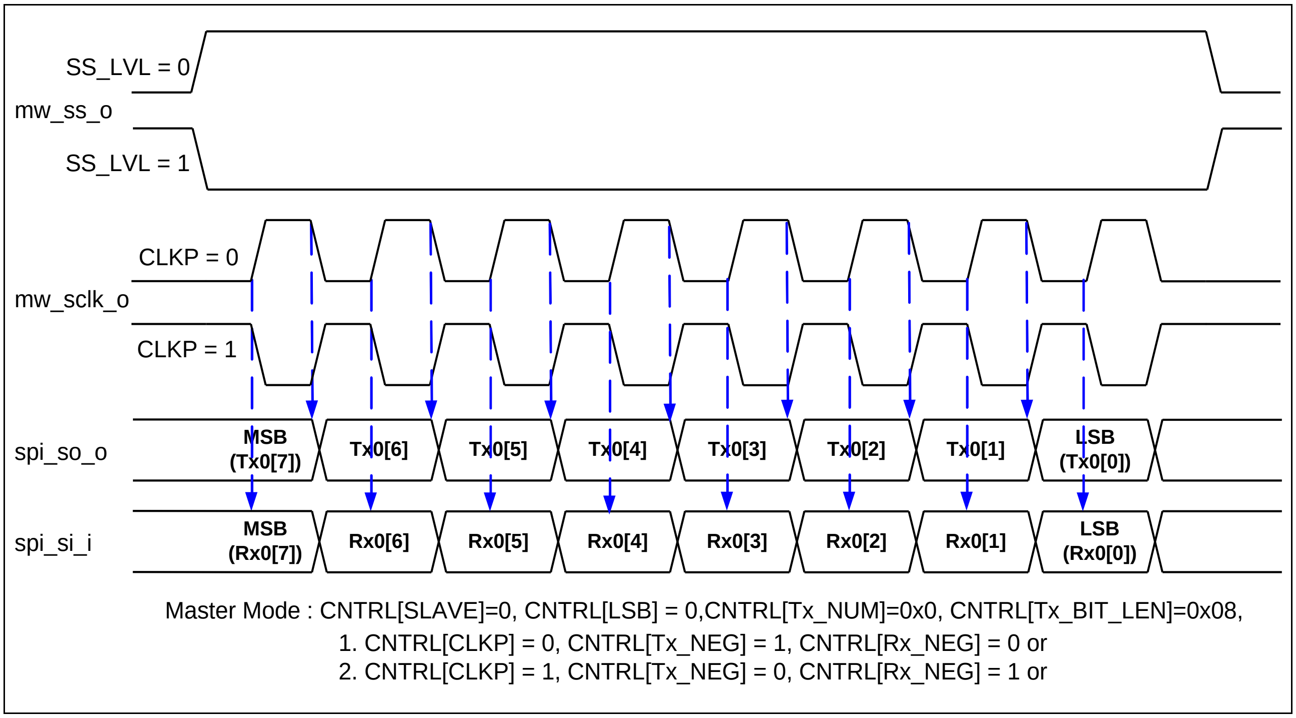


Figure 11-2 SPI Timing (Master)

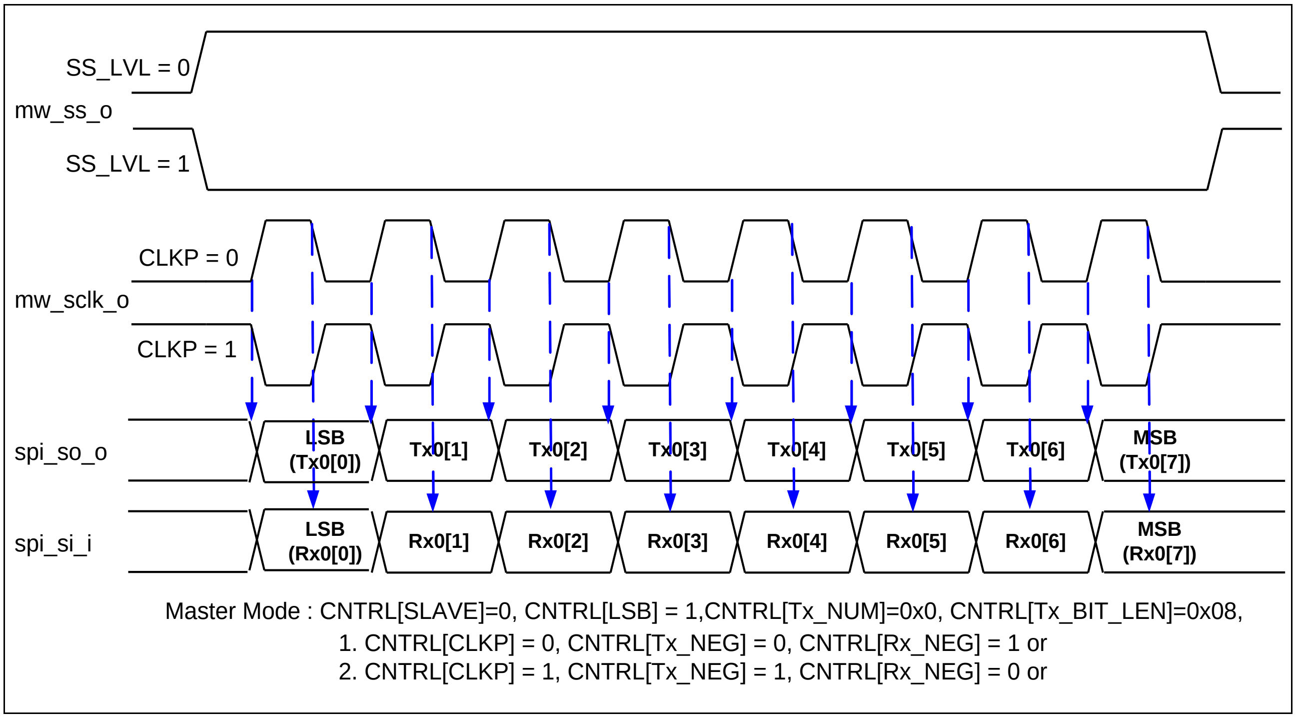


Figure 11-3 Alternate Phase SCLK clock Timing (Master)

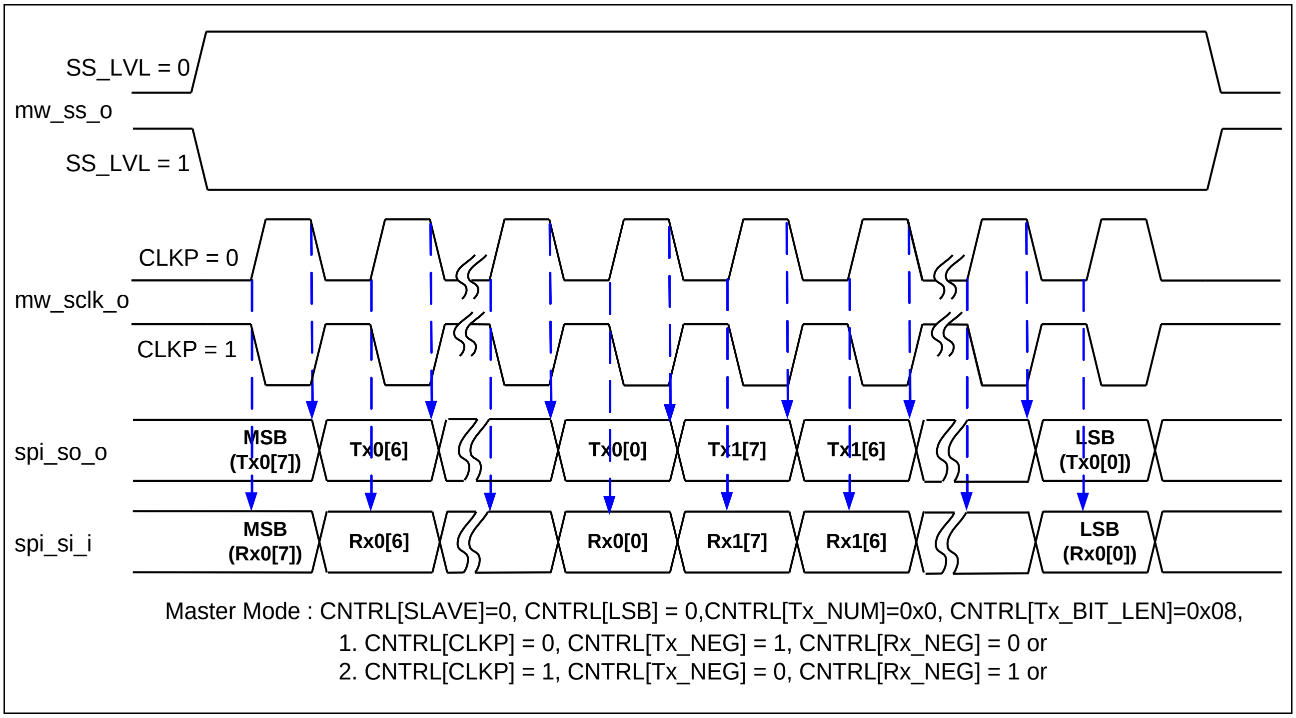


Figure 11-4 SPI Timing (Slave)

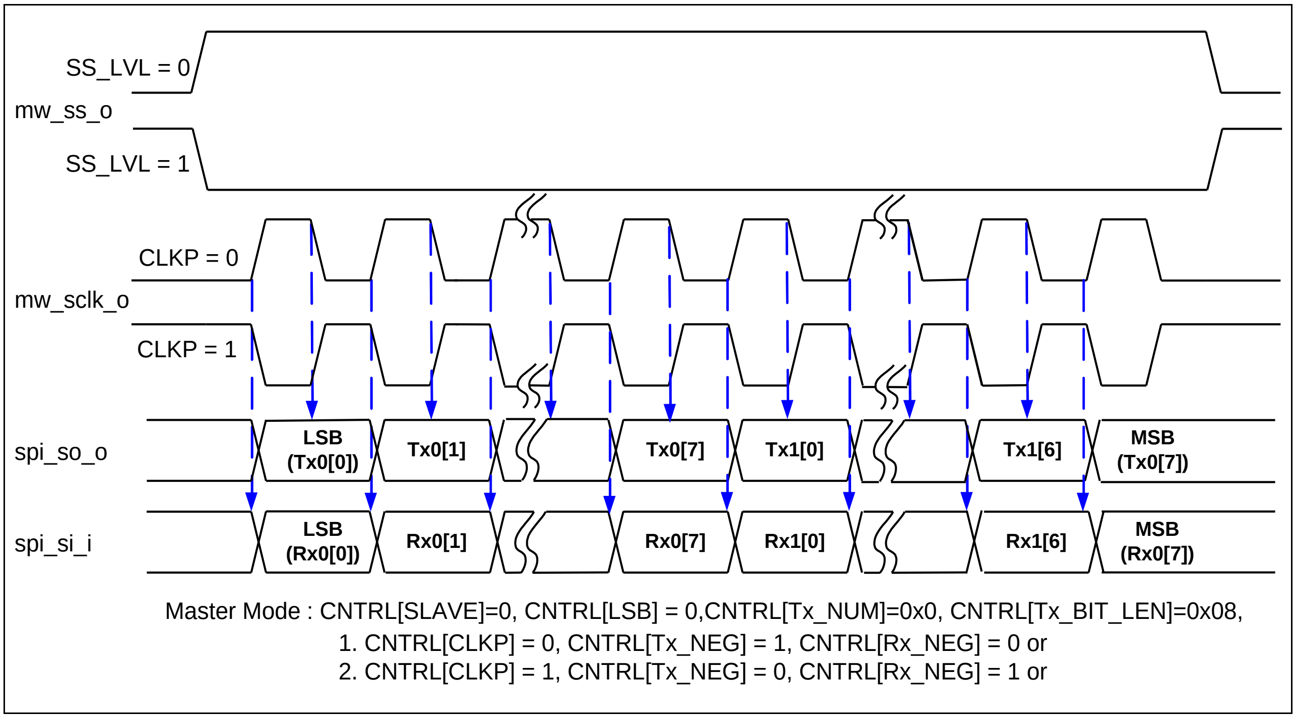


Figure 11-5 Alternate Phase SCLK Clock Timing (Slave)

11.3. Registers

R: read only, **W**: write only, **R/W**: both read and write

Register	Offset	R/W	Description	Reset Value
SPI_BA = 0xB800_A000				
CNTRL	SPI_BA + 0x00	R/W	Control and Status Register	0x0000_0004
DIVIDER	SPI_BA + 0x04	R/W	Clock Divider Register	0x0000_0000
SSR	SPI_BA + 0x08	R/W	Slave Select Register	0x0000_0000
Reserved	SPI_BA + 0x0C	N/A	**Reserved**	N/A
Rx0	SPI_BA + 0x10	R	Data Receive Register 0	0x0000_0000
Rx1	SPI_BA + 0x14	R	Data Receive Register 1	0x0000_0000
Rx2	SPI_BA + 0x18	R	Data Receive Register 2	0x0000_0000
Rx3	SPI_BA + 0x1C	R	Data Receive Register 3	0x0000_0000
Tx0	SPI_BA + 0x10	W	Data Transmit Register 0	0x0000_0000
Tx1	SPI_BA + 0x14	W	Data Transmit Register 1	0x0000_0000
Tx2	SPI_BA + 0x18	W	Data Transmit Register 2	0x0000_0000
Tx3	SPI_BA + 0x1C	W	Data Transmit Register 3	0x0000_0000

NOTE 1: When software programs CNTRL, the GO_BUSY bit should be written last.

11.4. Functional Description

11.4.1. Active SPI Controller

To activate the SPI, please follow the steps below:

1. Set the *TX_BIT_LEN* bit of **CNTRL** register to set the transmit bit length
2. Set the *TX_NUM* bit of **CNTRL** register to set the transfer numbers
3. Set the *GO_BUSY* bit of **CNTRL** register to activate SPI Controller
4. Polling *GO_BUSY* bit of **CNTRL** register until it was cleared, or waiting *IF* interrupt of **CNTRL** register.

11.4.2. Initialize SPI Controller

To initial the SPI Controller, please follow the steps below:

1. Configure GPIO SPI Multiple function
2. Set **DIVIDER** register to generate the serial clock on output clock
3. Set **SSR** register to select the access device
4. Set *LSB* bit of **CNTRL** register to send LSB or MSB first
5. Set the *IE* bit of **CNTRL** register to enable SPI Controller interrupt

11.4.3. SPI Controller Transmit/Receive

To transmit/receive the data, please follow the steps below:

1. Fill the data into **Tx0** ~ **Tx3** registers
2. Activate the SPI Controller
3. Receive the data from **Rx0** ~ **Rx3** registers

11.4.4. SPI Programming Example

The programming example is for accessing a device with following specifications

- Data bit latches on positive edge of serial clock
- Data bit drives on negative edge of serial clock
- Data is transferred with the MSB first
- Only one byte transmits/receives in a transfer
- Chip select signal is active low

You should do following actions basically (you should refer to the specification of device for the detailed steps):

1. Write a divisor into **DIVIDER** to determine the frequency of serial clock.
2. Write in **SSR**, set ASS = 0, SS_LVL = 0 and SSR[0] or SSR[1] to 1 to activate the device you want to access.

When transmit (write) data to device:

3. Write the data you want to transmit into **Tx0**[7:0].

When receive (read) data from device:

4. Write 0xFFFFFFFF into **Tx0**.
5. Write in **CNTRL**, set Rx_NEG = 0, Tx_NEG = 1, Tx_BIT_LEN = 0x08, Tx_NUM = 0x0, LSB = 0, SLEEP = 0x0 and GO_BUSY = 1 to start the transfer.
   — Wait for interrupt (if IE = 1) or polling the GO_BUSY bit until it turns to 0 —
6. Read out the received data from **Rx0**.
7. Go to 3) to continue data transfer or set SSR[0] or SSR[1] to 0 to inactivate the device.

12. Timer and WDT

12.1. Overview

12.1.1. General Timer Controller

The timer allows user to easily implement a counting scheme for use. The timer can perform functions like frequency measurement, event counting, interval measurement, pulse generation, delay timing, and so on. The timer possesses features such as adjustable resolution, programmable counting period. See descriptions below for more detailed information. The timer can generate an interrupt signal upon timeout, or provide the current value of count during operation.

The general TIMER Controller includes the following features

- Compliant with the AMBA APB
- One channel with a 32-bit counter and an interrupt request.
- Maximum uninterrupted time = (1 / 12 MHz) * (2^8) * (2^32 - 1), if TCLK = 12 MHz

12.1.2. Watchdog Timer

The purpose of watchdog timer is to perform a system restart after the software running into a problem. This recovers system from crash for some reasons. It is a free running timer with programmable time-out intervals. When the specified time internal expires, a system reset can be generated. If the watchdog timer reset function is enabled and the watchdog timer is not being reset before timing out, then the watchdog reset is activated after 1024 WDT clocks. Setting **WTE** in the register **WTCR** enables the watchdog timer.

The **WTR** should be set before making use of watchdog timer. This ensures that the watchdog timer restarts from a known state. The watchdog timer will start counting and time-out after a specified period of time. The time-out interval is selected by two bits, **WTIS[1:0]**. The **WTR** is self-clearing, i.e., after setting it, the hardware will automatically reset it.
When timeout occurs, Watchdog Timer interrupt flag is set. Watchdog Timer waits for an additional 1024 WDT clock cycles before issuing a reset signal, if the **WTRE** is set. The **WTRF** will be set and the reset signal will last for 16128 WDT clock cycles long. When used as a simple timer, the reset function is disabled. Watchdog Timer will set the **WTIF** each time a timeout occurs. The **WTIF** can be polled to check the status, and software can restart the timer by setting the **WTR**. The Watchdog Timer can be put in the test mode by setting **WTTME** in the register WTCR.

12.2. Block Diagram

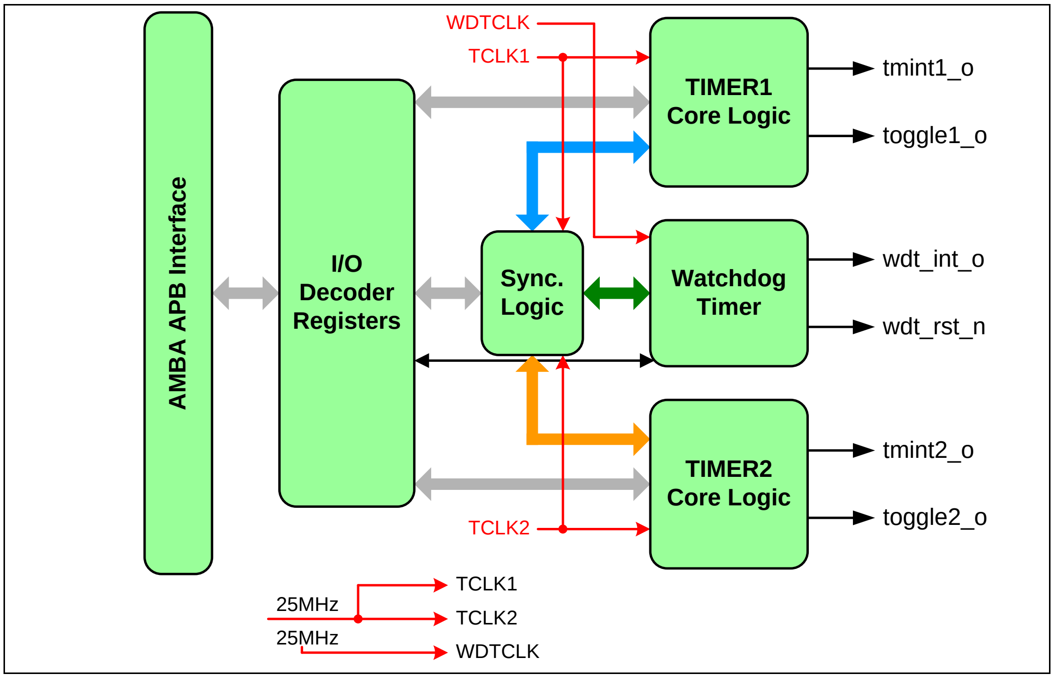


Figure 12-1 Timer Block Diagram

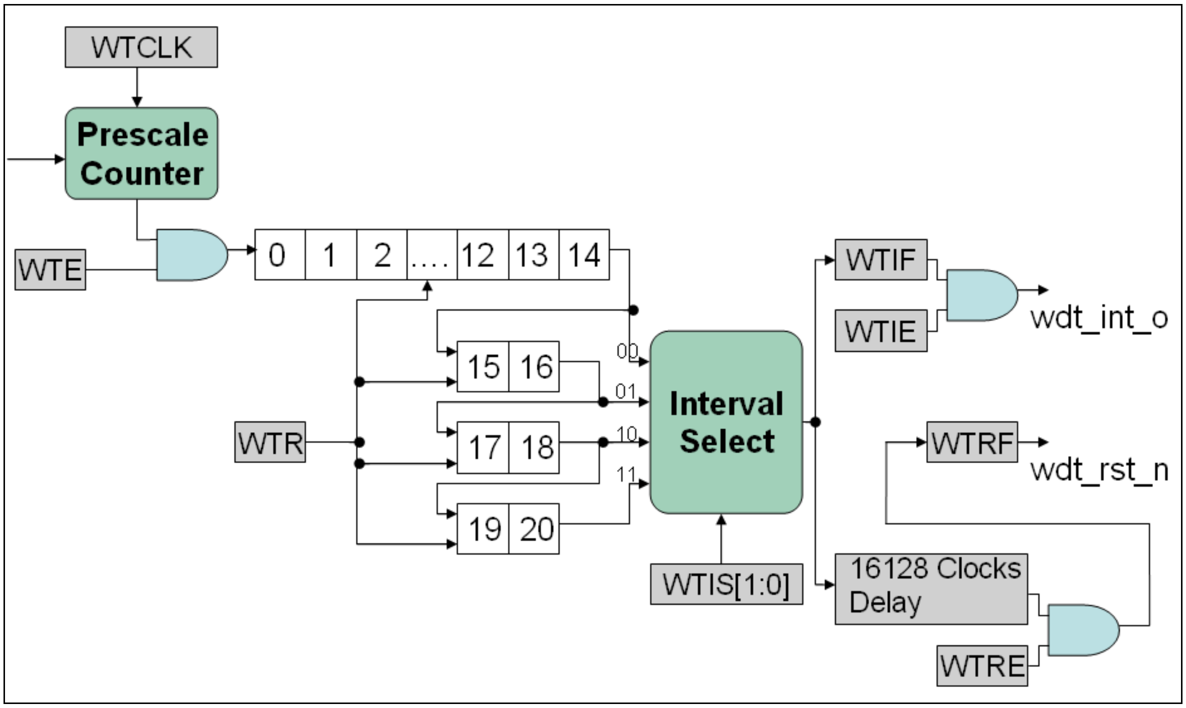


Figure 12-2 Watchdog Timer Block Diagram

12.3. Registers

R : Read only, **W** : Write only, **R/W** : Both read and write, **C** : Only value 0 can be written

Register	Address	R/W/C	Description	Reset Value
TMR_BA = 0xB800_B000				
TCSR0	TMR_BA+00	R/W	Timer Control and Status Register 0	0x0000_0005
TCSR1	TMR_BA+04	R/W	Timer Control and Status Register 1	0x0000_0005
TICR0	TMR_BA+08	R/W	Timer Initial Control Register 0	0x0000_0000
TICR1	TMR_BA+0C	R/W	Timer Initial Control Register 1	0x0000_0000
TDR0	TMR_BA+10	R	Timer Data Register 0	0x0000_0000
TDR1	TMR_BA+14	R	Timer Data Register 1	0x0000_0000
TISR	TMR_BA+18	R/W	Timer Interrupt Status Register	0x0000_0000
WTCR	TMR_BA+1C	R/W	Watchdog Timer Control Register	0x0000_0400

12.4. Functional Description

12.4.1. Interrupt Frequency

The frequency of timer interrupt depends on the following equation:

Freq. = Crystal clock / ((pre-scaler+1) * counter))

For example, the crystal clock input is 12 MHZ. According to the equation, user can decide the values of pre-scalar and counter to get the desired interrupt frequency. Table 2 demonstrates several reference values.

Frequency (1/sec)	[Pre-Scalar]	[Counter]
1	0xC	0xF4240
100	0xC	0x2710
1000	0xC	0x3E8

Table 2 Timer Reference Setting Values

12.4.2. Initialization

The driver should set the operating mode, pre-scalar and counter before enable the timer interrupt. The timer supports *one-shot, periodic, toggle and uninterrupted mode* for user to implement the counting scheme.

- In *one-shot* mode, the interrupt signal is generated once and it's not happen again unless the timer is re-enabled later.
- In *periodic* mode, the interrupt signal is generated periodically.
- Toggle mode
- Uninterrupted mode

Figure 12-3 shows the initialization sequence.

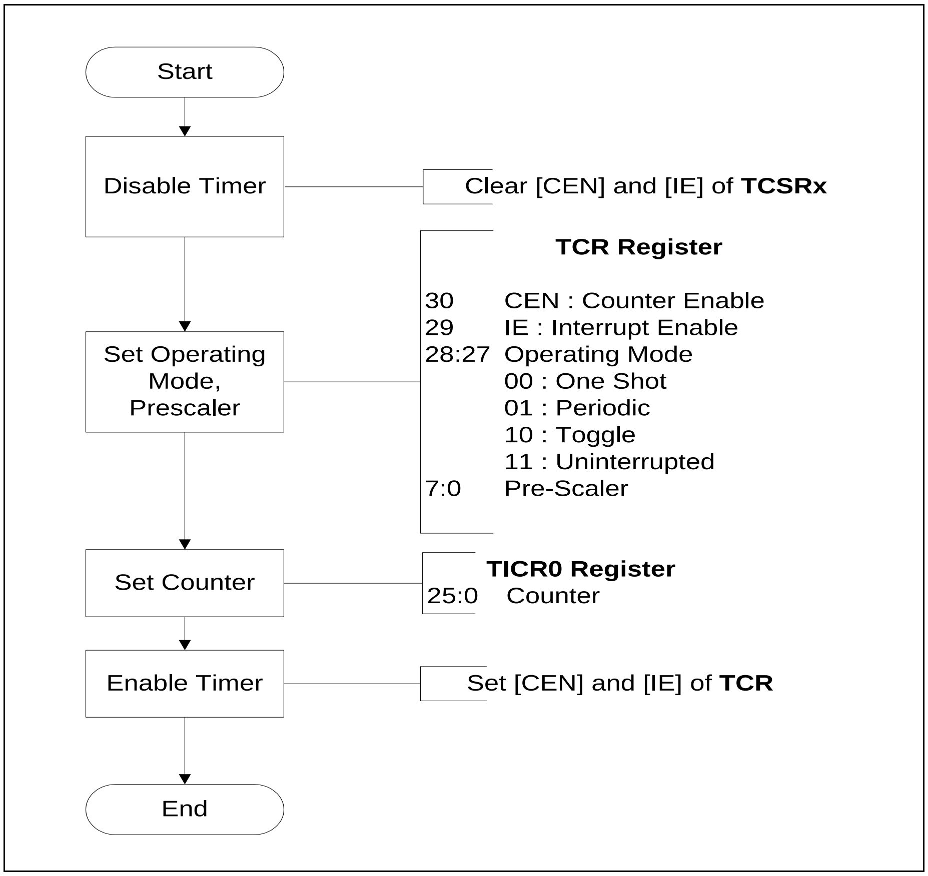


Figure 12-3 Timer Initialization Sequence

12.4.3. Timer Interrupt Service Routine

A common timer interrupt service routine is very simple. It increases the software counter and clears the timer interrupt status. Figure 12-4 shows the flow chart of such an interrupt service routine.

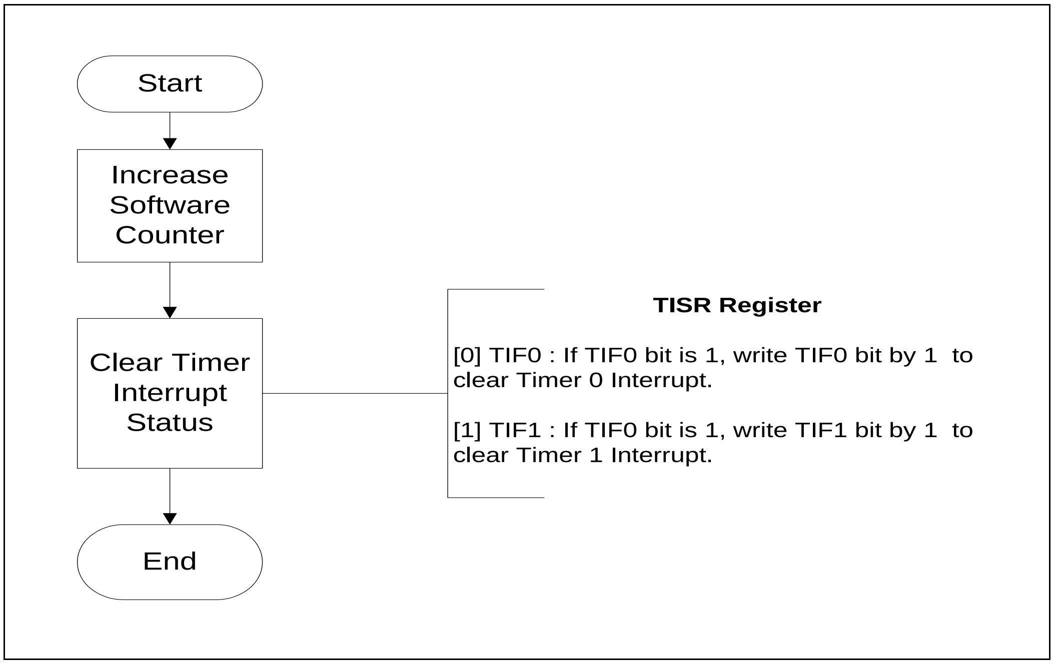


Figure 12-4 Timer Interrupt Service Routine

12.4.4. Watchdog Timer

The register WTCR is used to control watchdog timer. The bit WTR should be set before enable watchdog timer. It ensures that the watchdog timer restarts from a known state. Table 3 lists the Watchdog Timeout period. Figure 12-5 and Figure 12-6 illustrate how to use watchdog timer.

WTIS[5:4]	Interrupt Time-out	Reset Time-out	Actual time *WTCLK* = 1	Actual time *WTCLK* = 0
00	2^{14} clocks	2^{14} + 1024 clocks	0.371 sec	1.450 msec
01	2^{16} clocks	2^{16} + 1024 clocks	1.419 sec	5.546 msec
10	2^{18} clocks	2^{18} + 1024 clocks	5.614 sec	21.93 msec
11	2^{20} clocks	2^{20} + 1024 clocks	22.39 sec	87.46 msec

Table 3 WatchDog Timer Reset Time (Using 12MHz crystal)

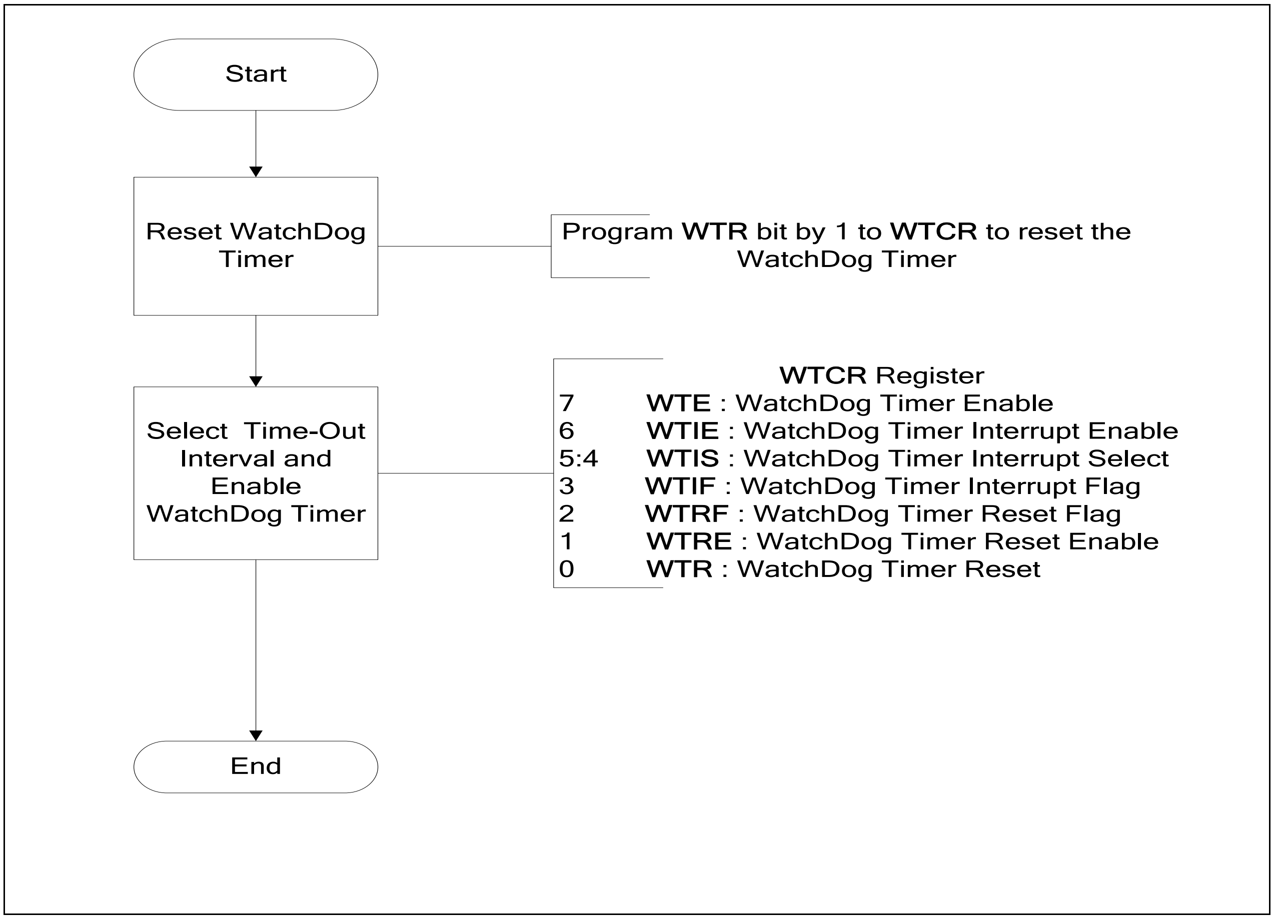


Figure 12-5 Enable Watchdog Timer

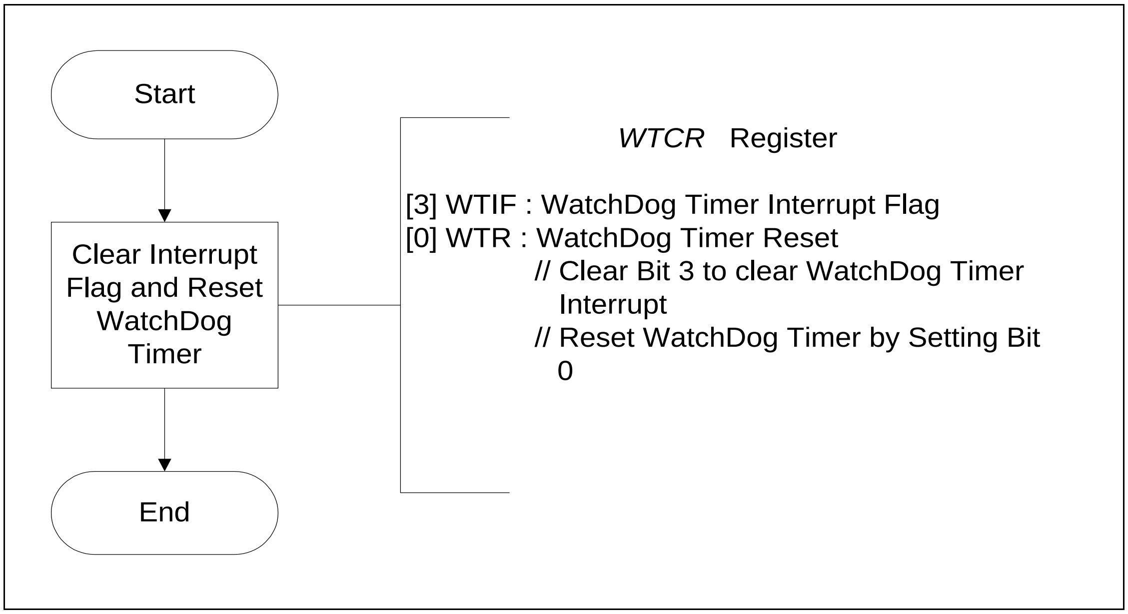


Figure 12-6 Watchdog Timer ISR

13. UART

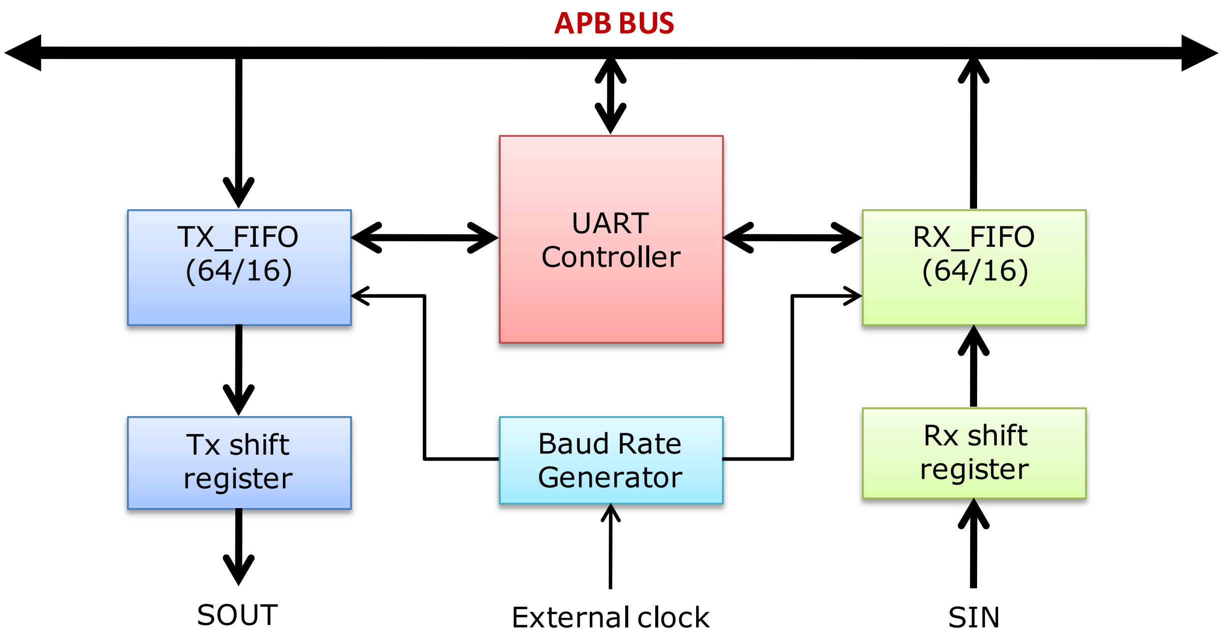
13.1. Overview

The UART interface controller module includes two channels, UART0~UART1. One of them is equipped with flow control function High Speed UART and the other is a Normal Speed UART. The Universal Asynchronous Receiver/Transmitter **(UART)** performs a serial-to-parallel conversion on data characters received from the peripheral, and a parallel-to-serial conversion on data characters received from the CPU. There are six types of interrupts, they are, transmitter FIFO empty interrupt**(Int_THRE)**, receiver threshold level reaching interrupt **(Int_RDA),** line status interrupt (overrun error or parity error or framing error or break interrupt) **(Int_RLS)** , time out interrupt **(Int_Tout)**, MODEM status interrupt **(Int_Modem)** and Wake up status interrupt **(Int_WakeUp).**
The two UART Interface Controller that one have a **64-byte** transmitter FIFO (TX_FIFO) and a **64-byte** (plus 3-bit of error data per byte) receiver FIFO (RX_FIFO) has been built in to reduce the number of interrupts presented to the CPU and the other have a **16-byte** transmitter FIFO (TX_FIFO) and a **16-byte** (plus 3-bit of error data per byte) receiver FIFO (RX_FIFO) has been built in to reduce the number of interrupts presented to the CPU. The CPU can completely read the status of the UART at any time during the operation. The reported status information includes the type and condition of the transfer operations being performed by the UART, as well as any error conditions (parity error, overrun error, framing error, or break interrupt) found. The UART includes a programmable baud rate generator that is capable of dividing crystal clock input by divisors to produce the clock that transmitter and receiver needed. The baud rate equation is **Baud Out = crystal clock / 16 * [Divisor + 2].**

The UART includes the following features:

- 64 byte/16 byte entry FIFOs for received and transmitted data payloads
- Flow control functions (CTS, RTS) are supported.
- Programmable baud-rate generator that allows the internal clock to be divided by 2 to (2^16 + 1) to generate an internal 16X clock.
- Fully programmable serial-interface characteristics:
  - 5-, 6-, 7-, or 8-bit character
  - Even, odd, or no-parity bit generation and detection
  - 1-, 1&1/2, or 2-stop bit generation
  - Baud rate generation
  - False start bit detection.
- Loop back mode for internal diagnostic testing

13.2. Block Diagram



13.3. Registers

R : Read only, **W** : Write only, **R/W** : Both read and write, **C** : Only value 0 can be written

UART_BA (UA_BA) = B800_C000
Channel0: UART_Base0 (High Speed) = B800_C000
Channel1: UART_Base1 (Normal Speed) = B800_C100

Register	Address	R/W	Description	Reset Value
UA_RBR	UA_BA + 0x00	R	Receive Buffer Register (DLAB = 0)	Undefined
UA_THR	UA_BA + 0x00	W	Transmit Holding Register (DLAB = 0)	Undefined
UA_IER	UA_BA + 0x04	R/W	Interrupt Enable Register (DLAB = 0)	0x0000_0000
UA_DLL	UA_BA + 0x00	R/W	Divisor Latch Register (LS) (DLAB = 1)	0x0000_0000
UA_DLM	UA_BA + 0x04	R/W	Divisor Latch Register (MS) (DLAB = 1)	0x0000_0000
UA_IIR	UA_BA + 0x08	R	Interrupt Identification Register	0x8181_8181
UA_FCR	UA_BA + 0x08	W	FIFO Control Register	Undefined
UA_LCR	UA_BA + 0x0C	R/W	Line Control Register	0x0000_0000
UA_MCR	UA_BA + 0x10	R/W	Modem Control Register	0x0000_0000

UA_LSR	UA_BA + 0x14	R	Line Status Register	0x6060_6060
UA_MSR	UA_BA + 0x18	R/W	Modem Status Register	0x0000_0000
UA_TOR	UA_BA + 0x1C	R/W	Time Out Register	0x0000_0000

13.4. Functional Description

13.4.1. Clock Source

The UART clock source can be External Crystal, PLL, and PLL/2. It can be set in the Clock Source Select Control Register.

Note: UART clock must slower than APB clock

13.4.2. Baud Rate

The UART includes a programmable baud rate generator. The crystal clock input is divided by divisor to produce the clock that transmitter and receiver need. The equation is

Baud Rate = APB clock / (16 * [Divisor + 2])

The UA_DLL and UA_DLM registers consist of the low byte and high byte of the divisor. The DLL and DLM registers aren't accessible until the DLAB bit of LCR register is set 1. The driver should program, the correct value into the UA_DLL/UA_DLM registers according to the desired baud rate. Table-4 lists some general baud rate settings.

Baud Rate	DLM	DLL	Real	Error rate [%]
115200	0	6	117187.5	1.725
57600	0	14	58593.75	1.725
38400	0	22	39062.5	1.725
19200	0	47	19132.65	-0.35
9600	0	96	9566.33	-0.35

Table-4 Baud rate sample

13.4.3. Initializations

Before the transfer operation starts, the serial interface of UART must be programmed. The driver should set the baud rate, parity bit, data bit and stop bit. If the transfer operation is done triggered by interrupt, the TX, RX and RLS interrupts need to be enabled. Figure 13-1 shows the initialization flow of UART.

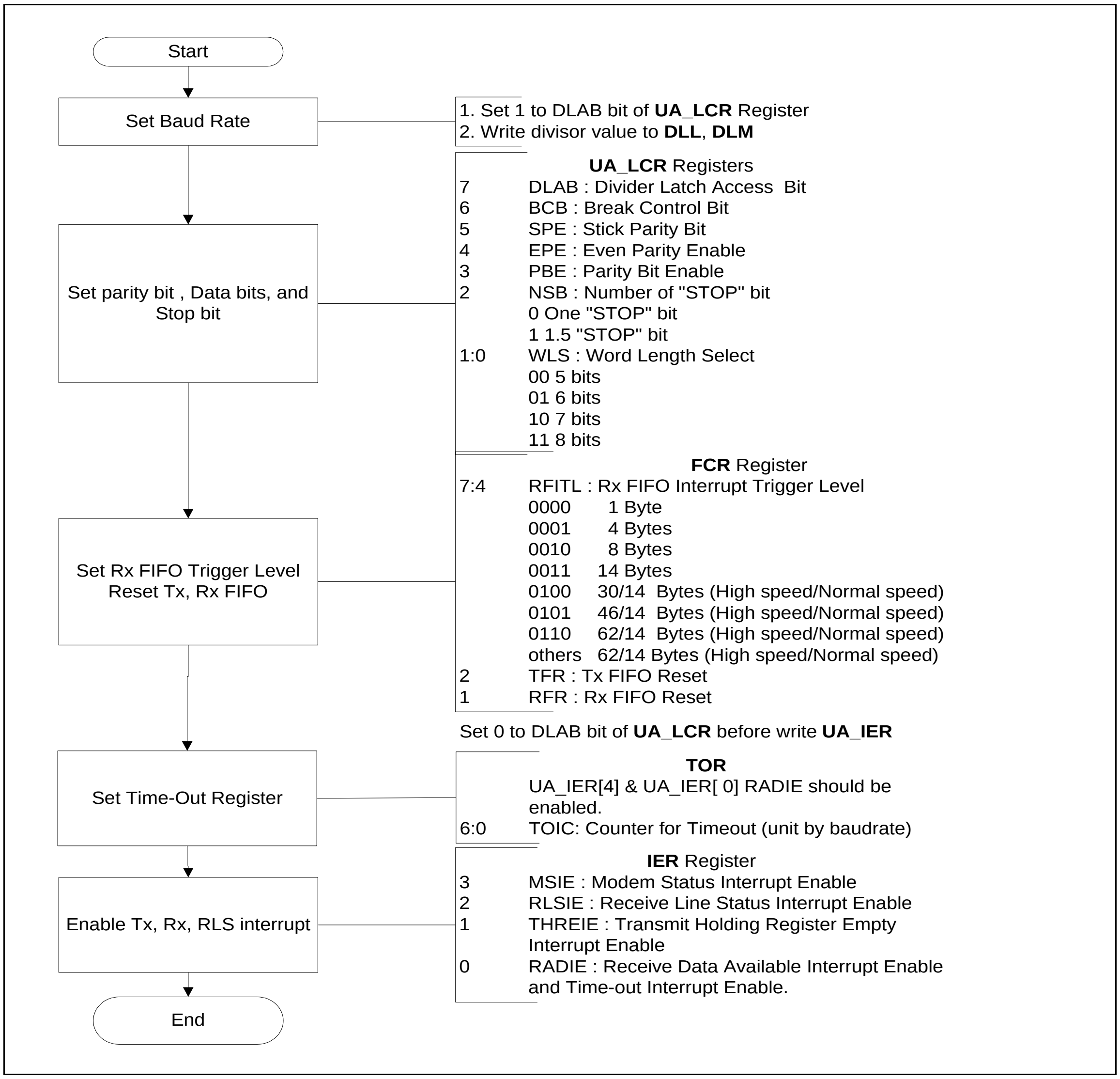


Figure 13-1 UART initialization

13.4.4. Polled I/O Functions

The driver can transmit and receive data through UART by polling mode. The poll functions check UART buffer by reading status register. If there's at least one data byte available in receive FIFO, the [RFDR] bit is set 1. It indicates that driver can read receive FIFO to get new data bytes. If the transmitter is empty, the [TE] bit is set 1. Then the data bytes can be written into the transmit FIFO. The data bytes in the transmit FIFO will be shifted to SOUT serially. Figure13-2 and Figure13-3 show the programming flow of transmit data and receive data in polling mode.

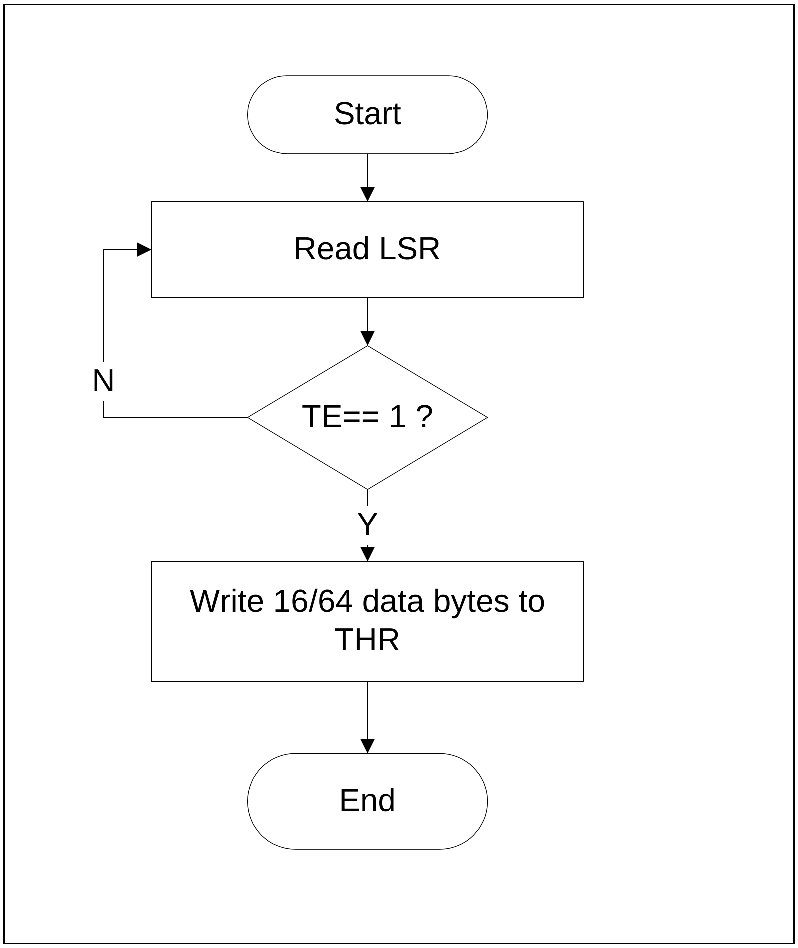


Figure 13-2 Transmit data in polling mode

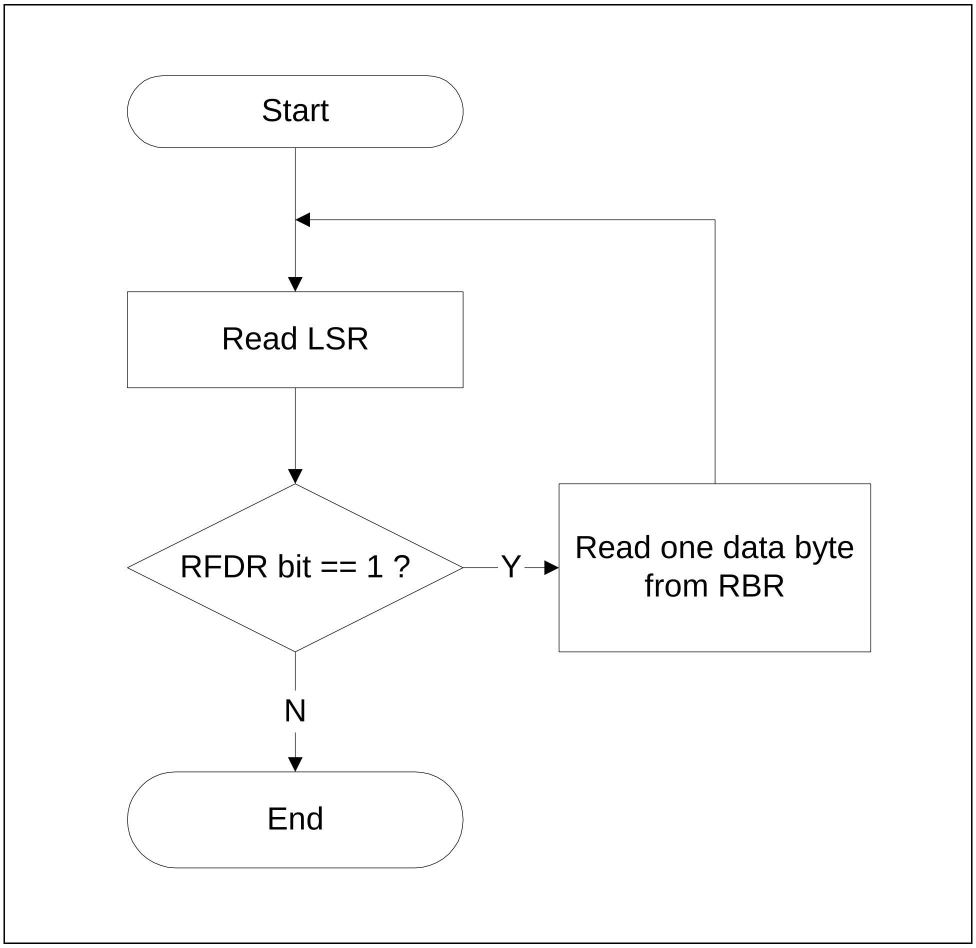


Figure 13-3 Receive data in polling mode

13.4.5. Interrupted I/O Functions

The data bytes also can be transmitted and received through UART by interrupt control. The interrupt service routine is responsible to move data bytes from driver's buffer to transmit FIFO whenever the THRE interrupt happens. If RDA or TOUT interrupts occurs, the interrupt service routine should move the data bytes from receive FIFO to driver's buffer.

In interrupt mode, the input and output functions are different from the polling functions. They read or write the driver's buffer instead of Tx /Rx FIFO. The output function writes the data bytes into driver's buffer and then enables THRE interrupt. The ISR will read the data bytes from driver's buffer and write them to the Tx FIFO when the transmitter FIFO empty interrupt occurs, or get the data bytes from Rx FIFO the driver receiving buffer when the *receiver threshold level reaching interrupt* occurs. When the input function is called, it reads the data bytes from driver's receiving buffer and then returns. Figure13-4, Figure13-5 and Figure13-6 show the flow of output function, input function, and interrupt service routine.

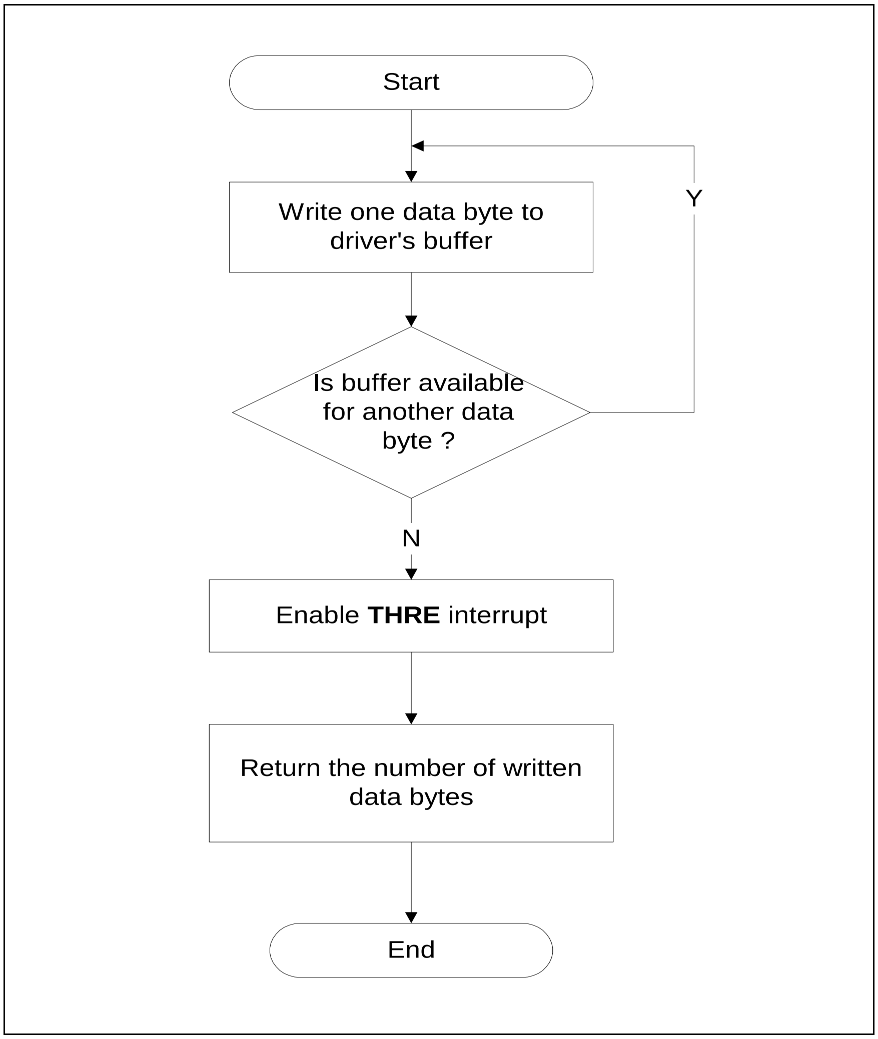


Figure 13-4 Output function in interrupt mode

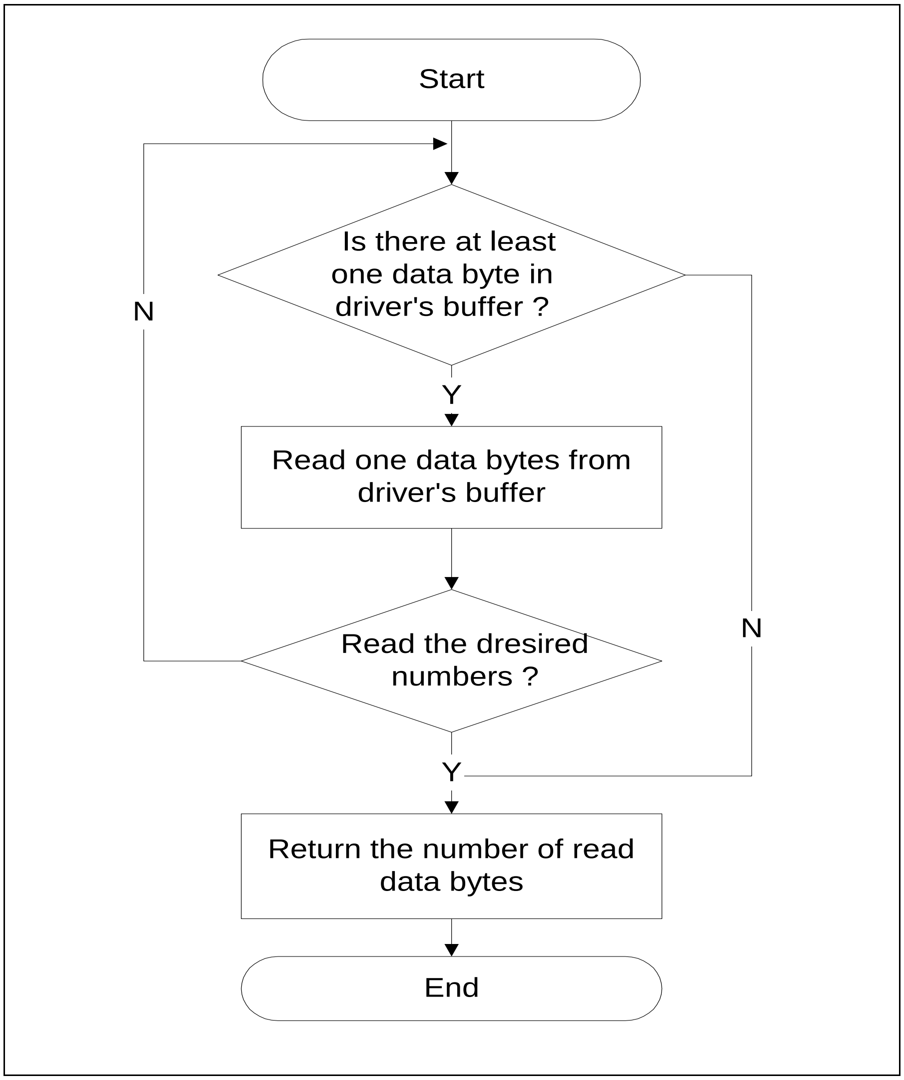


Figure 13-5 Input functions in interrupt mode

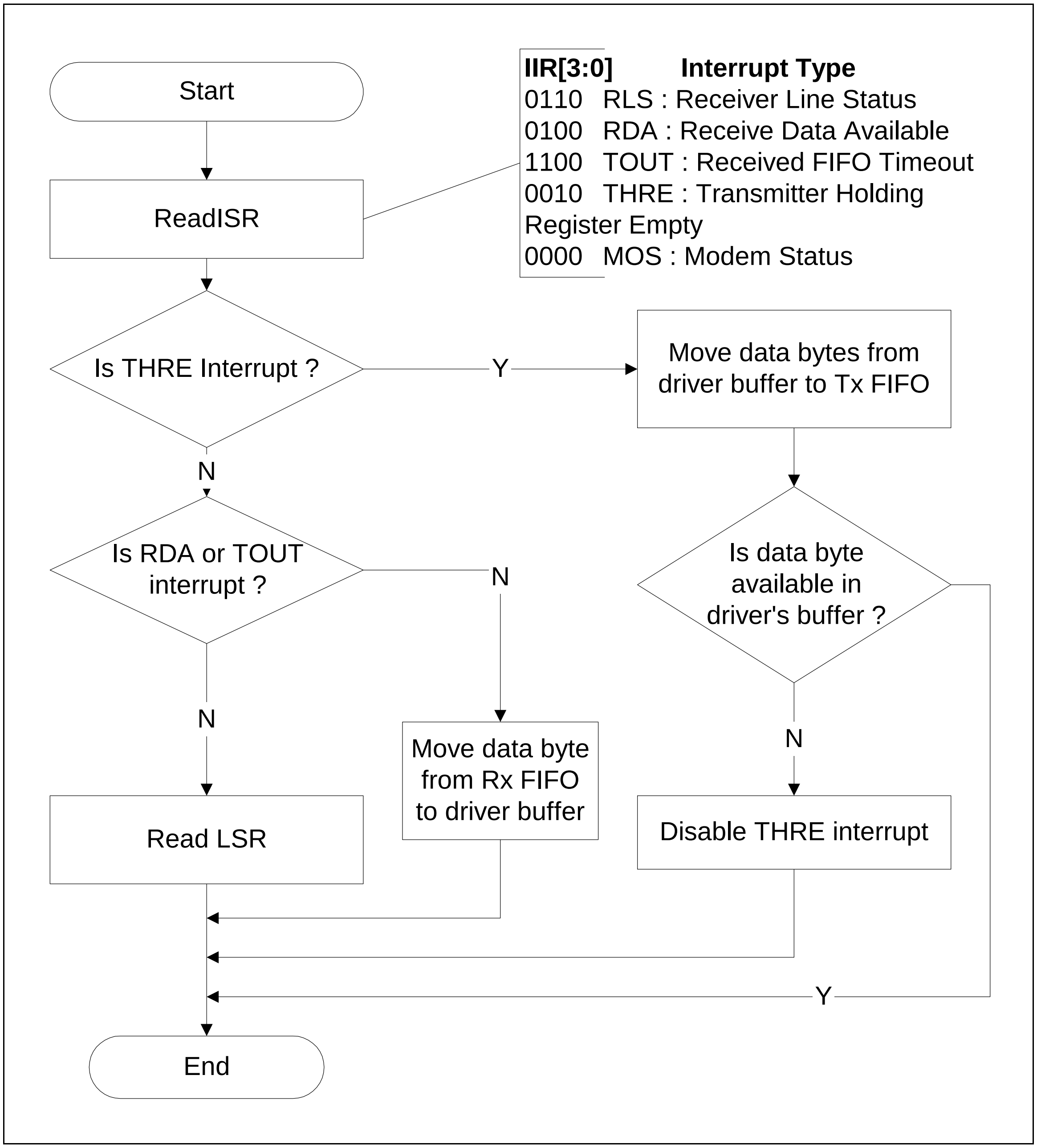


Figure 13-6 Interrupt Service Routine

14. USB

This document introduces how to properly program the USB device controller in this chips. In section 14.4, it takes a brief introduction of USB, and illustrates the fundamental flows in USB. Section 14.5 shows how to control the registers of the USB device controller to accomplish data transfer in USB.

14.1. Block Diagram

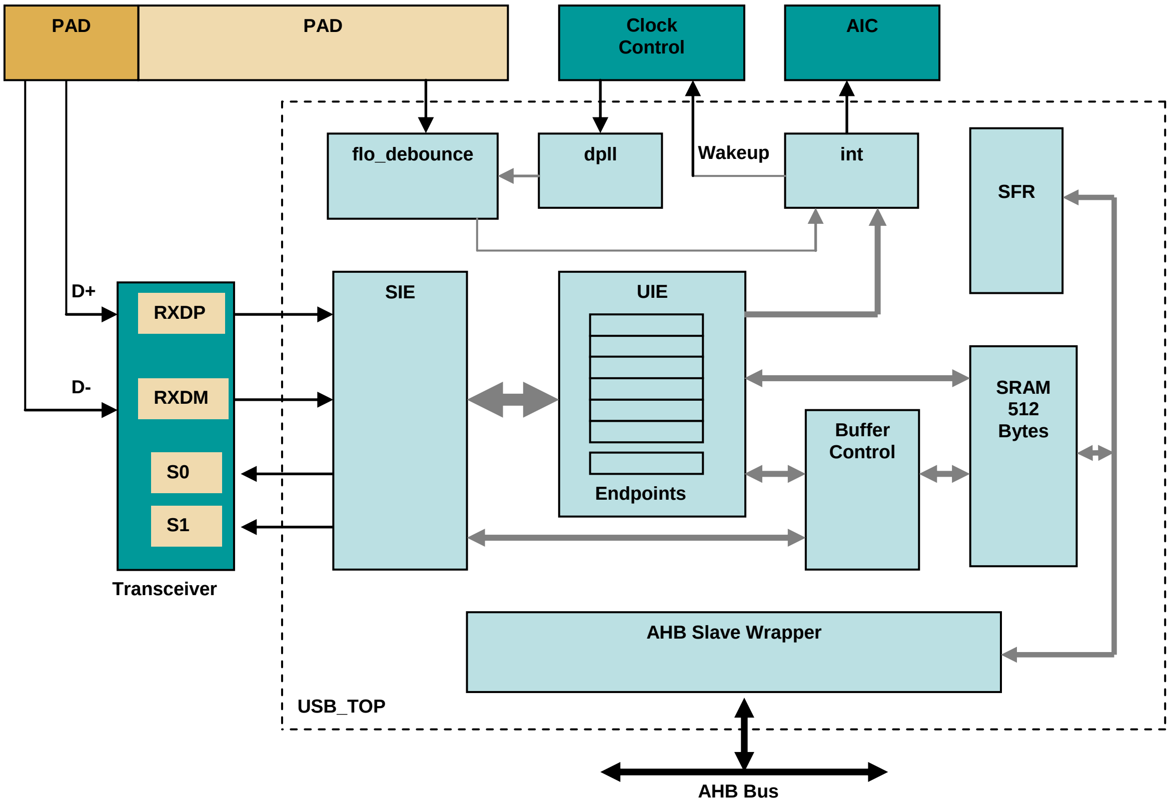


Figure 14-1 USB Block Diagram

14.2. Registers

Register	Address	R/W	Description	Reset Value
USB_BA = 0xB100_9000				

IEF	USB_BA+0x000	R/W	Interrupt Enable Flag	0x0000_0000
EVF	USB_BA+0x004	R	Interrupt Event Flag	0x0000_0000
FADDR	USB_BA+0x008	R/W	Function Address	0x0000_0000
STS	USB_BA+0x00C	R,/W	System state	0x0000_00x0
ATTR	USB_BA+0x010	R/W	Bus state & attribution	0x0000_0040
FLODET	USB_BA+0x014	R	Floating detect	0x0000_0000
BUFSEG	USB_BA+0x018	R/W	Buffer Segmentation	0x0000_0000
USBCFG	USB_BA+0x01C	R/W	USB configuration, internal test only	0x0000_0000
BUFSEG0	USB_BA+0x020	R/W	Buffer Segmentation of endpoint 0	0x0000_0000
MXPLD0	USB_BA+0x024	R/W	Maximal payload of endpoint 0	0x0000_0000
CFG0	USB_BA+0x028	R/W	Configuration of endpoint 0	0x0000_0000
CFGP0	USB_BA+0x02C	R/W	stall control register and In/out ready clear flag of endpoint 0	0x0000_0000
BUFSEG1	USB_BA+0x030	R/W	Buffer Segmentation of endpoint 1	0x0000_0000
MXPLD1	USB_BA+0x034	R/W	Maximal payload of endpoint 1	0x0000_0000
CFG1	USB_BA+0x038	R/W	Configuration of endpoint 1	0x0000_0000
CFGP1	USB_BA+0x03C	R/W	stall control register and In/out ready clear flag of endpoint 1	0x0000_0000
BUFSEG2	USB_BA+0x040	R/W	Buffer Segmentation of endpoint 2	0x0000_0000
MXPLD2	USB_BA+0x044	R/W	Maximal payload of endpoint 2	0x0000_0000
CFG2	USB_BA+0x048	R/W	Configuration of endpoint 2	0x0000_0000
CFGP2	USB_BA+0x04C	R/W	stall control register and In/out ready clear flag of endpoint 2	0x0000_0000
BUFSEG3	USB_BA+0x050	R/W	Buffer Segmentation of endpoint 3	0x0000_0000
MXPLD3	USB_BA+0x054	R/W	Maximal payload of endpoint 3	0x0000_0000
CFG3	USB_BA+0x058	R/W	Configuration of endpoint 3	0x0000_0000
CFGP3	USB_BA+0x05C	R/W	stall control register and In/out ready clear flag of endpoint 3	0x0000_0000
BUFSEG4	USB_BA+0x060	R/W	Buffer Segmentation of endpoint 4	0x0000_0000
MXPLD4	USB_BA+0x064	R/W	Maximal payload of endpoint 4	0x0000_0000
CFG4	USB_BA+0x068	R/W	Configuration of endpoint 4	0x0000_0000
CFGP4	USB_BA+0x06C	R/W	stall control register and In/out ready clear flag of endpoint 4	0x0000_0000
BUFSEG5	USB_BA+0x070	R/W	Buffer Segmentation of endpoint 5	0x0000_0000
MXPLD5	USB_BA+0x074	R/W	Maximal payload of endpoint 5	0x0000_0000

CFG5	USB_BA+0x078	R/W	Configuration of endpoint 5	0x0000_0000
CFGP5	USB_BA+0x07C	R/W	In ready clear flag of endpoint 5	0x0000_0000
USBBIST	USB_BA+0x0A0	R/W	USB buffer test register	0x0000_0000

14.3. Introduction to USB

There are four types of pipes in USB, and each of them defines a specified purpose to carry data on USB. They are Control Pipe, Bulk Pile, Isochronous Pipe and Interrupt Pipe. Control Pipe is the default pipe to be established once the USB is connected. It provides the service for USB host to retrieve the basic information of an USB device. We take "Get Device Descriptor" as an example. This operation can be divided into three stages. The first is setup stage that host sends setup command to a device. The second is data stage that host receive data from a device. The third is status stage that host send ACK to device to complete the operation. The following figures help reader to understand the concept.

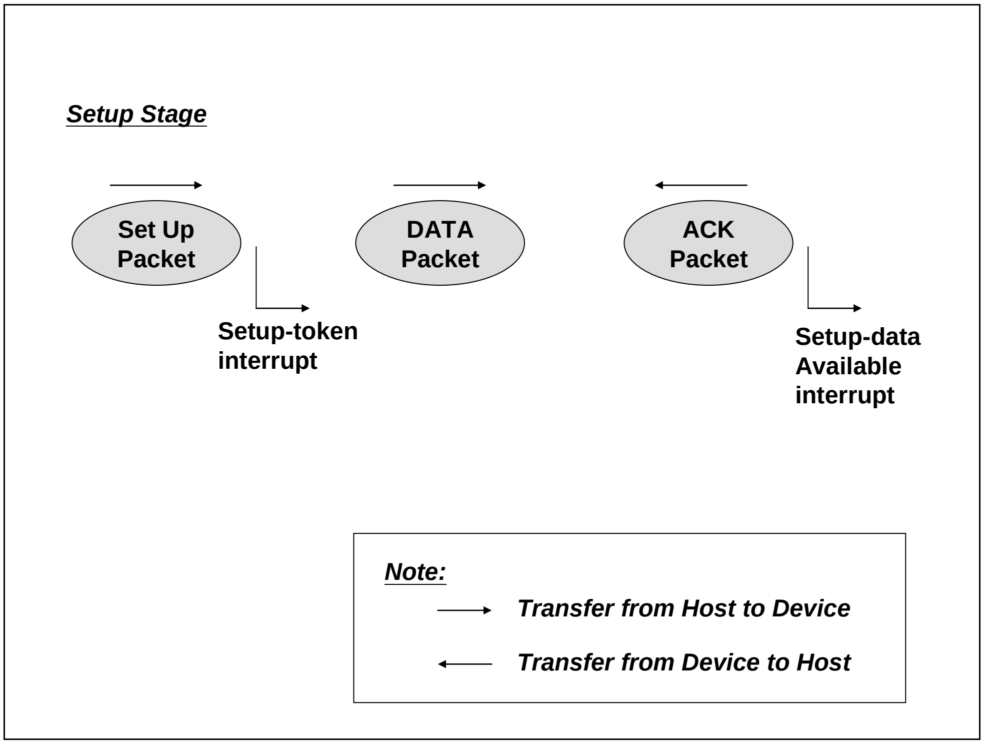


Figure 14-2   DATA Packet above represents the command of this stage.

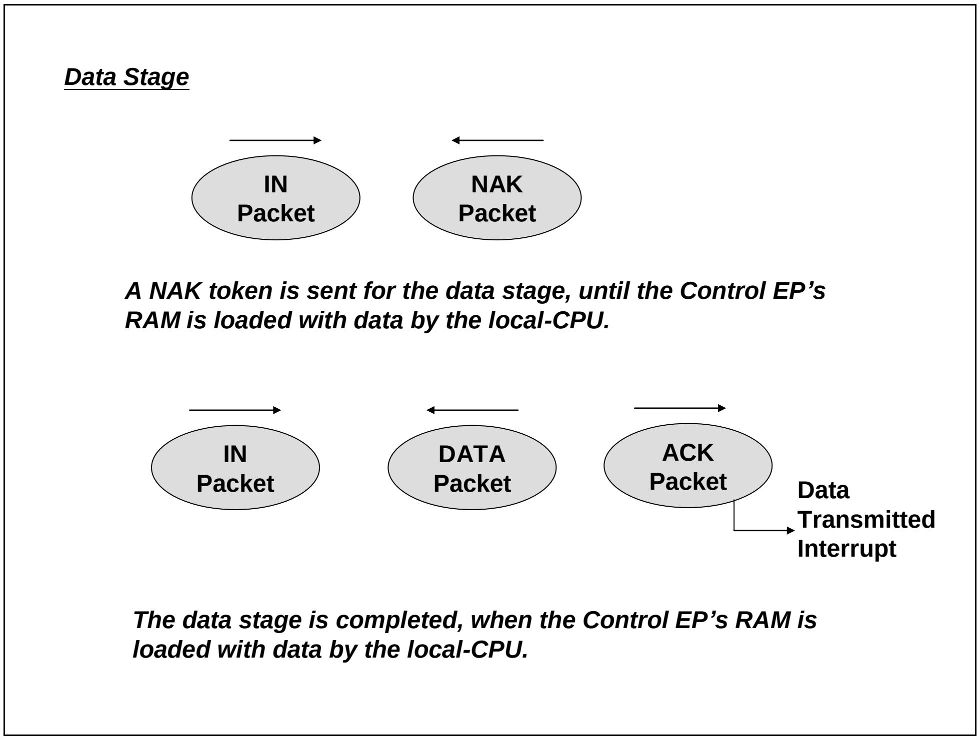


Figure 14-3 Data Packet above represents the device information.

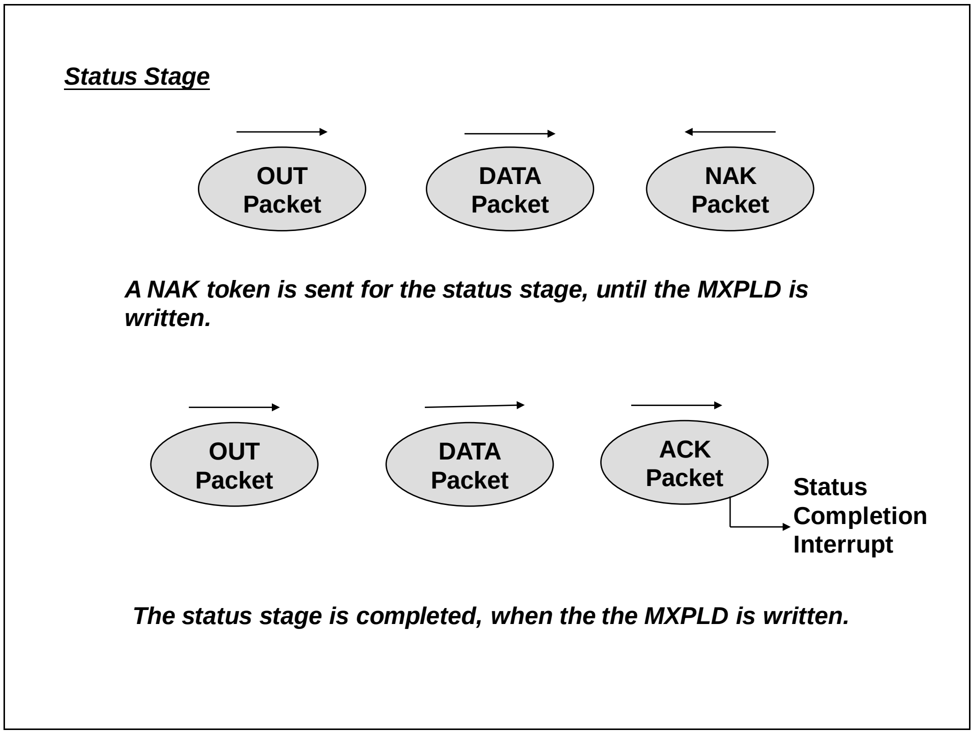


Figure 14-4 Data Packet above usually is zero-packet which represents ACK.

As to other pipes, they only have data stage. Bulk Pipe and Interrupt Pipe have ACK or NACK Packet, but Isochronous does not have.

14.4. Function Descriptions

14.4.1. Registers Programming

This section shows readers to program registers to operate the USB device controller properly. Readers can know how to initialize the USB device controller and how to transfer data through the USB device controller.

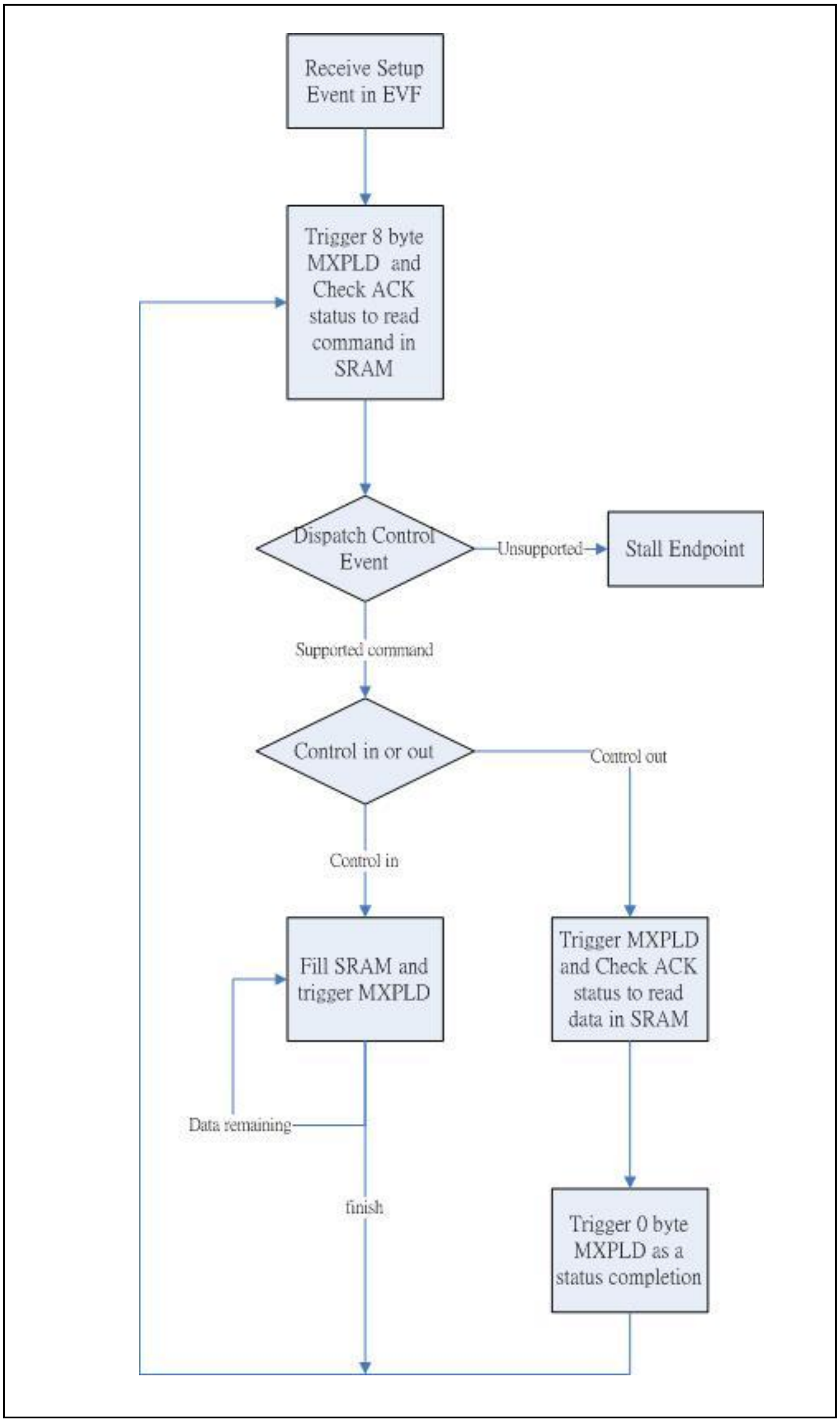
14.4.2. Initialization

- Give endpoints with proper settings such as endpoint number, endpoint direction and isochronous bit.
- Allocate buffer for endpoints.
- Set device address with initial value zero.

14.4.3. Control Transfer

- Allocate buffer for setup packet (BUFSEG, 0x18).
- Allocate buffer for Control-In and Control-Out endpoints (BUFSEG0 ~ BUFSEG5)
- Parse Setup Packet (Setup in EVF asserted) from allocated buffer
- If it is a control out command, fill MXPLDx to retrieve data from buffer of the endpoint.
- If it is a control in command, fill data into buffer and write MXPLDx with the size of data.

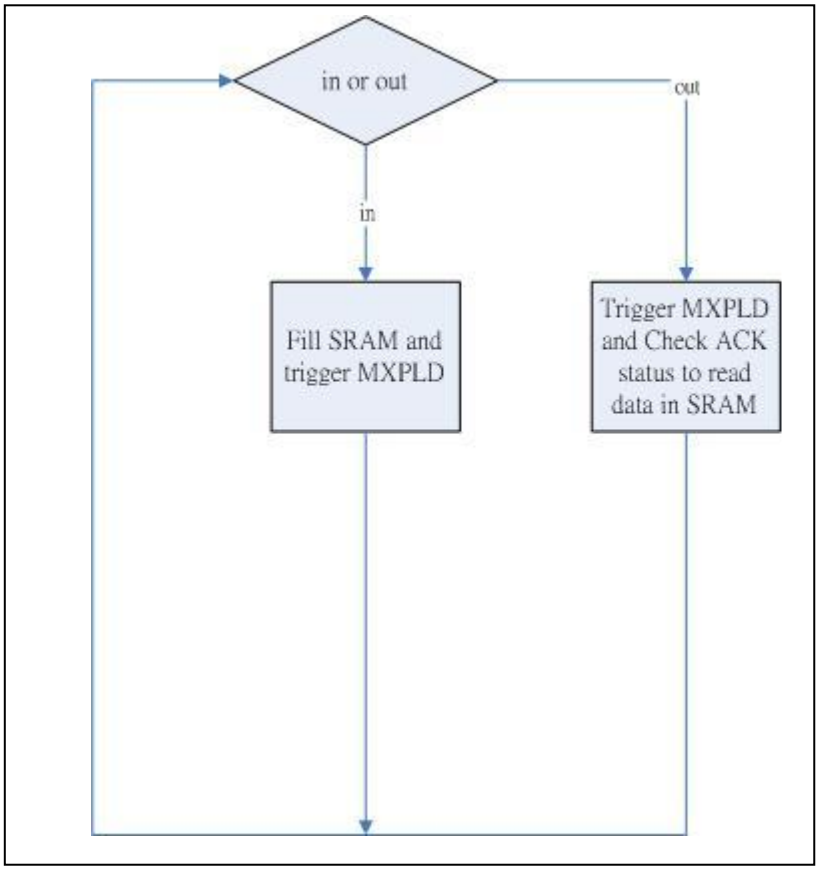
Here is an illustration of control flow.


Receive Setup Event in EVF
Trigger 8 byte MXPLD and Check ACK status to read command in SRAM
Dispatch Control Event
Unsupported
Stall Endpoint
Supported command
Control in or out
Control out
Control in
Fill SRAM and trigger MXPLD
Trigger MXPLD and Check ACK status to read data in SRAM
Data remaining
finish
Trigger 0 byte MXPLD as a status completion

14.4.4. Others' Transfer

- Fill MXPLDx to retrieve data from buffer of the endpoint for data out.
- Fill data into buffer and write MXPLDx with the size of data for data in.

An illustration is shown as follows.



14.5. Code Section

14.5.1. Initialization

```
void UsbIbrCfg (void)
{
    outp8 (FADDR, 0x00);
    outp16 (BUFSEG, BUF_SETUP);     // Setup
    outp8 (CFGP0, EPT_CFGP);   // EP0 CTRL IN
    outp16 (CFG0, CFG0_SETTING);
```

```
    outp16 (BUFSEG0, BUF0_SETTING);
    outp8 (CFGP1, EPT_CFGP);   // EP0 CTRL OUT
    outp16 (CFG1, CFG1_SETTING);
    outp16 (BUFSEG1, BUF1_SETTING);
    outp8 (CFGP2, EPT_CFGP);   // EP2 BULK IN
    outp16 (CFG2, CFG2_SETTING);
    outp16 (BUFSEG2, BUF_BULK1);
    outp8 (CFGP3, EPT_CFGP);   // EP3 BULK OUT
    outp16 (CFG3, CFG3_SETTING);
    outp16 (BUFSEG3, BUF_BULK0);
}
```

14.5.2. Control Transfer - Get Descriptor

```
// Get Descriptor
        case GET_DESCRIPTOR:
        {
            g_u16Len = g_au8UsbSetup[6] + (g_au8UsbSetup[7] << 8);
            switch (g_au8UsbSetup[3])
            {
                // Get Device Descriptor
                case DESC_DEVICE:
                {
                    g_u16Len = Minimum (g_u16Len, LEN_DEVICE);
                    for (g_u8i = 0; g_u8i < g_u16Len; g_u8i++)
                        g_au8UsbCtrl[g_u8i] = c_au8DeviceDesc[g_u8i];
                    break;
                }
                // Get Configuration Descriptor
                case DESC_CONFIG:
                {
                    g_u16Len = Minimum (g_u16Len, c_au8ConfigDesc[2]);
                    for (g_u8i = 0; g_u8i < g_u16Len; g_u8i++)
                        g_au8UsbCtrl[g_u8i] = c_au8ConfigDesc[g_u8i];
                    break;
                }
```

```
                    // Get String Descriptor
                    case DESC_STRING:
                    {
                         // Get Language
                         if (g_au8UsbSetup[4] == 0)
                         {
                              g_u16Len = Minimum (g_u16Len, c_au8StringLang[0]);
                              for (g_u8i = 0; g_u8i < g_u16Len; g_u8i++)
                                   g_au8UsbCtrl[g_u8i] = c_au8StringLang[g_u8i];
                              break;
                         }
                         // Get String Descriptor
                         g_u16Len = Minimum (g_u16Len,
c_pau8String[g_au8UsbSetup[2]][0]);
                         for (g_u8i = 0; g_u8i < g_u16Len; g_u8i++)
                              g_au8UsbCtrl[g_u8i] =
c_pau8String[g_au8UsbSetup[2]][g_u8i];
                         break;
                    }
                    default:
                         return FALSE;
               }
               outp16 (CFG0, DATA1 (CFG0_SETTING));
               outp16 (MXPLD0, g_u16Len);
               g_u8Flag = FLAG_OUT_ACK;
```

14.5.3. Bulk Out

```
void UsbIbrWrite (void)
{
     if (g_u32Length > MAX_PACKET_SIZE)
     {
          if (inp16 (BUFSEG3) == BUF_BULK0)
          {
               outp16 (BUFSEG3, BUF_BULK1);
               outp16 (MXPLD3, MAX_PACKET_SIZE);
               for (g_u8i = 0; g_u8i < MAX_PACKET_SIZE; g_u8i++)
```

```
                    *(UINT8 *)(g_u32Address + g_u8i) = g_au8UsbBulk0[g_u8i];
            }
            else
            {
                outp16 (BUFSEG3, BUF_BULK0);
                outp16 (MXPLD3, MAX_PACKET_SIZE);
                for (g_u8i = 0; g_u8i < MAX_PACKET_SIZE; g_u8i++)
                    *(UINT8 *)(g_u32Address + g_u8i) = g_au8UsbBulk1[g_u8i];
            }
            g_u32Address += MAX_PACKET_SIZE;
            g_u32Length -= MAX_PACKET_SIZE;
        }
        else
        {
            if (inp16 (BUFSEG3) == BUF_BULK0)
            {
                for (g_u8i = 0; g_u8i < g_u32Length; g_u8i++)
                    *(UINT8 *)(g_u32Address + g_u8i) = g_au8UsbBulk0[g_u8i];
            }
            else
            {
                for (g_u8i = 0; g_u8i < g_u32Length; g_u8i++)
                    *(UINT8 *)(g_u32Address + g_u8i) = g_au8UsbBulk1[g_u8i];
            }
            g_u32Address += g_u32Length;
            g_u32Length = 0;
            g_u8BulkState = BULK_IN;
        }
}
```

15. Revision History

Version	Date	Description
V1.00	Feb. 20, 2009	● Created

Important Notice

Nuvoton products are not designed, intended, authorized or warranted for use as components in equipment or systems intended for surgical implantation, atomic energy control instruments, aircraft or spacecraft instruments, transportation instruments, traffic signal instruments, combustion control instruments, or for any other applications intended to support or sustain life. Furthermore, Nuvoton products are not intended for applications whereby failure could result or lead to personal injury, death or severe property or environmental damage.

Nuvoton customers using or selling these products for such applications do so at their own risk and agree to fully indemnify Nuvoton for any damages resulting from their improper use or sales.