Autonomous environmental monitoring probe for aquaculture sites

alireza ramezaniakhmareh

Embedded Computing Systems
Submission date: December 2012
Supervisor: Jo Arve Alfredsen, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Norwegian University of Science and Technology

Department of Engineering Cybernetics

Master of Science in Electronic Engineering

Embedded Systems

MS Thesis Report

Alireza Ramezani

Version : 0.1 rev3

Build Date : November 30, 2012

Abstract

Proper knowledge of the composition and quality of the local underwater environment is very important for the management of sea-based aquaculture farms. This project aims to develop an instrument that is capable of moving vertically in a controlled manner along a taut string from the sea surface to a predetermined depth while measuring pertinent seawater parameters. The main focus will be to specify, design and implement an embedded computer system that realizes the instrument.

Contents

List of Figures

List of Tables

1 Project Specification

1.1 Basics about water analysis

Salinity - is a measure of the total concentration of all ions found in water. Seven ions make up the bulk of a salinity reading. These are sodium, potassium, calcium, magnesium, chloride, sulfate, and bicarbonate. Salinity is typically measured in parts per thousand (ppt). Full-strength seawater is typically around 35 ppt. Estuarine water is classified as having 3 ppt while freshwater has less than 0.3 ppt. Any water having more than 1 ppt salinity can be expected to have a salty taste. Penaeid shrimp generally exhibit ideal growth at salinity values between 15-25 ppt, but it is a highly adaptable species and has been grown in commercial settings at salinities ranging from 2-38 ppt. Tilapias show varying degrees of salinity tolerance depending on species but all can tolerate salinities lower than 5 ppt.

Potassium - concentrations in water tend to be highly variable and tend to vary with salinity. The concentration of potassium in seawater is normally around 380 ppm.

pH - values between 7.0 and 8.0 are generally considered ideal for aquaculture.

Temperature - the ideal water temperature for the culture of tropical fishes and shrimps is between 82F to 86F.

Total Hardness, Calcium & Magnesium - total hardness is the concentration of calcium and magnesium in water expressed in milligrams per liter (mg/l) of equivalent calcium carbonate ($CaCO_3$). Water having total hardness values of 0-75 mg/l is generally classified as soft water, while values above 150 mg/l is categorized as hard water. Full strength seawater has a total hardness of 6,600 mg/l. Hardness is important to crustaceans because they have heavily mineralized exoskeletons and it is often thought that low hardness levels may limit their growth (Greenway, 1974). The lower limit for total hardness for the proper development of crustacean's exoskeletons is believed to be 50 mg/l (Boyd, 1990).

Phosphorous - is perhaps the most important nutrient influencing the natural productivity of aquatic systems. Phosphorous is critical for proper development of phytoplankton and phytoplankton is one of the basic building blocks of aquatic productivity. Estuarine areas typically have much higher concentrations of phosphorous in the water and this is why they are so productive.

Open ocean environments typically have very low concentrations of phosphorous and productivity is generally low. Concentration of phosphorous in pure seawater is usually around 0.07 mg/l.

Total Ammonia Nitrogen (TAN) - TAN consists of two fractions, un-ionized ammonia (NH3) and ionized ammonia (NH4+) and is the by-product of protein metabolism. TAN is excreted from the gills of fish as they assimilate feed and is produced when bacteria decompose organic waste solids within the culture system. The un-ionized form of ammonia-nitrogen is extremely toxic to fish. The fraction of TAN in the un-ionized form is dependent upon the pH and temperature of the water. At a pH of 7.0, most of the TAN is in the ionized form, while at a pH of 8.0 the majority is in the un-ionized form, While the lethal concentration of ammonia-nitrogen for many species has been established, the sublethal effects of ammonia-nitrogen have not been well defined. Reduction in growth rates may be the most important sublethal effect. In general, the concentration of unionized ammonia-nitrogen should not exceed 0.05 mg/1. Nitrite-nitrogen (N02-) - is a product of the oxidation of ammonia-nitrogen. Vitrifying bacteria (Nitosomonas) in the production system utilize ammonia-nitrogen as an energy source for growth and produce nitrite-nitrogen as a by-product. These bacteria are the basis for biological filtration. The vitrifying bacteria grow on the surface of the biofilter substrate and to some extent on all production system components including pipes, valves, tank walls, etc. While nitrite-nitrogen is not as toxic as ammonia-nitrogen, it is harmful to aquatic species and must be removed from the system. Concentrations of nitrite-nitrogen should not exceed 0.5 mg/l for long periods of time.

Nitrate-nitrogen (NO3) - fortunately, Nitrobacter bacteria, which are also present in most biological filters, utilize nitrite-nitrogen as an energy source and produce nitrate as a by-product. Nitrates are not generally of great concern to the aquaculturist. Studies have shown that aquatic species can tolerate extremely high levels (greater than 100 mg/l) of nitrate-nitrogen in production systems. Nitrate-nitrogen is either flushed from a system during system maintenance operations (such as settled solids removal or filter backwashing) or denitrification occurs within a treatment system component such as a settling tank. Denitrification is mainly due to the metabolism of nitrate-nitrogen by anaerobic bacteria producing nitrogen gas that is released to the atmosphere during aeration processes.

Sulfate - the most common form of sulfur in seawater is sulfate. Concentrations can vary with the nature of the geological materials in the watershed and with hydrological conditions. Pure seawater normally contains around

885 mg/l sulfate.

Heavy Metals - according to the recommendations of the United States Environmental Protection Agency (EPA), safe levels of cadmium, chromium, copper, lead, and zinc are 10, 100, 25, 100, and 100 micrograms/liter (ug/L) respectively. Most of these metals form a dietary requirement at very low levels (less than 10 ug/L) while being toxic at high concentrations.

1.2 Environmental Parameters Measurement Requirements

Temperature: the Probe should be able to measure the temperature range from -2° to +30° C. Because, water temperature usually ranges from 1° to 30°C, during the year in the Trondheim fjord. Also, the required accuracy for mentioned measurement is about ± 0.1°C.

Salinity: Salinity usually ranges from 0.5ppt to 35ppt(case study). As a result, the sensor which is used in the probe should be capable of measuring this range of salinity and accurate enough for a valid data to be obtained. And, the resolution acceptable for this experiment would be 1ppt.

Depth: It should be measured continuously(non stopping measurement-free running) to calculate at which depth the probe is at each time instant. And, requested Resolution shall be 0.1m with the accuracy of ±0.1m. Worth mentioning that the maximum depth would be around 50 meters.

1.3 Deployment time and power consumption

The power consumption of the system should be as low as possible, because the device is powered with the batteries which cannot persist much operation time if the power consumption is not regulated properly. And, because of difficulty in accessing the probe for battery replacement, it would be designed to work for at least 3 months without the need for battery recharge or replacement.

2 System Level Design

The main goal of this phase is to identify the systems main functional modules (modularization) and define the logical interfaces between them.

2.1 Systems Main Functional Modules

2.1.1 Microcontroller

As the main controller of our system, we have considered a 32-bit AVR microcontroller which has some good features such low power consumption, high performance, and low cost that are three important characteristics of Embedded Systems. By using an 32-bit family of AVRs, we could have a system capable of running a realtime operating system (such as Embedded linux). For this project, we have implemented an embedded system based on FreeRTOS which has a realtime kernel and a small code size suitable for embedded applications.

2.1.2 Stepper Motor

As we should have a strategy for making the probe to move along a taut string, we have made use of a stepper motor which task is controlling the vertical motion. Its worth mentioning that implementing the task for motor control would be out of this project focus.

2.1.3 Power-on indicator

Our probe needs to signal the power-on state to the user. For this purpose, we have considered a LED as a power-on indicator.

2.1.4 GSM/GPRS Module

There should be a module through which the Microcontroller communicates and sends the data to GSM network. So, we have assigned this task to a GSM/GPRS module. This module should have the capability to send the gathered data as a text message to a predefined GSM network.

2.1.5 External Non Volatile Memory

By considering the internal memory of microcontrollers usually is not sufficient for storing large amount of data (maximum few hundred Kb), we have to think about having another high capacity external memory for storing our samples which are crucial for future data analysis.

2.1.6 Power Supply

One of main characteristics of Embedded systems is being low power. As a result, we have considered some facilities to reduce the power consumption, and make use of a number of normal AA batteries as our system's power supply.

2.1.7 Digital Sensor Module

In this project we have considered using a digital sensor module with RS232 interface, with which the microcontroller could communicate and receive the sensor acquired data fast and efficient. This sensor measures water quality factors such as temperature, depth, and salinity which are the main characteristics of water quality measurement.

2.2 Required Communication Interfaces

RS232 communication port: the microcontroller needs a port for communication with both sensor module and GSM module. RS232 is a simple and user-friendly communication port through which we send and receive data to and from sensor module easily and fast. By considering this fact that both AVR controller and GSM module have the same working voltage range, but AVR controller and sensor module have different voltage levels, we need a voltage convertor between AVR and sensor module. By having so, we have the proper communication interface between main system modules.

Debug Interface: for the testing and debug purposes, there should be a J-tag interface through which the maintenance staff could test the system while required. And, if it is necessary, upgrade the firmware of the MCU through J-Tag interface.

GSM/GPRS Network Connection: By having the monitoring probe connected to a GSM network, we could send the gathered data to devices which are connected to that network either(a cell phone, for example).

USB communication port: Taking into account that the GSM network connection might fail because of whichever reason, there should be an alternative way for monitoring the environment which would be through a USB interface. By making available a USB communication port, we could guarantee the constant observation with a PC having minimum hardware and software requirements.

3 Module Level Design

The main goal of this section is to make Further modularization and detailed specification of each hardware and software component providing the required basis/documentation for implementation.

3.1 Microcontroller's Peripherials

Here we list the peripherals that have been used for this project, and their related specific characteristics.

3.1.1 General-Purpose Input/Output Controller(GPIO)

Features Each I/O line of the GPIO features:

• Configurable pin-change, rising-edge or falling-edge interrupt on any I/O line

• A glitch filter providing rejection of pulses shorter than one clock cycle

• Input visibility and output control

• Multiplexing of up to four peripheral functions per I/O line

• Programmable internal pull-up resistor

Overview The General Purpose Input/Output Controller manages the I/O pins of the microcontroller. Each I/O line may be dedicated as a general-purpose I/O or be assigned to a function of an embedded peripheral. This assures effective optimization of the pins of a product.

Block Diagram Here is the block diagram of GPIO controller:

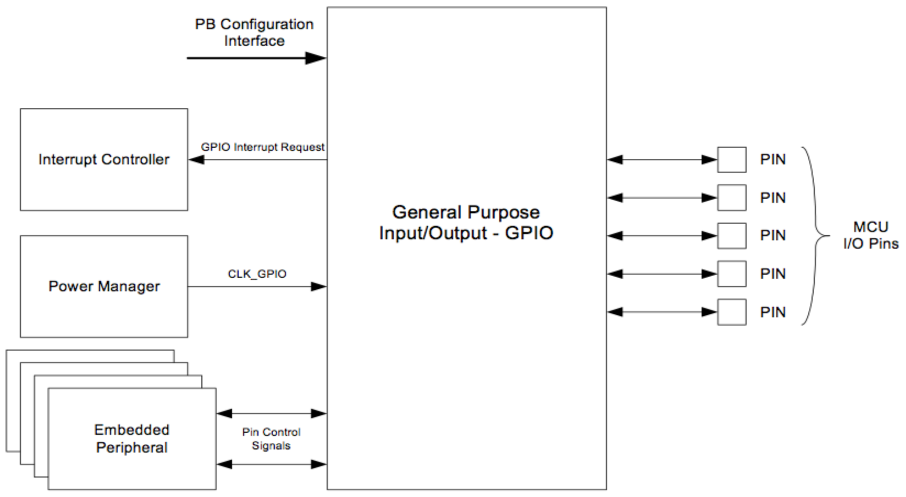


Figure 1: GPIO Block Diagram

Product Dependencies In order to use this module, other parts of the system must be configured correctly, as described below.

Module Configuration Most of the features of the GPIO are configurable for each product.

Product specific settings includes:

- Number of I/O pins.
- Functions implemented on each pin
- Peripheral function(s) multiplexed on each I/O pin
- Reset value of registers

Clocks The clock for the GPIO bus interface (CLK_GPIO) is generated by the Power Manager. This clock is enabled at reset, and can be disabled in the Power Manager.

The CLK_GPIO must be enabled in order to access the configuration registers of the GPIO or to use the GPIO interrupts. After configuring the GPIO, the CLK_GPIO can be disabled if interrupts are not used.

Interrupts The GPIO interrupt lines are connected to the interrupt controller. Using the GPIO interrupt requires the interrupt controller to be configured first.

Functional Description The GPIO controls the I/O lines of the microcontroller. The control logic associated with each pin is represented in the figure below:

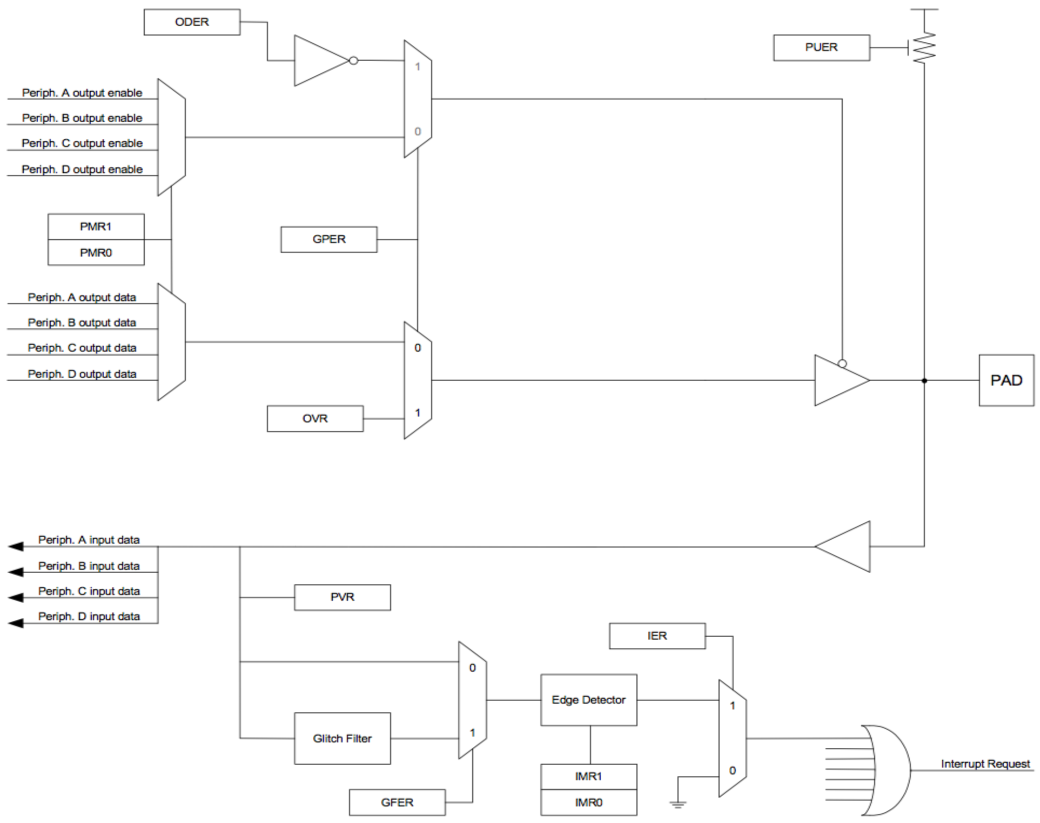


Figure 2: Overview of the GPIO Pad Connections

Basic Operation

I/O Line or peripheral function selection When a pin is multiplexed with one or more peripheral functions, the selection is controlled with the GPIO Enable Register (GPER). If a bit in GPER is written to one, the corresponding pin is controlled by the GPIO. If a bit is written to zero, the corresponding pin is controlled by a peripheral function.

Peripheral selection The GPIO provides multiplexing of up to four peripheral functions on a single pin. The selection is performed by accessing Peripheral Mux Register 0 (PMR0) and Peripheral Mux Register 1 (PMR1).

Output control When the I/O line is assigned to a peripheral function, i.e. the corresponding bit in GPER is written to zero, the drive of the I/O line is controlled by the peripheral. The peripheral, depending on the value in PMR0 and PMR1, determines whether the pin is driven or not.

When the I/O line is controlled by the GPIO, the value of the Output Driver Enable Register (ODER) determines if the pin is driven or not. When a bit in this register is written to one, the corresponding I/O line is driven by the GPIO. When the bit is written to zero, the GPIO does not drive the line.

The level driven on an I/O line can be determined by writing to the Output Value Register (OVR).

Inputs The level on each I/O line can be read through the Pin Value Register (PVR). This register indicates the level of the I/O lines regardless of whether the lines are driven by the GPIO or by an external component. Note that due to power saving measures, the PVR register can only be read when GPER is written to one for the corresponding pin or if interrupt is enabled for the pin.

Output line timings The figure below shows the timing of the I/O line when writing a one and a zero to OVR. The same timing applies when performing a 'set' or 'clear' access, i.e., writing a one to the Output Value Set Register (OVRS) or the Output Value Clear Register (OVRC). The timing of PVR is also shown.

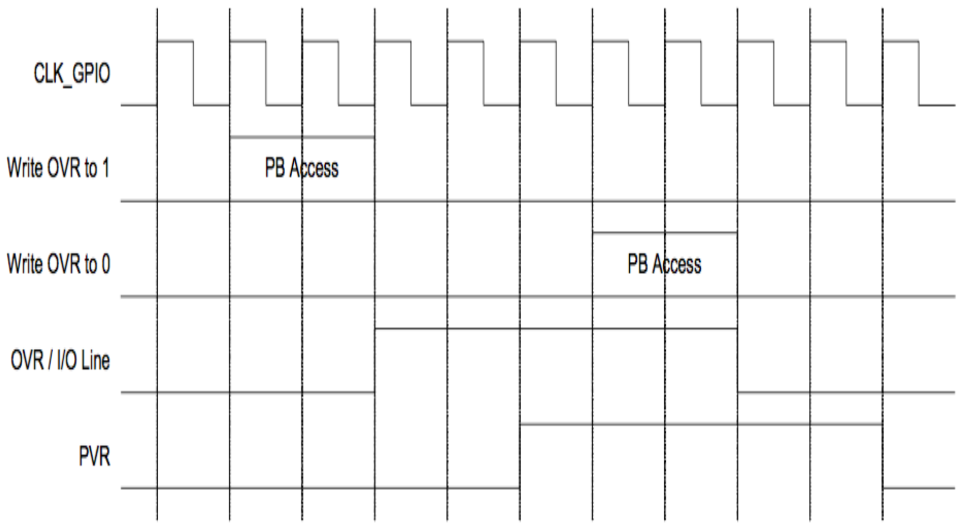


Figure 3: Output Line Timings

Advanced Operation

Pull-up resistor control Each I/O line is designed with an embedded pull-up resistor. The pull-up resistor can be enabled or disabled by writing a one or a zero to the corresponding bit in the Pull-up Enable Register (PUER). Control of the pull-up resistor is possible whether an I/O line is controlled by a peripheral or the GPIO.

Input glitch filter Optional input glitch filters can be enabled on each I/O line. When the glitch filter is enabled, a glitch with duration of less than 1 clock cycle is automatically rejected, while a pulse with duration of 2 clock cycles or more is accepted. For pulse durations between 1 clock cycle and 2 clock cycles, the pulse may or may not be taken into account, depending on the precise timing of its occurrence. Thus for a pulse to be guaranteed visible it must exceed 2 clock cycles, whereas for a glitch to be reliably filtered out, its duration must not exceed 1 clock cycle. The filter introduces 2 clock cycles of latency.

The glitch filters are controlled by the Glitch Filter Enable Register (GFER). When a bit is written to one in GFER, the glitch filter on the corresponding pin is enabled. The glitch filter affects only interrupt inputs. Inputs to peripherals or the value read through PVR are not affected by the glitch filters.

Interrupts The GPIO can be configured to generate an interrupt when it detects an input change on an I/O line. The module can be configured to signal an interrupt whenever a pin changes value or only to trigger on rising edges or falling edges. Interrupts are enabled on a pin by writing a one to the corresponding bit in the Interrupt Enable Register (IER). The interrupt mode is set by writing to the Interrupt Mode Register 0 (IMR0) and the Interrupt Mode Register 1(IMR1). Interrupts can be enabled on a pin, regardless of the configuration of the I/O line, i.e. whether it is controlled by the GPIO or assigned to a peripheral function.

In every port there are four interrupt lines connected to the interrupt controller. Groups of eight interrupts in the port are ORed together to form an interrupt line.

When an interrupt event is detected on an I/O line, and the corresponding bit in IER is written to one, the GPIO interrupt request line is asserted. A number of interrupt signals are ORed-wired together to generate a single interrupt signal to the interrupt controller.

The Interrupt Flag Register (IFR) can by read to determine which pin(s) caused the interrupt. The interrupt bit must be cleared by writing a one to the Interrupt Flag Clear Register (IFRC). To take effect, the clear operation must be performed when the interrupt line is enabled in IER. Otherwise, it will be ignored.

GPIO interrupts can only be triggered when the CLK_GPIO is enabled.

Interrupt Timings The figure below shows the timing for rising edge (or pin-change) interrupts when the glitch filter is disabled. For the pulse to be registered, it must be sampled at the rising edge of the clock. In this example, this is not the case for the first pulse. The second pulse is however sampled on a rising edge and will trigger an interrupt request.

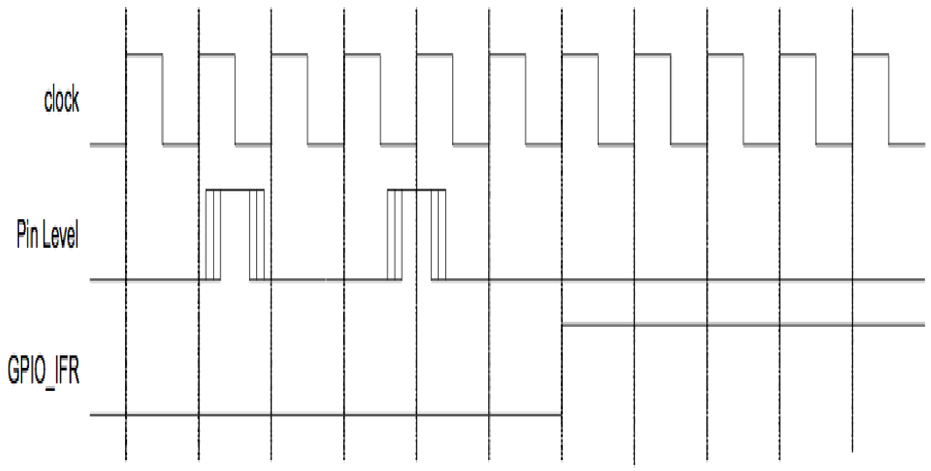


Figure 4: Interrupt Timing With Glitch Filter Disabled

The figure below shows the timing for rising edge (or pin-change) interrupts when the glitch filter is enabled. For the pulse to be registered, it must be sampled on two subsequent rising edges. In the example, the first pulse is rejected while the second pulse is accepted and causes an interrupt request.

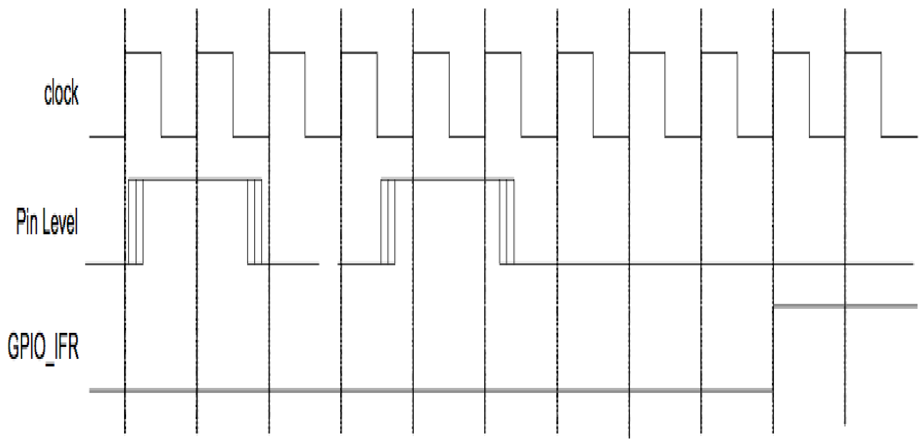


Figure 5: Interrupt Timing With Glitch Filter Enabled

User Interface The GPIO controls all the I/O pins on the AVR32 microcontroller. The pins are managed as 32- bit ports that are configurable through a PB interface. Each port has a set of configuration registers. The overall memory map of the GPIO is shown below. The number of pins and hence the number of ports are product specific.

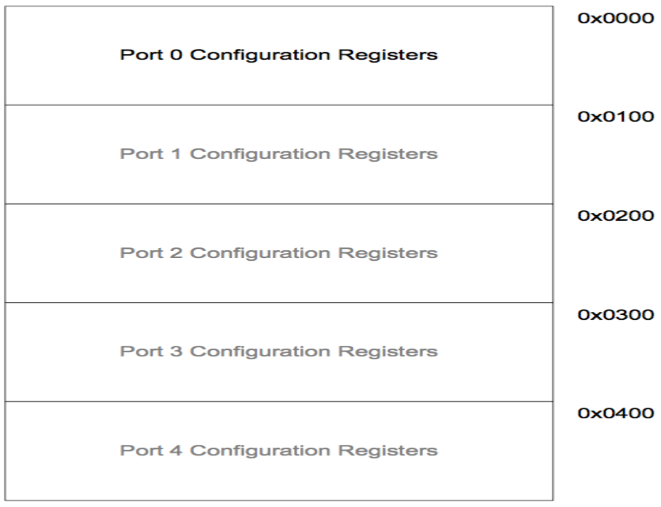

0x0000
Port 0 Configuration Registers
0x0100
Port 1 Configuration Registers
0x0200
Port 2 Configuration Registers
0x0300
Port 3 Configuration Registers
0x0400
Port 4 Configuration Registers

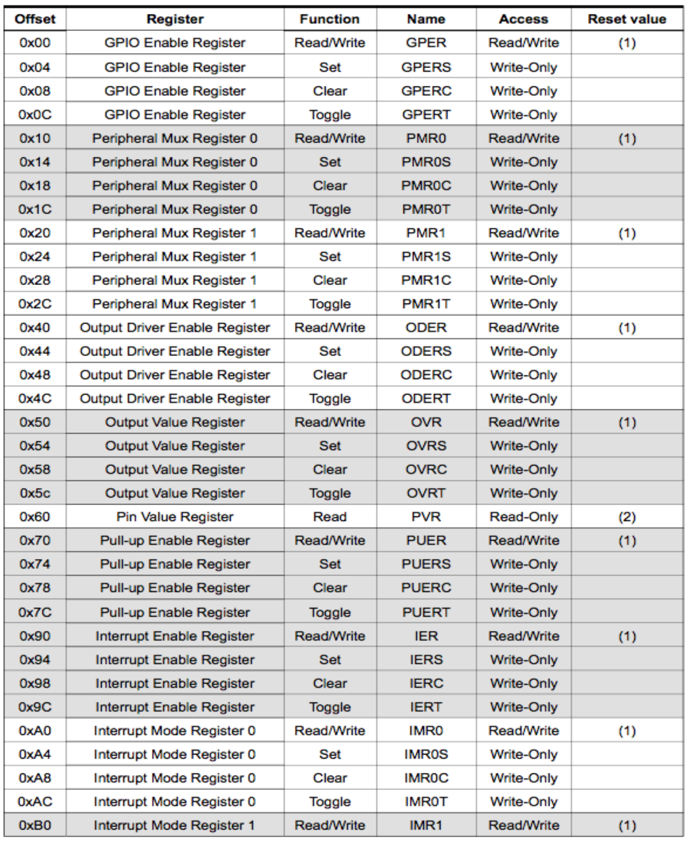
Figure 6: Overall Mermory Map

In the GPIO Controller Function Multiplexingtable in the Package and Pinout chapter, each GPIO line has a unique number. Note that the PA, PB, PC and PX ports do not directly correspond to the GPIO ports. To find the corresponding port and pin the following formula can be used:

GPIO port = floor((GPIO number) / 32), example: floor((36)/32) = 1

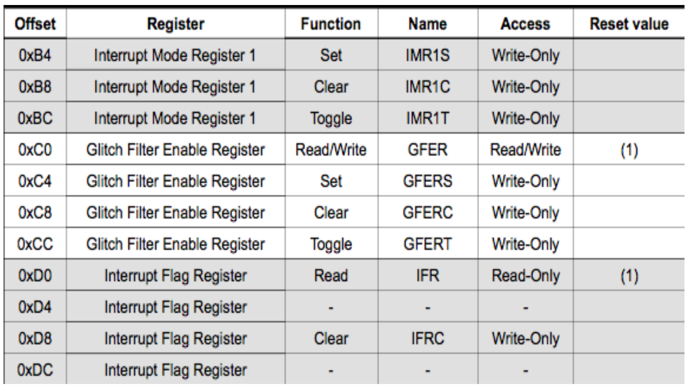
GPIO pin = GPIO number mod 32, example: 36 mod 32 = 4

The table below shows the configuration registers for one port. Addresses shown are relative to the port address offset. The specific address of a configuration register is found by adding the register offset and the port offset to the GPIO start address. One bit in each of the configuration registers corresponds to an I/O pin.



Offset	Register	Function	Name	Access	Reset value
0x00	GPIO Enable Register	Read/Write	GPER	Read/Write	(1)
0x04	GPIO Enable Register	Set	GPERS	Write-Only	
0x08	GPIO Enable Register	Clear	GPERC	Write-Only	
0x0C	GPIO Enable Register	Toggle	GPERT	Write-Only	
0x10	Peripheral Mux Register 0	Read/Write	PMR0	Read/Write	(1)
0x14	Peripheral Mux Register 0	Set	PMR0S	Write-Only	
0x18	Peripheral Mux Register 0	Clear	PMR0C	Write-Only	
0x1C	Peripheral Mux Register 0	Toggle	PMR0T	Write-Only	
0x20	Peripheral Mux Register 1	Read/Write	PMR1	Read/Write	(1)
0x24	Peripheral Mux Register 1	Set	PMR1S	Write-Only	
0x28	Peripheral Mux Register 1	Clear	PMR1C	Write-Only	
0x2C	Peripheral Mux Register 1	Toggle	PMR1T	Write-Only	
0x40	Output Driver Enable Register	Read/Write	ODER	Read/Write	(1)
0x44	Output Driver Enable Register	Set	ODERS	Write-Only	
0x48	Output Driver Enable Register	Clear	ODERC	Write-Only	
0x4C	Output Driver Enable Register	Toggle	ODERT	Write-Only	
0x50	Output Value Register	Read/Write	OVR	Read/Write	(1)
0x54	Output Value Register	Set	OVRS	Write-Only	
0x58	Output Value Register	Clear	OVRC	Write-Only	
0x5c	Output Value Register	Toggle	OVRT	Write-Only	
0x60	Pin Value Register	Read	PVR	Read-Only	(2)
0x70	Pull-up Enable Register	Read/Write	PUER	Read/Write	(1)
0x74	Pull-up Enable Register	Set	PUERS	Write-Only	
0x78	Pull-up Enable Register	Clear	PUERC	Write-Only	
0x7C	Pull-up Enable Register	Toggle	PUERT	Write-Only	
0x90	Interrupt Enable Register	Read/Write	IER	Read/Write	(1)
0x94	Interrupt Enable Register	Set	IERS	Write-Only	
0x98	Interrupt Enable Register	Clear	IERC	Write-Only	
0x9C	Interrupt Enable Register	Toggle	IERT	Write-Only	
0xA0	Interrupt Mode Register 0	Read/Write	IMR0	Read/Write	(1)
0xA4	Interrupt Mode Register 0	Set	IMR0S	Write-Only	
0xA8	Interrupt Mode Register 0	Clear	IMR0C	Write-Only	
0xAC	Interrupt Mode Register 0	Toggle	IMR0T	Write-Only	
0xB0	Interrupt Mode Register 1	Read/Write	IMR1	Read/Write	(1)

Table 1: GPIO Register Memory Map



Offset	Register	Function	Name	Access	Reset value
0xB4	Interrupt Mode Register 1	Set	IMR1S	Write-Only	
0xB8	Interrupt Mode Register 1	Clear	IMR1C	Write-Only	
0xBC	Interrupt Mode Register 1	Toggle	IMR1T	Write-Only	
0xC0	Glitch Filter Enable Register	Read/Write	GFER	Read/Write	(1)
0xC4	Glitch Filter Enable Register	Set	GFERS	Write-Only	
0xC8	Glitch Filter Enable Register	Clear	GFERC	Write-Only	
0xCC	Glitch Filter Enable Register	Toggle	GFERT	Write-Only	
0xD0	Interrupt Flag Register	Read	IFR	Read-Only	(1)
0xD4	Interrupt Flag Register	-	-	-	
0xD8	Interrupt Flag Register	Clear	IFRC	Write-Only	
0xDC	Interrupt Flag Register	-	-	-	

Access Types Each configuration register can be accessed in four different ways. The first address location can be used to write the register directly. This address can also be used to read the register value. The following addresses facilitate three different types of write access to the register. Performing a "set" access, all bits written to one will be set. Bits written to zero will be unchanged by the operation. Performing a “clear” access, all bits written to one will be cleared. Bits written to zero will be unchanged by the operation. Finally, a toggle access will toggle the value of all bits written to one. Again all bits written to zero remain unchanged. Note that for some registers (e.g. IFR), not all access methods are permitted.

Note that for ports with less than 32 bits, the corresponding control registers will have unused bits. This is also the case for features that are not implemented for a specific pin. Writing to an unused bit will have no effect. Reading unused bits will always return 0.

3.1.2 Interrupt Controller(INTC)

Features

- Autovectored low latency interrupt service with programmable priority
  - 4 priority levels for regular, maskable interrupts
  - One Non-Maskable Interrupt
- Up to 64 groups of interrupts with up to 32 interrupt requests in each group

Overview The INTC collects interrupt requests from the peripherals, prioritizes them, and delivers an interrupt request and an autovector to the CPU. The AVR32 architecture supports 4 priority levels for regular, maskable interrupts, and a Non-Maskable Interrupt (NMI).

The INTC supports up to 64 groups of interrupts. Each group can have up to 32 interrupt request lines, these lines are connected to the peripherals. Each group has an Interrupt Priority Register (IPR) and an Interrupt Request Register (IRR). The IPRs are used to assign a priority level and an autovector to each group, and the IRRs are used to identify the active interrupt request within each group. If a group has only one interrupt request line, an active interrupt group uniquely identifies the active interrupt request line, and the corresponding IRR is not needed. The INTC also provides one Interrupt Cause Register (ICR) per priority level. These registers identify the group that has a pending interrupt of the corresponding priority level. If several groups have a pending interrupt of the same level, the group with the lowest number takes priority.

Block Diagram Figure 12 gives an overview of the INTC. The grey boxes represent registers that can be accessed via the user interface. The interrupt requests from the peripherals (IREQn) and the NMI are input on the left side of the figure. Signals to and from the CPU are on the right side of the figure.

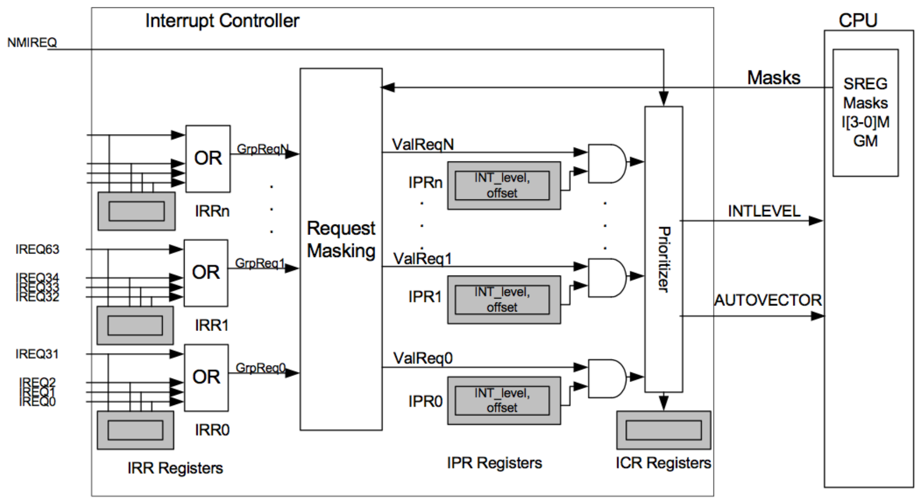


Figure 7: INTC Block Diagram

Product Dependencies In order to use this module, other parts of the system must be configured correctly, as described below.

Power Management If the CPU enters a sleep mode that disables CLK_SYNC, the INTC will stop functioning and resume operation after the system wakes up from sleep mode.

Clocks The clock for the INTC bus interface (CLK_INTC) is generated by the Power Manager. This clock is enabled at reset, and can be disabled in the Power Manager. The INTC sampling logic runs on a clock which is stopped in any of the sleep modes where the system RC oscillator is not running. This clock is referred to as CLK_SYNC. This clock is enabled at reset, and only turned off in sleep modes where the system RC oscillator is stopped.

Debug Operation When an external debugger forces the CPU into debug mode, the INTC continues normal operation.

Functional Description All of the incoming interrupt requests (IREQs) are sampled into the corresponding Interrupt Request Register (IRR). The IRRs must be accessed to identify which IREQ within a group that is active. If several IREQs within the same group are active, the interrupt service routine must prioritize between them. All of the input lines in each group are logically ORed together to form the GrpReqN lines, indicating if there is a pending interrupt in the corresponding group.

The Request Masking hardware maps each of the GrpReq lines to a priority level from INT0 to INT3 by associating each group with the Interrupt Level (INTLEVEL) field in the corresponding Interrupt Priority Register (IPR). The GrpReq inputs are then masked by the mask bits from the CPU status register. Any interrupt group that has a pending interrupt of a priority level that is not masked by the CPU status register, gets its corresponding ValReq line asserted.

Masking of the interrupt requests is done based on five interrupt mask bits of the CPU status register, namely Interrupt Level 3 Mask (I3M) to Interrupt Level 0 Mask (I0M), and Global Interrupt Mask (GM). An interrupt request is masked if either the GM or the corresponding interrupt level mask bit is set.

The Prioritizer hardware uses the ValReq lines and the INTLEVEL field in the IPRs to select the pending interrupt of the highest priority. If an NMI interrupt request is pending, it automatically gets the highest priority of any pending interrupt. If several interrupt groups of the highest pending interrupt level have pending interrupts, the interrupt group with the lowest number is selected.

The INTLEVEL and handler autovector offset (AUTOVECTOR) of the selected interrupt are transmitted to the CPU for interrupt handling and context switching. The CPU does not need to know which interrupt is requesting handling, but only the level and the offset of the handler address. The IRR registers contain the interrupt request lines of the groups and can be read via user interface registers for checking which interrupts of the group are actually active.

The delay through the INTC from the peripheral interrupt request is set until the interrupt request to the CPU is set is three cycles of CLK_SYNC.

Non-Maskable Interrupts A NMI request has priority over all other interrupt requests. NMI has a dedicated exception vector address defined by the AVR32 architecture, so AUTOVECTOR is undefined when INTLEVEL indicates that an NMI is pending.

CPU Response When the CPU receives an interrupt request it checks if any other exceptions are pending. If no exceptions of higher priority are pending, interrupt handling is initiated. When initiating interrupt handling, the corresponding interrupt mask bit is set automatically for this and lower levels in status register. E.g, if an interrupt of level 3 is approved for handling, the interrupt mask bits I3M, I2M, I1M, and I0M are set in status register. If an interrupt of level 1 is approved, the masking bits I1M and I0M are set in status register. The handler address is calculated by logical OR of the AUTOVECTOR to the CPU system register Exception Vector Base Address (EVBA). The CPU will then jump to the calculated address and start executing the interrupt handler.

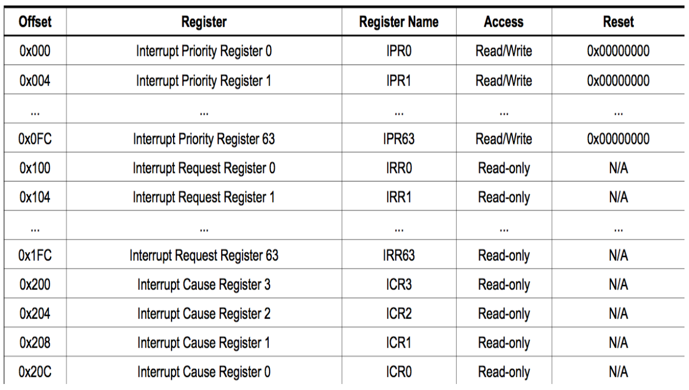
Setting the interrupt mask bits prevents the interrupts from the same and lower levels to be passed through the interrupt controller. Setting of the same level mask bit prevents also multiple requests of the same interrupt to happen.

It is the responsibility of the handler software to clear the interrupt request that caused the interrupt before returning from the interrupt handler. If the conditions that caused the interrupt are not cleared, the interrupt request remains active.

Clearing an Interrupt Request Clearing of the interrupt request is done by writing to registers in the corresponding peripheral module, which then clears the corresponding NMIREQ/IREQ signal.

The recommended way of clearing an interrupt request is a store operation to the controlling peripheral register, followed by a dummy load operation from the same register. This causes a pipeline stall, which prevents the interrupt from accidentally re-triggering in case the handler is exited and the interrupt mask is cleared before the interrupt request is cleared.

User Interface Here are the registers accessible by the user:



Offset	Register	Register Name	Access	Reset
0x000	Interrupt Priority Register 0	IPR0	Read/Write	0x00000000
0x004	Interrupt Priority Register 1	IPR1	Read/Write	0x00000000
...	...	...	...	...
0x0FC	Interrupt Priority Register 63	IPR63	Read/Write	0x00000000
0x100	Interrupt Request Register 0	IRR0	Read-only	N/A
0x104	Interrupt Request Register 1	IRR1	Read-only	N/A
...	...	...	...	...
0x1FC	Interrupt Request Register 63	IRR63	Read-only	N/A
0x200	Interrupt Cause Register 3	ICR3	Read-only	N/A
0x204	Interrupt Cause Register 2	ICR2	Read-only	N/A
0x208	Interrupt Cause Register 1	ICR1	Read-only	N/A
0x20C	Interrupt Cause Register 0	ICR0	Read-only	N/A

Table 2: INTC Register Memory Map

3.1.3 Power Manager(PM)

Overview The Power Manager (PM) controls the oscillators and PLLs, and generates the clocks and resets in the device. The PM controls two fast crystal oscillators, as well as two PLLs, which can multiply the clock from either oscillator to provide higher frequencies. Additionally, a low-power 32KHz oscillator is used to generate the real-time counter clock for high accuracy real-time measurements. The PM also contains a low-power RC oscillator with fast start-up time, which can be used to clock the digital logic.

The provided clocks are divided into synchronous and generic clocks. The synchronous clocks are used to clock the main digital logic in the device, namely the CPU, and the modules and peripherals connected to the HSB, PBA, and PBB buses. The generic clocks are asynchronous clocks, which can be tuned precisely within a wide frequency range, which makes them suitable for peripherals that require specific frequencies, such as timers and communication modules.

The PM also contains advanced power-saving features, allowing the user to optimize the power consumption for an application. The synchronous clocks are divided into three clock domains, one for the CPU and HSB, one for modules on the PBA bus, and one for modules on the PBB bus.The three clocks can run at different speeds, so the user can save power by running peripherals at a relatively low clock, while maintaining a high CPU performance. Additionally, the clocks can be independently changed on-the-fly, without halting any peripherals. This enables the user to adjust the speed of the CPU and memories to the dynamic load of the application, without disturbing or re-configuring active peripherals.

Each module also has a separate clock, enabling the user to switch off the clock for inactive modules, to save further power. Additionally, clocks and oscillators can be automatically switched off during idle periods by using the sleep instruction on the CPU. The system will return to normal on occurrence of interrupts.

The Power Manager also contains a Reset Controller, which collects all possible reset sources, generates hard and soft resets, and allows the reset source to be identified by software.

Block Diagram

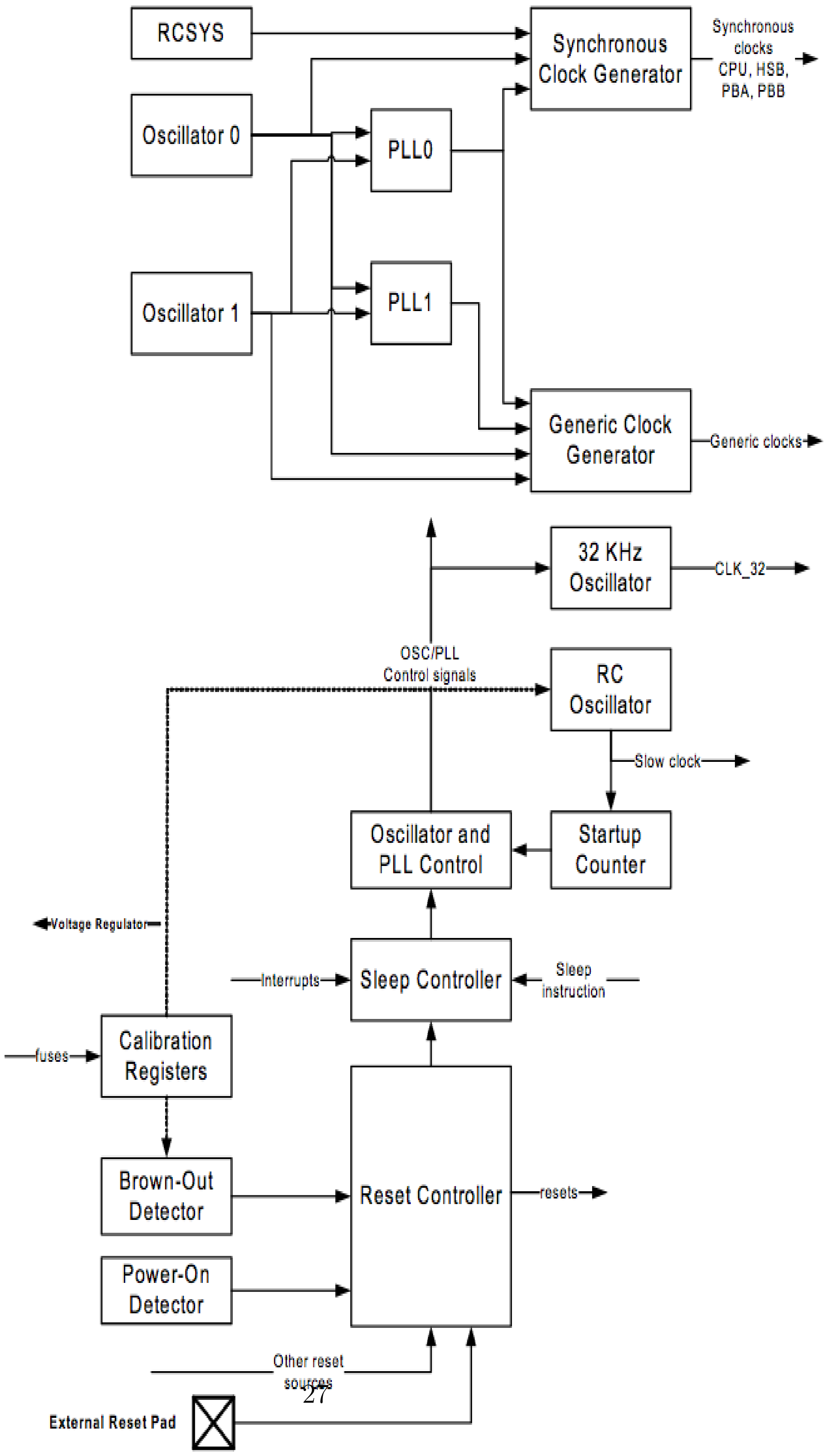


Figure 8: Power Manager Block Diagram

Product Dependencies

I/O Lines The PM provides a number of generic clock outputs, which can be connected to output pins, multiplexed with I/O lines. The user must first program the I/O controller to assign these pins to their peripheral function. If the I/O pins of the PM are not used by the application, they can be used for other purposes by the I/O controller.

Interrupt The PM interrupt line is connected to one of the internal sources of the interrupt controller. Using the PM interrupt requires the interrupt controller to be programmed first.

Functional Description

Slow Clock The slow clock is generated from an internal RC oscillator which is always running, except in Static mode. The slow clock can be used for the main clock in the device. The slow clock is also used for the Watchdog Timer and measuring various delays in the Power Manager.

The RC oscillator has a 3 cycles startup time, and is always available when the CPU is running. The RC oscillator operates at approximately 115 kHz. Software can change RC oscillator calibration through the use of the RCCR register.

RC oscillator can also be used as the RTC clock when crystal accuracy is not required.

Oscillator 0 and 1 Operation The two main oscillators are designed to be used with an external crystal and two biasing capacitors, as shown in Figure 9. Oscillator 0 can be used for the main clock in the device. Both oscillators can be used as source for the generic clocks.

The oscillators are disabled by default after reset. When the oscillators are disabled, the XIN and XOUT pins can be used as general purpose I/Os. When the oscillators are configured to use an external clock, the clock must be applied to the XIN pin while the XOUT pin can be used as a general purpose I/O.

The oscillators can be enabled by writing to the OSCnEN bits in MCCTRL. Operation mode (external clock or crystal) is chosen by writing to the MODE field in OSCCTRLn. Oscillators are automatically switched off in certain sleep modes to reduce power consumption.

After a hard reset, or when waking up from a sleep mode that disabled the oscillators, the oscillators may need a certain amount of time to stabilize on the correct frequency. This start-up time can be set in the OSCCTRLn register.

The PM masks the oscillator outputs during the start-up time, to ensure that no unstable clocks propagate to the digital logic. The OSCnRDY bits in POSCSR are automatically set and cleared according to the status of the oscillators. A zero to one transition on these bits can also be configured to generate an interrupt.

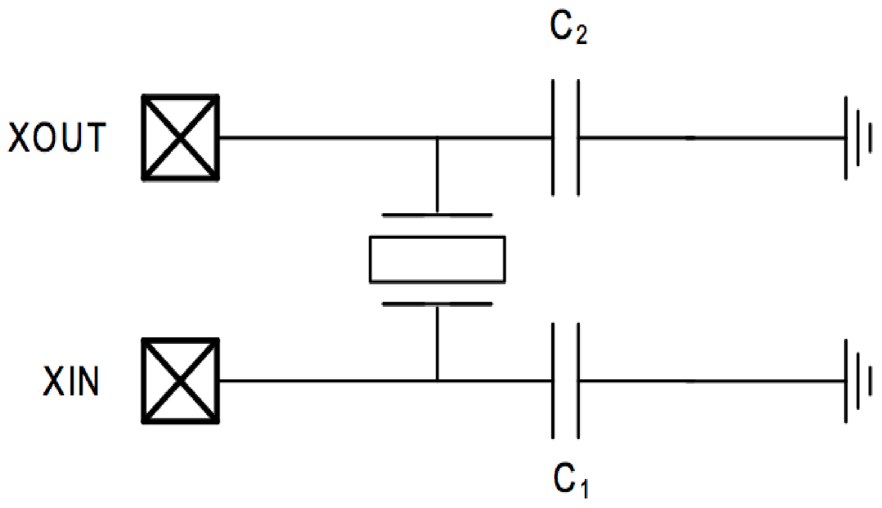


Figure 9: Oscillator Connections

32 KHz Oscillator Operation The 32 KHz oscillator operates as described for Oscillator 0 and 1 above. The 32 KHz oscillator is used as source clock for the Real-Time Counter.

The oscillator is disabled by default, but can be enabled by writing OSC32EN in OSCCTRL32. The oscillator is an ultra-low power design and remains enabled in all sleep modes except Static mode.

While the 32 KHz oscillator is disabled, the XIN32 and XOUT32 pins are available as general purpose I/Os. When the oscillator is configured to work with an external clock (MODE field in OSCCTRL32 register), the external clock must be connected to XIN32 while the XOUT32 pin can be used as a general purpose I/O.

The startup time of the 32 KHz oscillator can be set in the OSCCTRL32, after which OSC32RDY in POSCSR is set. An interrupt can be generated on a zero to one transition of OSC32RDY.

As a crystal oscillator usually requires a very long startup time (up to 1 second), the 32 KHz oscillator will keep running across resets, except Power-On-Reset.

PLL Operation The device contains two PLLs, PLL0 and PLL1. These are disabled by default, but can be enabled to provide high frequency source clocks for synchronous or generic clocks. The PLLs can take either Oscillator 0 or 1 as reference clock. The PLL output is divided by a multiplication factor, and the PLL compares the resulting clock to the reference clock. The PLL will adjust its output frequency until the two compared clocks are equal, thus locking the output frequency to a multiple of the reference clock frequency.

When the PLL is switched on, or when changing the clock source or multiplication factor for the PLL, the PLL is unlocked and the output frequency is undefined. The PLL clock for the digital logic is automatically masked when the PLL is unlocked, to prevent connected digital logic from receiving a too high frequency and thus become unstable.

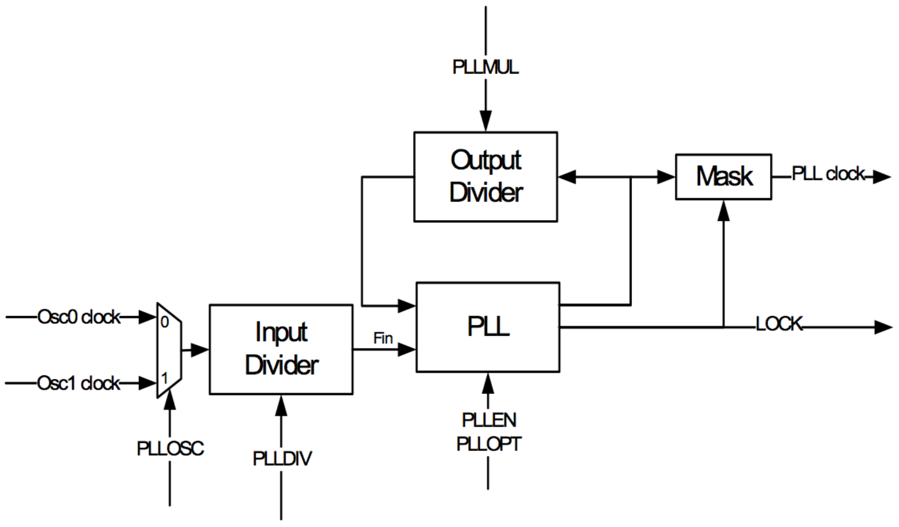


Figure 10: PLL with Control Logic and Filters

Enabling the PLL: PLLn is enabled by writing the PLLEN bit in the PLLn register. PLLOSC selects Oscillator 0 or 1 as clock source. The PLLMUL and PLLDIV bitfields must be written with the multiplication and division factors, respectively, creating the voltage controlled oscillator frequency f_VCO and the PLL frequency f_PLL:

if $PLLDIV > 0$

$f_{IN} = \frac{f_{OSC}}{2.PLL_{DIV}}$

$f_{VCO} = \frac{(PLLMUL+1)}{(PLLDIV).f_{OSC}}$

if $PLLDIV = 0$

$f_{IN} = f_{OSC}$

$f_{VCO} = 2.(PLLMUL + 1).f_{OSC}$

Note: Refer to Electrical Characteristics section for F_{IN} and F_{VCO} frequency range.

if $PLLOPT[1]$ field is set to 0:

$f_{PLL} = f_{VCO}$

if $PLLOPT[1]$ field is set to 1:

$f_{PLL} = f_{VCO}/2$

The PLLn: PLLOPT field should be set to proper values according to the PLL operating frequency. The PLLOPT field can also be set to divide the output frequency of the PLLs by 2.

The lock signal for each PLL is available as a LOCKn flag in POSCSR. An interrupt can be generated on a 0 to 1 transition of these bits.

Synchronous Clocks The slow clock (default), Oscillator 0, or PLL0 provide the source for the main clock, which is the common root for the synchronous clocks for the CPU/HSB, PBA, and PBB modules. The main clock is divided by an 8-bit prescaler, and each of these four synchronous clocks can run from any tapping of this prescaler, or the undivided main clock, as long as $f_{CPU} \geq f_{PBA,B}$.

The synchronous clock source can be changed on-the fly, responding to varying load in the application. The clock domains can be shut down in sleep mode. Additionally, the clocks for each module in the four domains can be individually masked, to avoid power consumption in inactive modules.

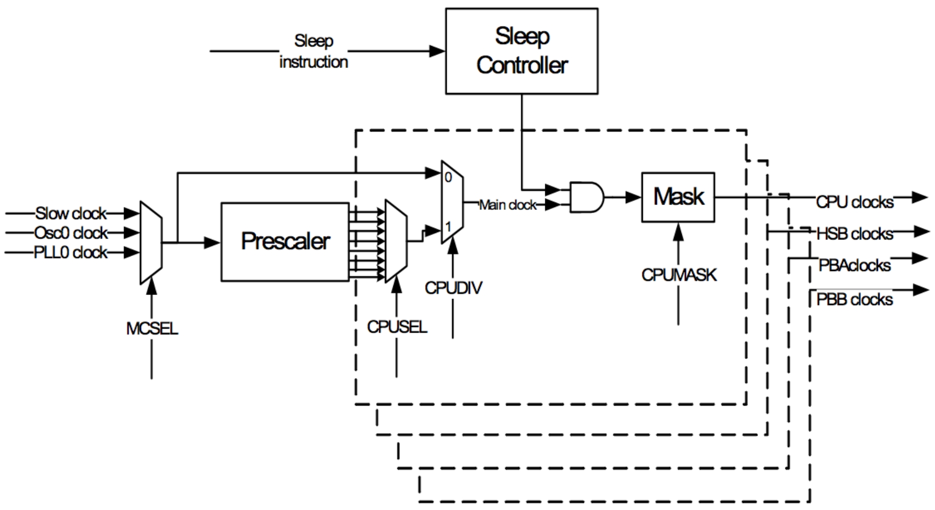


Figure 11: Synchronous Clock Generation

Selecting PLL or oscillator for the main clock The common main clock can be connected to the slow clock, Oscillator 0, or PLL0. By default, the main clock will be connected to the slow clock. The user can connect the main clock to Oscillator 0 or PLL0 by writing the MCSEL field in the Main Clock Control Register (MCCTRL). This must only be done after that unit has been enabled, otherwise a deadlock will occur. Care should also be taken that the new frequency of the synchronous clocks does not exceed the maximum frequency for each clock domain.

Selecting synchronous clock division ratio The main clock feeds an 8-bit prescaler, which can be used to generate the synchronous clocks. By default, the synchronous clocks run on the undivided main clock. The user can select a prescaler division for the CPU clock by writing CKSEL.CPUDIV to 1 and CPUSEL to the prescaling value, resulting in a CPU clock frequency:

$f_{CPU} = f_{main}/2^{(}CPUSEL + 1)$

Similarly, the clock for the PBA, and PBB can be divided by writing their respective fields. To ensure correct operation, frequencies must be selected so that$f_{CPU} \geq f_{PBA,B}$. Also, frequencies must never exceed the specified maximum frequency for each clock domain.

CKSEL can be written without halting or disabling peripheral modules. Writing CKSEL allows a new clock setting to be written to all synchronous clocks at the same time. It is possible to keep one or more clocks unchanged by writing the same value a before to the xxxDIV and xxxSEL fields. This way, it is possible to e.g. scale CPU and HSB speed according to the required performance, while keeping the PBA and PBB frequency constant.

For modules connected to the HSB bus, the PB clock frequency must be set to the same frequency than the CPU clock.

Clock ready flag There is a slight delay from CKSEL is written and the new clock setting becomes effective. During this interval, the Clock Ready (CKRDY) flag in ISR will read as 0. If IER.CKRDY is written to one, the Power Manager interrupt can be triggered when the new clock setting is effective. CKSEL must not be re-written while CKRDY is zero, or the system may become unstable or hang.

Peripheral Clock Masking By default, the clock for all modules are enabled, regardless of which modules are actually being used. It is possible to disable the clock for a module in the CPU, HSB, PBA, or PBB clock domain by writing the corresponding bit in the Clock Mask register (CPU/HSB/PBA/PBB) to 0. When a module is not clocked, it will cease operation, and its registers cannot be read or written. The module can be re-enabled later by writing the corresponding mask bit to 1.

A module may be connected to several clock domains, in which case it will have several mask bits.

Cautionary note The OCD clock must never be switched off if the user wishes to debug the device with a JTAG debugger.

Note that clocks should only be switched off if it is certain that the module will not be used. Switching off the clock for the internal RAM will cause a problem if the stack is mapped there. Switching off the clock to the Power Manager (PM), which contains the mask registers, or the corresponding PBx bridge, will make it impossible to write the mask registers again. In this case, they can only be re-enabled by a system reset.

Mask ready flag Due to synchronisation in the clock generator, there is a slight delay from a mask register is written until the new mask setting goes into effect. When clearing mask bits, this delay can usually be ignored.

However, when setting mask bits, the registers in the corresponding module must not be written until the clock has actually be re-enabled. The status flag MSKRDY in ISR pro- vides the required mask status information. When writing either mask register with any value, this bit is cleared. The bit is set when the clocks have been enabled and disabled according to the new mask setting. Optionally, the Power Manager interrupt can be enabled by writing the MSKRDY bit in IER.

Sleep Modes In normal operation, all clock domains are active, allowing software execution and peripheral operation. When the CPU is idle, it is possible to switch off the CPU clock and optionally other clock domains to save power. This is activated by the sleep instruction, which takes the sleep mode index number as argument.

Entering and exiting sleep modes The sleep instruction will halt the CPU and all modules belonging to the stopped clock domains. The modules will be halted regardless of the bit settings of the mask registers.

Oscillators and PLLs can also be switched off to save power. Some of these modules have a relatively long start-up time, and are only switched off when very low power consumption is required.

The CPU and affected modules are restarted when the sleep mode is exited. This occurs when an interrupt triggers. Note that even if an interrupt is enabled in sleep mode, it may not trigger if the source module is not clocked.

Supported sleep modes The following sleep modes are supported.

Idle: The CPU is stopped, the rest of the chip is operating. Wake-up sources are any interrupt.

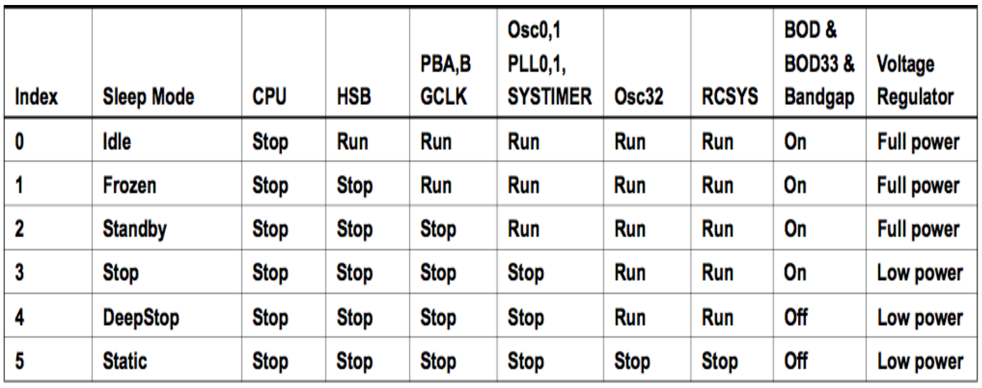
Frozen: The CPU and HSB modules are stopped, peripherals are operating. Wake-up sources are any interrupt from PB modules.

Standby: All synchronous clocks are stopped, but oscillators and PLLs are running, allowing quick wake-up to normal mode. Wake-up sources are RTC or external interrupt.

Stop: As Standby, but Oscillator 0 and 1, and the PLLs are stopped. 32 KHz (if enabled) and RC oscillators and RTC/WDT still operate. Wake-up sources are RTC, external interrupt, or external reset pin.

DeepStop: All synchronous clocks, Oscillator 0 and 1 and PLL 0 and 1 are stopped. 32 KHz oscillator can run if enabled. RC oscillator still operates. Bandgap voltage reference, BOD and BOD33 are turned off. Wake-up sources are RTC, external interrupt (EIC) or external reset pin.

Static: All oscillators, including 32 KHz and RC oscillator are stopped. Bandgap voltage reference, BOD and BOD33 detectors are turned off. Wake-up sources are external interrupt (EIC) in asynchronous mode only or external reset pin.



Index	Sleep Mode	CPU	HSB	PBA,B GCLK	Osc0,1 PLL0,1, SYSTIMER	Osc32	RCSYS	BOD & BOD33 & Bandgap	Voltage Regulator
0	Idle	Stop	Run	Run	Run	Run	Run	On	Full power
1	Frozen	Stop	Stop	Run	Run	Run	Run	On	Full power
2	Standby	Stop	Stop	Stop	Run	Run	Run	On	Full power
3	Stop	Stop	Stop	Stop	Stop	Run	Run	On	Low power
4	DeepStop	Stop	Stop	Stop	Stop	Run	Run	Off	Low power
5	Static	Stop	Stop	Stop	Stop	Stop	Stop	Off	Low power

Table 3: Sleep Modes

The power level of the internal voltage regulator is also adjusted according to the sleep mode to reduce the internal regulator power consumption.

Precautions when entering sleep mode Modules communicating with external circuits should normally be disabled before entering a sleep mode that will stop the module operation. This prevents erratic behavior when entering or exiting sleep mode.

Communication between the synchronous clock domains is disturbed when entering and exiting sleep modes. This means that bus transactions are not allowed between clock domains affected by the sleep mode. The system may hang if the bus clocks are stopped in the middle of a bus transaction.

The CPU is automatically stopped in a safe state to ensure that all CPU bus operations are complete when the sleep mode goes into effect. Thus, when entering Idle mode, no further action is necessary.

When entering a sleep mode (except Idle mode), all HSB masters must be stopped before entering the sleep mode. Also, if there is a chance that any PB write operations are incomplete, the CPU should perform a read operation from any register on the PB bus before executing the sleep instruction.

This will stall the CPU while waiting for any pending PB operations to complete.

When entering a sleep mode deeper or equal to DeepStop, the VBus asynchronous interrupt should be disabled (USBCON.VBUSTE = 0).

Wake Up The USB can be used to wake up the part from sleep modes through register AWEN of the Power Manager.

Generic Clocks Timers, communication modules, and other modules connected to external circuitry may require specific clock frequencies to operate correctly. The Power Manager contains an implementation defined number of generic clocks that can provide a wide range of accurate clock frequencies.

Each generic clock module runs from either Oscillator 0 or 1, or PLL0 or 1. The selected source can optionally be divided by any even integer up to 512. Each clock can be independently enabled and disabled, and is also automatically disabled along with peripheral clocks by the Sleep Controller.

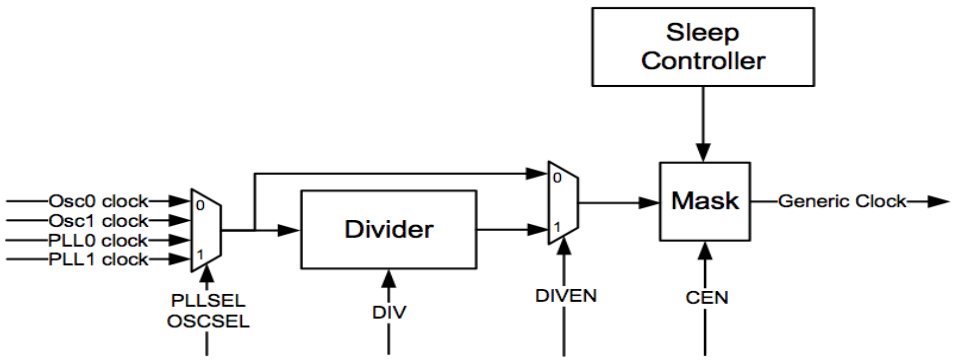


Figure 12: Generic Clock Generation

Enabling a generic clock A generic clock is enabled by writing the CEN bit in GCCTRL to 1. Each generic clock can use either Oscillator 0 or 1 or PLL0 or 1 as source, as selected by the PLLSEL and OSCSEL bits. The source clock can optionally be divided by writing DIVEN to 1 and the division factor to DIV, resulting in the output frequency:

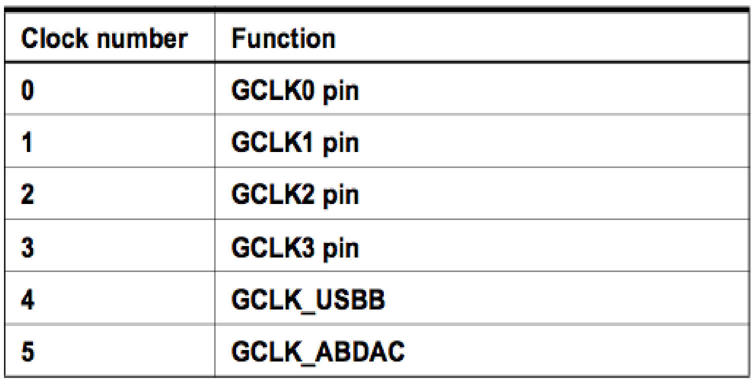
$f_{GCLK} = f_{SRC}/(2.(DIV + 1))$

Disabling a generic clock The generic clock can be disabled by writing CEN to zero or entering a sleep mode that disables the PB clocks. In either case, the generic clock will be switched off on the first falling edge after the disabling event, to ensure that no glitches occur. If CEN is written to 0, the bit will still read as 1 until the next falling edge occurs, and the clock is actually switched off. When writing CEN to 0, the other bits in GCCTRL should not be changed until CEN reads as 0, to avoid glitches on the generic clock.

When the clock is disabled, both the prescaler and output are reset.

Changing clock frequency When changing generic clock frequency by writing GCCTRL, the clock should be switched off by the procedure above, before being re-enabled with the new clock source or division setting. This prevents glitches during the transition.

Generic clock implementation The generic clocks are allocated to different functions as shown in Table 4.



Clock number	Function
0	GCLK0 pin
1	GCLK1 pin
2	GCLK2 pin
3	GCLK3 pin
4	GCLK_USBB
5	GCLK_ABDAC

Table 4: Generic Clock Allocation

Divided PB Clocks The clock generator in the Power Manager provides divided PBA and PBB clocks for use by peripherals that require a prescaled PBx clock.

The divided clocks are not directly maskable, but are stopped in sleep modes where the PBx clocks are stopped.

Debug Operation The OCD clock must never be switched off if the user wishes to debug the device with a JTAG debugger.

During a debug session, the user may need to halt the system to inspect memory and CPU registers. The clocks normally keep running during this debug operation, but some peripherals may require the clocks to be stopped, e.g. to prevent timer overflow, which would cause the program to fail. For this reason, peripherals on the PBA and PBB buses may use “debug qualified” PBx clocks.The divided PBx clocks are always debug qualified clocks.

Debug qualified PBx clocks are stopped during debug operation. The debug system can optionally keep these clocks running during the debug operation.

Reset Controller The Reset Controller collects the various reset sources in the system and generates hard and soft resets for the digital logic.

The device contains a Power-On Detector, which keeps the system reset until power is stable. This eliminates the need for external reset circuitry to guarantee stable operation when powering up the device.

It is also possible to reset the device by asserting the RESET_N pin. This pin has an internal pullup, and does not need to be driven externally when negated. Table 5 lists these and other reset sources supported by the Reset Controller.

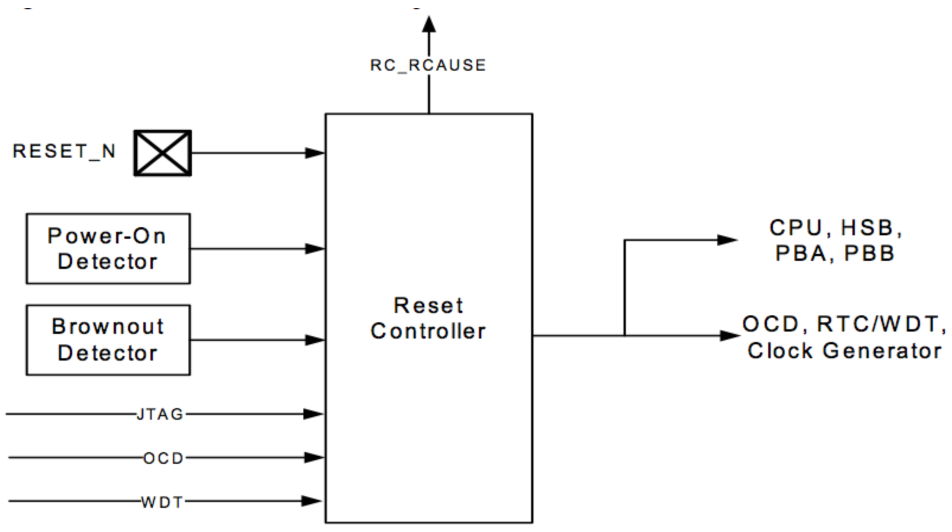
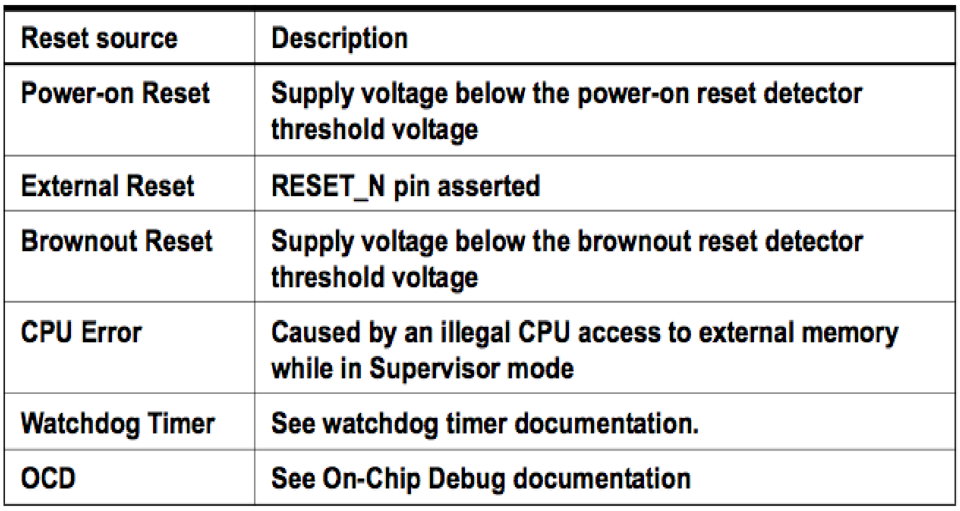


Figure 13: Reset Controller Block Diagram

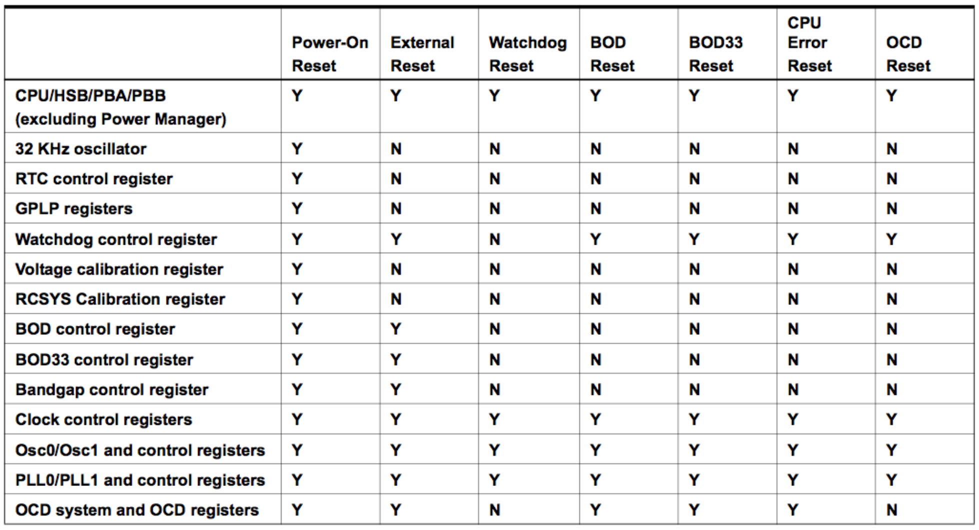
In addition to the listed reset types, the JTAG can keep parts of the device statically reset through the JTAG Reset Register.



Reset source	Description
Power-on Reset	Supply voltage below the power-on reset detector threshold voltage
External Reset	RESET_N pin asserted
Brownout Reset	Supply voltage below the brownout reset detector threshold voltage
CPU Error	Caused by an illegal CPU access to external memory while in Supervisor mode
Watchdog Timer	See watchdog timer documentation.
OCD	See On-Chip Debug documentation

Table 5: Reset Description

When a reset occurs, some parts of the chip are not necessarily reset, depending on the reset source. Only the Power On Reset (POR) will force a reset of the whole chip.



	Power-On Reset	External Reset	Watchdog Reset	BOD Reset	BOD33 Reset	CPU Error Reset	OCD Reset
CPU/HSB/PBA/PBB (excluding Power Manager)	Y	Y	Y	Y	Y	Y	Y
32 KHz oscillator	Y	N	N	N	N	N	N
RTC control register	Y	N	N	N	N	N	N
GPLP registers	Y	N	N	N	N	N	N
Watchdog control register	Y	Y	N	Y	Y	Y	Y
Voltage calibration register	Y	N	N	N	N	N	N
RCSYS Calibration register	Y	N	N	N	N	N	N
BOD control register	Y	Y	N	N	N	N	N
BOD33 control register	Y	Y	N	N	N	N	N
Bandgap control register	Y	Y	N	N	N	N	N
Clock control registers	Y	Y	Y	Y	Y	Y	Y
Osc0/Osc1 and control registers	Y	Y	Y	Y	Y	Y	Y
PLL0/PLL1 and control registers	Y	Y	Y	Y	Y	Y	Y
OCD system and OCD registers	Y	Y	N	Y	Y	Y	N

Table 6: Effect of the Different Reset Events

The cause of the last reset can be read from the RCAUSE register. This register contains one bit for each reset source, and can be read during the boot sequence of an application to determine the proper action to be taken.

Power-On detector The Power-On Detector monitors the VDDCORE supply pin and generates a reset when the device is powered on. The reset is active until the supply voltage from the linear regulator is above the power-on threshold level. The reset will be re-activated if the voltage drops below the power-on threshold level.

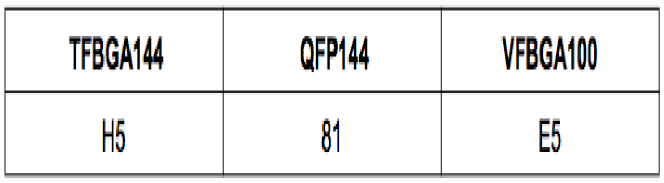
Brown-Out detector The Brown-Out Detector (BOD) monitors the VDDCORE supply pin and compares the supply voltage to the brown-out detection level, as set in BOD.LEVEL. The BOD is disabled by default, but can be enabled either by software or by flash fuses. The Brown-Out Detector can either generate an interrupt or a reset when the supply voltage is below the brown-out detection level. In any case, the BOD output is available in bit POSCSR.BODDET bit.

Note that any change to the BOD.LEVEL field of the BOD register should be done with the BOD deactivated to avoid spurious reset or interrupt.

Brown-Out detector 3V3 The Brown-Out Detector 3V3 (BOD33) monitors one VDDIO supply pin and compares the supply voltage to the brown-out detection 3V3 level, which is typically calibrated at 2V7. The BOD33 is enabled by default, but can be disabled by software. The Brown-Out Detector 3V3 can either generate an interrupt or a reset when the supply voltage is below the brown-out detection3V3 level. In any case, the BOD33 output is available in bit POSCSR.BOD33DET bit.

Note that any change to the BOD33.LEVEL field of the BOD33 register should be done with the BOD33 deactivated to avoid spurious reset or interrupt.

The BOD33.LEVEL default value is calibrated to 2V7



TFBGA144	QFP144	VFBGA100
H5	81	E5

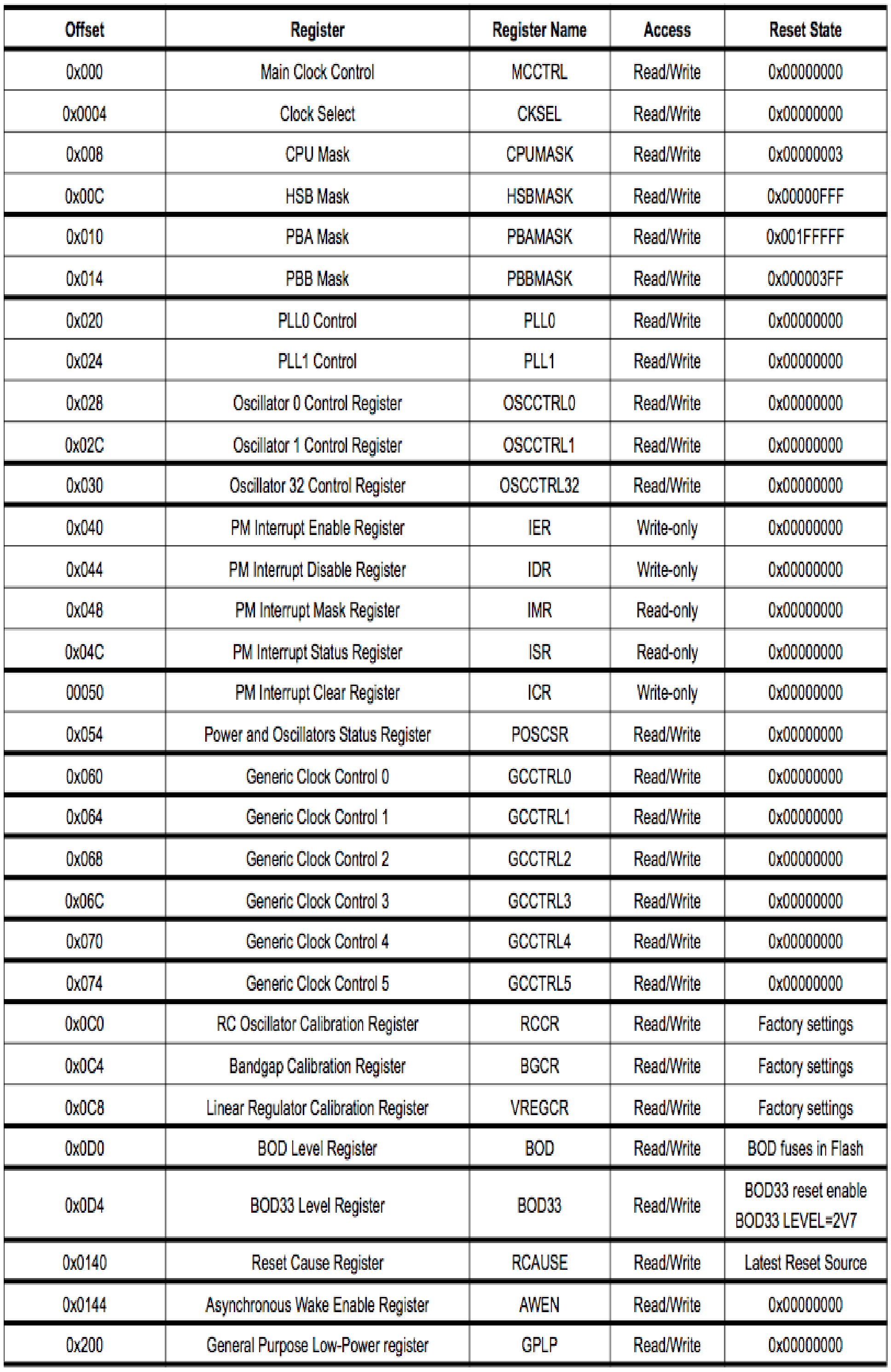
Table 7: VDDIO pin monitored by BOD33

External reset The external reset detector monitors the state of the RESET_N pin. By default, a low level on this pin will generate a reset.

Calibration Registers The Power Manager controls the calibration of the RC oscillator, voltage regulator, bandgap voltage reference through several calibrations registers.

Those calibration registers are loaded after a Power On Reset with default values stored in factory-programmed flash fuses.

Although it is not recommended to override default factory settings, it is still possible to override these default values by writing to those registers. To prevent unexpected writes due to software bugs, write access to these registers is protected by a “key”. First, a write to the register must be made with the field “KEY” equal to 0x55 then a second write must be issued with the “KEY” field equal to 0xAA.



Offset	Register	Register Name	Access	Reset State
0x000	Main Clock Control	MCCTRL	Read/Write	0x00000000
0x0004	Clock Select	CKSEL	Read/Write	0x00000000
0x008	CPU Mask	CPUMASK	Read/Write	0x00000003
0x00C	HSB Mask	HSBMASK	Read/Write	0x00000FFF
0x010	PBA Mask	PBAMASK	Read/Write	0x001FFFFF
0x014	PBB Mask	PBBMASK	Read/Write	0x000003FF
0x020	PLL0 Control	PLL0	Read/Write	0x00000000
0x024	PLL1 Control	PLL1	Read/Write	0x00000000
0x028	Oscillator 0 Control Register	OSCCTRL0	Read/Write	0x00000000
0x02C	Oscillator 1 Control Register	OSCCTRL1	Read/Write	0x00000000
0x030	Oscillator 32 Control Register	OSCCTRL32	Read/Write	0x00000000
0x040	PM Interrupt Enable Register	IER	Write-only	0x00000000
0x044	PM Interrupt Disable Register	IDR	Write-only	0x00000000
0x048	PM Interrupt Mask Register	IMR	Read-only	0x00000000
0x04C	PM Interrupt Status Register	ISR	Read-only	0x00000000
00050	PM Interrupt Clear Register	ICR	Write-only	0x00000000
0x054	Power and Oscillators Status Register	POSCSR	Read/Write	0x00000000
0x060	Generic Clock Control 0	GCCTRL0	Read/Write	0x00000000
0x064	Generic Clock Control 1	GCCTRL1	Read/Write	0x00000000
0x068	Generic Clock Control 2	GCCTRL2	Read/Write	0x00000000
0x06C	Generic Clock Control 3	GCCTRL3	Read/Write	0x00000000
0x070	Generic Clock Control 4	GCCTRL4	Read/Write	0x00000000
0x074	Generic Clock Control 5	GCCTRL5	Read/Write	0x00000000
0x0C0	RC Oscillator Calibration Register	RCCR	Read/Write	Factory settings
0x0C4	Bandgap Calibration Register	BGCR	Read/Write	Factory settings
0x0C8	Linear Regulator Calibration Register	VREGCR	Read/Write	Factory settings
0x0D0	BOD Level Register	BOD	Read/Write	BOD fuses in Flash
0x0D4	BOD33 Level Register	BOD33	Read/Write	BOD33 reset enable BOD33 LEVEL=2V7
0x0140	Reset Cause Register	RCAUSE	Read/Write	Latest Reset Source
0x0144	Asynchronous Wake Enable Register	AWEN	Read/Write	0x00000000
0x200	General Purpose Low-Power register	GPLP	Read/Write	0x00000000

Table 8: PM Register Memory Map

3.1.4 Serial peripheral Interface(SPI)

Overview The Serial Peripheral Interface (SPI) circuit is a synchronous serial data link that provides communication with external devices in Master or Slave mode. It also enables communication between processors if an external processor is connected to the system.

The Serial Peripheral Interface is essentially a shift register that serially transmits data bits to other SPIs. During a data transfer, one SPI system acts as the "master" which controls the data flow, while the other devices act as "slaves" which have data shifted into and out by the master. Different CPUs can take turn being masters (Multiple Master Protocol opposite to Single Master Protocol where one CPU is always the master while all of the others are always slaves) and one master may simultaneously shift data into multiple slaves. However, only one slave may drive its output to write data back to the master at any given time.

A slave device is selected when the master asserts its NSS signal. If multiple slave devices exist, the master generates a separate slave select signal for each slave (NPCS).

The SPI system consists of two data lines and two control lines:

Master Out Slave In (MOSI): this data line supplies the output data from the master shifted into the input(s) of the slave(s).

Master In Slave Out (MISO): this data line supplies the output data from a slave to the input of the master. There may be no more than one slave transmitting data during any particular transfer.

Serial Clock (SPCK): this control line is driven by the master and regulates the flow of the data bits. The master may transmit data at a variety of baud rates; the SPCK line cycles once for each bit that is transmitted.

Slave Select (NSS): this control line allows slaves to be turned on and off by hardware.

Block Diagram

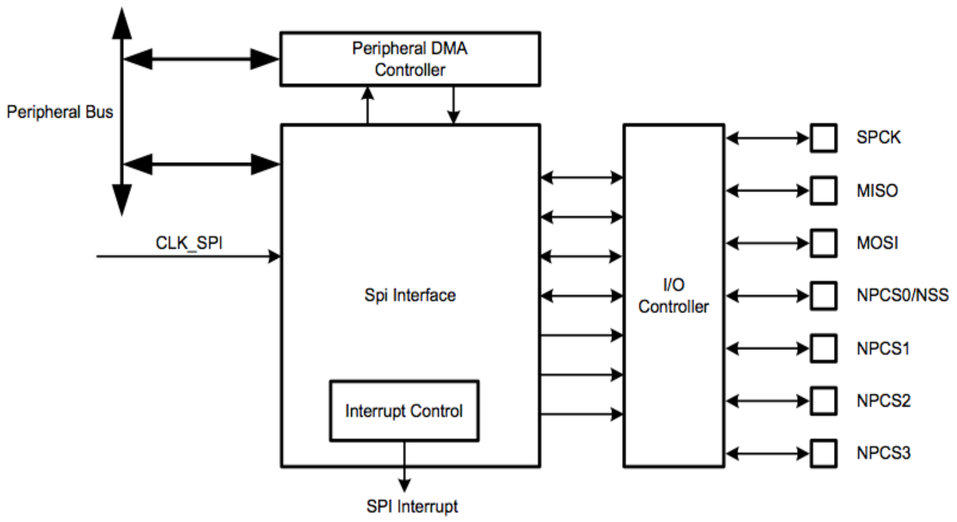


Figure 14: SPI Block Diagram

Application Block Diagram

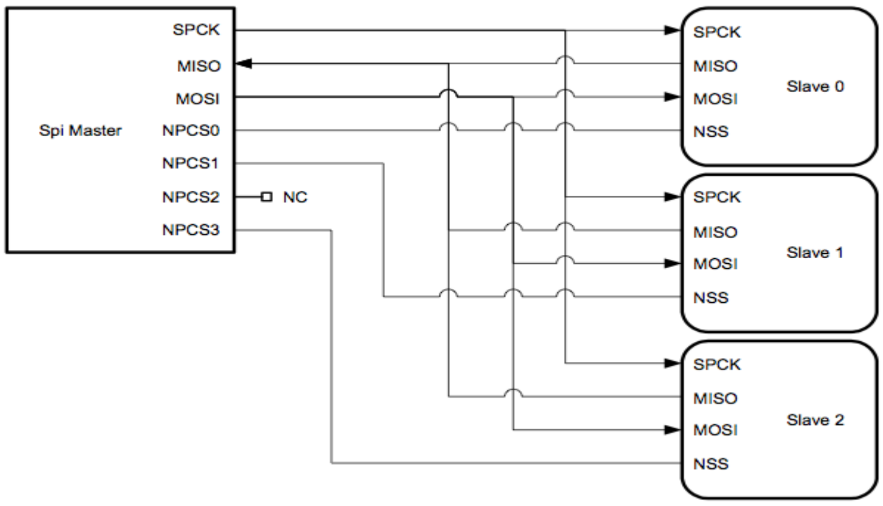
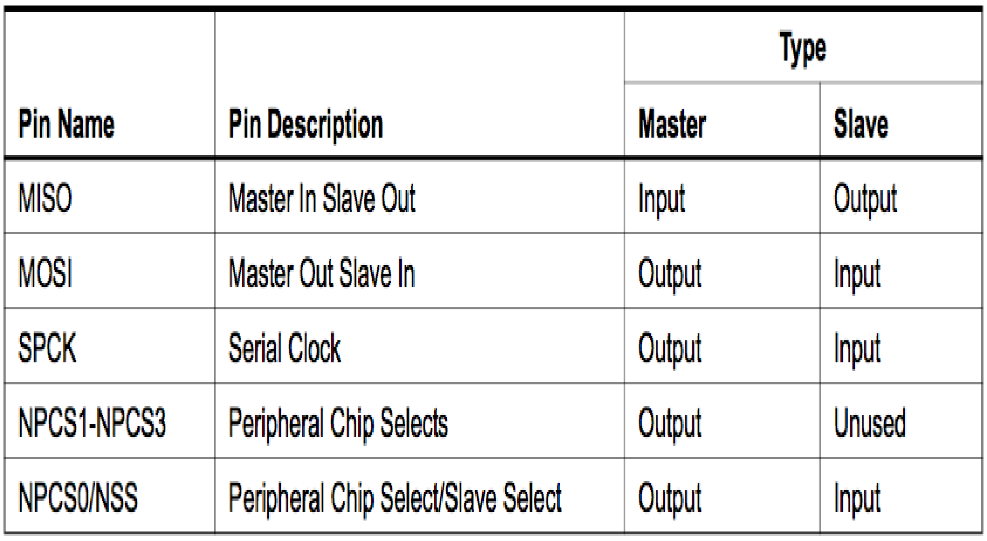


Figure 15: Application Block Diagram: Single Master/Multiple Slave Implementation

I/O Lines Description



Pin Name	Pin Description	Type	
		Master	Slave
MISO	Master In Slave Out	Input	Output
MOSI	Master Out Slave In	Output	Input
SPCK	Serial Clock	Output	Input
NPCS1-NPCS3	Peripheral Chip Selects	Output	Unused
NPCS0/NSS	Peripheral Chip Select/Slave Select	Output	Input

Table 9: I/O Lines Description

Product Dependencies In order to use this module, other parts of the system must be configured correctly, as described below.

I/O Lines The pins used for interfacing the compliant external devices may be multiplexed with I/O lines. The user must first configure the I/O Controller to assign the SPI pins to their peripheral functions.

Clocks The clock for the SPI bus interface (CLK_SPI) is generated by the Power Manager. This clock is enabled at reset, and can be disabled in the Power Manager. It is recommended to disable the SPI before disabling the clock, to avoid freezing the SPI in an undefined state.

Interrupts The SPI interrupt request line is connected to the interrupt controller. Using the SPI interrupt requires the interrupt controller to be programmed first.

Functional Description

Modes of Operation The SPI operates in master mode or in slave mode.

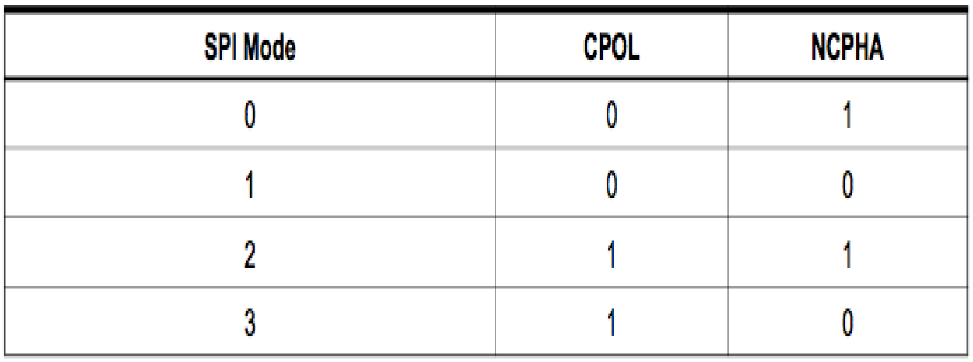
Operation in master mode is configured by writing a one to the Master/Slave Mode bit in the Mode Register (MR.MSTR). The pins NPCS0 to NPCS3 are all configured as outputs, the SPCK pin is driven, the MISO line is wired on the receiver input and the MOSI line driven as an output by the transmitter.

If the MR.MSTR bit is written to zero, the SPI operates in slave mode. The MISO line is driven by the transmitter output, the MOSI line is wired on the receiver input, the SPCK pin is driven by the transmitter to synchronize the receiver. The NPCS0 pin becomes an input, and is used as a Slave Select signal (NSS). The pins NPCS1 to NPCS3 are not driven and can be used for other purposes.

The data transfers are identically programmable for both modes of operations. The baud rate generator is activated only in master mode.

Data Transfer Four combinations of polarity and phase are available for data transfers. The clock polarity is configured with the Clock Polarity bit in the Chip Select Registers (CSRn.CPOL). The clock phase is configured with the Clock Phase bit in the CSRn registers (CSRn.NCPHA). These two bits determine the edges of the clock signal on which data is driven and sampled. Each of the two bits has two possible states, resulting in four possible combinations that are incompatible with one another. Thus, a master/slave pair must use the same parameter pair values to communicate. If multiple slaves are used and fixed in different configurations, the master must reconfigure itself each time it needs to communicate with a different slave.

Table 10 shows the four modes and corresponding parameter settings.



SPI Mode	CPOL	NCPHA
0	0	1
1	0	0
2	1	1
3	1	0

Table 10: SPI modes

Figure 16 and Figure 17 show examples of data transfers.

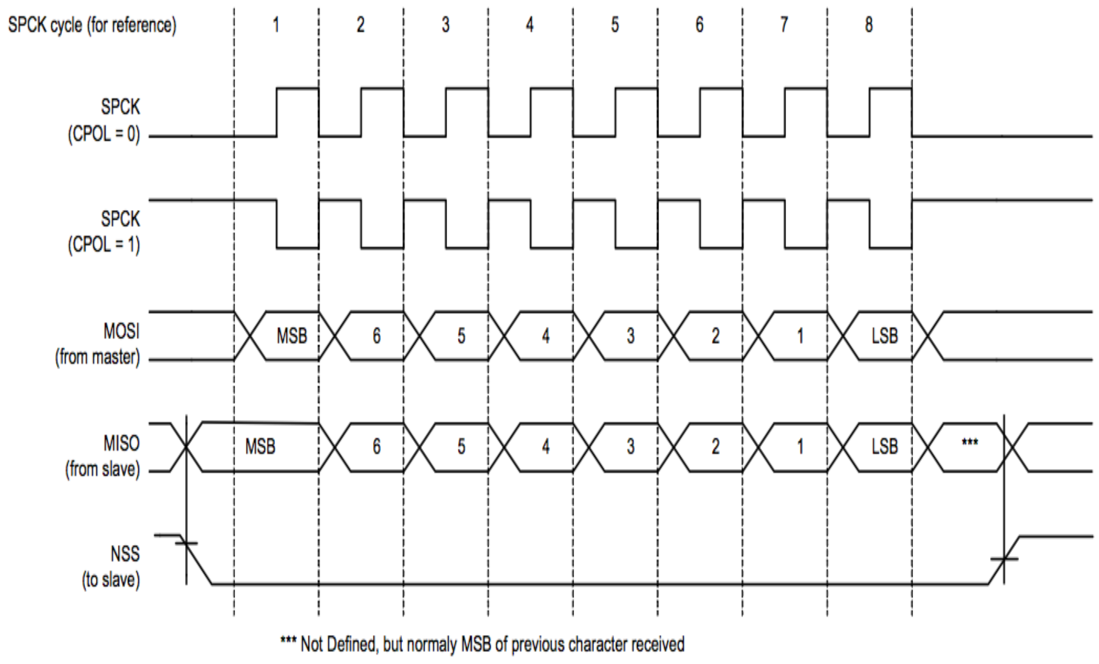


Figure 16: Application Block Diagram: Single Master/Multiple Slave Implementation

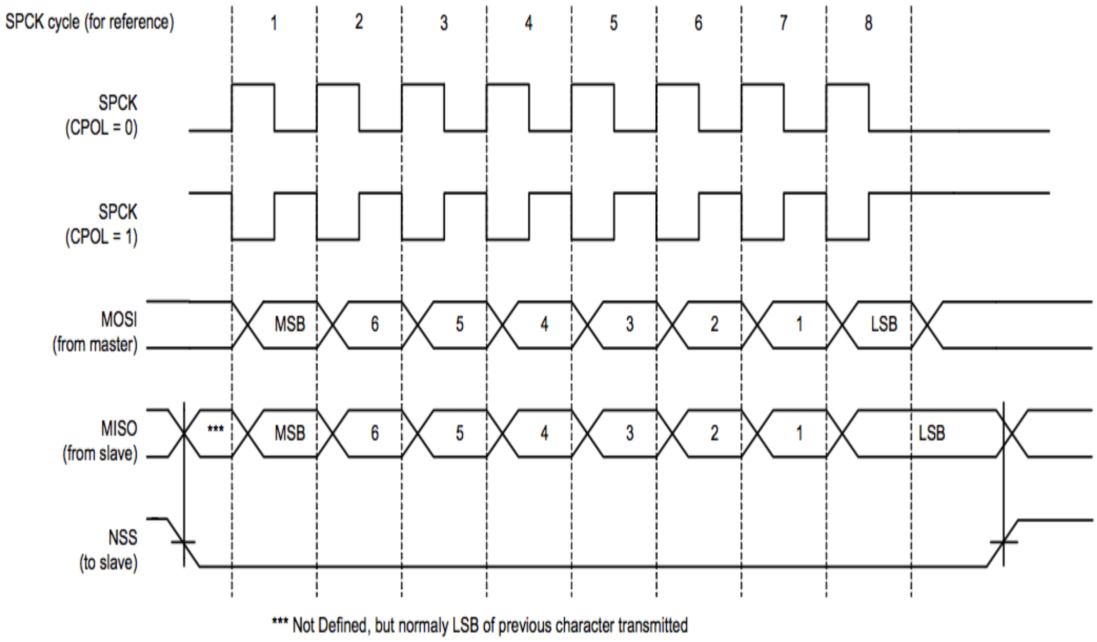


Figure 17: Application Block Diagram: Single Master/Multiple Slave Implementation

Master Mode Operations When configured in master mode, the SPI uses the internal programmable baud rate generator as clock source. It fully controls the data transfers to and from the slave(s) connected to the SPI bus. The SPI drives the chip select line to the slave and the serial clock signal (SPCK).

The SPI features two holding registers, the Transmit Data Register (TDR) and the Receive Data Register (RDR), and a single Shift Register. The holding registers maintain the data flow at a constant rate.

After enabling the SPI, a data transfer begins when the processor writes to the TDR register. The written data is immediately transferred in the Shift Register and transfer on the SPI bus starts. While the data in the Shift Register is shifted on the MOSI line, the MISO line is sampled and shifted in the Shift Register. Transmission cannot occur without reception.

Before writing to the TDR, the Peripheral Chip Select field in TDR (TDR.PCS) must be written in order to select a slave.

If new data is written to TDR during the transfer, it stays in it until the current transfer is completed. Then, the received data is transferred from the Shift Register to RDR, the data in TDR is loaded in the Shift Register and a new transfer starts.

The transfer of a data written in TDR in the Shift Register is indicated by the Transmit Data Register Empty bit in the Status Register (SR.TDRE). When new data is written in TDR, this bit is cleared. The SR.TDRE bit is used to trigger the Transmit Peripheral DMA Controller channel.

The end of transfer is indicated by the Transmission Registers Empty bit in the SR register (SR.TXEMPTY). If a transfer delay (CSRn.DLYBCT) is greater than zero for the last transfer, SR.TXEMPTY is set after the completion of said delay. The CLK_SPI can be switched off at this time.

During reception, received data are transferred from the Shift Register to the reception FIFO. The FIFO can contain up to 4 characters (both Receive Data and Peripheral Chip Select fields). While a character of the FIFO is unread, the Receive Data Register Full bit in SR remains high (SR.RDRF). Characters are read through the RDR register. If the four characters stored in the FIFO are not read and if a new character is stored, this sets the Overrun Error Status bit in the SR register (SR.OVRES). The procedure to follow is described later.

In master mode, if the received data is not read fast enough compared to the transfer rhythm imposed by the write accesses in the TDR, some overrun errors may occur, even if the FIFO is enabled. To insure a perfect data integrity of received data (especially at high data rate), the mode Wait Data Read Before Transfer can be enabled in the MR register (MR.WDRBT). When this mode is activated, no transfer starts while received data remains unread in the RDR. When data is written to the TDR and if unread received data is stored in the RDR, the transfer is paused until the RDR is read. In this mode no overrun error can occur. Please note that if this mode is enabled, it is useless to activate the FIFO in reception.

Figure 18 shows a block diagram of the SPI when operating in master mode. Figure 19 shows a flow chart describing how transfers are handled.

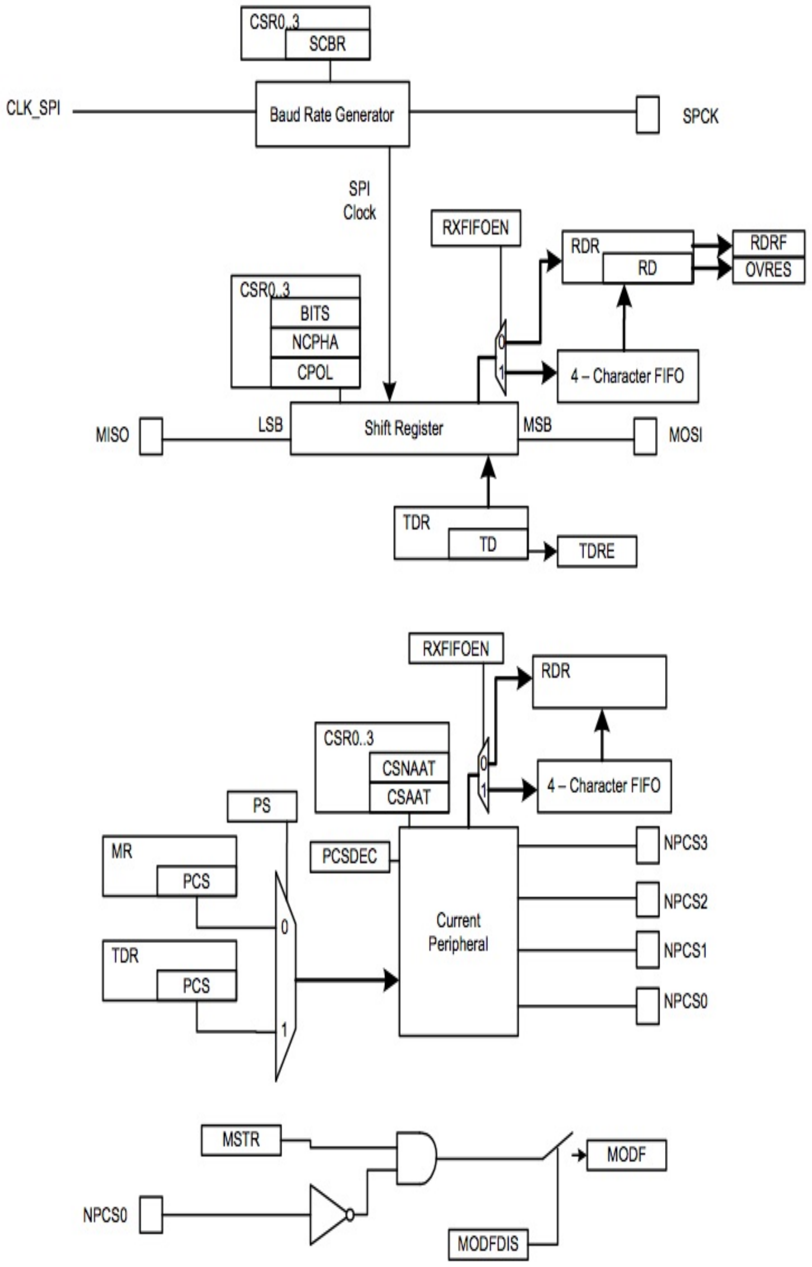


Figure 18: Master Mode Block Diagram

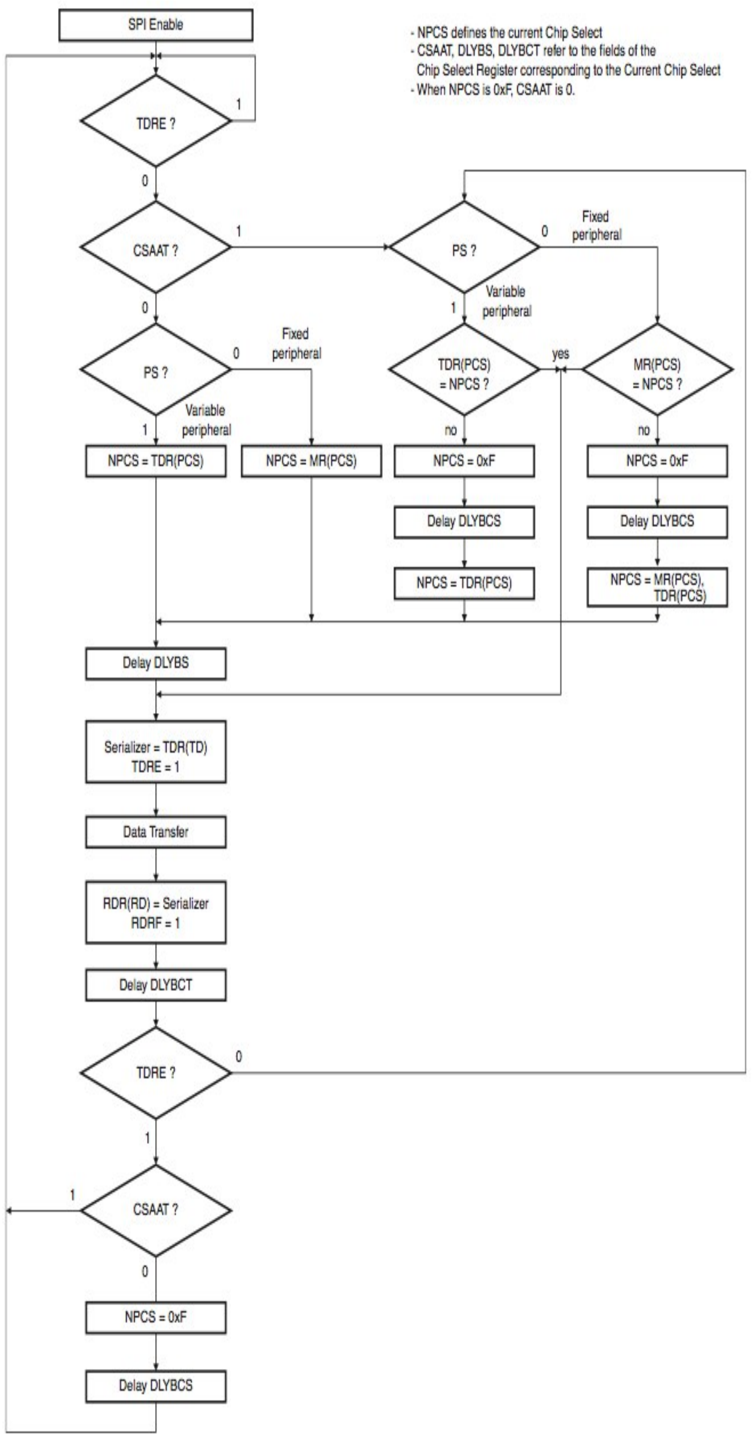


Figure 19: Master Mode Flow Diagram

Clock generation The SPI Baud rate clock is generated by dividing the CLK_SPI , by a value between 1 and 255.

This allows a maximum operating baud rate at up to CLK_SPI and a minimum operating baud rate of CLK_SPI divided by 255.

Writing the Serial Clock Baud Rate field in the CSRn registers (CSRn.SCBR) to zero is forbidden. Triggering a transfer while CSRn.SCBR is zero can lead to unpredictable results.

At reset, CSRn.SCBR is zero and the user has to configure it at a valid value before performing the first transfer.

The divisor can be defined independently for each chip select, as it has to be configured in the CSRn.SCBR field. This allows the SPI to automatically adapt the baud rate for each interfaced peripheral without reprogramming.

Transfer delays Figure 20 shows a chip select transfer change and consecutive transfers on the same chip select. Three delays can be configured to modify the transfer waveforms:

- The delay between chip selects, programmable only once for all the chip selects by writing to the Delay Between Chip Selects field in the MR register (MR.DLYBCS). Allows insertion of a delay between release of one chip select and before assertion of a new one
- The delay before SPCK, independently programmable for each chip select by writing the Delay Before SPCK field in the CSRn registers (CSRn.DLYBS). Allows the start of SPCK to be delayed after the chip select has been asserted.
- The delay between consecutive transfers, independently programmable for each chip select by writing the Delay Between Consecutive Transfers field in the CSRn registers (CSRn.DLYBCT). Allows insertion of a delay between two transfers occurring on the same chip select

These delays allow the SPI to be adapted to the interfaced peripherals and their speed and bus release time.

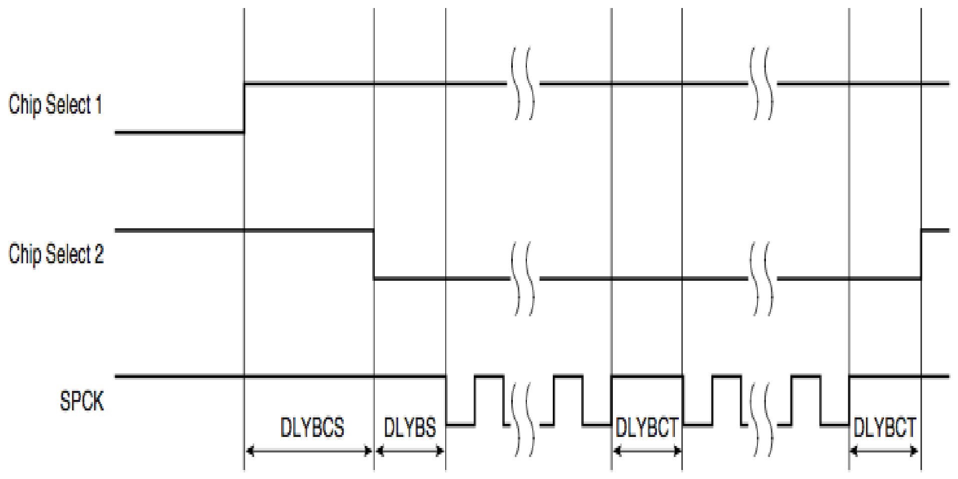


Figure 20: Programmable Delays

Peripheral selection The serial peripherals are selected through the assertion of the NPCS0 to NPCS3 signals. By default, all the NPCS signals are high before and after each transfer.

The peripheral selection can be performed in two different ways:

- Fixed Peripheral Select: SPI exchanges data with only one peripheral
- Variable Peripheral Select: Data can be exchanged with more than one peripheral

Fixed Peripheral Select is activated by writing a zero to the Peripheral Select bit in MR (MR.PS). In this case, the current peripheral is defined by the MR.PCS field and the TDR.PCS field has no effect.

Variable Peripheral Select is activated by writing a one to the MR.PS bit . The TDR.PCS field is used to select the current peripheral. This means that the peripheral selection can be defined for each new data.

The Fixed Peripheral Selection allows buffer transfers with a single peripheral. Using the Peripheral DMA Controller is an optimal means, as the size of the data transfer between the memory and the SPI is either 4 bits or 16 bits. However, changing the peripheral selection requires the Mode Register to be reprogrammed.

The Variable Peripheral Selection allows buffer transfers with multiple peripherals without reprogramming the MR register. Data written to TDR is

32-bits wide and defines the real data to be transmitted and the peripheral it is destined to. Using the Peripheral DMA Controller in this mode requires 32-bit wide buffers, with the data in the LSBs and the PCS and LASTXFER fields in the MSBs, however the SPI still controls the number of bits (8 to16) to be transferred through MISO and MOSI lines with the CSRn registers. This is not the optimal means in term of memory size for the buffers, but it provides a very effective means to exchange data with several peripherals without any intervention of the processor.

Peripheral chip select decoding

The user can configure the SPI to operate with up to 15 peripherals by decoding the four Chip Select lines, NPCS0 to NPCS3 with an external logic. This can be enabled by writing a one to the Chip Select Decode bit in the MR register (MR.PCSDEC).

When operating without decoding, the SPI makes sure that in any case only one chip select line is activated, i.e. driven low at a time. If two bits are defined low in a PCS field, only the lowest numbered chip select is driven low.

When operating with decoding, the SPI directly outputs the value defined by the PCS field of either the MR register or the TDR register (depending on PS).

As the SPI sets a default value of 0xF on the chip select lines (i.e. all chip select lines at one) when not processing any transfer, only 15 peripherals can be decoded.

The SPI has only four Chip Select Registers, not 15. As a result, when decoding is activated, each chip select defines the characteristics of up to four peripherals. As an example, the CRS0 register defines the characteristics of the externally decoded peripherals 0 to 3, corresponding to the PCS values 0x0 to 0x3. Thus, the user has to make sure to connect compatible peripherals on the decoded chip select lines 0 to 3, 4 to 7, 8 to 11 and 12 to 14.

Peripheral deselection

When operating normally, as soon as the transfer of the last data written in TDR is completed, the NPCS lines all rise. This might lead to runtime error if the processor is too long in responding to an interrupt, and thus might lead to difficulties for interfacing with some serial peripherals requiring the chip select line to remain active during a full set of transfers.

To facilitate interfacing with such devices, the CSRn registers can be configured with the Chip Select Active After Transfer bit written to one (CSRn.CSAAT) . This allows the chip select lines to remain in their current state (low = active) until transfer to another peripheral is required.

When the CSRn.CSAAT bit is written to qero, the NPCS does not rise in all cases between two transfers on the same peripheral. During a transfer on a Chip Select, the SR.TDRE bit rises as soon as the content of the TDR is transferred into the internal shifter. When this bit is detected the TDR can be reloaded. If this reload occurs before the end of the current transfer and if the next transfer is performed on the same chip select as the current transfer, the Chip Select is not de-asserted between the two transfers. This might lead to difficulties for interfacing with some serial peripherals requiring the chip select to be de-asserted after each transfer. To facilitate interfacing with such devices, the CSRn registers can be configured with the Chip Select Not Active After Transfer bit (CSRn.CSNAAT) written to one. This allows to de-assert systematically the chip select lines during a time DLYBCS. (The value of the CSRn.CSNAAT bit is taken into account only if the CSRn.CSAAT bit is written to zero for the same Chip Select).

Figure 21 shows different peripheral deselection cases and the effect of the CSRn.CSAAT and CSRn.CSNAAT bits.

FIFO management

A FIFO has been implemented in Reception FIFO (both in master and in slave mode), in order to be able to store up to 4 characters without causing an overrun error. If an attempt is made to store a fifth character, an overrun error rises. If such an event occurs, the FIFO must be flushed. There are two ways to Flush the FIFO:

- By performing four read accesses of the RDR (the data read must be ignored)
- By writing a one to the Flush Fifo Command bit in the CR register (CR.FLUSHFIFO).

After that, the SPI is able to receive new data.

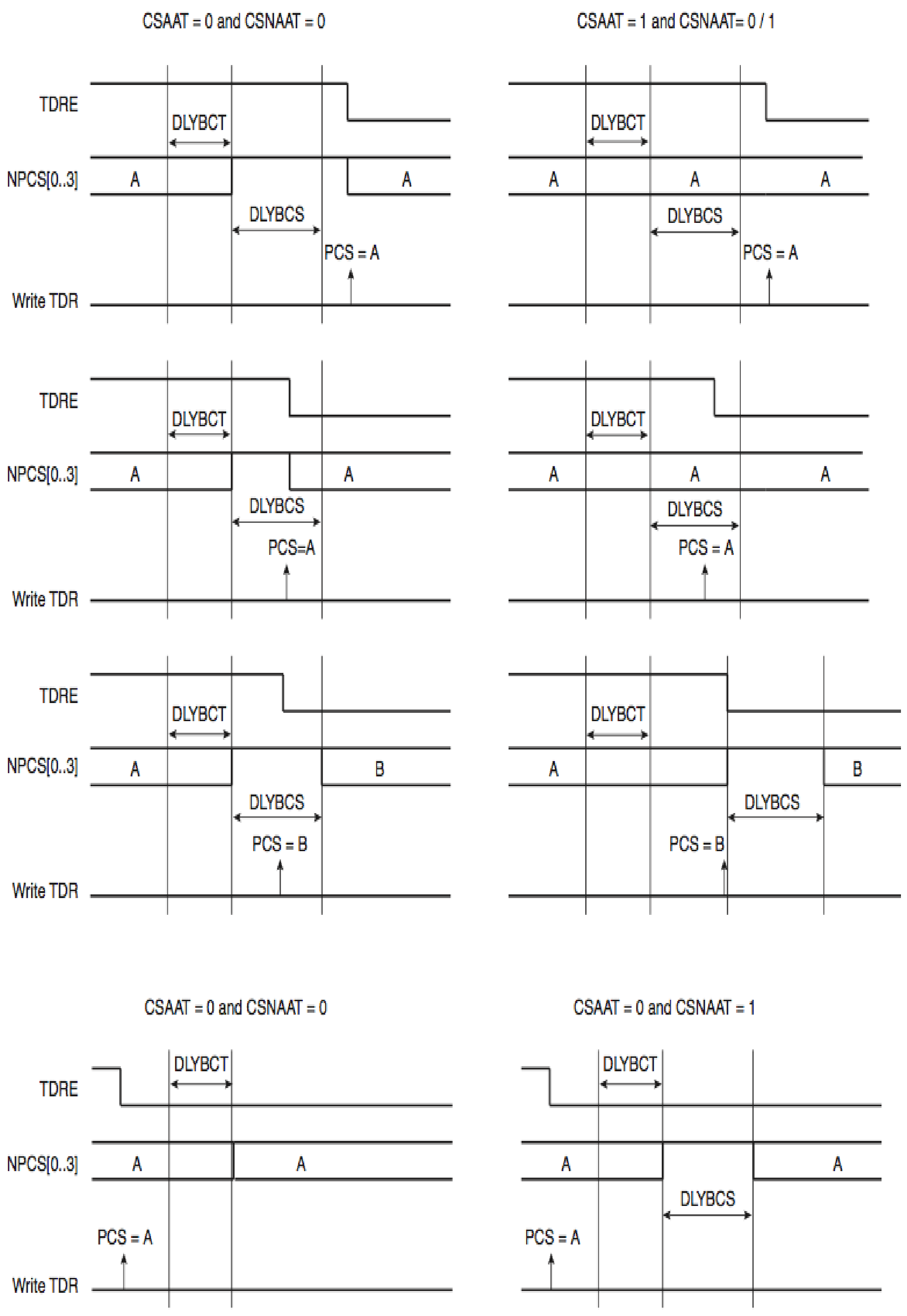


Figure 21: Peripheral Deselection

User Interface

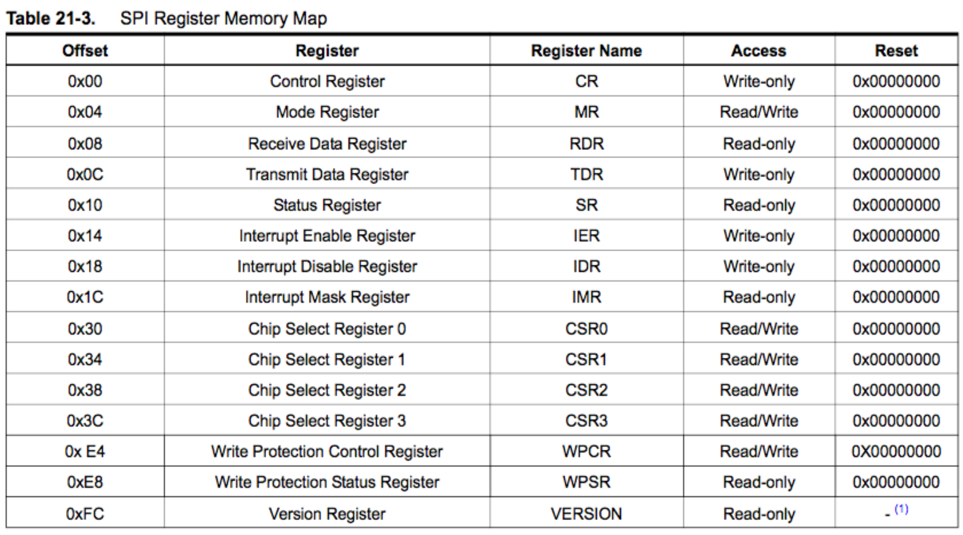


Table 21-3. SPI Register Memory Map

Offset	Register	Register Name	Access	Reset
0x00	Control Register	CR	Write-only	0x00000000
0x04	Mode Register	MR	Read/Write	0x00000000
0x08	Receive Data Register	RDR	Read-only	0x00000000
0x0C	Transmit Data Register	TDR	Write-only	0x00000000
0x10	Status Register	SR	Read-only	0x00000000
0x14	Interrupt Enable Register	IER	Write-only	0x00000000
0x18	Interrupt Disable Register	IDR	Write-only	0x00000000
0x1C	Interrupt Mask Register	IMR	Read-only	0x00000000
0x30	Chip Select Register 0	CSR0	Read/Write	0x00000000
0x34	Chip Select Register 1	CSR1	Read/Write	0x00000000
0x38	Chip Select Register 2	CSR2	Read/Write	0x00000000
0x3C	Chip Select Register 3	CSR3	Read/Write	0x00000000
0x E4	Write Protection Control Register	WPCR	Read/Write	0X00000000
0xE8	Write Protection Status Register	WPSR	Read-only	0x00000000
0xFC	Version Register	VERSION	Read-only	- [(1)]

Table 11: SPI Register Memory Map

In this project we have not used the SPI in slave mode, so we refer for more details to the official datasheet released by ATMEL.

3.1.5 Universal synchronous and Asynchronous Receiver and Transmitter(USART)

Overview The Universal Synchronous Asynchronous Receiver Transceiver (USART) provides a full duplex, universal, synchronous/asynchronous serial link. Data frame format is widely configurable, including basic length, parity, and stop bit settings, maximizing standards support. The receiver implements parity-, framing-, and overrun error detection, and can handle un-fixed frame lengths with the time-out feature. The USART supports several operating modes, providing an interface to RS485, LIN, and SPI buses, with ISO7816 T=0 and T=1 smart card slots, infrared transceivers, and modem port connections. Communication with slow and remote devices is eased by the timeguard. Duplex multidrop communication is supported by address and data differentiation through the parity bit. The hardware handshaking feature enables an out-of-band flow control, automatically managing RTS and CTS pins. The Peripheral DMA Controller connection enables memory transactions, and the USART supports chained buffer management without processor intervention. Automatic echo, remote-, and local loopback -test modes are also supported.

Block Diagram

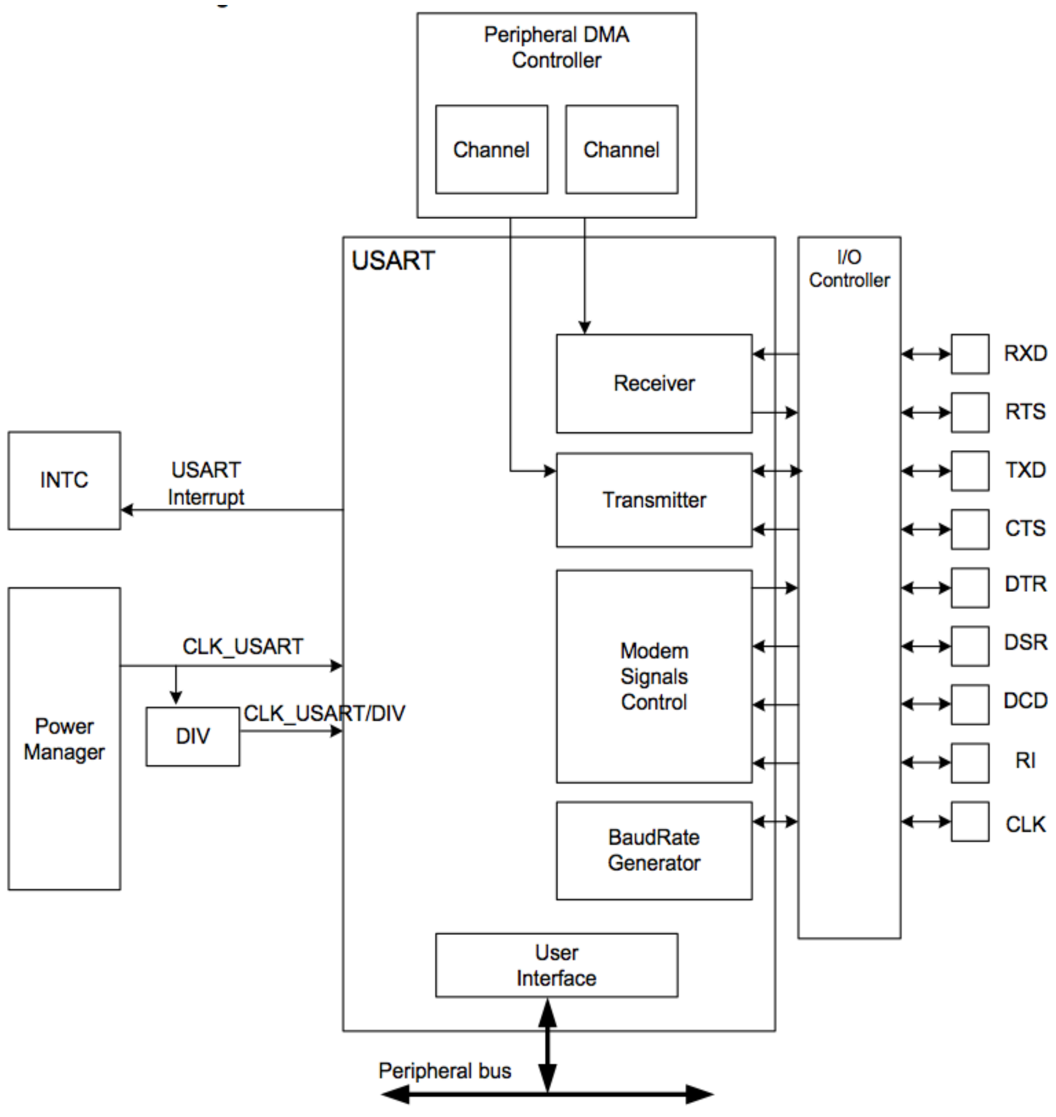
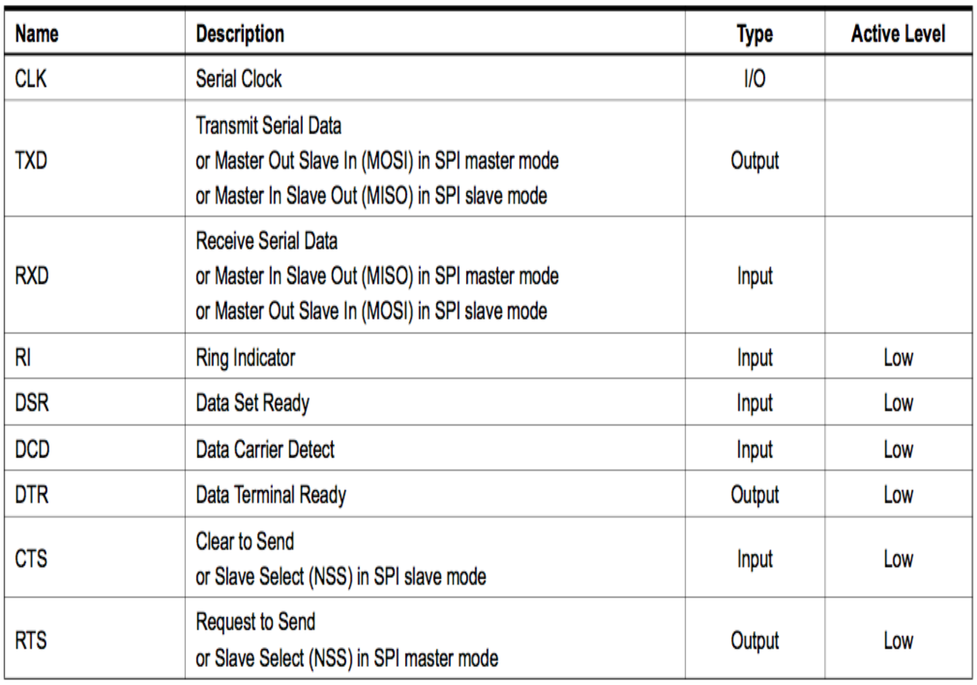


Figure 22: USART Block Diagram

I/O Lines Description



Name	Description	Type	Active Level
CLK	Serial Clock	I/O	
TXD	Transmit Serial Data or Master Out Slave In (MOSI) in SPI master mode or Master In Slave Out (MISO) in SPI slave mode	Output	
RXD	Receive Serial Data or Master In Slave Out (MISO) in SPI master mode or Master Out Slave In (MOSI) in SPI slave mode	Input	
RI	Ring Indicator	Input	Low
DSR	Data Set Ready	Input	Low
DCD	Data Carrier Detect	Input	Low
DTR	Data Terminal Ready	Output	Low
CTS	Clear to Send or Slave Select (NSS) in SPI slave mode	Input	Low
RTS	Request to Send or Slave Select (NSS) in SPI master mode	Output	Low

Table 12: I/O Lines Description

Product Dependencies

I/O Lines The USART pins may be multiplexed with the I/O Controller lines. The user must first configure the I/O Controller to assign these pins to their peripheral functions. Unused I/O lines may be used for other purposes.

To prevent the TXD line from falling when the USART is disabled, the use of an internal pull up is required. If the hardware handshaking feature or modem mode is used, the internal pull up on TXD must also be enabled.

All the pins of the modems may or may not be implemented on the USART. On USARTs not equipped with the corresponding pins, the associated control bits and statuses have no effect on the behavior of the USART.

Clocks The clock for the USART bus interface (CLK_USART) is generated by the Power Manager. This clock is enabled at reset, and can be disabled in the Power Manager. It is recommended to disable the USART before disabling the clock, to avoid freezing the USART in an undefined state.

Interrupts The USART interrupt request line is connected to the interrupt controller. Using the USART interrupt requires the interrupt controller to be programmed first.

Functional Description

Baud Rate Generator The baud rate generator provides the bit period clock named the Baud Rate Clock to both receiver and transmitter. It is based on a 16-bit divider, which is specified in the Clock Divider field in the Baud Rate Generator Register (BRGR.CD). A non-zero value enables the generator, and if CD is one, the divider is bypassed and inactive. The Clock Selection field in the Mode Register (MR.USCLKS) selects clock source between:

- CLK_USART (internal clock)
- CLK_USART/DIV (a divided CLK_USART, refer to Module Configuration section)
- CLK (external clock, available on the CLK pin)

If the external CLK clock is selected, the duration of the low and high levels of the signal provided on the CLK pin must be at least 4.5 times longer than those provided by CLK_USART.

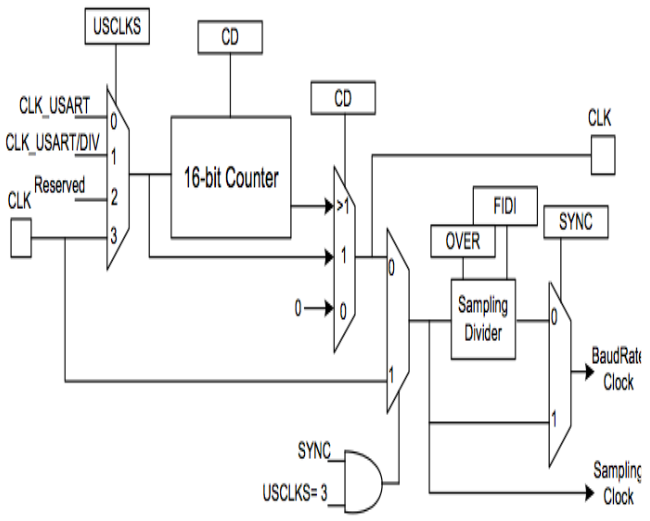


Figure 23: Baud Rate Generator

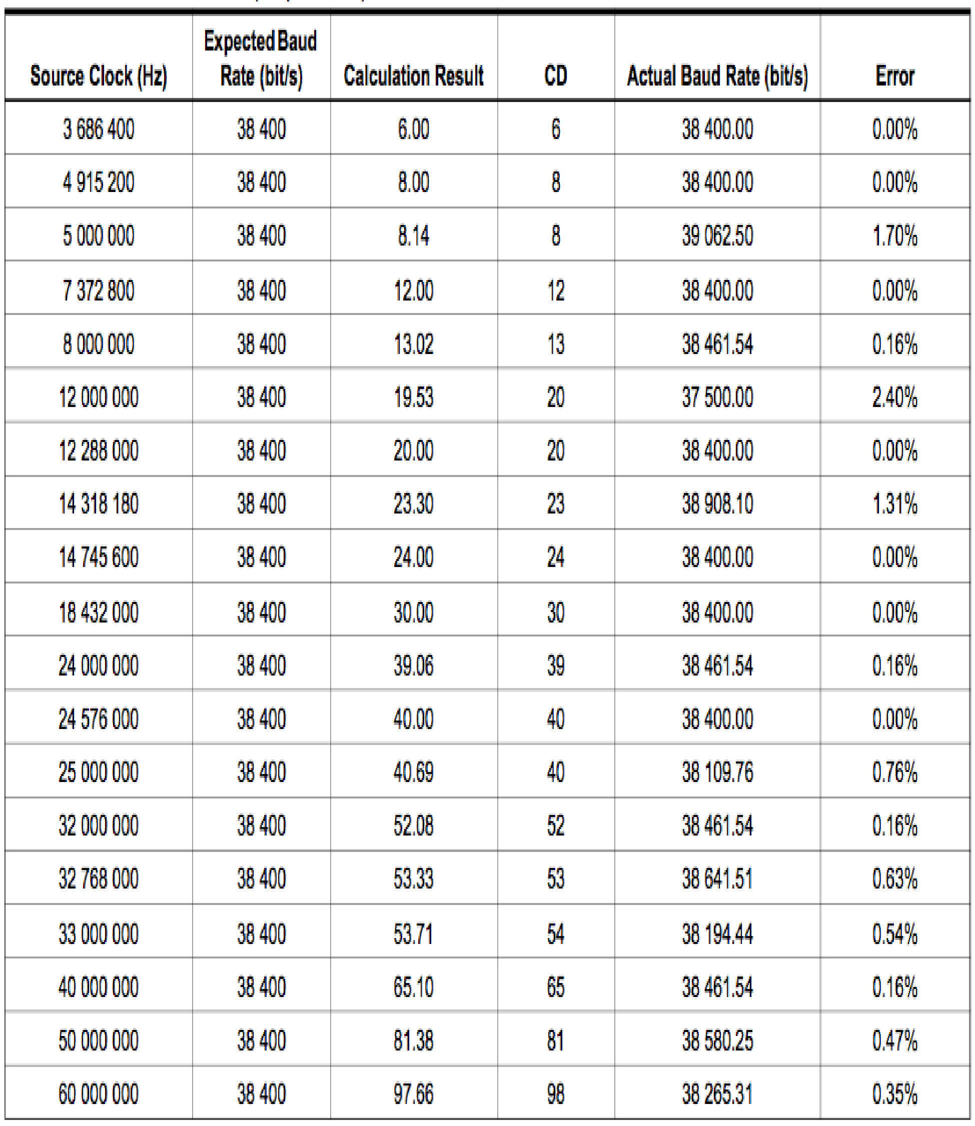
Baud Rate in Asynchronous Mode If the USART is configured to operate in an asynchronous mode, the selected clock is divided by the CD value before it is provided to the receiver as a sampling clock. Depending on the Over- sampling Mode bit (MR.OVER) value, the clock is then divided by either 8 (OVER=1), or 16 (OVER=0). The baud rate is calculated with the following formula:

$BaudRate = SelectedClock/(8(2–OVER)CD)$

This gives a maximum baud rate of CLK_USART divided by 8, assuming that CLK_USART is the fastest clock possible, and that OVER is one.

Baud Rate Calculation Example:

Table 13 shows calculations based on the CD field to obtain 38400 baud from different source clock frequencies. This table also shows the actual resulting baud rate and error.



Source Clock (Hz)	Expected Baud Rate (bit/s)	Calculation Result	CD	Actual Baud Rate (bit/s)	Error
3 686 400	38 400	6.00	6	38 400.00	0.00%
4 915 200	38 400	8.00	8	38 400.00	0.00%
5 000 000	38 400	8.14	8	39 062.50	1.70%
7 372 800	38 400	12.00	12	38 400.00	0.00%
8 000 000	38 400	13.02	13	38 461.54	0.16%
12 000 000	38 400	19.53	20	37 500.00	2.40%
12 288 000	38 400	20.00	20	38 400.00	0.00%
14 318 180	38 400	23.30	23	38 908.10	1.31%
14 745 600	38 400	24.00	24	38 400.00	0.00%
18 432 000	38 400	30.00	30	38 400.00	0.00%
24 000 000	38 400	39.06	39	38 461.54	0.16%
24 576 000	38 400	40.00	40	38 400.00	0.00%
25 000 000	38 400	40.69	40	38 109.76	0.76%
32 000 000	38 400	52.08	52	38 461.54	0.16%
32 768 000	38 400	53.33	53	38 641.51	0.63%
33 000 000	38 400	53.71	54	38 194.44	0.54%
40 000 000	38 400	65.10	65	38 461.54	0.16%
50 000 000	38 400	81.38	81	38 580.25	0.47%
60 000 000	38 400	97.66	98	38 265.31	0.35%

Table 13: Baud Rate Example (OVER=0)

The baud rate is calculated with the following formula (OVER=0):

$BaudRate = CLKUSART/(CD \times 16)$

The baud rate error is calculated with the following formula. It is not recommended to work with an error higher than 5%.

$Error = 1 - (\frac{ExpectedBaudRate}{ActualBaudRate})$

Fractional Baud Rate in Asynchronous Mode

The baud rate generator has a limitation: the source frequency is always a multiple of the baud rate. An approach to this problem is to integrate a high resolution fractional N clock generator, outputting fractional multiples of the reference source clock. This fractional part is selected with the Fractional Part field (BRGR.FP), and is activated by giving it a non-zero value. The resolution is one eighth of CD. The resulting baud rate is calculated using the following formula:

$BaudRate = \frac{SelectedClock}{8(2-OVER)(CD+\frac{FP}{8})}$

The modified architecture is presented below:

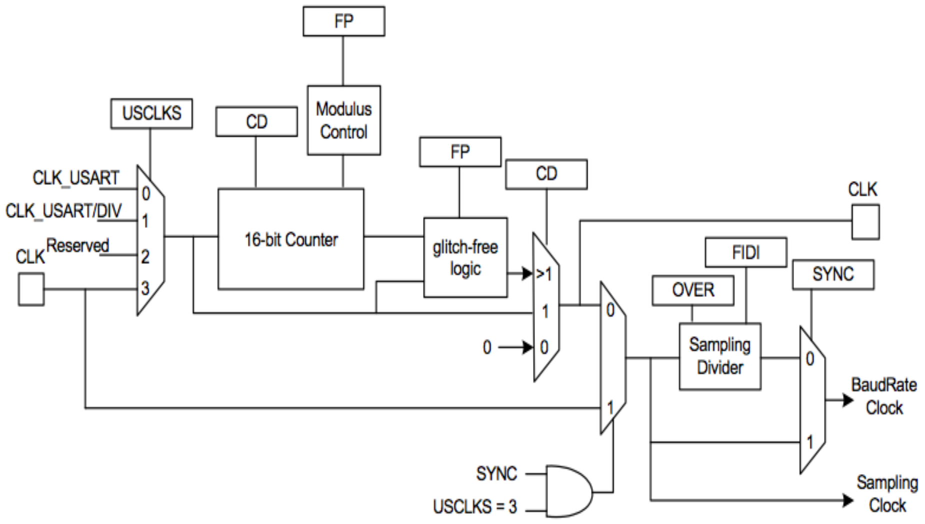


Figure 24: Fractional Baud Rate Generator)

Receiver and Transmitter Control After a reset, the transceiver is disabled. The receiver/transmitter is enabled by writing a one to either the Receiver Enable, or Transmitter Enable bit in the Control Register (CR.RXEN, or CR.TXEN). They may be enabled together and can be configured both before and after they have been enabled. The user can reset the USART receiver/transmitter at any time by writing a one to either the Reset Receiver (CR.RSTRX), or Reset Transmitter (CR.RSTTX) bit. This soft- ware reset clears status bits and resets internal state machines, immediately halting any communication. The user interface configuration registers will retain their values.

The user can disable the receiver/transmitter by writing a one to either the Receiver Disable, or Transmitter Disable bit (CR.RXDIS, or CR.TXDIS). If the receiver is disabled during a character reception, the USART will wait for the current character to be received before disabling. If the transmitter is disabled during transmission, the USART will wait until both the current character and the character stored in the Transmitter Holding Register (THR) are transmitted before dis- abling. If a timeguard has been implemented it will remain functional during the transaction.

Synchronous and Asynchronous Modes

Transmitter Operations The transmitter performs equally in both synchronous and asynchronous operating modes (MR.SYNC). One start bit, up to 9 data bits, an optional parity bit, and up to two stop bits are successively shifted out on the TXD pin at each falling edge of the serial clock. The number of data bits is selected by the Character Length field (MR.CHRL) and the MR.MODE9 bit. Nine bits are selected by writing a one to MODE9, overriding any value in CHRL. The parity bit configura- tion is selected in the MR.PAR field. The Most Significant Bit First bit (MR.MSBF) selects which data bit to send first. The number of stop bits is selected by the MR.NBSTOP field. The 1.5 stop bit configuration is only supported in asynchronous mode.

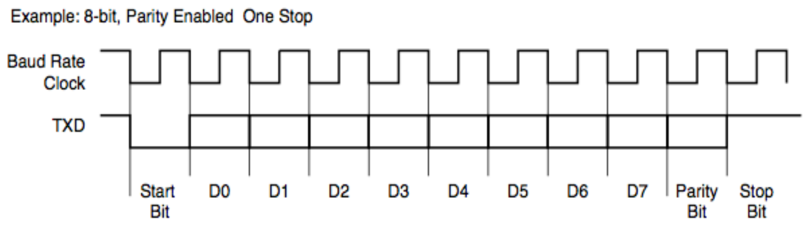


Figure 25: Character Transmit

The characters are sent by writing to the Character to be Transmitted field (THR.TXCHR). The transmitter reports status with the Transmitter Ready (TXRDY) and Transmitter Empty (TXEMPTY) bits in the Channel Status Register (CSR). TXRDY is set when THR is empty. TXEMPTY is set when both THR and the transmit shift register are empty (transmission complete). Both TXRDY and TXEMPTY are cleared when the transmitter is disabled. Writing a character to THR while TXRDY is zero has no effect and the written character will be lost.

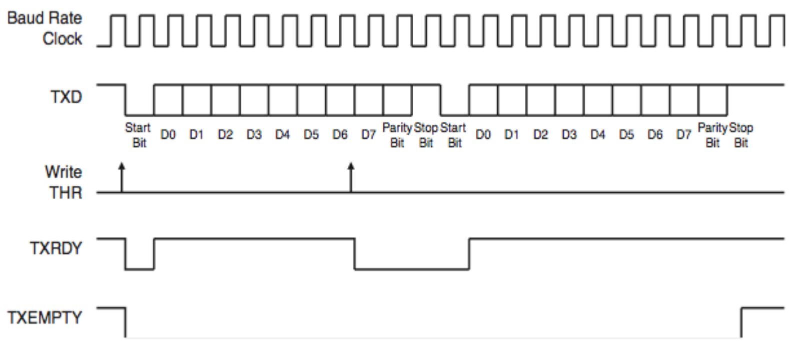


Figure 26: Transmitter Status

Asynchronous Receiver If the USART is configured in an asynchronous operating mode (MR.SYNC = 0), the receiver will oversample the RXD input line by either 8 or 16 times the baud rate clock, as selected by the Oversampling Mode bit (MR.OVER). If the line is zero for half a bit period (four or eight consecutive samples, respectively), a start bit will be assumed, and the following 8th or 16th sample will determine the logical value on the line, in effect resulting in bit values being determined at the middle of the bit period.

The number of data bits, endianess, parity mode, and stop bits are selected by the same bits and fields as for the transmitter (MR.CHRL, MODE9, MSBF, PAR, and NBSTOP). The synchronization mechanism will only consider one stop bit, regardless of the used protocol, and when the first stop bit has been sampled, the receiver will automatically begin looking for a new start bit, enabling resynchronization even if there is a protocol miss-match. Figure 25-11 and Figure 25-12 illustrate start bit detection and character reception in asynchronous mode.

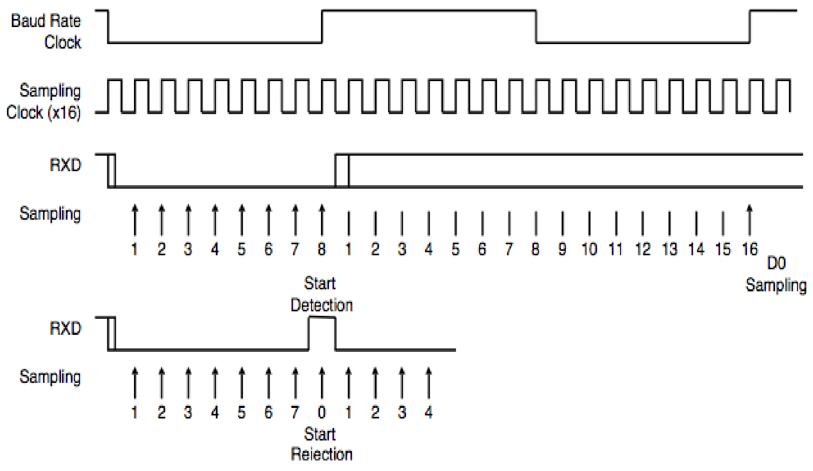


Figure 27: Asynchronous Start Bit Detection

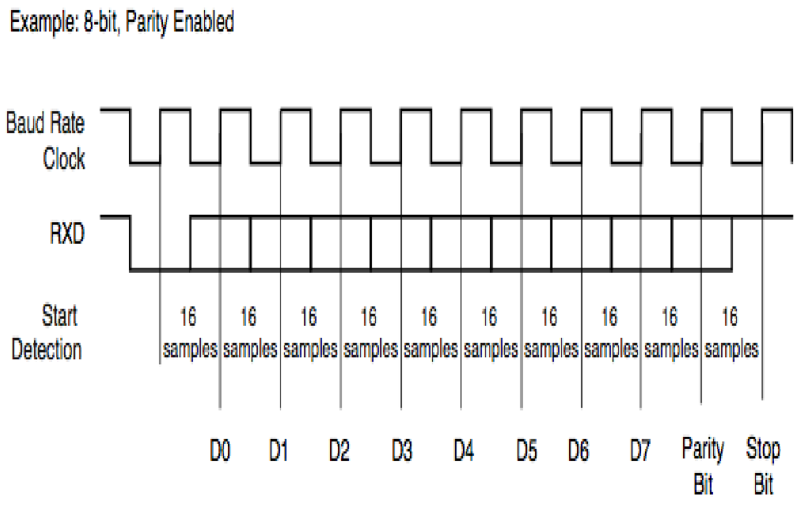
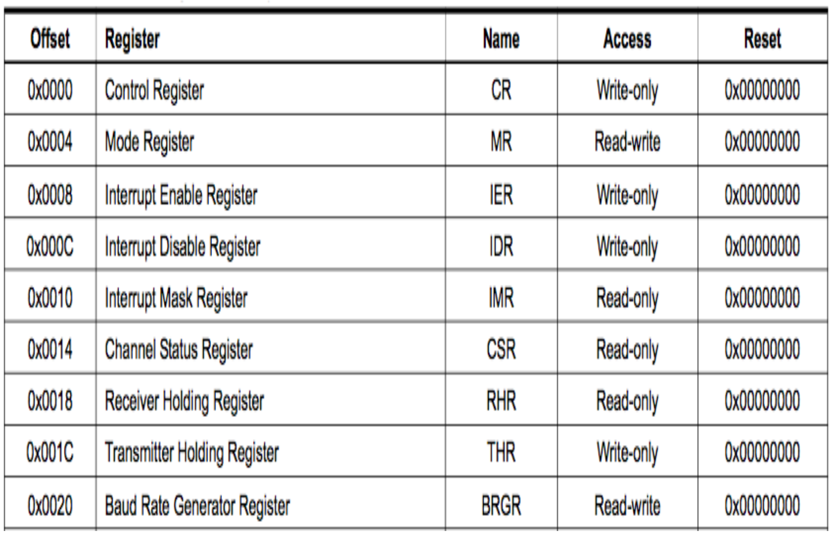


Figure 28: Asynchronous Character Reception

User Interface



Offset	Register	Name	Access	Reset
0x0000	Control Register	CR	Write-only	0x00000000
0x0004	Mode Register	MR	Read-write	0x00000000
0x0008	Interrupt Enable Register	IER	Write-only	0x00000000
0x000C	Interrupt Disable Register	IDR	Write-only	0x00000000
0x0010	Interrupt Mask Register	IMR	Read-only	0x00000000
0x0014	Channel Status Register	CSR	Read-only	0x00000000
0x0018	Receiver Holding Register	RHR	Read-only	0x00000000
0x001C	Transmitter Holding Register	THR	Write-only	0x00000000
0x0020	Baud Rate Generator Register	BRGR	Read-write	0x00000000

Table 14: USART Register Memory Map

3.1.6 Hi-Speed USB Interface (USBB)

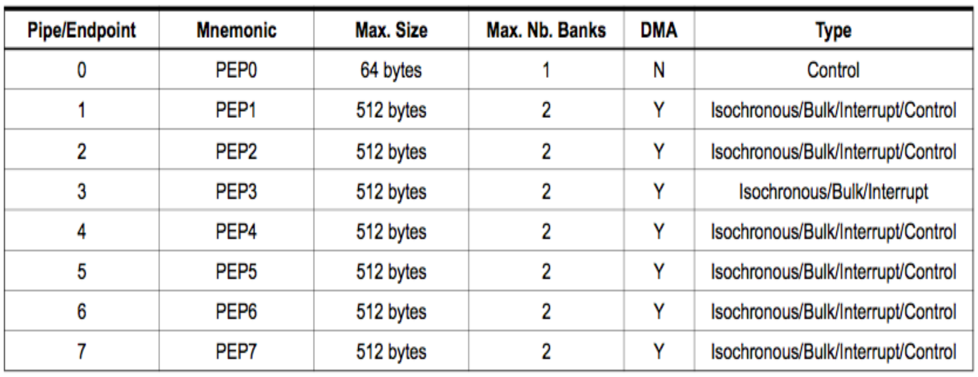
Features

- Compatible with the USB 2.0 specification
- Supports High (480Mbit/s), Full (12Mbit/s) and Low (1.5Mbit/s) speed Device and Embedded Host
- eight pipes/endpoints
- 2368bytes of Embedded Dual-Port RAM (DPRAM) for Pipes/Endpoints
- Up to 2 memory banks per Pipe/Endpoint (Not for Control Pipe/Endpoint)
- Flexible Pipe/Endpoint configuration and management with dedicated DMA channels
- On-Chip UTMI transceiver including Pull-Ups/Pull-downs
- On-Chip pad including VBUS analog comparator

Overview The Universal Serial Bus (USB) MCU device complies with the Universal Serial Bus (USB) 2.0 specification, in all speeds.

Each pipe/endpoint can be configured in one of several transfer types. It can be associated with one or more banks of a dual-port RAM (DPRAM) used to store the current data payload. If several banks are used ("ping-pong" mode), then one DPRAM bank is read or written by the CPU or the DMA while the other is read or written by the USBB core. This feature is mandatory for isochronous pipes/endpoints.

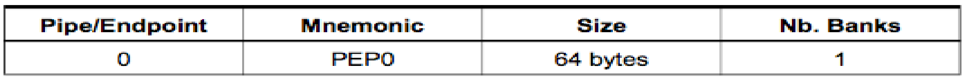
Table 15 describes the hardware configuration of the USB MCU device.



Pipe/Endpoint	Mnemonic	Max. Size	Max. Nb. Banks	DMA	Type
0	PEP0	64 bytes	1	N	Control
1	PEP1	512 bytes	2	Y	Isochronous/Bulk/Interrupt/Control
2	PEP2	512 bytes	2	Y	Isochronous/Bulk/Interrupt/Control
3	PEP3	512 bytes	2	Y	Isochronous/Bulk/Interrupt
4	PEP4	512 bytes	2	Y	Isochronous/Bulk/Interrupt/Control
5	PEP5	512 bytes	2	Y	Isochronous/Bulk/Interrupt/Control
6	PEP6	512 bytes	2	Y	Isochronous/Bulk/Interrupt/Control
7	PEP7	512 bytes	2	Y	Isochronous/Bulk/Interrupt/Control

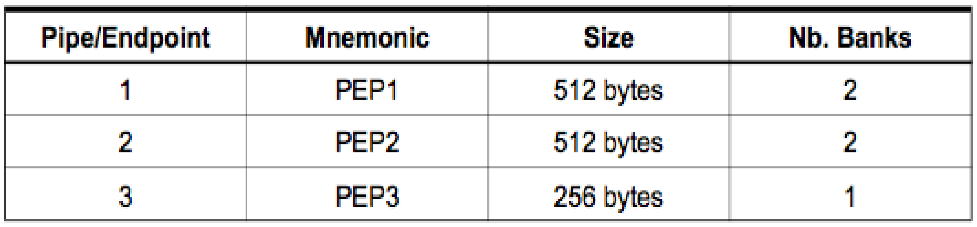
Table 15: Description of USB Pipes/Endpoints

The theoretical maximal pipe/endpoint configuration (3648bytes) exceeds the real DPRAM size (2368bytes). The user needs to be aware of this when configuring pipes/endpoints. To fully use the 2368bytes of DPRAM, the user could for example use the configuration described in Table 16 and 17.



Pipe/Endpoint	Mnemonic	Size	Nb. Banks
0	PEP0	64 bytes	1

Table 16: Example of Configuration of Pipes/Endpoints Using the Whole DPRAM



Pipe/Endpoint	Mnemonic	Size	Nb. Banks
1	PEP1	512 bytes	2
2	PEP2	512 bytes	2
3	PEP3	256 bytes	1

Table 17: Example of Configuration of Pipes/Endpoints Using the Whole DPRAM

Block Diagram The USBB provides a hardware device to interface a USB link to a data flow stored in a dual-port RAM (DPRAM).

The UTMI transceiver requires an external 12MHz clock as a reference to its internal 480MHz PLL. The internal 480MHz PLL is used to clock an internal DLL module to recover the USB differential data at 480Mbit/s.

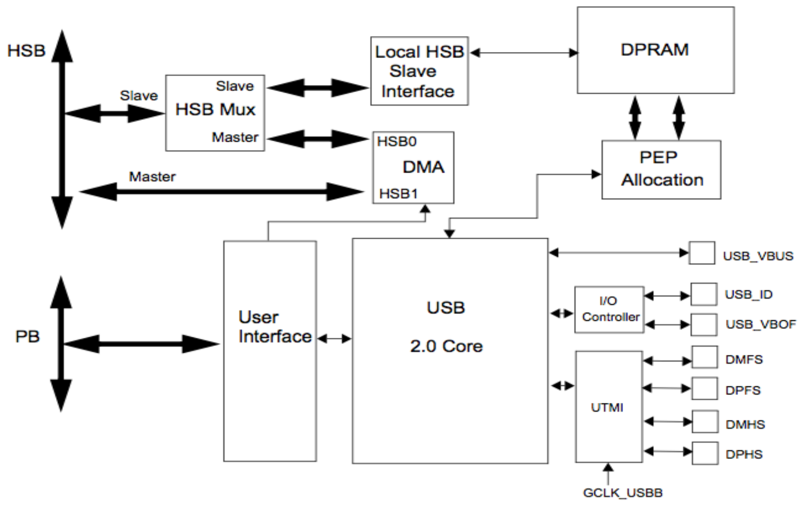


Figure 29: USBB Block Diagram

Application Block Diagram Depending on the USB operating mode (device-only, reduced-host modes) and the power source (bus-powered or self-powered), there are different typical hardware implementations.

Device Mode

Self-Powered device

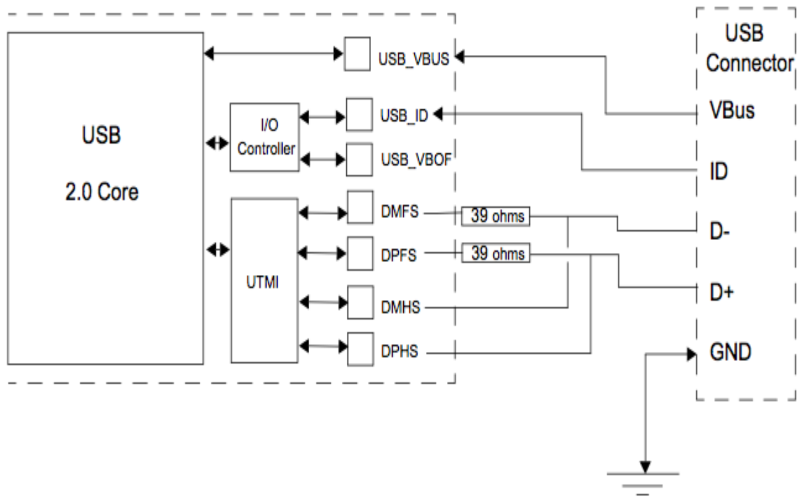
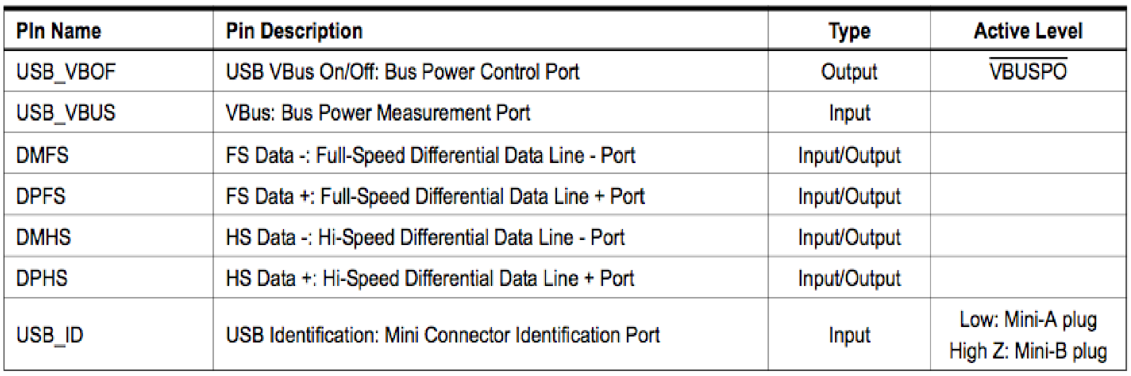


Figure 30: Self-powered Device Application Block Diagram

I/O Lines Description



PIn Name	Pin Description	Type	Active Level
USB_VBOF	USB VBus On/Off: Bus Power Control Port	Output	$\overline{\text{VBUSPO}}$
USB_VBUS	VBus: Bus Power Measurement Port	Input	
DMFS	FS Data -: Full-Speed Differential Data Line - Port	Input/Output	
DPFS	FS Data +: Full-Speed Differential Data Line + Port	Input/Output	
DMHS	HS Data -: Hi-Speed Differential Data Line - Port	Input/Output	
DPHS	HS Data +: Hi-Speed Differential Data Line + Port	Input/Output	
USB_ID	USB Identification: Mini Connector Identification Port	Input	Low: Mini-A plug High Z: Mini-B plug

Table 18: I/O Lines Description

Product Dependencies In order to use this module, other parts of the system must be configured correctly, as described below.

I/O Lines The USB_VBOF and USB_ID pins are multiplexed with I/O Controller lines and may also be multiplexed with lines of other peripherals. In order to use them with the USB, the user must first configure the I/O Controller to assign them to their USB peripheral functions.

If USB_ID is used, the I/O Controller must be configured to enable the internal pull-up resistor of its pin.

If USB_VBOF or USB_ID is not used by the application, the corresponding pin can be used for other purposes by the I/O Controller or by other peripherals.

Clocks The clock for the USBB bus interface (CLK_USBB) is generated by the Power Manager. This clock is enabled at reset, and can be disabled in the Power Manager. It is recommended to disable the USBB before disabling the clock, to avoid freezing the USBB in an undefined state.

The UTMI transceiver needs a 12MHz clock as a clock reference for its internal 480MHz PLL. Before using the USB, the user must ensure that this 12MHz clock is available.

The 12MHz input is connected to a Generic Clock (GCLK_USBB) provided by the Power Manager.

Interrupts The USBB interrupt request line is connected to the interrupt controller. Using the USBB interrupt requires the interrupt controller to be programmed first.

Functional Description

USB General Operation Introduction

After a hardware reset, the USBB is disabled. When enabled, the USBB runs either in device mode or in host mode according to the ID detection.

If the USB_ID pin is not connected to ground, the USB_ID Pin State bit in the General Status register (USBSTA.ID) is set (the internal pull-up resistor of the USB_ID pin must be enabled by the I/O Controller) and device mode is engaged.

The USBSTA.ID bit is cleared when a low level has been detected on the USB_ID pin. Host mode is then engaged.

Power-On and reset

Figure 31 describes the USBB main states.

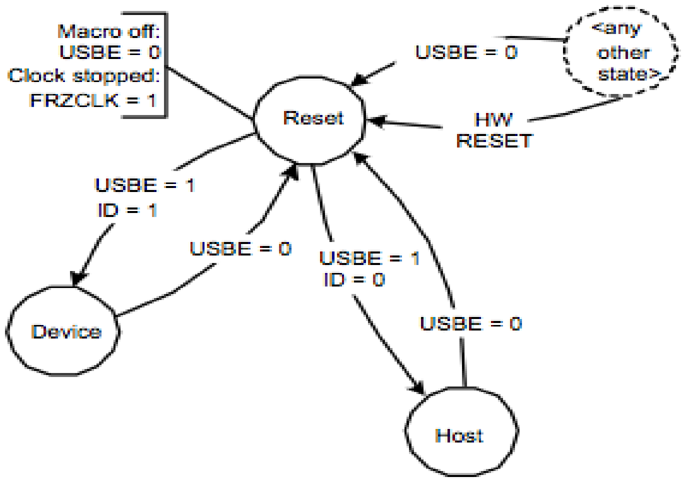


Figure 31: General States

After a hardware reset, the USBB is in the Reset state. In this state:

- The macro is disabled. The USBB Enable bit in the General Control register (USBCON.USBE) is zero.

- The macro clock is stopped in order to minimize power consumption. The Freeze USB Clock bit in USBCON (USBON.FRZCLK) is set.
- The UTMI is in suspend mode.
- The internal states and registers of the device and host modes are reset.
- The DPRAM is not cleared and is accessible.
- The USBSTA.ID bit and the VBus Level bit in the UBSTA (UB-STA.VBUS) reflect the states of the USB_ID and USB_VBUS input pins.
- The OTG Pad Enable (OTGPADE) bit, the VBus Polarity (VBUSPO) bit, the FRZCLK bit, the USBE bit, the USB_ID Pin Enable (UIDE) bit, the USBB Mode (UIMOD) bit in USBCON, and the Low-Speed Mode Force bit in the Device General Control (UDCON.LS) register can be written by software, so that the user can program pads and speed before enabling the macro, but their value is only taken into account once the macro is enabled and unfrozen.

After writing a one to USBCON.USBE, the USBB enters the Device or the Host mode (according to the ID detection) in idle state.

The USBB can be disabled at any time by writing a zero to USBCON.USBE. In fact, writing a zero to USBCON.USBE acts as a hardware reset, except that the OTGPADE, VBUSPO, FRZCLK, UIDE, UIMOD and, LS bits are not reset.

Interrupts

One interrupt vector is assigned to the USB interface. Figure 32 shows the structure of the USB interrupt system.

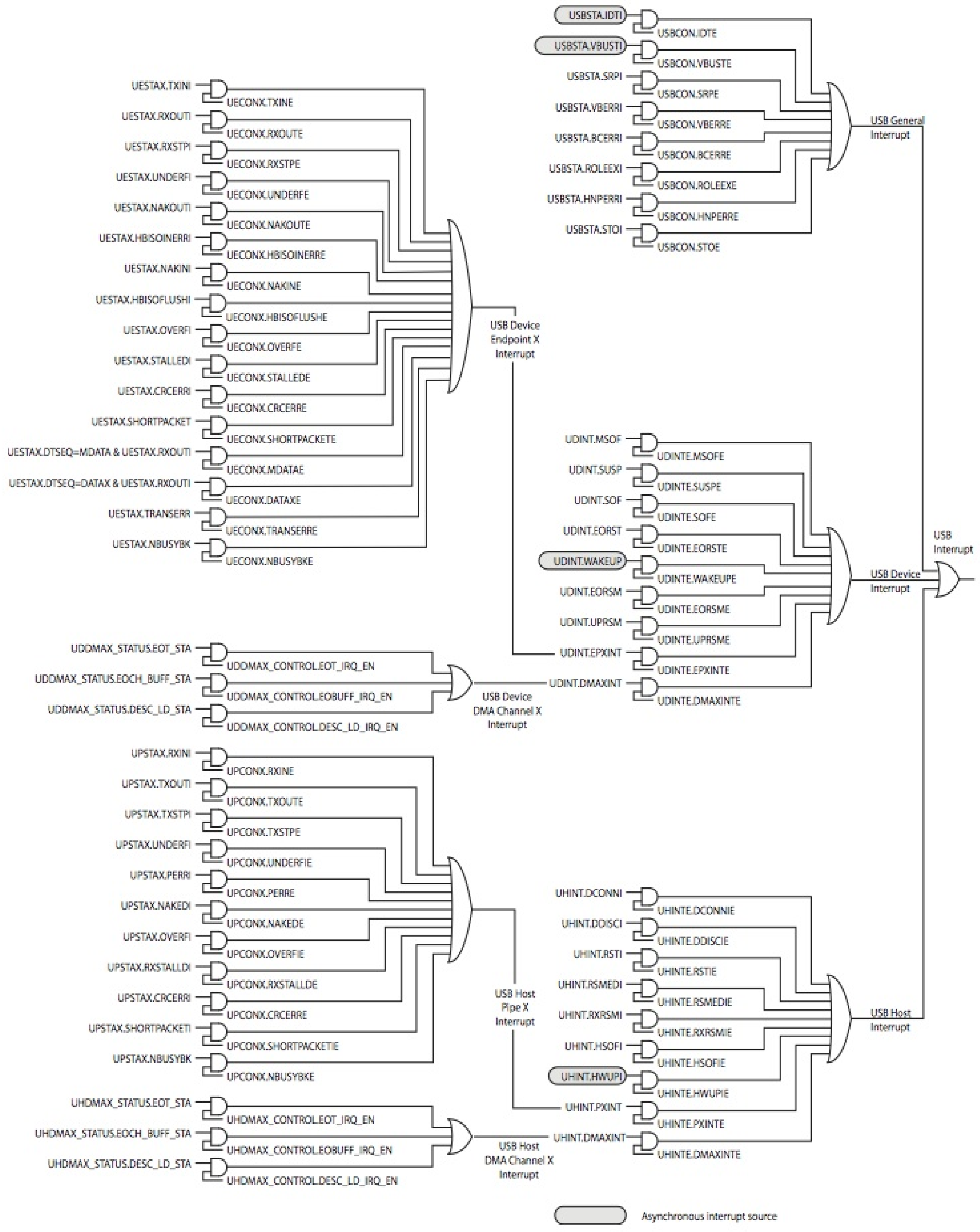


Figure 32: Interrupt System

There are two kinds of general interrupts: processing, i.e. their generation is part of the normal processing, and exception, i.e. errors (not related to CPU exceptions).

The processing general interrupts are:

- The ID Transition Interrupt (IDTI)
- The VBus Transition Interrupt (VBUSTI)
- The Role Exchange Interrupt (ROLEEXI)

The exception general interrupts are:

- The VBus Error Interrupt (VBERRI)
- The B-Connection Error Interrupt (BCERRI)
- The Suspend Time-Out Interrupt (STOI)

MCU Power modes

Run mode

In this mode, all MCU clocks can run, including the USB clock.

Idle mode

In this mode, the CPU is halted, i.e. the CPU clock is stopped. The Idle mode is entered what- ever the state of the USBB. The MCU wakes up on any USB interrupt.

Frozen mode

Same as the Idle mode, except that the HSB module is stopped, so the USB DMA, which is an HSB master, can not be used. Moreover, the USB DMA must be stopped before entering this sleep mode in order to avoid erratic behavior. The MCU wakes up on any USB interrupt.

Standby, Stop, DeepStop and Static modes

Same as the Frozen mode, except that the USB generic clock and other clocks are stopped, so the USB macro is frozen. Only the asynchronous USB interrupt sources can wake up the MCU in these modes . The Power Manager (PM) may have to be configured to enable asynchro- nous wake up from USB. The USB module must be frozen by writing a one to the FRZCLK bit.

USB clock frozen

In the run, idle and frozen MCU modes, the USBB can be frozen when the USB line is in the suspend mode, by writing a one to the FRZCLK bit, what reduces power consumption.

In deeper MCU power modes (from StandBy mode), the USBC must be frozen.

In this case, it is still possible to access the following elements, but only in Run mode:

- The OTGPADE, VBUSPO, FRZCLK, USBE, UIDE, UIMOD and LS bits in the USBCON register
- The DPRAM (through the USB Pipe/Endpoint n FIFO Data (USB-FIFOnDATA) registers, but not through USB bus transfers which are frozen)

Moreover, when FRZCLK is written to one, only the asynchronous interrupt sources may trigger the USB interrupt:

- The ID Transition Interrupt (IDTI)
- The VBus Transition Interrupt (VBUSTI) • The Wake-up Interrupt (WAKEUP)
- The Host Wake-up Interrupt (HWUPI)

USB Suspend mode

In peripheral mode, the Suspend Interrupt bit in the Device Global Interrupt register (UDINT.SUSP)indicates that the USB line is in the suspend mode. In this case, the transceiver is automatically set in suspend mode to reduce the consumption.The 480MHz internal PLL is stopped. The USBSTA.CLKUSABLE bit is cleared.

Speed control

- Device mode
  - When the USB interface is in device mode, the speed selection (full-speed or high-speed) is per- formed automatically by the USBB during the USB reset according to the host speed capability. At the end of the USB reset, the USBB enables or disables high-speed terminations and pull-up.
  - It is possible to restraint the USBB to full-speed or low-speed mode by handling the LS and the Speed Configuration (SPDCONF) bits in UDCON.

- Host mode
  - When the USB interface is in host mode, internal pull-down resistors are connected on both D+ and D- and the interface detects the speed of the connected device, which is reflected by the Speed Status (SPEED) field in USBSTA.

DPRAM management

Pipes and endpoints can only be allocated in ascending order (from the pipe/endpoint 0 to the last pipe/endpoint to be allocated). The user shall therefore configure them in the same order.

The allocation of a pipe/endpoint n starts when the Endpoint Memory Allocate bit in the Endpoint n Configuration register (UECFGn.ALLOC) is written to one. Then, the hardware allocates a memory area in the DPRAM and inserts it between the n-1 and n+1 pipes/endpoints. The n+1 pipe/endpoint memory window slides up and its data is lost. Note that the following pipe/end- point memory windows (from n+2) do not slide.

Disabling a pipe, by writing a zero to the Pipe n Enable bit in the Pipe Enable/Reset register (UPRST.PENn), or disabling an endpoint, by writing a zero to the Endpoint n Enable bit in the Endpoint Enable/Reset register (UERST.EPENn), resets neither the UECFGn.ALLOC bit nor its configuration (the Pipe Banks (PBK) field, the Pipe Size (PSIZE) field, the Pipe Token (PTO- KEN) field, the Pipe Type (PTYPE) field, the Pipe Endpoint Number (PEPNUM) field, and the Pipe Interrupt Request Frequency (INTFRQ) field in the Pipe n Configuration (UPCFGn) register/the Endpoint Banks (EPBK) field, the Endpoint Size (EPSIZE) field, the Endpoint Direction (EPDIR) field, and the Endpoint Type (EPTYPE) field in UECFGn).

To free its memory, the user shall write a zero to the UECFGn.ALLOC bit. The n+1 pipe/end- point memory window then slides down and its data is lost. Note that the following pipe/endpoint memory windows (from n+2) does not slide.

Figure 33 illustrates the allocation and reorganization of the DPRAM in a typical example.

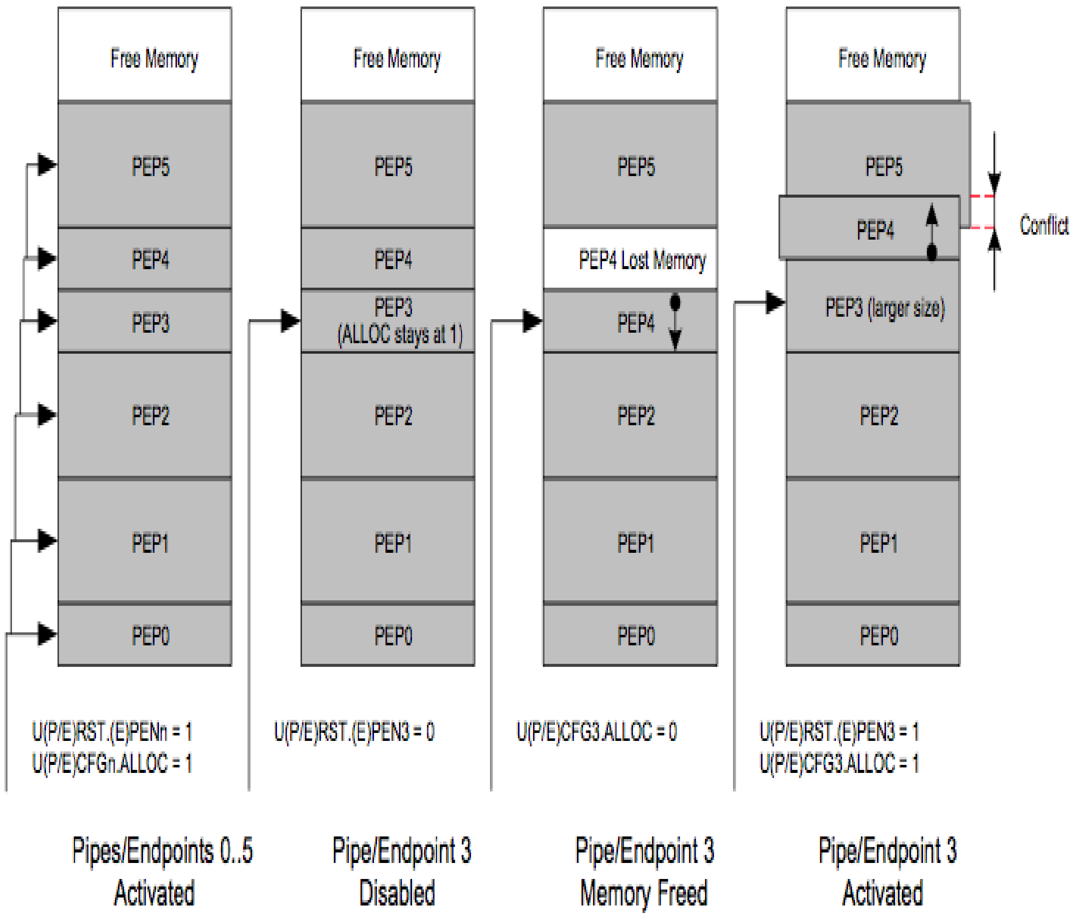


Figure 33: Allocation and Reorganization of the DPRAM

1. The pipes/endpoints 0 to 5 are enabled, configured and allocated in ascending order. Each pipe/endpoint then owns a memory area in the DPRAM.

2. The pipe/endpoint 3 is disabled, but its memory is kept allocated by the controller.

3. In order to free its memory, its ALLOC bit is written to zero. The pipe/endpoint 4 mem- ory window slides down, but the pipe/endpoint 5 does not move.

4. If the user chooses to reconfigure the pipe/endpoint 3 with a larger size, the controller allocates a memory area after the pipe/endpoint 2 memory area and automatically slides up the pipe/endpoint 4 memory

window. The pipe/endpoint 5 does not move and a memory conflict appears as the memory windows of the pipes/endpoints 4 and 5 overlap. The data of these pipes/endpoints is potentially lost.

Note that:

- There is no way the data of the pipe/endpoint 0 can be lost (except if it is de-allocated) as memory allocation and de-allocation may affect only higher pipes/endpoints.
- Deactivating then reactivating a same pipe/endpoint with the same configuration only modifies temporarily the controller DPRAM pointer and size for this pipe/endpoint, but nothing changes in the DPRAM, so higher endpoints seem to not have been moved and their data is preserved as far as nothing has been written or received into them while changing the allocation state of the first pipe/endpoint.
- When the user write a one to the ALLOC bit, the Configuration OK Status bit in the Endpoint n Status register (UESTAn.CFGOK) is set only if the configured size and number of banks are correct compared to their maximal allowed values for the endpoint and to the maximal FIFO size (i.e. the DPRAM size), so the value of CFGOK does not consider memory allocation conflicts.

Pad Suspend

Figure 34 shows the pad behaviour.

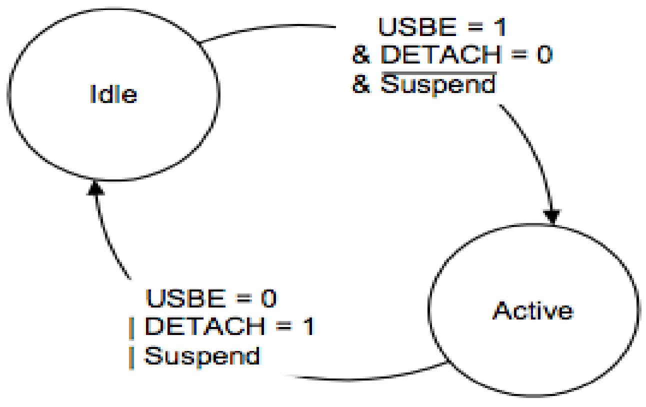


Figure 34: Pad Behavior

- In the Idle state, the pad is put in low power consumption mode, i.e., the differential receiver of the USB pad is off, and internal pull-down with strong value(15K) are set in both DP/DM to avoid floating lines.
- In the Active state, the pad is working.

Figure 35 illustrates the pad events leading to a PAD state change.

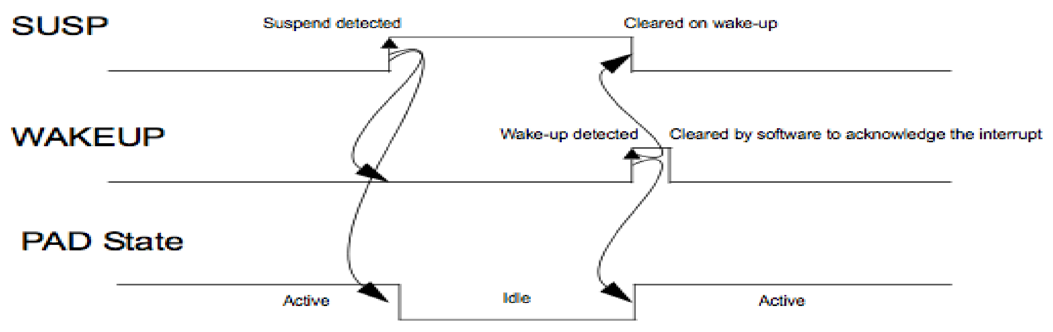


Figure 35: Pad Events

The SUSP bit is set and the Wake-Up Interrupt (WAKEUP) bit in UDINT is cleared when a USB “Suspend” state has been detected on the USB bus. This event automatically puts the USB pad in the Idle state. The detection of a non-idle event sets WAKEUP, clears SUSP and wakes up the USB pad.

Moreover, the pad goes to the Idle state if the macro is disabled or if the DETACH bit is written to one. It returns to the Active state when USBE is written to one and DETACH is written to zero.

Plug-In detection

The USB connection is detected from the USB_VBUS pad. Figure 36 shows the architecture of the plug-in detector.

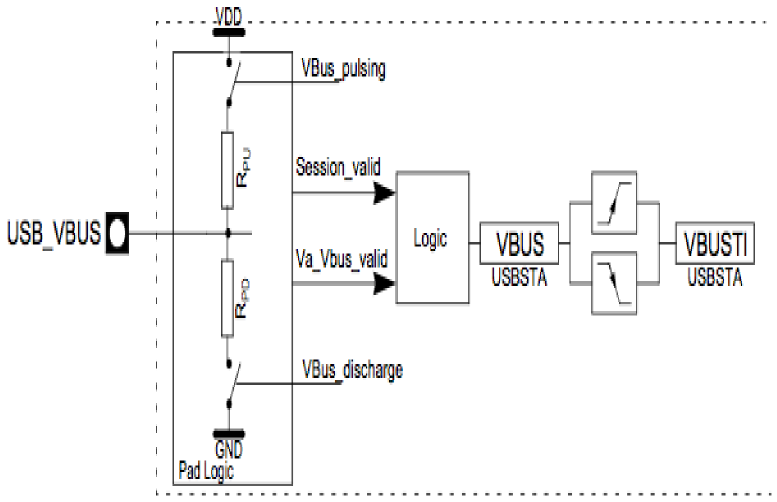


Figure 36: Plug-In Detection Input Block Diagram

The control logic of the USB_VBUS pad outputs two signals:

- The Session_valid signal is high when the voltage on the USB_VBUS pad is higher than or equal to 1.4V.
- The Va_Vbus_valid signal is high when the voltage on the USB_VBUS pad is higher than or equal to 4.4V.

In device mode, the USBSTA.VBUS bit follows the Session_valid comparator output:

- It is set when the voltage on the USB_VBUS pad is higher than or equal to 1.4V.
- It is cleared when the voltage on the VBUS pad is lower than 1.4V.

n host mode, the USBSTA.VBUS bit follows an hysteresis based on Session_valid and Va_Vbus_valid:

- It is set when the voltage on the USB_VBUS pad is higher than or equal to 4.4V.
- It is cleared when the voltage on the USB_VBUS pad is lower than 1.4V.

The VBus Transition interrupt (VBUSTI) bit in USBSTA is set on each transition of the USB- STA.VBUS bit.

The USBSTA.VBUS bit is effective whether the USBB is enabled or not.

ID detection

Figure 37 shows how the ID transitions are detected.

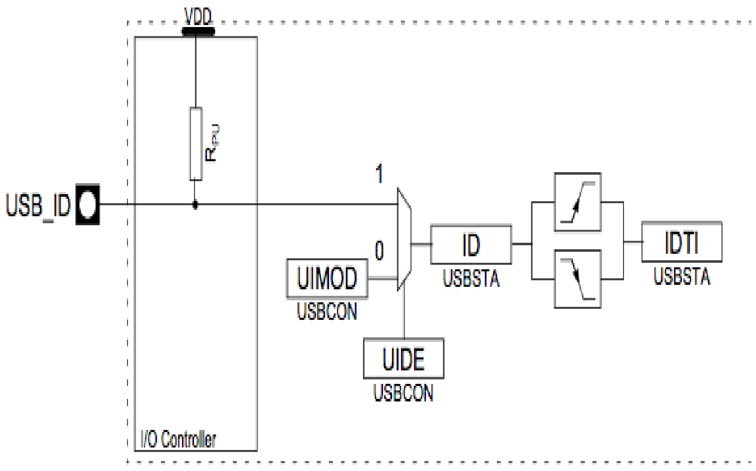


Figure 37: Pad Events

The USB mode (device or host) can be either detected from the USB_ID pin or software selected by writing to the UIMOD bit, according to the UIDE bit. This allows the USB_ID pin to be used as a general purpose I/O pin even when the USB interface is enabled.

By default, the USB_ID pin is selected (UIDE is written to one) and the USBB is in device mode (UBSTA.ID is one), what corresponds to the case where no Mini-A plug is connected, i.e. no plug or a Mini-B plug is connected and the USB_ID pin is kept high by the internal pull-up resistor from the I/O Controller (which must be enabled if USB_ID is used).

The ID Transition Interrupt (IDTI) bit in USBSTA is set on each transition of the ID bit, i.e. when a Mini-A plug (host mode) is connected or disconnected. This does not occur when a Mini-B plug (device mode) is connected or disconnected.

The USBSTA.ID bit is effective whether the USBB is enabled or not.

USB Device Operation

Introduction In device mode, the USBB supports hi- full- and low-speed data transfers.

In addition to the default control endpoint, seven endpoints are provided, which can be configured with the types isochronous, bulk or interrupt.The device mode starts in the Idle state, so the pad consumption is reduced to the minimum.

Power-On and reset

Figure 38 describes the USBB device mode main states.

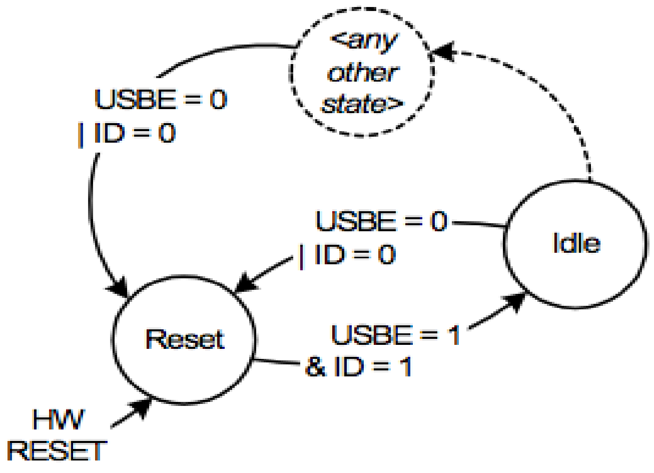


Figure 38: Device Mode States

After a hardware reset, the USBB device mode is in the Reset state. In this state:

- The macro clock is stopped in order to minimize power consumption (FRZCLK is written to one).
- The internal registers of the device mode are reset.
- The endpoint banks are de-allocated.
- Neither D+ nor D- is pulled up (DETACH is written to one).

D+ or D- will be pulled up according to the selected speed as soon as the DETACH bit is written to zero and VBus is present. See “Device mode” for further details.

When the USBB is enabled (USBE is written to one) in device mode (ID is one), its device mode state goes to the Idle state with minimal power consumption. This does not require the USB clock to be activated.

The USBB device mode can be disabled and reset at any time by disabling the USBB (by writing a zero to USBE) or when host mode is engaged (ID is zero).

USB reset

The USB bus reset is managed by hardware. It is initiated by a connected host.

When a USB reset is detected on the USB line, the following operations are performed by the controller:

- All the endpoints are disabled, except the default control endpoint.
- The default control endpoint is reset.
- The data toggle sequence of the default control endpoint is cleared.
- At the end of the reset process, the End of Reset (EORST) bit in UDINT interrupt is set.
- During a reset, the USBB automatically switches to the Hi-Speed mode if the host is Hi- Speed capable (the reset is called a Hi-Speed reset). The user should observe the USBSTA.SPEED field to know the speed running at the end of the reset (EORST is one).

Endpoint reset

An endpoint can be reset at any time by writing a one to the Endpoint n Reset (EPRSTn) bit in the UERST register. This is recommended before using an endpoint upon hardware reset or when a USB bus reset has been received. This resets:

- The internal state machine of this endpoint.
- The receive and transmit bank FIFO counters.
- All the registers of this endpoint (UECFGn, UESTAn, the Endpoint n Control (UECONn) register), except its configuration (ALLOC, EPBK, EPSIZE, EPDIR, EPTYPE) and the Data Toggle Sequence (DTSEQ) field of the UESTAn register.

Note that the interrupt sources located in the UESTAn register are not cleared when a USB bus reset has been received.

The endpoint configuration remains active and the endpoint is still enabled.

The endpoint reset may be associated with a clear of the data toggle sequence as an answer to the CLEAR_FEATURE USB request. This can be achieved by writing a one to the Reset Data Toggle Set bit in the Endpoint n Control

Set register (UECONnSET.RSTDTS).(This will set the Reset Data Toggle (RSTD) bit in UECONn).

In the end, the user has to write a zero to the EPRSTn bit to complete the reset operation and to start using the FIFO.

Endpoint activation

The endpoint is maintained inactive and reset as long as it is disabled (EPENn is written to zero). DTSEQ is also reset.

The algorithm represented on Figure 39 must be followed in order to activate an endpoint.

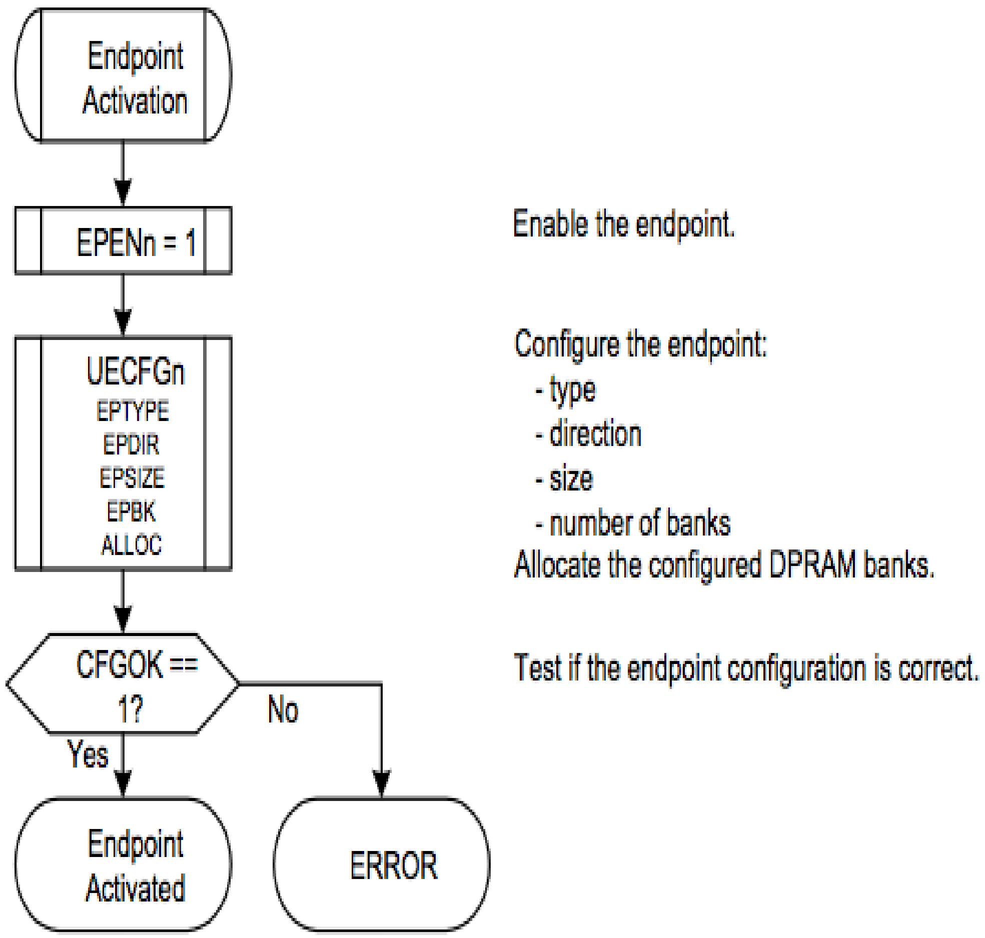


Figure 39: Endpoint Activation Algorithm

As long as the endpoint is not correctly configured (CFGOK is zero), the controller does not acknowledge the packets sent by the host to this endpoint.

The CFGOK bit is set only if the configured size and number of banks are correct compared to their maximal allowed values for the endpoint and to the maximal FIFO size (i.e. the DPRAM size).

Address setup

The USB device address is set up according to the USB protocol.

- After all kinds of resets, the USB device address is 0.
- The host starts a SETUP transaction with a SET_ADDRESS(addr) request.
- The user write this address to the USB Address (UADD) field in UDCON, and write a zero to the Address Enable (ADDEN) bit in UDCON, so the actual address is still 0.
- The user sends a zero-length IN packet from the control endpoint.
- The user enables the recorded USB device address by writing a one to ADDEN.

Once the USB device address is configured, the controller filters the packets to only accept those targeting the address stored in UADD.

UADD and ADDEN shall not be written all at once.

UADD and ADDEN are cleared:

- On a hardware reset.
- When the USBB is disabled (USBE written to zero).
- When a USB reset is detected.

When UADD or ADDEN is cleared, the default device address 0 is used.

Suspend and wake-up

When an idle USB bus state has been detected for 3 ms, the controller set the Suspend (SUSP) interrupt bit in UDINT. The user may then write a one to the FRZCLK bit to reduce power consumption. The MCU can also enter the Idle or Frozen sleep mode to lower again power consumption.

To recover from the Suspend mode, the user shall wait for the Wake-Up (WAKEUP) interrupt bit, which is set when a non-idle event is detected, then write a zero to FRZCLK.

As the WAKEUP interrupt bit in UDINT is set when a non-idle event is detected, it can occur whether the controller is in the Suspend mode or not. The SUSP and WAKEUP interrupts are thus independent of each other except that one bit is cleared when the other is set.

Detach

The reset value of the DETACH bit is one.

It is possible to initiate a device re-enumeration simply by writing a one then a zero to DETACH.

DETACH acts on the pull-up connections of the D+ and D- pads.

Remote wake-up

The Remote Wake-Up request (also known as Upstream Resume) is the only one the device may send on its own initiative, but the device should have beforehand been allowed to by a DEVICE_REMOTE_WAKEUP request from the host.

- First, the USBB must have detected a “Suspend” state on the bus, i.e. the Remote Wake-Up request can only be sent after a SUSP interrupt has been set.
- The user may then write a one to the Remote Wake-Up (RMWKUP) bit in UDCON to send an upstream resume to the host for a remote wake-up. This will automatically be done by the controller after 5ms of inactivity on the USB bus.
- When the controller sends the upstream resume, the Upstream Resume (UPRSM) interrupt is set and SUSP is cleared.
- RMWKUP is cleared at the end of the upstream resume.
- If the controller detects a valid “End of Resume” signal from the host, the End of Resume (EORSM) interrupt is set.

STALL request

For each endpoint, the STALL management is performed using:

- The STALL Request (STALLRQ) bit in UECONn to initiate a STALL request.
- The STALLed Interrupt (STALLEDI) bit in UESTAn is set when a STALL handshake has been sent.

To answer the next request with a STALL handshake, STALLRQ has to be set by writing a one to the STALL Request Set (STALLRQS) bit. All following requests will be discarded (RXOUTI, etc. will not be set) and handshaked with a STALL until the STALLRQ bit is cleared, what is done when a new SETUP packet is received (for control endpoints) or when the STALL Request Clear (STALLRQC) bit is written to one.

Each time a STALL handshake is sent, the STALLEDI bit is set by the USBB and the EPnINT interrupt is set.

- Special considerations for control endpoints
  - If a SETUP packet is received into a control endpoint for which a STALL is requested, the Received SETUP Interrupt (RXSTPI) bit in UESTAn is set and STALLRQ and STALLEDI are cleared. The SETUP has to be ACKed.

    This management simplifies the enumeration process management. If a command is not sup- ported or contains an error, the user requests a STALL and can return to the main task, waiting for the next SETUP request.
- STALL handshake and retry mechanism
  - If a SETUP packet is received into a control endpoint for which a STALL is requested, the Received SETUP Interrupt (RXSTPI) bit in UESTAn is set and STALLRQ and STALLEDI are cleared. The SETUP has to be ACKed.

    This management simplifies the enumeration process management. If a command is not supported or contains an error, the user requests a STALL and can return to the main task, waiting for the next SETUP request.

Management of control endpoints

- Overview

  A SETUP request is always ACKed. When a new SETUP packet is received, the RXSTPI is set, but not the Received OUT Data Interrupt (RXOUTI) bit.

  The FIFO Control (FIFOCON) bit in UECONn and the Read/Write Allowed (RWALL) bit in UESTAn are irrelevant for control endpoints. The user shall therefore never use them on these endpoints. When read, their value are always zero.

Control endpoints are managed using:

- The RXSTPI bit which is set when a new SETUP packet is received and which shall be cleared by firmware to acknowledge the packet and to free the bank.
- The RXOUTI bit which is set when a new OUT packet is received and which shall be cleared by firmware to acknowledge the packet and to free the bank.
- The Transmitted IN Data Interrupt (TXINI) bit which is set when the current bank is ready to accept a new IN packet and which shall be cleared by firmware to send the packet.

- Control write Figure 40 shows a control write transaction. During the status stage, the controller will not necessarily send a NAK on the first IN token:
  - If the user knows the exact number of descriptor bytes that must be read, it can then anticipate the status stage and send a zero-length packet after the next IN token.
  - Or it can read the bytes and wait for the NAKed IN Interrupt (NAKINI) which tells that all the bytes have been sent by the host and that the transaction is now in the status stage.

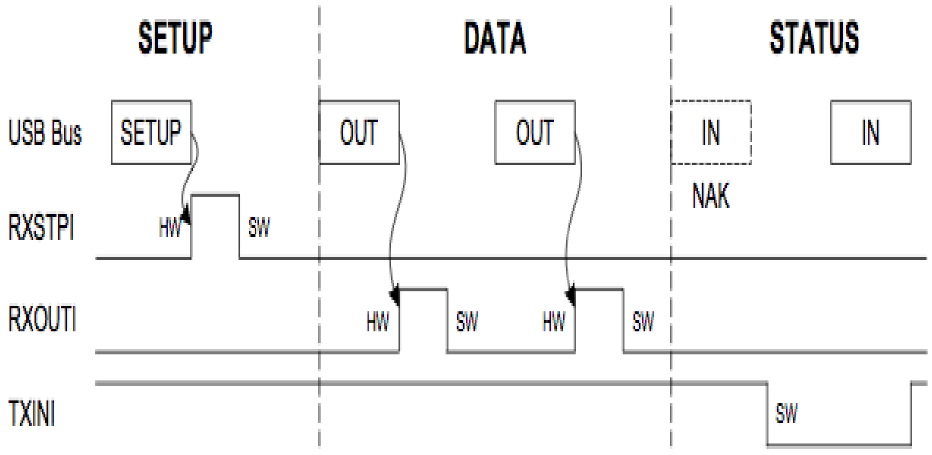


Figure 40: Control Write

- Control read Figure 41 shows a control read transaction. The USBB has to manage the simultaneous write requests from the CPU and the

USB host.

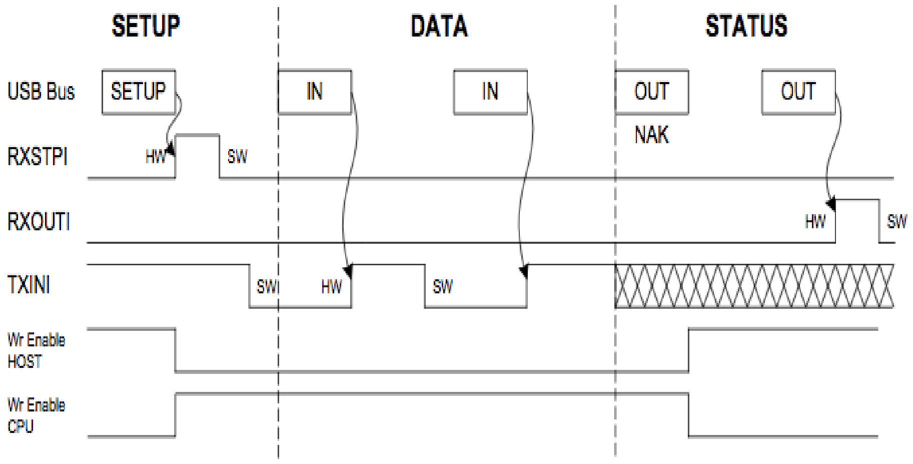


Figure 41: Control Read

A NAK handshake is always generated on the first status stage command.

When the controller detects the status stage, all the data written by the CPU are lost and clear- ing TXINI has no effect.

The user checks if the transmission or the reception is complete.

The OUT retry is always ACKed. This reception sets RXOUTI and TXINI. Handle this with the following software algorithm:

```
set TXINI
wait for RXOUTI OR TXINI
if RXOUTI, then clear bit and return
if TXINI, then continue
```

Once the OUT status stage has been received, the USBB waits for a SETUP request. The SETUP request has priority over any other request and has to be ACKed. This means that any other bit should be cleared and the FIFO reset when a SETUP is received.

The user has to take care of the fact that the byte counter is reset when a zero-length OUT packet is received.

Management of IN endpoints

- Overview

IN packets are sent by the USB device controller upon IN requests from the host. All the data can be written which acknowledges or not the bank when it is full. The endpoint must be configured first.

The TXINI bit is set at the same time as FIFOCON when the current bank is free. This triggers an EPnINT interrupt if the Transmitted IN Data Interrupt Enable (TXINE) bit in UECONn is one.

TXINI shall be cleared by software (by writing a one to the Transmitted IN Data Interrupt Enable Clear bit in the Endpoint n Control Clear register (UECONnCLR.TXINIC)) to acknowledge the interrupt, what has no effect on the endpoint FIFO.

The user then writes into the FIFO, and write a one to the FIFO Control Clear (FIFOCONC) bit in UECONnCLR to clear the FIFOCON bit. This allows the USBB to send the data. If the IN end- point is composed of multiple banks, this also switches to the next bank. The TXINI and FIFOCON bits are updated in accordance with the status of the next bank.

TXINI shall always be cleared before clearing FIFOCON.

The RWALL bit is set when the current bank is not full, i.e. the software can write further data into the FIFO.

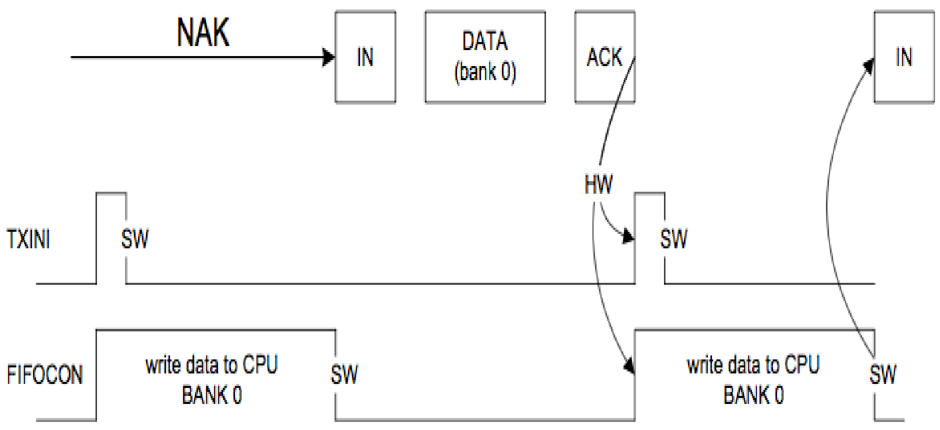


Figure 42: Example of an IN Endpoint with 1 Data Bank

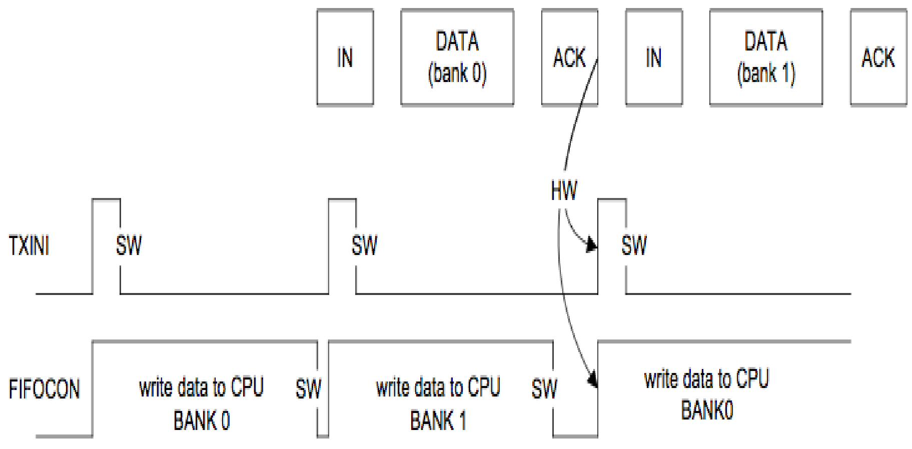


Figure 43: Example of an IN Endpoint with 2 Data Banks

- Detailed description

  The data is written, following the next flow:

  - When the bank is empty, TXINI and FIFOCON are set, what triggers an EPnINT interrupt if TXINE is one.
  - The user acknowledges the interrupt by clearing TXINI.
  - The user writes the data into the current bank by using the USB Pipe/Endpoint nFIFO Data virtual segment (see "USB Pipe/Endpoint n FIFO Data Register, until all the data frame is written or the bank is full (in which case RWALL is cleared and the Byte Count (BYCT) field in UESTAn reaches the endpoint size).
  - The user allows the controller to send the bank and switches to the next bank (if any) by clearing FIFOCON.

  If the endpoint uses several banks, the current one can be written while the previous one is being read by the host. Then, when the user clears FIFOCON, the following bank may already be free and TXINI is set immediately.

  An "Abort" stage can be produced when a zero-length OUT packet is received during an IN stage of a control or isochronous IN transaction. The Kill IN Bank (KILLBK) bit in UECONn is used to kill the last written bank. The best way to manage this abort is to apply the algorithm represented on Figure 44.

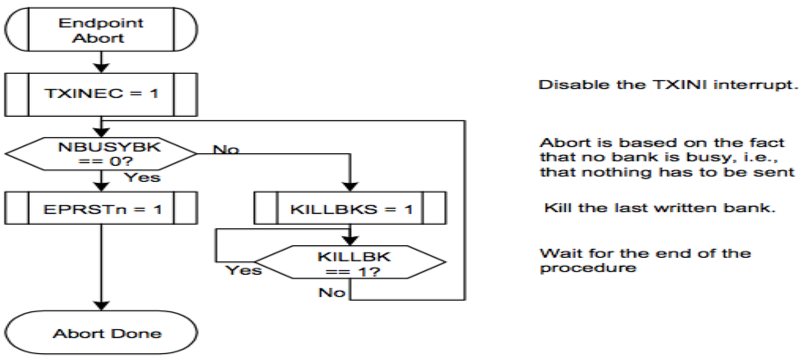

Endpoint Abort
TXINEC = 1
Disable the TXINI interrupt.
NBUSYBK == 0?
No
Yes
Abort is based on the fact that no bank is busy, i.e., that nothing has to be sent
EPRSTn = 1
KILLBKS = 1
Kill the last written bank.
KILLBK == 1?
Yes
No
Wait for the end of the procedure
Abort Done

Figure 44: Abort Algorithm

Management of OUT endpoints

- Overview OUT packets are sent by the host. All the data can be read which acknowledges or not the bank when it is empty.

  The endpoint must be configured first.

  The RXOUTI bit is set at the same time as FIFOCON when the current bank is full. This triggers an EPnINT interrupt if the Received OUT Data Interrupt Enable (RXOUTE) bit in UECONn is one.

  RXOUTI shall be cleared by software (by writing a one to the Received OUT Data Interrupt Clear (RXOUTIC) bit) to acknowledge the interrupt, what has no effect on the endpoint FIFO.

  The user then reads from the FIFO and clears the FIFOCON bit to free the bank. If the OUT endpoint is composed of multiple banks, this also switches to the next bank. The RXOUTI and FIFOCON bits are updated in accordance with the status of the next bank.

  RXOUTI shall always be cleared before clearing FIFOCON.

  The RWALL bit is set when the current bank is not empty, i.e. the software can read further data from the FIFO.

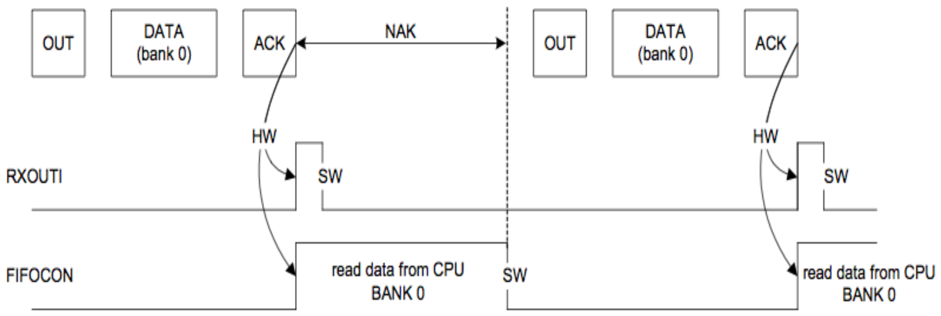


Figure 45: Example of an OUT Endpoint with one Data Bank

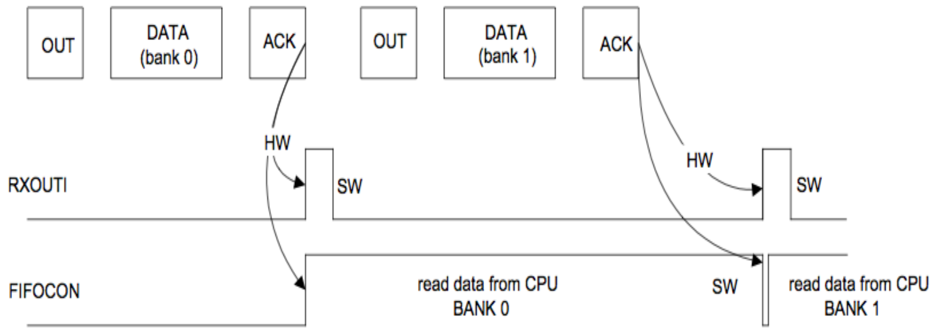


Figure 46: Example of an OUT Endpoint with two Data Banks

- Detailed description

  The data is read, following the next flow:

  - When the bank is full, RXOUTI and FIFOCON are set, what triggers an EPnINT interrupt if RXOUTE is one.
  - The user acknowledges the interrupt by writing a one to RX-OUTIC in order to clear RXOUTI.
  - The user can read the byte count of the current bank from BYCT to know how many bytes to read, rather than polling RWALL.
  - The user reads the data from the current bank by using the USB-FIFOnDATA register, until all the expected data frame is read or the bank is empty (in which case RWALL is cleared and BYCT reaches zero).
  - The user frees the bank and switches to the next bank (if any) by clearing FIFOCON.

If the endpoint uses several banks, the current one can be read while the following one is being written by the host. Then, when the user clears FIFOCON, the following bank may already be ready and RXOUTI is set immediately.

In Hi-Speed mode, the PING and NYET protocol is handled by the USBB. For single bank, a NYET handshake is always sent to the host (on Bulk-out transaction) to indicate that the current packet is acknowledged but there is no room for the next one. For double bank, the USBB responds to the OUT/DATA transaction with an ACK handshake when the endpoint accepted the data successfully and has room for another data payload (the second bank is free).

Underflow

This error exists only for isochronous IN/OUT endpoints. It set the Underflow Interrupt (UNDERFI) bit in UESTAn, what triggers an EPnINT interrupt if the Underflow Interrupt Enable (UNDERFE) bit is one.

An underflow can occur during IN stage if the host attempts to read from an empty bank. A zero- length packet is then automatically sent by the USBB.

An underflow can not occur during OUT stage on a CPU action, since the user may read only if the bank is not empty (RXOUTI is one or RWALL is one).

An underflow can also occur during OUT stage if the host sends a packet while the bank is already full. Typically, the CPU is not fast enough. The packet is lost.

An underflow can not occur during IN stage on a CPU action, since the user may write only if the bank is not full (TXINI is one or RWALL is one).

Overflow

This error exists for all endpoint types. It set the Overflow interrupt (OVERFI) bit in UESTAn, what triggers an EPnINT interrupt if the Overflow Interrupt Enable (OVERFE) bit is one.

An overflow can occur during OUT stage if the host attempts to write into a bank that is too small for the packet. The packet is acknowledged and the RXOUTI bit is set as if no overflow had occurred. The bank is filled with all the first bytes of the packet that fit in.

An overflow can not occur during IN stage on a CPU action, since the user may write only if the bank is not full (TXINI is one or RWALL is one).

HB IsoIn error

This error exists only for high-bandwidth isochronous IN endpoints if the high-bandwidth isochronous feature is supported by the device (see the UFEATURES register for this).

At the end of the micro-frame, if at least one packet has been sent to the host, if less banks than expected has been validated (by clearing the FIFOCON) for this micro-frame, it set the HBISOINERRORI bit in UESTAn, what triggers an EPnINT interrupt if the High Bandwidth Isochronous IN Error Interrupt Enable (HBISOINERRORE) bit is one.

For instance, if the Number of Transaction per MicroFrame for Isochronous Endpoint (NBTRANS field in UECFGn is three (three transactions per micro-frame), only two banks are filled by the CPU (three expected) for the current micro-frame. Then, the HBISOINERRI interrupt is generated at the end of the micro-frame. Note that an UNDERFI interrupt is also generated (with an automatic zero-length-packet), except in the case of a missing IN token.

HB IsoFlush

This error exists only for high-bandwidth isochronous IN endpoints if the high-bandwidth isochronous feature is supported by the device (see the UFEATURES register for this).

At the end of the micro-frame, if at least one packet has been sent to the host, if there is missing IN token during this micro-frame, the bank(s) destined to this micro-frame is/are flushed out to ensure a good data synchronization between the host and the device.

For instance, if NBTRANS is three (three transactions per micro-frame), if only the first IN token (among 3) is well received by the USBB, then the two last banks will be discarded.

CRC error

This error exists only for isochronous OUT endpoints. It set the CRC Error Interrupt (CRCERRI) bit in UESTAn, what triggers an EPnINT interrupt if the CRC Error Interrupt Enable (CRCERRE) bit is one.

A CRC error can occur during OUT stage if the USBB detects a corrupted received packet. The OUT packet is stored in the bank as if no CRC error had occurred (RXOUTI is set).

Interrupts

See the structure of the USB device interrupt system on Figure 32.

There are two kinds of device interrupts: processing, i.e. their generation is part of the normal processing, and exception, i.e. errors (not related to CPU exceptions).

- Global interrupts

  The processing device global interrupts are:

  - The Suspend (SUSP) interrupt
  - The Start of Frame (SOF) interrupt with no frame number CRC error (the Frame Number CRC Error (FNCERR) bit in the Device Frame Number (UDFNUM) register is zero)
  - The Micro Start of Frame (MSOF) interrupt with no CRC error.
  - The End of Reset (EORST) interrupt
  - The Wake-Up (WAKEUP) interrupt
  - The End of Resume (EORSM) interrupt
  - The Upstream Resume (UPRSM) interrupt
  - The Endpoint n (EPnINT) interrupt
  - The DMA Channel n (DMAnINT) interrupt

  The exception device global interrupts are:

  - The Start of Frame (SOF) interrupt with a frame number CRC error (FNCERR is one)
  - The Micro Start of Frame (MSOF) interrupt with a CRC error

- Endpoint interrupts

  The processing device endpoint interrupts are:

  - The Transmitted IN Data Interrupt (TXINI)
  - The Received OUT Data Interrupt (RXOUTI)
  - The Received SETUP Interrupt (RXSTPI)
  - The Short Packet (SHORTPACKET) interrupt
  - The Number of Busy Banks (NBUSYBK) interrupt

- The Received OUT isochronous Multiple Data Interrupt (MDATAI)
- The Received OUT isochronous DataX Interrupt (DATAXI)

The exception device endpoint interrupts are:

- The Underflow Interrupt (UNDERFI)
- The NAKed OUT Interrupt (NAKOUTI)
- The High-bandwidth isochronous IN error Interrupt (HBISOIN-ERRI) if the high-bandwidth isochronous feature is supported by the device (see the UFEATURES register for this)
- The NAKed IN Interrupt (NAKINI)
- The High-bandwidth isochronous IN Flush error Interrupt (HBISOFLUSHI) if the high- bandwidth isochronous feature is supported by the device (see the UFEATURES register for this)
- The Overflow Interrupt (OVERFI)
- The STALLed Interrupt (STALLEDI)
- The CRC Error Interrupt (CRCERRI)
- The Transaction error (ERRORTRANS) interrupt if the high-bandwidth isochronous feature is supported by the device (see the UFEATURES register for this)

3.2 GSM/GPRS Module Specifications

Here we mention some characteristics of the used GSM module in this design. The V2 family is the new generation of Telit modules which offers a GSM/GPRS protocol stack 3GPP Release 4, the Downlink Advance Receiver Performance (DARP) feature for Single Antenna Interference Cancellation (SAIC), the Enhanced Measurement Report, GERAN Feature package 1, which assists in supporting the Extended Uplink TBF and Network Assisted Cell Change (NACC), the control via remote AT Commands and Event Monitor.

The GC864 product family is one of the smallest GSM/GPRS quad-band modules with indus- trial connectors in the market.

According to Telit Unified Form Factor, the V2 modules are designed to be compatible with Telit's GSM/GPRS products in the compact, unified form

Figure 47: GC 864-QUAD V2 Compact

factor family, allowing easy scalability and shorter time-to-market for new design.

With its ultra-compact design and extended temperature range, the Telit GC864-QUAD V2 is the perfect platform for medium-volume m2m applications and mobile data devices. Additional features such as integrated TCP/IP protocol stack and serial multiplexer give extend functionality of the application at no additional cost.

The GC864-QUAD V2 makes it furthermore possible to run the customer's application inside the module, thus making it one of the smallest, complete platforms for m2m solutions. GC864-QUAD V2 is also available with integrated SIM Holder.

All Telit modules, support Over-the-Air firmware update by means Premium FOTA Management. By embedding RedBend's vRapid® agent, a proven and battle-tested technol- ogy powering hundreds of millions of cellular handsets world-wide Telit is able to update its products by transmitting only a delta file, which represents the difference between one firmware version and another.

As a part of Telit's corporate policy of environmental protection, all products comply to the RoHS (Restriction of Hazardous Substances) directive of the European Union (EU Directive 2002/95/EG).

Product features

- Quad-band EGSM 850 / 900 / 1800 / 1900 MHz
- Output power

  - Class 4 (2W) @ 850 / 900 MHz
  - Class 1 (1W) @ 1800 / 1900 MHz
- Control via AT commands according to 3GPP TS 27.005, 27.007 and Telit custom AT commands
- Serial port multiplexer 3GPP TS 27.010
- SIM access profile
- SIM application toolkit 3GPP TS 51.014
- Supply voltage range: 3.22–4.5 V DC (3.8 V DC recommended)
- TCP/IP stack access via AT commands
- Sensitivity:
  - 107 dBm (typ.) @ 850 / 900 MHz -106 dBm (typ.) @ 1800 / 1900 MHz
- Power consumption (typical values)
  - Power off: 62 uA
  - Idle (registered, power saving): 1.5 mA @ DRX=9
  - Dedicated mode: $<$ 240 mA @ max power level
  - GPRS cl.10: $<$ 420 mA @ max power level
- Dimensions: 30 x 36.2 x 3.2 mm
- Weight: 6.1 grams
- Extended temperature range
  - -40 °C to +85°C (operational)
  - -40 °C to +85°C (storage temperature)
- RoHS compliant
- DARP/SAIC support

Interfaces

- 80-pin Molex connector
- 10 I/O ports maximum

- Analog audio (balanced)
- 2 A/D plus 1 D/A converters
- Buzzer output
- ITU-T V.24 serial link through UART:
  - CMOS level
  - Baud rate from 300 to 115,200 bps
  - Autobauding up to 115,000 bps
- 50 Ohm murata GSC antenna connector

Audio

- Telephony, emergency call
- Half rate, full rate, enhanced full rate and adaptive multi rate voice codecs (HR, FR, EFR, AMR)
- Superior echo cancellation & noise reduction
- Multiple audio profiles pre-programmed and fully configurable by mean AT commands
- DTMF

SMS

- Point-to-point mobile originated and mobile terminated SMS
- Concatenated SMS supported
- SMS cell broadcast
- Text and PDU mode
- SMS over GPRS

Approvals

- Fully type approved conforming with R&TTE
- CE, GCF, FCC, PTCRB, IC, Anatel

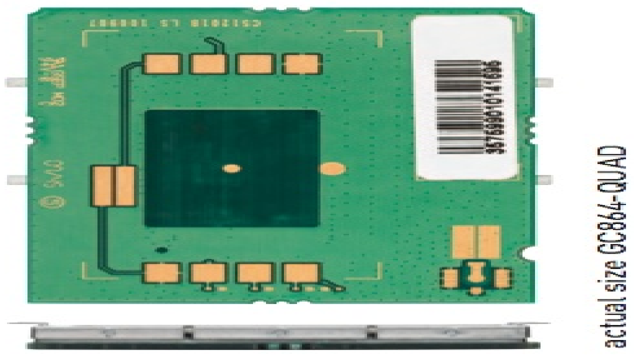


Figure 48: actual size GC864-QUAD

Circuit switched data transmission

- Asynchronous non-transparent CSD up to 9.6 kbps
- V.110

GPRS data

- GPRS class 10
- Mobile station class B
- Coding scheme 1 to 4
- PBCCH support
- GERAN Feature Package 1 support (NACC, Extended TBF)

GSM supplementary

- Call forwarding
- Call barring
- Call waiting & call hold
- Advice of charge
- Calling line identification presentation (CLIP)
- Calling line identification restriction (CLIR)
- Unstructured supplementary services mobile originated data (USSD)

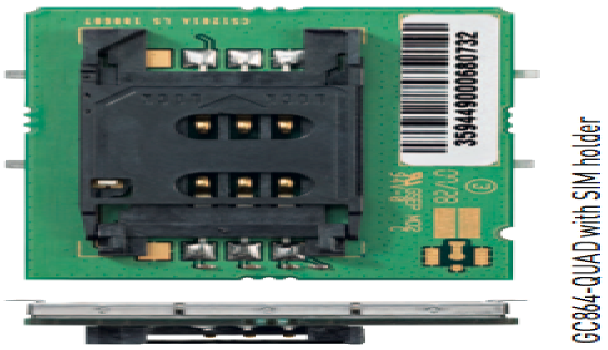


Figure 49: GC864-QUAD with SIM holder

- Closed user group

Additional features

- SIM phonebook
- SIM Holder (only for GC864-QUAD V2 variant with SIM holder)
- Fixed dialing number (FDN)
- Real-time clock
- Alarm management
- Network LED support
- IRA, GSM, 8859-1 and UCS2 character set
- Jamming detection
- Embedded TCP/IP stack, including TCP, IP, UDP, SMTP, ICMP and FTP protocols
- PFM (Premium FOTA Management) Over-The-Air update service
- Remote AT commands
- Event monitor

Telit's EASY features

- EASY SCAN ® automatic scan over GSM frequencies (also without SIM card)

4 Prototyping

The main goal of this section is the implementation and testing of each module independently with simulated inputs. Module integration, system implementation and testing of the complete system.

4.1 FreeRTOS

For this purpose we have chosen to use FreeRTOS for its advantages such as modularity and being free source. Also, by using RTOS we make sure that realtime requirements are met. As a result, we would have a fully modular system in which we could be able to add each single module without modifying the source code.

4.1.1 FreeRTOS Modules

Features The following standard features are provided.

- Choice of RTOS scheduling policy:
  - Pre-emptive: Always runs the highest available task. Tasks of identical priority share CPU time (fully pre- emptive with round robin time slicing).
  - Cooperative: Context switches only occur if a task blocks, or explicitly calls taskYIELD().
- Message queues
- Semaphores [via macros]
- Trace visualisation ability (requires more RAM)
- Majority of source code common to all supported development tools
- Additional features can quickly and easily be added.

Design Philosophy FreeRTOS is designed to be:

- Simple
- Portable

- Concise

Nearly all the code is written in C, with only a few assembler functions where completely unavoidable. This does not result in tightly optimized code, but does mean the code is readable, maintainable and easy to port. If performance were an issue it could easily be improved at the cost of portability. This will not be necessary for most applications.

The RTOS kernel uses multiple priority lists. This provides maximum application design flexibility. Unlike bitmap kernels any number of tasks can share the same priority.

Tasks & Priorities

Real Time Task Priorities Low priority numbers denote low priority tasks, with the default idle priority defined by tskIDLE_PRIORITY as being zero.

The number of available priorities is defined by tskMAX_PRIORITIES within FreeRTOSConfig.h. This should be set to suit your application.

Any number of real time tasks can share the same priority - facilitating application design. User tasks can also share a priority of zero with the idle task.

Priority numbers should be chosen to be as close and as low as possible. For example, if your application has 3 user tasks that must all be at different priorities then use priorities 3 (highest), 2 and 1 (lowest - the idle task uses priority 0).

Implementing a Task A task should have the following structure:

```
void vATaskFunction( void *pvParameters ) {
for( ;; ) {
-- Task application code here. -- }
}
```

The type pdTASK_CODE is defined as a function that returns void and takes a void pointer as it's only parameter. All functions that implement a task should be of this type. The parameter can be used to pass any information into the task. Task functions should never return so are typically implemented as a continuous loop.

Tasks are created by calling xTaskCreate() and deleted by calling vTaskDelete().

The prototype for the function shown above can be written as:

```
void vATaskFunction( void *pvParameters );
```

or,

```
portTASK_FUNCTION_PROTO( vATaskFunction, pvParameters );
```

Likewise the function above could equally be written as:

```
portTASK_FUNCTION( vATaskFunction, pvParameters ) {
for( ;; ) {
-- Task application code here. -- }
}
```

The Idle Task The idle task is created automatically by the first call to xTaskCreate (). The idle task is responsible for freeing memory allocated by the RTOS to tasks that have since been deleted. It is therefore important in applications that make use of the vTaskDelete() function to ensure the idle task is not starved of processing time. The activity visualisation utility can be used to check the microcontroller time allocated to the idle task. The idle task has no other active functions so can legitimately be starved of microcontroller time under all other conditions. It is acceptable for application tasks to share the idle task priority. (tskIDLE_PRIORITY).

The Idle Task Hook An idle task hook is a function that is called during each cycle of the idle task. If you want application functionality to run at the idle priority then there are two options:

- Implement the functionality in an idle task hook.

  There must always be at least one task that is ready to run. It is therefore imperative that the hook function does not call any API functions that might cause the task to block (vTaskDelay() for example).

- Create an idle priority task to implement the functionality.

  This is a more flexible solution but has a higher RAM usage overhead.

To create an idle hook:

- Set configUSE_IDLE_HOOK to 1 within FreeRTOSConfig.h.

- Define a function that has the following prototype:

```
void vApplicationIdleHook( void );
```

A common use for an idle hook is to simply put the processor into a power saving mode.

Start/Stopping the Real Time Kernel The real time kernel is started by calling vTaskStartScheduler(). The call will not return unless an application task calls vTaskEndScheduler() or the function cannot complete.

RTOS Kernel Utilities

Queue Implementation Items are placed in a queue by copy - not by reference. It is therefore preferable, when queuing large items, to only queue a pointer to the item.

Semaphore Implementation Binary semaphore functionality is provided by a set of macros. The macros use the queue implementation as this provides everything necessary with no extra code or testing overhead. The macros can easily be extended to provide counting semaphores if required.

Memory Management The RTOS kernel has to allocate RAM each time a task, queue or semaphore is created. The malloc() and free() functions can sometimes be used for this purpose, but ...

1. they are not always available on embedded systems,
2. take up valuable code space,
3. are not thread safe, and
4. are not deterministic (the amount of time taken to execute the function will differ from call to call)

.. so more often than not an alternative scheme is required.

One embedded / real time system can have very different RAM and timing requirements to another - so a single RAM allocation algorithm will only ever be appropriate for a subset of applications.

To get around this problem the memory allocation API is included in the RTOS portable layer - where an application specific implementation appropriate for the real time system being developed can be provided. When the real time kernel requires RAM, instead of calling malloc() it makes a call to pvPortMalloc(). When RAM is being freed, instead of calling free() the real time kernel makes a call to vPortFree().

Three sample RAM allocation schemes are included in the FreeRTOS source code download (V2.5.0 onwards). These are used by the various demo applications as appropriate. But, we here mention the simplest scheme that we made use of in the target system.

Scheme 1 - heap_1.c This is the simplest scheme of all. It does not permit memory to be freed once it has been allocated, but despite this is suitable for a surprisingly large number of applications. The algorithm simply subdivides a single array into smaller blocks as requests for RAM are made. The total size of the array is set by the definition configTOTAL_HEAP_SIZE - which is defined in FreeRTOSConfig.h.

This scheme:

Can be used if your application never deletes a task or queue (no calls to vTaskDelete() or vQueueDelete() are ever made). l Is always deterministic (always takes the same amount of time to return a block). l Is used by the PIC, AVR and 8051 demo applications - as these do not dynamically create or delete tasks after vTaskStartScheduler() has been called.

heap_1.c is suitable for a lot of small real time systems provided that all tasks and queues are created before the kernel is started.

Application programming interface(API) Here we mention just the functions have been used in our design, the rest could be found in the official FreeRTOS reference document.

Task Creation

Modules

- xTaskCreate
- vTaskDelete

xTaskCreate Create a new task and add it to the list of tasks that are ready to run.

```
task. h

portBASE_TYPE xTaskCreate(pdTASK_CODE pvTaskCode,const
   portCHAR * const pcName, unsigned portSHORT usStackDepth,
   void *pvParameters,unsigned portBASE_TYPE uxPriority,
   xTaskHandle  *pvCreatedTask);
```

Parameters:

pvTaskCode: Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).

pcName: A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by tskMAX_TASK_NAME_LEN - default is 16.

usStackDepth: The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage. The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type size_t.

pvParameters: Pointer that will be used as the parameter for the task being created.

uxPriority: The priority at which the task should run.

pvCreatedTask: Used to pass back a handle by which the created task can be referenced.

Returns:

pdPASS if the task was successfully created and added to a ready list, otherwise an error code defined in the file projdefs. h

Task Control

Modules

- vTaskDelay
- vTaskDelayUntil

- vTaskSuspend
- vTaskResume

vTaskDelay INCLUDE_vTaskDelay must be defined as 1 for this function to be available. See the configuration section for more information.

```
task. h
void vTaskDelay( portTickType xTicksToDelay );
```

Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK_RATE_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

Parameters:

xTicksToDelay: The amount of time, in tick periods, that the calling task should block.

vTaskDelayUntil INCLUDE_vTaskDelayUntil must be defined as 1 for this function to be available. See the configuration section for more information.

```
task. h
void vTaskDelayUntil( portTickType *pxPreviousWakeTime,
    portTickType xTimeIncrement );
```

Delay a task until a specified time. This function can be used by cyclical tasks to ensure a constant execution frequency.

This function differs from vTaskDelay() in one important aspect: vTaskDelay() will cause a task to block for the specified number of ticks from the time vTaskDelay() is called. It is therefore difficult to use vTaskDelay() by itself to generate a fixed execution frequency as the time between a task starting to execute and that task calling vTaskDelay() may not be fixed [the task may take a different path though the code between calls, or may get interrupted or preempted a different number of times each time it executes].

Whereas vTaskDelay() specifies a wake time relative to the time at which the function is called, vTaskDelayUntil() specifies the absolute (exact) time at which it wishes to unblock.

The constant configTICK_RATE_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

Parameters:

pxPreviousWakeTime: Pointer to a variable that holds the time at which the task was last unblocked. The variable must be initialised with the current time prior to its first use. Following this the variable is automatically updated within vTaskDelayUntil().

xTimeIncrement: The cycle time period. The task will be unblocked at time (* pxPreviousWakeTime + xTimeIncrement). Calling vTaskDelayUntil with the same xTimeIncrement parameter value will cause the task to execute with a fixed interval period.

vTaskSuspend INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

```
task. h
void vTaskSuspend( xTaskHandle pxTaskToSuspend );
```

Suspend any task. When suspended a task will never get any microcontroller processing time, no matter what its priority.

Calls to vTaskSuspend are not accumulative - i.e. calling vTaskSuspend () twice on the same task still only requires one call to vTaskResume () to ready the suspended task.

Parameters:

pxTaskToSuspend: Handle to the task being suspended. Passing a NULL handle will cause the calling task to be suspended.

vTaskResume INCLUDE_vTaskSuspend must be defined as 1 for this function to be available. See the configuration section for more information.

```
task. h
void vTaskResume( xTaskHandle pxTaskToResume );
```

Resumes a suspended task.

A task that has been suspended by one of more calls to vTaskSuspend () will be made available for running again by a single call to vTaskResume ().

Parameters:

pxTaskToResume: Handle to the task being readied.

Kernel Control

Modules

- vTaskStartScheduler
- vTaskEndScheduler
- vTaskSuspendAll
- xTaskResumeAll

vTaskStartScheduler Starts the real time kernel tick processing. After calling the kernel has control over which tasks are executed and when.

```
task. h
void vTaskStartScheduler( void );
```

The idle task is created automatically when vTaskStartScheduler() is called.

If vTaskStartScheduler() is successful the function will not return until an executing task calls vTaskEndScheduler(). The function might fail and return immediately if there is insufficient RAM available for the idle task to be created.

vTaskEndScheduler Stops the real time kernel tick. All created tasks will be automatically deleted and multitasking (either preemptive or cooperative) will stop. Execution then resumes from the point where vTaskStartScheduler() was called, as if vTaskStartScheduler() had just returned.

```
task. h
void vTaskEndScheduler( void );
```

vTaskEndScheduler () requires an exit function to be defined within the portable layer (see vPortEndScheduler () in port. c for the PC port). This performs hardware specific operations such as stopping the kernel tick.

vTaskEndScheduler () will cause all of the resources allocated by the kernel to be freed - but will not free resources allocated by application tasks.

vTaskSuspendAll Suspends all real time kernel activity while keeping interrupts (including the kernel tick) enabled.

```
task. h
void vTaskSuspendAll( void );
```

After calling vTaskSuspendAll () the calling task will continue to execute without risk of being swapped out until a call to xTaskResumeAll () has been made.

xTaskResumeAll Resumes real time kernel activity following a call to vTaskSuspendAll (). After a call to xTaskSuspendAll () the kernel will take control of which task is executing at any time.

Returns:

If resuming the scheduler caused a context switch then pdTRUE is returned, otherwise pdFALSE is returned.

Queue Management

Modules

- xQueueCreate
- xQueueSend
- xQueueReceive

xQueueCreate Creates a new queue instance. This allocates the storage required by the new queue and returns a handle for the queue.

```
queue. h
xQueueHandle xQueueCreate(unsigned portBASE_TYPE
    uxQueueLength, unsigned portBASE_TYPE uxItemSize);
```

Parameters:

uxQueueLength: The maximum number of items that the queue can contain. uxItemSize: The number of bytes each item in the queue will require. Items are queued by copy, not by reference, so this is the number of bytes that will be copied for each posted item. Each item on the queue must be the same size.

Returns:

If the queue is successfully create then a handle to the newly created queue is returned. If the queue cannot be created then 0 is returned.

xQueueSend Post an item on a queue. The item is queued by copy, not by reference. This function must not be called from an interrupt service routine.

```
queue.h
portBASE_TYPE xQueueSend(xQueueHandle xQueue,const void *
    pvItemToQueue, portTickType xTicksToWait);
```

Parameters:

xQueue: The handle to the queue on which the item is to be posted.

pvItemToQueue: A pointer to the item that is to be placed on the queue. The size of the items the queue will hold was defined when the queue was created, so this many bytes will be copied from pvItemToQueue into the queue storage area.

xTicksToWait: The maximum amount of time the task should block waiting for space to become available on the queue, should it already be full. The call will return immediately if this is set to 0. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required.

Returns:

pdTRUE if the item was successfully posted, otherwise errQUEUE_FULL.

xQueueReceive Receive an item from a queue. The item is received by copy so a buffer of adequate size must be provided. The number of bytes copied into the buffer was defined when the queue was created.

```
queue. h
portBASE_TYPE xQueueReceive(xQueueHandle xQueue, void *
    pcBuffer, portTickType xTicksToWait);
```

This function must not be used in an interrupt service routine.

Parameters:

pxQueue: The handle to the queue from which the item is to be received.

pcBuffer: Pointer to the buffer into which the received item will be copied.

xTicksToWait: The maximum amount of time the task should block waiting for an item to receive should the queue be empty at the time of the call. The time is defined in tick periods so the constant portTICK_RATE_MS should be used to convert to real time if this is required.

Returns:

pdTRUE if an item was successfully received from the queue, otherwise pdFALSE.

Semaphores

Modules

- vSemaphoreCreateBinary
- xSemaphoreTake
- xSemaphoreGive
- xSemaphoreGiveFromISR

vSemaphoreCreateBinary Macro that implements a semaphore by using the existing queue mechanism. The queue length is 1 as this is a binary semaphore. The data size is 0 as we don't want to actually store any data - we just want to know if the queue is empty or full.

```
semphr.h
vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore )
```

Parameters:

xSemaphore: Handle to the created semaphore. Should be of type xSemaphoreHandle.

xSemaphoreTake Macro to obtain a semaphore. The semaphore must of been created using vSemaphoreCreateBinary ().

```
semphr.h
xSemaphoreTake(xSemaphoreHandle xSemaphore,portTickType
    xBlockTime )
```

Parameters:

xSemaphore: A handle to the semaphore being obtained. This is the handle returned by vSemaphoreCreateBinary ();

xBlockTime: The time in ticks to wait for the semaphore to become available. The macro portTICK_RATE_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore.

Returns:

pdTRUE if the semaphore was obtained. pdFALSE if xBlockTime expired without the semaphore becoming available.

xSemaphoreGive Macro to release a semaphore. The semaphore must of been created using vSemaphoreCreateBinary (), and obtained using sSemaphoreTake ().

This must not be used from an ISR.

Parameters:

xSemaphore: A handle to the semaphore being released. This is the handle returned by vSemaphoreCreateBinary ();

Returns:

pdTRUE if the semaphore was released. pdFALSE if an error occurred. Semaphores are implemented using queues. An error can occur if there is no space on the queue to post a message - indicating that the semaphore was not first obtained correctly.

xSemaphoreGiveFromISR Macro to release a semaphore. The semaphore must of been created using vSemaphoreCreateBinary(), and obtained using xSemaphoreTake().

```
semphr. h
xSemaphoreGiveFromISR( xSemaphoreHandle xSemaphore,
   portBASE_TYPE xTaskPreviouslyWoken)
```

This macro can be used from an ISR.

Parameters:

xSemaphore: A handle to the semaphore being released. This is the handle returned by vSemaphoreCreateBinary ();

xTaskPreviouslyWoken: This is included so an ISR can make multiple calls to xSemaphoreGiveFromISR() from a single interrupt. The first call should always pass in pdFALSE. Subsequent calls should pass in the value returned from the previous call. See the file serial .c in the PC port for a good example of using xSemaphoreGiveFromISR().

Returns:

pdTRUE: if a task was woken by releasing the semaphore. This is used by the ISR to determine if a context switch may be required following the ISR.

4.1.2 FreeRTOS Implementation Modules

RTOS Fundamentals This section provides a very brief introduction to real time and multitasking concepts.

Multitasking The kernel is the core component within an operating system. Operating systems such as Linux employ kernels that allow users access to the computer seemingly simultaneously. Multiple users can execute multiple programs apparently concurrently.

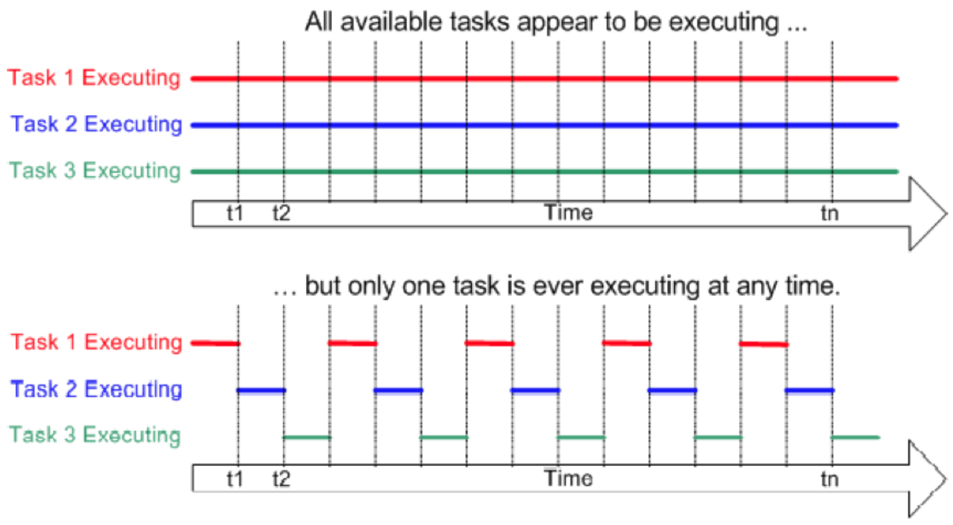
Each executing program is a task under control of the operating system. If an operating system can execute multiple tasks in this manner it is said to be multitasking.

The use of a multitasking operating system can simplify the design of what would otherwise be a complex software application:

The multitasking and inter-task communications features of the operating system allow the complex application to be partitioned into a set of smaller and more manageable tasks. l The partitioning can result in easier software testing, work breakdown within teams, and code reuse. l Complex timing and sequencing details can be removed from the application code and become the responsibility of the operating system.

Multitasking Vs Concurrency:

A conventional processor can only execute a single task at a time - but by rapidly switching between tasks a multitasking operating system can make it appear as if each task is executing concurrently. This is depicted by the diagram below which shows the execution pattern of three tasks with respect to time. The task names are color coded and written down the left hand.



Time moves from left to right, with the colored lines showing which task is executing at any particular time. The upper diagram demonstrates the perceived concurrent execution pattern, and the lower the actual multitasking execution pattern.

Scheduling The scheduler is the part of the kernel responsible for deciding which task should be executing at any particular time. The kernel can suspend and later resume a task many times during the task lifetime.

The scheduling policy is the algorithm used by the scheduler to decide which task to execute at any point in time. The policy of a (non real time) multi user system will most likely allow each task a "fair" proportion of processor time. The policy used in real time / embedded systems is described later.

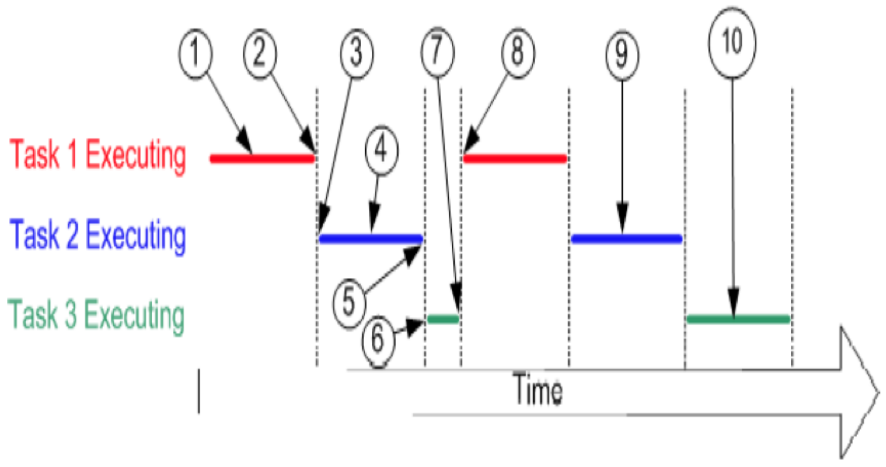
In addition to being suspended involuntarily by the RTOS kernel a task can choose to suspend itself. It will do this if it either wants to delay (sleep) for a fixed period, or wait (block) for a resource to become available (eg a serial port) or an event to occur (eg a key press). A blocked or sleeping task is not able to execute, and will not be allocated any processing time.

Referring to the numbers in the diagram above:

At (1) task 1 is executing.

At (2) the kernel suspends task 1 ...

... and at (3) resumes task 2.



While task 2 is executing (4), it locks a processor peripheral for it's own exclusive access.

At (5) the kernel suspends task 2 ...

... and at (6) resumes task 3.

Task 3 tries to access the same processor peripheral, finding it locked task 3 cannot continue so suspends itself at (7).

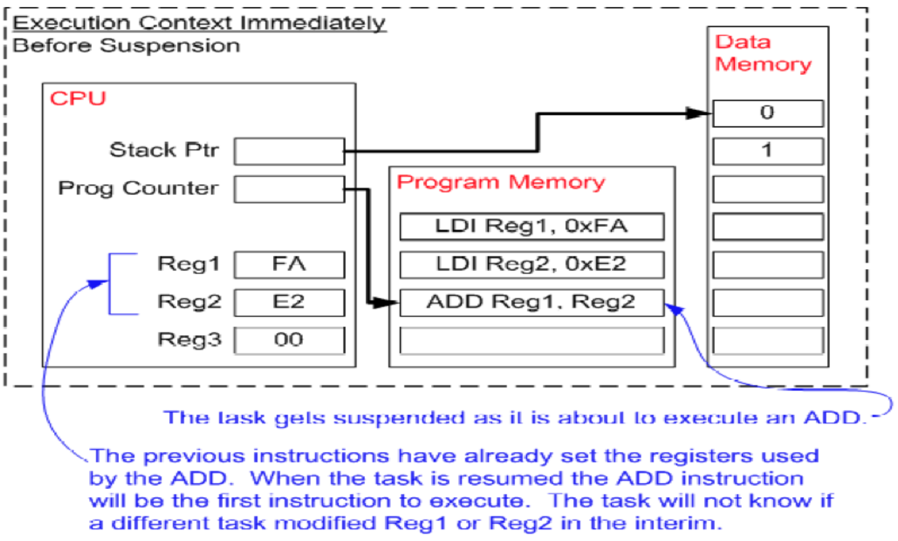
At (8) the kernel resumes task 1. Etc.

The next time task 2 is executing (9) it finishes with the processor peripheral and unlocks it.

The next time task 3 is executing (10) it finds it can now access the processor peripheral and this time executes until suspended by the kernel.

Context Switching As a task executes it utilizes the processor / microcontroller registers and accesses RAM and ROM just as any other program. These resources together (the processor registers, stack, etc.) comprise the task execution context.

A task is a sequential piece of code - it does not know when it is going to get suspended or resumed by the kernel and does not even know when this has happened. Consider the example of a task being suspended immediately before executing an instruction that sums the values contained within two processor registers.


Execution Context Immediately Before Suspension
CPU
Stack Ptr
Prog Counter
Reg1
FA
Reg2
E2
Reg3
00
Program Memory
LDI Reg1, 0xFA
LDI Reg2, 0xE2
ADD Reg1, Reg2
Data Memory
0
1
The task gets suspended as it is about to execute an ADD.
The previous instructions have already set the registers used by the ADD. When the task is resumed the ADD instruction will be the first instruction to execute. The task will not know if a different task modified Reg1 or Reg2 in the interim.

While the task is suspended other tasks will execute and may modify the processor register values. Upon resumption the task will not know that the processor registers have been altered - if it used the modified values the summation would result in an incorrect value.

To prevent this type of error it is essential that upon resumption a task has a context identical to that immediately prior to its suspension. The operating system kernel is responsible for ensuring this is the case - and does so by saving the context of a task as it is suspended. When the task is resumed its saved context is restored by the operating system kernel prior to its execution. The process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching.

Real Time Applications Real time operating systems (RTOS's) achieve multitasking using these same principals - but their objectives are very different to those of non real time systems. The different objective is reflected in the scheduling policy. Real time / embedded systems are designed to provide a timely response to real world events. Events occurring in the real world can have deadlines before which the real time / embedded system must respond and the RTOS scheduling policy must ensure these deadlines are met.

To achieve this objective the software engineer must first assign a priority to each task. The scheduling policy of the RTOS is then to simply ensure that the highest priority task that is able to execute is the task given processing time. This may require sharing processing time "fairly" between tasks of equal priority if they are ready to run simultaneously.

4.2 Target Application Development

In this part we have described the application development for the target Microcontroller in ANCI C programming language using AVRSTUDIO 6 as the IDE for this purpose.

4.2.1 Embedded System Block Diagram

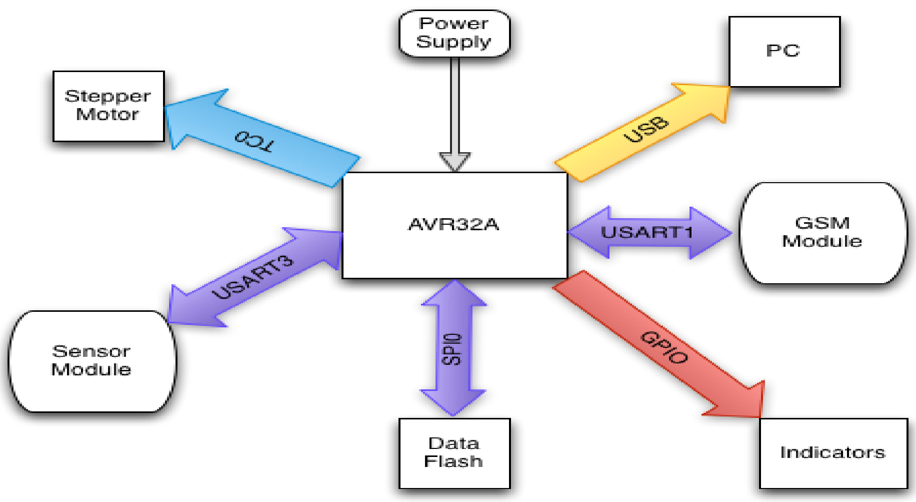


Figure 50: Application Block Diagram

AVR32A The AVR32A microarchitecture is targeted at cost-sensitive, lower-end applications like smaller microcontrollers. This microarchitecture does not provide dedicated hardware registers for shad- owing of register file registers in interrupt contexts. Additionally, it does not provide hardware registers for the return address registers and return status registers. Instead, all this information is stored on the system stack. This saves chip area at the expense of slower interrupt handling.

Upon interrupt initiation, registers R8-R12 are automatically pushed to the system stack. These registers are pushed regardless of the priority level of the pending interrupt. The return address and status register are also automatically pushed to stack. The interrupt handler can therefore use R8-R12 freely. Upon interrupt completion, the old R8-R12 registers and status register are restored, and execution continues at the return address stored popped from stack.

The stack is also used to store the status register and return address for exceptions and scall. Executing the rete or rets instruction at the completion of an exception or system call will pop this status register and continue execution at the popped return address.

Tasks Implementation We have divided the application in some tasks, each one taking care of a specific functionality, for example we have defined the task named "vUsartGsmTask" for handling send and receive operations concerning the microcontroller, GSM module, and a Mobile phone as a receiver. In the following paragraphs, we shall explain each task functionality, and include the piece of code implementing the task.

vUsartGsmTask This task is responsible for communication between microcontroller and the GSM module. At first we wait for a command received from GSM module indicating the gathered data should be sent to the specified cellphone number, and for this purpose we dedicated an interrupt service routine for USART1, responsible for handling the received data from Modem. When we have made sure that its the time for measurement to be done, we give a semaphore to "vUsartSensorTask", by doing so we are able to perform the required operation and send the gathered data through the USART1 back to the specified cellphone in a context of a SMS message. the sequence of operations in this task is as following:

- Create a semaphore for the synchronisation between this task and "vUsartSensorTask".
- Activating the USART1 peripheral in normal mode, and initialising the Receive interrupt for handling the received data from GSM Modem.
- getting the delay function for this task initialised.
- in the infinite loop of the task, we wait for "xSpiSemaphore" to be created and released the by "vSpiDataFlashTask" . (because at first we should store the acquired data in datafalsh memory, then release the semaphore to let the GSM module to send the data to the specified cellphone through GSM network.)
- ...then we could send the acquired data back to the GSM network by GSM modem and through USART1 peripheral. For doing so, we should follow the following steps:

  - At first we send the "ATE0" command to turn off ECHO feature. Because, we are not going to use any PC terminal for data observation, but we have the microcontoller communicating with modem, instead. So, there is no point in using this feature here.
  - in this step we should insert the pin code for having the sim card registered to a GSM network through which we would like to send the gathered data.
  - there are 2 mode for sending a SMS message as we are using ATCOMMANDs. Our preference would be TEXT mode that is quite straightforward and user-friendly. then we configure the module to save the received SMS messages to SIM memory.
  - then we set the cellphone number to which we need to send the required data, this number would be just for demonstration purposes.
  - Finally, we are ready for sending the data, so we point to the array containing the acquired data, when we reach to end of this array, we send a "CTRL+Z" character indicating its the time to message to be sent. especially, for this step we need to want a little bit more than usual (200 ms), to data be written to sending buffer properly.
- And we suspend the task for 20 ms to let other tasks to have the processor time, as well.

Here is the piece of code implementing the task functionality:

```
void vUsartGsmTask(void *pvparameteres)
{
            vSemaphoreCreateBinary(xSendDataSemaphore);
            xSemaphoreTake(xSendDataSemaphore,0);


                 sysclk_enable_peripheral_clock(&AVR32_USART1)
                    ;
                 usart_init_rs232(&AVR32_USART1,&usart_options
                    ,FOSC0);
                 INTC_register_interrupt((__int_handler)&
                    usart1_isr,AVR32_USART1_IRQ,
                    AVR32_INTC_INT0);
                 (&AVR32_USART1) -> ier = (1<<
                    AVR32_USART_RXRDY_OFFSET);


                 portTickType xLastWakeTime;
                 xLastWakeTime= xTaskGetTickCount();


     while(1)
         {

         if(xSpiSemaphore != NULL)
             {
                 if (xSemaphoreTake(xSpiSemaphore, (
                    portTickType)20)==pdTRUE)
                         {
                                  usart_write_line(&
                                     AVR32_USART1,"ATE0\r\n");
                             vTaskDelayUntil(&xLastWakeTime
                                ,20);
                                  usart_write_line(&
                                     AVR32_USART1,"AT+CPIN
                                     =6588\r\n");
                                 vTaskDelayUntil(&
                                    xLastWakeTime,20);
                                  usart_write_line(&
                                     AVR32_USART1,"AT+CMGF=1\r
                                     \n");
                                 vTaskDelayUntil(&
                                    xLastWakeTime,20);
                                  usart_write_line(&
                                     AVR32_USART1,"AT+CNMI
                                     =2,1\r\n");
                                  vTaskDelayUntil(&
                                     xLastWakeTime,20);
                                  usart_write_line(&
                                     AVR32_USART1,"AT+CMGS
                                     =+4745174817\r\n");
                                  vTaskDelayUntil(&
                                     xLastWakeTime,20);
                                  usart_write_line(&
                                     AVR32_USART1,SENSOR_DATA)
                                     ;
                                vTaskDelayUntil(&xLastWakeTime
                                   ,200);
                                  usart_write_line(&
                                     AVR32_USART1,"\x1A");
                                vTaskDelayUntil(&xLastWakeTime
                                   ,20);

                         }
             }
         vTaskDelayUntil(&xLastWakeTime,20);
```

```
        }

}
```

vUsartSensorTask This task is responsible for handling the required communication between the microcontroller and sensor module. When the command for sending the data received by the microcontoller, its the microcontollers to tell the sensor module to start measurement by its sensors, and send the gathered data back to microcontoller for sending to GSM module. this is possible by using USART3 peripheral between Micro and SENSOR module. So, we got a commutation channel through which the data and commands are exchanged between Micro and SENSOR. the required functionality is met by performing the following steps:

- At first we shall create a semaphore named "xReadSensorSemaphore" for synchronisation purpose between this task and "vSpiDataFlash-Task" for writing the acquired data to a Dataflash memory.
- at this point, we connect specific pins of microcontoller to USART3 functionality, and initialise the USART3 peripheral to the normal mode with no parity, 8 bit data, 1 stop bit, and no handshake. Then, we activate the interrupt upon data reception by USART3, for collecting the acquired sensors data by the microcontroller.
- initialise the tick count for delay function
- In the infinite loop we wait "xSendDataSemaphore" to be available and released, to be able to send the "READ" command to sensor module to start data acquisition.
- when the data acquisition is done, the sensor module sends the data back to microcontroller through USART3. And, in the interrupt service routine, we save the received data to a temporary buffer. and we are done, we release "xReadSensorSemaphore" for "vSpiDataFlashTask".
- And we suspend the task for 100 ms to other task have the processing time.

Here is the piece of code implementing the task functionality described above:

```
void vUsartSensorTask(void *pvparameteres)
{
        vSemaphoreCreateBinary(xReadSensorSemaphore);
        xSemaphoreTake(xReadSensorSemaphore,0);

        gpio_enable_module(COMPORT_3_GPIO_MAP,
        sizeof(COMPORT_3_GPIO_MAP) / sizeof(
           COMPORT_3_GPIO_MAP[3]));
        sysclk_enable_peripheral_clock(&AVR32_USART3);
    usart_init_rs232(&AVR32_USART3,&usart_options,FOSC0);
        INTC_register_interrupt((__int_handler)&usart3_isr,
           AVR32_USART3_IRQ,AVR32_INTC_INT0);
        (&AVR32_USART3) -> ier = (1<<
           AVR32_USART_RXRDY_OFFSET);

        portTickType xLastWakeTime;
        xLastWakeTime= xTaskGetTickCount();
        while(1)
        {
          if(xSendDataSemaphore!= NULL)
        {
                if (xSemaphoreTake(xSendDataSemaphore, (
                   portTickType)0)==pdTRUE)
                 usart_write_line(&AVR32_USART3,"READ\r\n");

        }
        vTaskDelayUntil(&xLastWakeTime,100);
        }

}
```

vSpiDataFlashTask This task is dedicated to handle the communication between DATAFALSH memory and Microcontoller. The SPI peripheral is responsible for this communication. When the data is received from sensor module it should be stored somewhere. For future analysis, this data should be kept in a non volatile memory, so we have used a DATAFALSH memory which is suitable for this application. this task is implemented as following:

- At first we create a semaphore named ”xSpiSemaphore” to be used by ”vUsartGsmTask” (when its released, we send the gathered sensor data to the GSM network). We store the data first, then we send it to GSM network.
- now its the SPI peripheral turn to be initialised and activated. For this purpose, we connect the Micro pins to SPI functionality, put the SPI in master mode, connect 12 MHZ crystal, activate the suitable chipselect, and finally activate the peripheral.

- at this point, we select the slave device connected to microcontroller to be DATAFLASH memory.
- In a infinite loop, we wait for ”xReadSensorSemaphore” semaphore to be available and released. Then we check if the memory is ready for write operation. If yes, we perform the write access, and write the array containing the sensor data to target dataflash memory. For finalising the writing operation, we deselect the dataflash, by doing so the write operation would be successfully done. For debugging purposes, we toggle a dedicated LED, to make sure the program control hit this piece of code.
- when we made sure the write operation is done, we could release the ”xSpiSemaphore”, letting ”vUsartGsmTask” continue its operation (which is sending the sensor data)
- As usual, we wait for 20 ms to other tasks gain processor time.

Here is the piece of code implementing the above mentioned functions:

```

void vSpiDataFlashTask( void *pvparameteres)
{
        vSemaphoreCreateBinary(xSpiSemaphore);
        xSemaphoreTake(xSpiSemaphore,0);

        sysclk_enable_peripheral_clock(&AVR32_SPI0);
        // initialize the spi module and AT45DBX dataflash
        spi_init_module();
        spi_select_device(AT45DBX_SPI,&spi_device_conf);

        portTickType xLastWakeTime;
        xLastWakeTime= xTaskGetTickCount();
        while (1)
        {
        vTaskDelayUntil(xLastWakeTime,200);
        if(xReadSensorSemaphore != NULL)
        {
        if (xSemaphoreTake(xReadSensorSemaphore, (
           portTickType)0)==pdTRUE)
        {
        for (int i=0; i<120 ; i++)
        {
       //Pattern =(uint8_t)NTC_f(&result);
        if(at45dbx_mem_check())
       {
        // Perform write access.
        if (at45dbx_write_open(i))
        {
        at45dbx_write_byte(SENSOR_DATA[i]);
        // at45dbx_write_close();
        spi_deselect_device(AT45DBX_SPI,&spi_device_conf);
        LED_Toggle(LED2);
        vTaskDelayUntil(xLastWakeTime,30);
        }

        }
        }
        if(xSpiSemaphore!=NULL)
        xSemaphoreGive(xSpiSemaphore);
        }
        }
        }
}
```

vButtonPollTask For data observation on a personal computer for maintenance staff (on field observation), we considered a task which is concerned with this functionality. We have dedicated a push button that if its pressed, a read operation from data flash memory is executed, and the content of memory is observed on a PC terminal (e.x. hyper terminal in windows) through a USB connection with the embedded system. When the board is connected to a PC by USB standard cable, PC recognises the board as a simple communication port (CDC), after the data can be sent from the board to the PC by configuring the terminal program with proper baud rate, handshake, number of data bits, and number of stop bits. In this task we describe the reading operation from data flash, and next task would be in charge of data communication between the system and a PC. The required steps would be as following:

- At first we create and take a semaphore named ”xButtonSemaphore” for synchronisation with ”vUsbSendDataTask” task .
- then we initialise tick count function for generating required delay time.
- in the infinite loop we have implemented a series of read operations from data flash memory which the data would be stored in an array for observations purposes.
- when the reading operation is done, we shall release the semaphore for ”vUsbSendDataTask” task to continue its operation.
- ...And we suspend the task for 10 ms to let other tasks to run

Here is the piece of code implementing the required functionality:

```
void vButtonPollTask(void *pvparameteres)
{
        vSemaphoreCreateBinary(xButtonSemaphore);
        xSemaphoreTake(xButtonSemaphore ,0);

        portTickType xLastWakeTime;
        xLastWakeTime= xTaskGetTickCount();
        uint8_t Button_Status;

        while(1)
        {
        vTaskDelayUntil(&xLastWakeTime ,10);
        Button_Status=gpio_get_pin_value(GPIO_PUSH_BUTTON_0);
        if(Button_Status==0)
         {

                for (u8 k=0; k<120 ; k++)
                        {
                                // Perform read access.
                                if(at45dbx_mem_check())
                                {
                                        if (at45dbx_read_open
                                           (k))
                                        {
                                        data_flash_buf[k] =
                                           at45dbx_read_byte
                                           ();

                                        at45dbx_read_close();
                                        }
                                }
                                vTaskDelayUntil(&
                                   xLastWakeTime ,30);

                        }

                         if(xButtonSemaphore != NULL)
                         xSemaphoreGive(xButtonSemaphore);

                }


        }

}
```

vUsbSendDataTask the functionality implemented in this task would be data transmission from an array to a personal computer through USB interface for on field data observation. In this task we shall wait for a predefined push button to be pressed, then we send the acquired sensors data to a PC terminal for debugging purposes. The required steps for this purpose would be as following:

- At first we initialise the USB peripheral with required parameters such as choosing the device mode, and communication class select, etc...
- in the infinite loop we wait for ”xButtonSemaphore” semaphore to be released...
- Upon releasing, we send the content of an dedicated array to the PC terminal through USB interface...
- When its done, we wait for 100 ms to other task have some processing time as well.

Here is the implementation of required functionality:

```
void vUsbSendDataTask(void *pvparameters){

                sysclk_enable_peripheral_clock(&AVR32_USART1)
                    ;
                usart_init_rs232(&AVR32_USART1,&usart_options
                    ,FOSC0);

                portTickType xLastWakeTime;
                xLastWakeTime = xTaskGetTickCount();
                taskENTER_CRITICAL();
                {
                 stdio_usb_init(&AVR32_USART1);
                }
                taskEXIT_CRITICAL();

        while(1){
          vTaskDelayUntil(&xLastWakeTime, 100);
                if(xButtonSemaphore!=NULL)
                        if(xSemaphoreTake(xButtonSemaphore,(
                            portTickType)0))
                                for(int j=0;j<120;j++)
                                {
                                        udi_cdc_putc(
                                            data_flash_buf[j])
                                            ;
                                         vTaskDelayUntil(&
                                            xLastWakeTime,
                                                      50);
                                        }
        }
}
```

vApplicationTickHook As an indicator, we have activated the tick hook, by which we toggle a dedicated a LED each time the task executes. So, we could see if system works, and delays are created properly. Here is the tick hook and its implementation:

```
void vApplicationTickHook(void)
{

        LED_Toggle(LED3);

}
```

vApplicationIdleHook One of operations useful for reducing the power consumption, is activating the idle hook, and put the processor in sleep mode each time the idle hook is called. By considering that this system should be a low power. using idle hook would be advantageous without an extra cost.

Here is the mentioned task:

```
void vApplicationIdleHook(void)
{
 // sleepmgr_enter_sleep();
SLEEP(0);
}
```

Task scheduling and Launch the main program At first we should create each task, and give the proper priority to each one as its comes in following:

```


#define USB_TASK_PRIORITY (tskIDLE_PRIORITY+2)
#define USART_GSM_TASK_PRIORITY (tskIDLE_PRIORITY+2)
#define USART_SENSOR_TASK_PRIORITY (tskIDLE_PRIORITY+2)
#define BUTTON_CHECK_TASK_PRIORITY (tskIDLE_PRIORITY+2)
#define RESET_TASK_PRIORITY (tskIDLE_PRIORITY+2)
#define SPI_FLASH_TASK_PRIORITY (tskIDLE_PRIORITY+2)
```

```
//#define MOTOR_CONTROL_TASK_PRIORITY (tskIDLE_PRIORITY+2)


xTaskCreate(vUsbSendDataTask,(signed char *) "USB_DATA_SEND", configMINIMAL_STACK_SIZE, NULL,USB_TASK_PRIORITY,NULL);
xTaskCreate(vUsartGsmTask,(signed char *)"USART_GSM_TASK",configMINIMAL_STACK_SIZE,NULL,USART_GSM_TASK_PRIORITY,NULL); xTaskCreate(vUsartSensorTask,(signed char *)"USART_SENSOR_TASK",configMINIMAL_STACK_SIZE,NULL,USART_SENSOR_TASK_PRIORITY,NULL);
        xTaskCreate(vButtonPollTask,(signed char *)"BUTTON_STATUS_TASK",configMINIMAL_STACK_SIZE,NULL,BUTTON_CHECK_TASK_PRIORITY,NULL);
        xTaskCreate(vReset,(signed char *)"SOFT_RESET_TASK",configMINIMAL_STACK_SIZE,NULL,RESET_TASK_PRIORITY,NULL);
        xTaskCreate(vSpiDataFlashTask,(signed char *) "SPI_FLASH_TASK",configMINIMAL_STACK_SIZE,NULL,SPI_FLASH_TASK_PRIORITY,NULL);
        //xTaskCreate(vMotorControlTask,(signed char *) "MOTOR_CONTROL_TASK",configMINIMAL_STACK_SIZE,NULL,MOTOR_CONTROL_TASK_PRIORITY,NULL);
```

Then we shall start the scheduler by the following command:

```
vTaskStartScheduler();
```

After this the operating system would take care of task scheduling automatically by considering the given task priorities.

Worth mentionig that the ”vMotorControlTask” task implementation would be the topic of another master student.

5 Conclusion and Future Work

We have tried to consider factors like modularity, small code size, and efficiency for developing the required embedded application. And, having in mind that this system is not the most optimal solution, and there are some other approaches by which we could get to a more optimal final solution. This approach could be a good starting point for other people. Also, this system implementation could be tested with actual hardware to see how would be on field response of the system, and whether we meet the specification requirements or not. If not, the application can be modified to reach the best possible one.

References

[1] William J. Emery and Richard E. Thomson Boulder, *Data Analysis Methods in Physical Oceanography*, Second and Revised Edition.

[2] Atmel Corporation, *AT32UC3A3/A4 Data Sheet*, 2011.

[3] Ted Van Sickle, *Programming Microcontrollers in C*, Second Edition.

[4] Ted Jensen, *A TUTORIAL ON POINTERS AND ARRAYS IN C*, Sept. 2003.

[5] Atmel Corporation, *AT45DB011 Data Sheet*, 2010.

[6] Telit Communications S.p.A. , *GC 864-QUAD V2 Software Manual*, 2011.

[7] Richard Barry, *A FREE RTOS for small real time embedded systems*, 2005.