Embedded Design and Development

- **ARM 7 and Cortex M3 Microcontroller**
- **FreeRTOS**
- **Automotive with CAN and AUTOSAR**

Cranes Varsity

Prerequisite	**Basic electronics - Analog and Digital.** **Basic knowledge of Microprocessors and Microcontrollers** **C programming: Data types, storage classes, looping, bitwise operators, pointers, function pointers.**

Cranes Varsity

CHAPTER – 1

INTRODUCTION TO ARM7 (LPC2129)

ARM (Advanced RISC Machine - formerly Acorn RISC Machine) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by British company ARM Holdings. The ARM architecture is the most widely used 32-bit instruction set architecture in numbers produced.

Originally conceived by Acorn Computers for use in its personal computers, the first ARM-based products were the co-processor modules for the BBC Micro series of computers. In 2005 about 98% of the more than one billion mobile phones sold each year used at least one ARM processor. As of 2009[update] ARM processors accounted for approximately 90% of all embedded 32-bit RISC processors and were used extensively in consumer electronics, including personal digital assistants (PDAs), tablets, mobile phones, digital media and music players, hand-held game consoles, calculators and computer peripherals such as hard drives and routers.

The ARM architecture is licensable. Companies that are current or former ARM licensees include Advanced Micro Devices, Inc. Alcatel-Lucent, Apple Inc., Applied Micro, Atmel, Broadcom, Cirrus Logic, CSR plc, Digital Equipment Corporation, Ember, Energy Micro, Freescale, Intel (through DEC), LG, Marvell Technology Group, Microsemi, Microsoft, NEC, Nintendo, Nuvoton, Nvidia, Sony, NXP (formerly Philips Semiconductor), Oki, ON Semiconductor, Psion, Qualcomm, Renesas, Research In Motion, Samsung, Sharp, Silicon Labs, ST-Ericsson, STMicroelectronics, Symbios Logic, Texas Instruments, VLSI Technology, Yamaha, Fuzhou Rockchip, and ZiiLABS.

In addition to the abstract architecture, ARM offers several microprocessor core designs, including the ARM7, ARM9, ARM11, Cortex-A8, Cortex-A9, and Cortex-A15. Companies often license these designs from ARM to manufacture and integrate into their own system on a chip (SoC) with other components like RAM, GPUs, or radio basebands (for mobile phones).

System-on-chip packages integrating ARM's core designs include Nvidia Tegra's first three generations, CSR plc's Quatro family, ST-Ericsson's Nova and NovaThor, Silicon Labs's Precision32 MCU, Texas Instruments OMAP products, Samsung's Hummingbird and Exynos products, Apple's A4, A5, and A5X chips, and Freescale's i.MX.

Companies can also obtain an ARM architectural license for designing their own, different CPU cores using the ARM instruction set. Distinct ARM architecture implementations by licensees include Apple's A6, AppliedMicro's X-Gene, Qualcomm's Snapdragon and Krait, DEC's StrongARM, Marvell (formerly Intel) XScale, and Nvidia's planned Project Denver.

The official Acorn RISC Machine project started in October 1983. VLSI Technology, Inc was chosen as silicon partner, since it already supplied Acorn with ROMs and some custom chips. The design was led by Wilson and Furber, and was consciously designed with a similar efficiency ethos as the 6502 It had a key design goal of achieving low-latency input/output (interrupt) handling like the 6502. The 6502's memory access architecture had allowed developers to produce fast machines without the use of costly direct memory access hardware. VLSI produced the first ARM silicon on 26 April 1985 – it worked the first time and came to be termed ARM1 by April 1985. The first "real" production systems named ARM2 were available the following year.

Its first practical application was as a second processor to the BBC Micro, where it was used to develop the simulation software to finish work on the support chips (VIDC, IOC, MEMC) and to speed up the

operation of the CAD software used in developing ARM2. Wilson subsequently rewrote BBC Basic in ARM assembly language, and the in-depth knowledge obtained from designing the instruction set allowed the code to be very dense, making ARM BBC Basic an extremely good test for any ARM emulator. The original aim of a principally ARM-based computer was achieved in 1987 with the release of the Acorn Archimedes.

In 1992 Acorn once more won the Queen's Award for Technology for the ARM. The ARM2 featured a 32-bit data bus, a 26-bit address space and twenty-seven 32-bit registers. Program code had to lie within the first 64Mbyte of the memory, as the program counter was limited to 24 bits because the top 6 and bottom 2 bits of the 32-bit register served as status flags. The ARM2 had a transistor count of just 30,000, compared to Motorola's six-year older 68000 model with 68,000. Much of this simplicity comes from not having microcode (which represents about one-quarter to one-third of the 68000) and, like most CPUs of the day, not including any cache. This simplicity led to its low power usage, while performing better than the Intel 80286. A successor, ARM3, was produced with a 4 KB cache, which further improved performance. The architecture has evolved over time, and starting with the Cortex series of cores, three "profiles" are defined:

Revision	Example core implementation	ISA enhancement
ARMv1	ARM1	First ARM processor 26-bit addressing
ARMv2	ARM2	32-bit multiplier 32-bit coprocessor support
ARMv2a	ARM3	On-chip cache Atomic swap instruction Coprocessor 15 for cache management
ARMv3	ARM6 and ARM7DI	32-bit addressing Separate *cpsr* and *spsr* New modes—*undefined instruction* and *abort* MMU support—virtual memory
ARMv3M	ARM7M	Signed and unsigned long multiply instructions
ARMv4	StrongARM	Load-store instructions for signed and unsigned halfwords/bytes New mode—*system* Reserve SWI space for architecturally defined operations 26-bit addressing mode no longer supported
ARMv4T	ARM7TDMI and ARM9T	Thumb
ARMv5TE	ARM9E and ARM10E	Superset of the ARMv4T Extra instructions added for changing state between ARM and Thumb Enhanced multiply instructions Extra DSP-type instructions Faster multiply accumulate
ARMv5TEJ	ARM7EJ and ARM926EJ	Java acceleration
ARMv6	ARM11	Improved multiprocessor instructions Unaligned and mixed endian data handling New multimedia instructions

- "Application" profile: Cortex-A series
- "Real-time" profile: Cortex-R series
- "Microcontroller" profile: Cortex-M series.

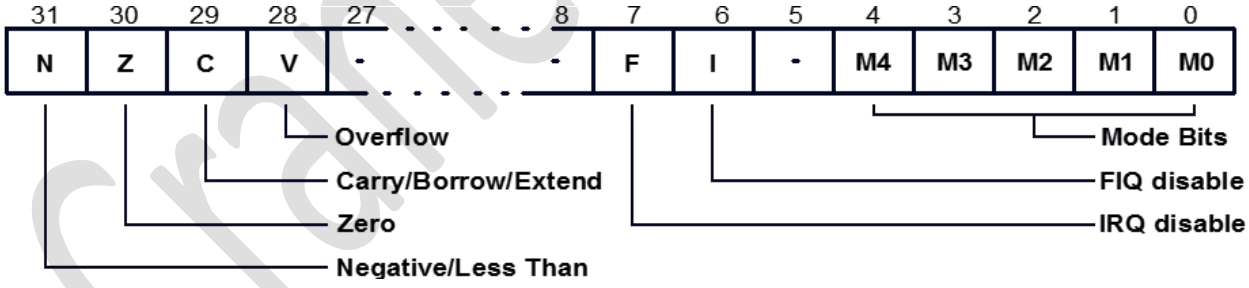
ARM 7 Registers

Registers R0-R7 are the same across all CPU modes; they are never banked.
R13 and R14 are banked across all privileged CPU modes except system mode. That is, each mode that can be entered because of an exception has its own R13 and R14. These registers generally contain the stack pointer and the return address from function calls, respectively.

Aliases:
R13 is also referred to as SP, the Stack Pointer.
R14 is also referred to as LR, the Link Register.
R15 is also referred to as PC, the Program Counter.

Register	
r0	Scratch Registers: r0-r3, r12 r0-r3 used to pass parameters r12 intra-procedure scratch will be overwritten by subroutines
r1	
r2	
r3	
r4	Preserved Registers: r4-r11 stack before using restore before returning
r5	
r6	
r7	
r8	
r9	
r10	
r11	
r12	
r13 'sp'	Stack Pointer: not much use on the stack
r14 'lr'	Link Register: set by BL or BLX on entry of routine overwritten by further use of BL or BLX
r15 'pc'	Program Counter

Register Use in the ARM Procedure Call Standard



The N, Z, C and V are condition code flags
may be changed as a result of arithmetic and logical operations in the processor
may be tested by all instructions to determine if the instruction is to be executed
The I and F bits are the interrupt disable bits
The M0, M1, M2, M3 and M4 bits are the mode bits

ARM CPU modes:

The ARM architecture specifies the following CPU modes. At any moment in time, the CPU can be in only one mode, but it can switch modes due to external events (interrupts) or programmatically.

User mode
The only non-privileged mode.

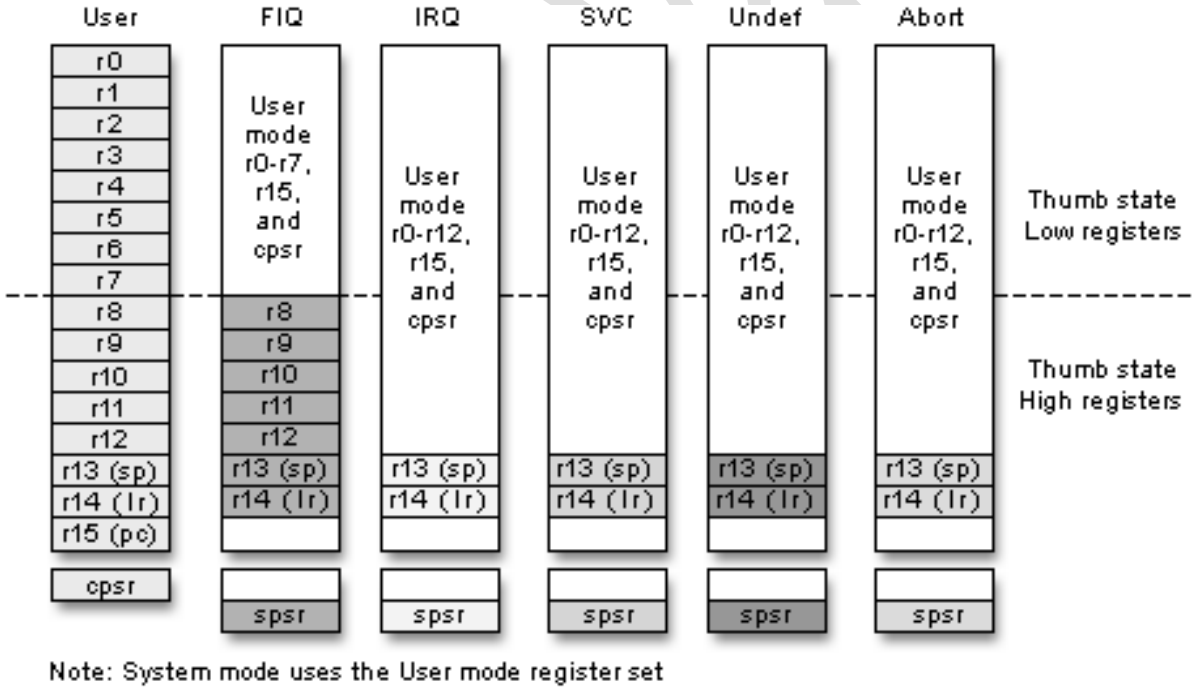
System mode
The only privileged mode that is not entered by an exception. It can only be entered by executing an instruction that explicitly writes to the mode bits of the CPSR.

Supervisor (svc) mode
A privileged mode entered whenever the CPU is reset or when a SWI instruction is executed.

Abort mode
A privileged mode that is entered whenever a prefetch abort or data abort exception occurs.

Undefined mode
A privileged mode that is entered whenever an undefined instruction exception occurs.

Interrupt mode
A privileged mode that is entered whenever the processor accepts an IRQ interrupts.
Fast interrupt mode. A privileged mode that is entered whenever the processor accepts an FIQ interrupts.



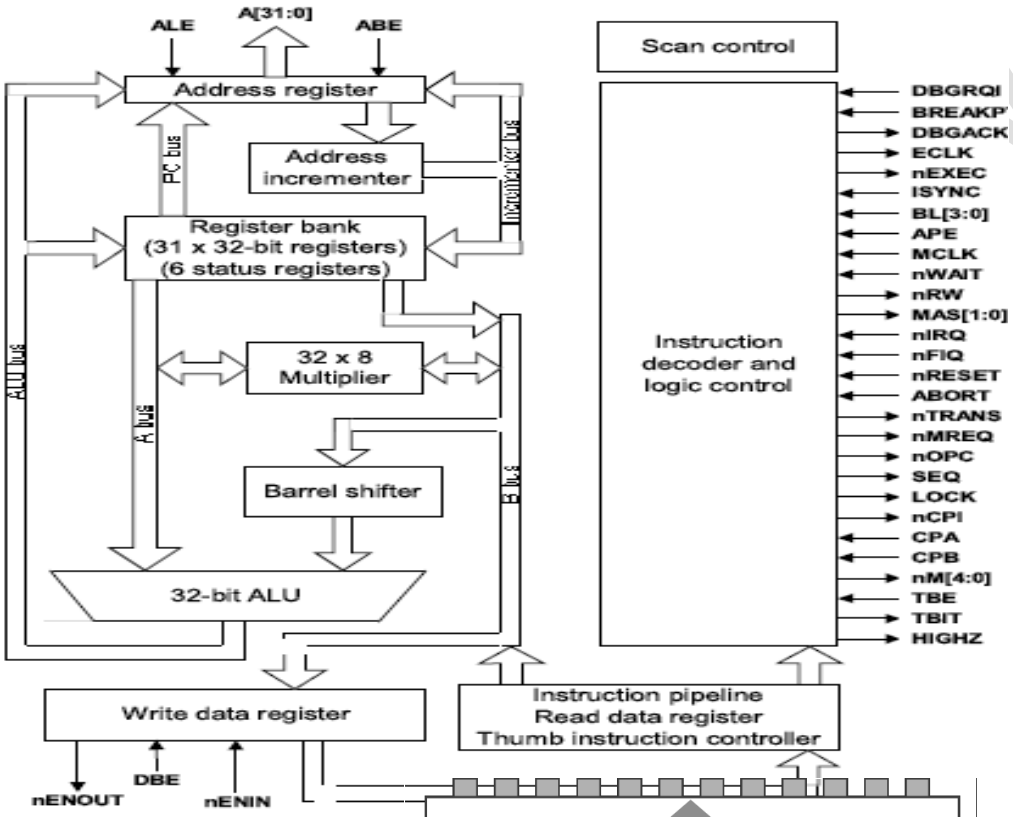
Note: System mode uses the User mode register set

ARM7 - Features

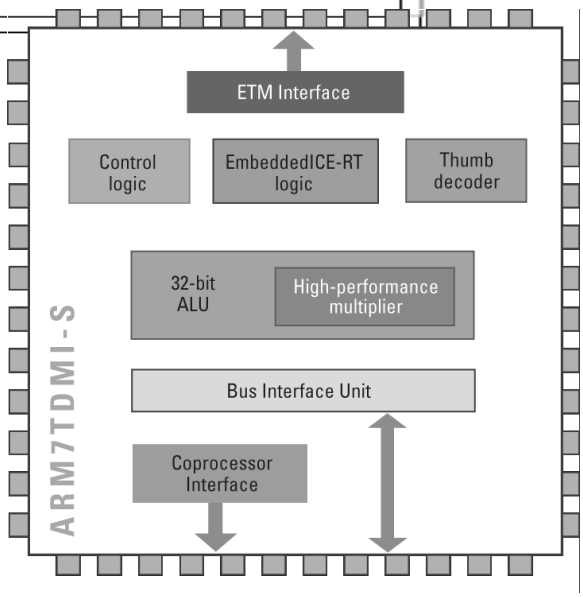
- 32-bit RISC processor (32-bit data & address bus)
- Big and Little Endean operating modes
- High performance RISC (17 MIPS sustained @ 25 MHz (25 MIPS peak) @ 3V)
- Low power consumption (0.6mA/MHz @ 3V fabricated in .8µm CMOS)

- Fully static operation (ideal for power-sensitive applications)
- Fast interrupt response (for real-time applications)
- Virtual Memory System Support
- Excellent high-level language support
- Simple but powerful instruction set

ARM7 – Block Diagram



ARM7-TDMI



NXP LPC21XX

INTRODUCTION TO LPC2129

- 16-bit/32-bit ARM7TDMI-S microcontroller in a 64 or 144 pin package.
- 8/16/64 kB of on-chip static RAM and 64/128/256 kB of on-chip flash program memory (except for flashless LPC2210/20/90). 128-bit wide interface/accelerator enables high-speed 60 MHz operation.
- In-System/In-Application Programming (ISP/IAP) via on-chip boot loader software. Single flash sector or full chip erase in 100 ms and programming of 256 bytes in 1 ms.
- Diversified Code Read Protection (CRP) enables different security levels to be implemented.
- External 8, 16, or 32-bit bus (144 pin packages).
- Embedded ICE RT and Embedded Trace offer real-time debugging with the on-chip Real Monitor software and high speed tracing of instruction execution.
- Up to four interconnected CAN interfaces with advanced acceptance filters.
- 10-bit A/D converter providing four/eight analog inputs with conversion times as low as 2.44 ms per channel and dedicated result registers to minimize interrupt overhead.
- Two 32-bit timers/external event counters with four capture and four compare channels each, PWM unit (6 outputs), Real Time Clock (RTC), and watchdog.
- Multiple serial interfaces including two UARTs (16C550), a fast I2C-bus (400 kbit/s), and two SPI interfaces.
- Vectored interrupt controller with configurable priorities and vector addresses.
- Up to forty-eight 5 V tolerant fast general purpose I/O pins.

Cranes Varsity

CHAPTER - 2

ARM9 and ARM11 Processors

ARM architecture forms the basis for every ARM processor. Over time, the ARM architecture has evolved to include architectural features to meet the growing demand for new functionality, high performance and the need of new and emerging markets.

The ARM architecture supports implementations across a wide range of performance points, establishing it as the leading architecture in many market segments. The ARM architecture supports a very broad range of performance points leading to very small implementations of ARM processors and very efficient implementations of advanced designs using state of the art micro-architecture techniques. Implementation size, performance and low power consumption are key attributes of the ARM architecture.

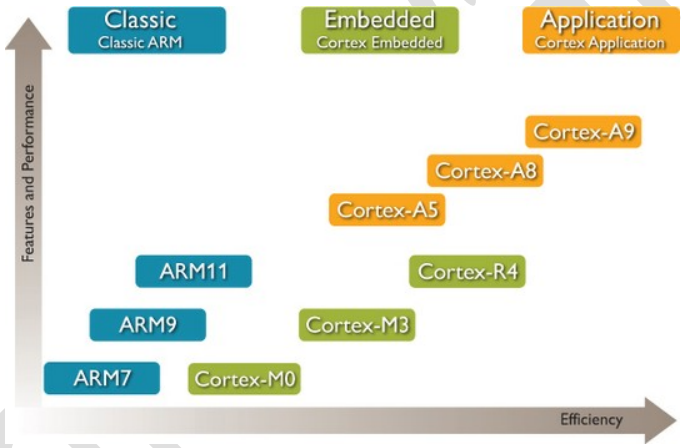


Figure1: The ARM families are divided in three macro areas

ARM9 Processor

The ARM9 processor family enables single processor solutions for microcontrollers, DSP and JAVA applications, offering savings in chip area and complexity, power consumption and time to market. The ARM 9 DSP enhanced processors are well suited for applications requiring a mix of DSP and microcontroller performance.ARM9 processors re the heart of advanced digital products across multiple applications. They deliver deterministic high performance and flexibility for demanding and cost sensitive embedded applications. The rich DSP extensions available remove the need for a separate DSP in the SOC design.

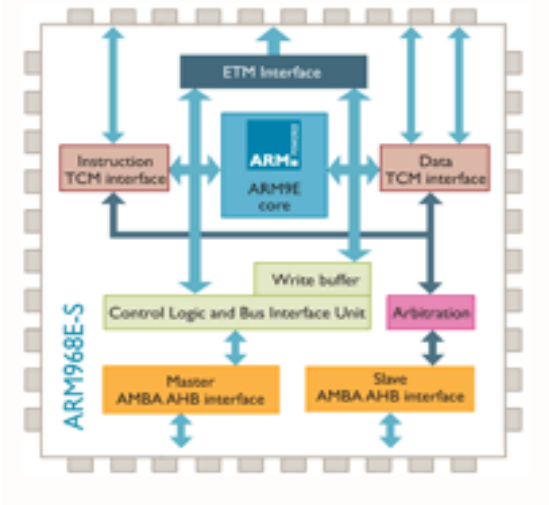


Figure2: ARM 9

ARM9 Features:

1. ARM920T (Dual 16k caches with MMU support multiple Oss).
2. ARM922T (Dual 8k caches for applications support multiple OSs1).
3. ARM940T™ (Dual 4k caches for embedded control applications running a RTOS)
4. 32-bit RISC processor core Super scaling 5-stage integer pipeline.
5. Achieves 1.1 MIPS/MHz, 300 MIPS (Dhrystone 2.1) in a typical 0.13µm process
6. ARM9 processor has Harvard architecture
7. ARM9E families use a Vector Floating Point (VFP) ARM coprocessor, which adds full floating point operands.VFP also provides fast development in SoC design when using tools like MatLab®

Key improvements over ARM7 cores, enabled by spending more transistors, include:

- Decreased heat production and lower overheating risk.
- Clock frequency improvements. Shifting from a three-stage pipeline to a five-stage one lets the clock speed be approximately doubled, on the same silicon fabrication process.
- Cycle count improvements.
- Faster loads and stores; many instructions now cost just one cycle. This is helped by both the modified Harvard architecture (reducing bus and cache contention) and the new pipeline stages.
- Exposing pipeline interlocks, enabling compiler optimizations to reduce blockage between stages.

Cranes Varsity

Additionally some ARM9 cores incorporate "Enhanced DSP" instructions, such as a multiply-to accumulate to support more efficient implementations of digital signal processing algorithms. Switching to a Harvard architecture entailed a non-unified cache, so that instruction fetches do not evict data (and vice versa). ARM9 cores have separate data and address bus signals, which chip designers use in various ways. In most cases they connect at least part of the address space in von Neumann style, used for both instructions and data, usually to an AHB interconnect connecting to a DRAM interface and an External Bus Interface usable with NOR flash memory. Such hybrids are no longer pure Harvard architecture processors.

ARM 11 Processors

The ARM11 processor family provides the engine that powers many smart phones in production today and is widely used in consumer, home and embedded applications. It delivers extreme low power and a range of performance from 350 MHz in small area designs up to 1 GHz in speed optimized designs in 45 and 65nm. ARM11 processor software is compatible with all previous generations of ARM processors and introduces 32 bit SIMD for media processing physically tagged caches to improve OS context switch performance, Trust zone for hardware enforced security and tightly coupled memories for real-time applications.

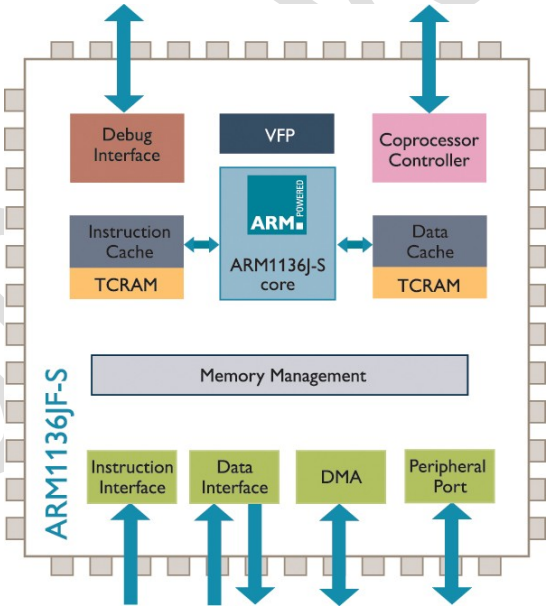


Figure:3 ARM 11

ARM11 Features:

1. Families with ARMv6 instruction set architecture that includes the Thumb® extensions for code density, Jazelle™ technology for Java™ acceleration, ARM DSP extensions, and SIMD media processing extensions. MMU supporting operating systems1 and palm OS

2. 32-bit RISC processor core with 8-stage integer pipeline and separate load-store and arithmetic pipelines to maximize instruction throughput.
3. Targets a performance range of Dhrystone MIPS 400 to 1200

Why ARM11:

Low risk and fast time to market

- Proven, well-understood and widely deployed family of processors
- Pre-verified supporting components and reference implementation flows

High performance in low-cost designs

- 800MHz to 1 GHz+ in 65G at under 2 mm^2
- 1 to 4 cores in an SMP cluster with ARM11 MP Core™
- ARM System IP, Physical IP, and available third party design support
- ARM926/AHB to ARM11/AXI migration simplified through AMBA® AHB-AXI bridge fabric

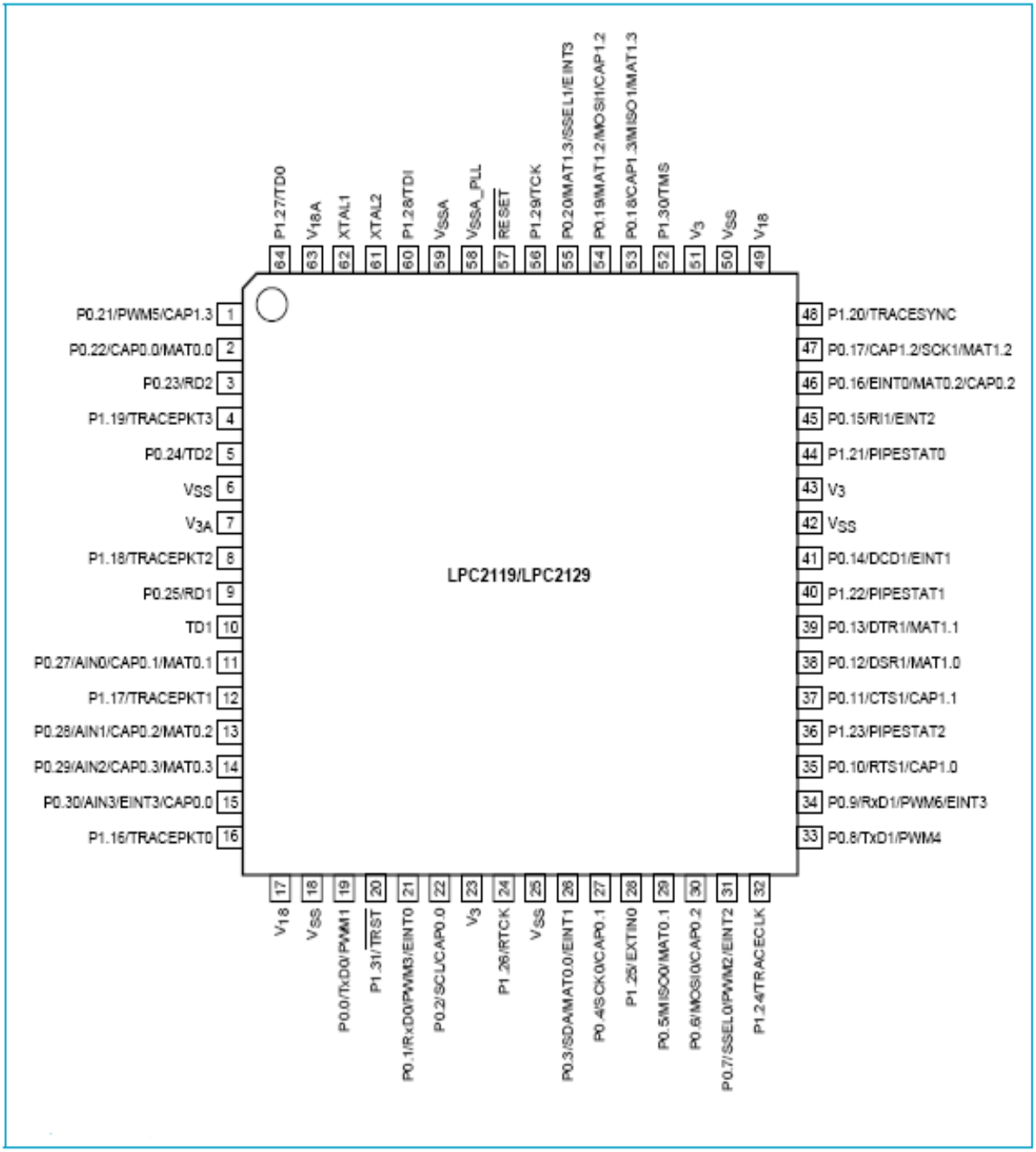
Compelling end-user experience

- Significant uplift over the ARM926EJ-S™ processor in media, OS, and browser performance
- Smartphone-class web browser support and extensive software and tool ecosystem

Cranes Varsity

CHAPTER – 3

PIN CONFIGURATION & PIN FUNCTION SELECT REGISTER



PINSEL0:

The PINSEL0 register controls the functions of the pins as per the settings listed in table, The direction control bit in the IO0DIR register is effective only when the GPIO function is selected for a pin. For other functions, direction is controlled automatically.

PINSEL0	Pin Name	Function when 00	Function when 01	Function when 10	Function when 11	Reset Value
1:0	P0.0	GPIO Port 0.0	TxD (UART0)	PWM1	Reserved	00
3:2	P0.1	GPIO Port 0.1	RxD (UART0)	PWM3	EINT0	00
5:4	P0.2	GPIO Port 0.2	SCL (I^2C)	Capture 0.0 (TIMER0)	Reserved	00
7:6	P0.3	GPIO Port 0.3	SDA (I^2C)	Match 0.0 (TIMER0)	EINT1	00
9:8	P0.4	GPIO Port 0.4	SCK (SPI0)	Capture 0.1 (TIMER0)	Reserved	00
11:10	P0.5	GPIO Port 0.5	MISO (SPI0)	Match 0.1 (TIMER0)	Reserved	00
13:12	P0.6	GPIO Port 0.6	MOSI (SPI0)	Capture 0.2 (TIMER0)	Reserved	00
15:14	P0.7	GPIO Port 0.7	SSEL (SPI0)	PWM2	EINT2	00
17:16	P0.8	GPIO Port 0.8	TxD UART1	PWM4	Reserved	00
19:18	P0.9	GPIO Port 0.9	RxD (UART1)	PWM6	EINT3	00
21:20	P0.10	GPIO Port 0.10	RTS (UART1)	Capture 1.0 (TIMER1)	Reserved	00
23:22	P0.11	GPIO Port 0.11	CTS (UART1)	Capture 1.1 (TIMER1)	Reserved	00
25:24	P0.12	GPIO Port 0.12	DSR (UART1)	Match 1.0 (TIMER1)	Reserved	00
27:26	P0.13	GPIO Port 0.13	DTR (UART1)	Match 1.1 (TIMER1)	Reserved	00
29:28	P0.14	GPIO Port 0.14	CD (UART1)	EINT1	Reserved	00
31:30	P0.15	GPIO Port 0.15	RI (UART1)	EINT2	Reserved	00

PINSEL1:

The PINSEL1 register controls the functions of the pins as per the settings listed in following tables. The direction control bit in the IO0DIR register is effective only when the GPIO function is selected for a pin. For other functions direction is controlled automatically.

PINSEL1	Pin Name	Function when 00	Function when 01	Function when 10	Function when 11	Reset Value
1:0	P0.16	GPIO Port 0.16	EINT0	Match 0.2 (TIMER0)	Capture 0.2 (TIMER0)	00
3:2	P0.17	GPIO Port 0.17	Capture 1.2 (TIMER1)	SCK (SPI1)	Match 1.2 (TIMER1)	00
5:4	P0.18	GPIO Port 0.18	Capture 1.3 (TIMER1)	MISO (SPI1)	Match 1.3 (TIMER1)	00
7:6	P0.19	GPIO Port 0.19	Match 1.2 (TIMER1)	MOSI (SPI1)	Match 1.3 (TIMER1)	00
9:8	P0.20	GPIO Port 0.20	Match 1.3 (TIMER1)	SSEL (SPI1)	EINT3	00
11:10	P0.21	GPIO Port 0.21	PWM5	Reserved	Capture 1.3 (TIMER1)	00
13:12	P0.22	GPIO Port 0.22	Reserved	Capture 0.0 (TIMER0)	Match 0.0 (TIMER0)	00
15:14	P0.23	GPIO Port 0.23	RD2 (CAN Controller 2)	Reserved	Reserved	00
17:16	P0.24	GPIO Port 0.24	TD2 (CAN Controller 2)	Reserved	Reserved	00
19:18	P0.25	GPIO Port 0.25	RD1 (CAN Controller 1)	Reserved	Reserved	00
21:20	P0.26	Reserved				00
23:22	P0.27	GPIO Port 0.27	AIN0 (A/D Converter)	Capture 0.1 (TIMER0)	Match 0.1 (TIMER0)	01
25:24	P0.28	GPIO Port 0.28	AIN1 (A/D Converter)	Capture 0.2 (TIMER0)	Match 0.2 (TIMER0)	01
27:26	P0.29	GPIO Port 0.29	AIN2 (A/D Converter)	Capture 0.3 (TIMER0)	Match 0.3 (TIMER0)	01
29:28	P0.30	GPIO Port 0.30	AIN3 (A/D Converter)	EINT3	Capture 0.0 (TIMER0)	01
31:30	P0.31	Reserved				00

PINSEL2

PINSEL2	Description	Reset Value
1:0	Reserved.	00
2	When 0, pins P1.36:26 are used as GPIO pins. When 1, P1.31:26 are used as a Debug port.	$\overline{\text{P1.26/RTCK}}$
3	When 0, pins P1.25:16 are used as GPIO pins. When 1, P1.25:16 are used as a Trace port.	$\overline{\text{P1.20/ TRACESYNC}}$

PINSEL2	Description	Reset Value
4:5	Reserved. **Note:** These bits must not be altered at any time. Changing them may result in an incorrect code execution.	11
6:31	Reserved.	NA

CHAPTER – 4

ARM7 GENERAL PURPOSE INPUT/OUTPUT (GPIO)

GPIO, or General Purpose Input/output, is the easiest way for you to interact with basic peripherals like buttons, LEDs, switches, and other components. It can also be used for more complex components like text and graphic LCDs, but for now we'll start with a few basic components that are relatively ease to get working.

Features

- Accelerated GPIO functions:
- All registers are byte and half-word addressable
- Entire port value can be written in one instruction
- Bit-level set and clear registers allow a single instruction set or clear of any number of bits in one port
- Direction control of individual bits
- All I/O default to inputs after reset

In order to get started with GPIO, you need to understand the four 'registers' that control it: IODIR, IOSET, IOCLR and IOPIN. Each of these registers is explained in detail below with some basic examples of how they work.

IODIR

IODIR controls the 'direction' of the GPIO pin. You use this register to set a GPIO pin to either Input (0) or Output (1). For example, if we want to use our GPIO pin to send signals 'out' from the microcontroller to some external device, we need to set a pin (for example GPIO 0.10) to 'Output' (1). We could do that with the following code:

```
IODIR0 |= (1 << 10);
```

If we want to use our GPIO pin to receive information from the outside world (reading it inside the microcontroller), we need to set GPIO 0.10 to 'Input' (0). That could be accomplished the following code:

```
IODIR0 &= ~ (1 << 10);
```

To set several pins to output at once we could do either of the following (which will produce identical code when compiled):

// Set GPIO 0.10, 0.11, and 0.15 to output

```
IODIR0 |= (1 << 10) | (1 << 11) | (1 << 15);
```

If your GPIO pin is set as *Output* (using the IODIR register mentioned above), you can use **IOSET** to set your GPIO pin to 'high' (providing a small 3.3V electrical current) or **IOCLR** to set it to 'low'

// Set GPIO 0.10, 0.11, and 0.15 to output using hexadecimal

```
IODIR0 |= 00008C00;
```

IOSET and IOCLR

If your GPIO pin is set as Output (using the IODIR register mentioned above), you can use IOSET to set your GPIO pin to 'high' (providing a small 3.3V electrical current) or IOCLR to set it to 'low' (providing a connection to GND). You shouldn't really think about high being 'on' and low being 'off', though, since ... as we'll see in the example below ... you can often turn a device 'on' by setting it low and 'off' by setting it 'high', depending on how the components are connected.

Let's take the LPC-P2148 board as an example. There are eight LEDs provided on this board for testing purposes, connected to GPIO pins of PORT 0. Looking at the schematic for this board, we can see that these two LEDs (located on the center right-hand side of the schematic) are connected to 3.3v on one side, and (GPIO pins of PORT 0) on the other side (with a 330 ohm resistor added to make sure the LEDs do not draw too much current and burn out).

// Make sure GPIO 0.10 and 0.11 are set to output

IODIR0 |= (1 << 10) | (1 << 11);

// Turn the LEDs on using IOSET

IOSET0 |= (1 << 10) | (1 << 11);

// Turn the LEDs off using IOCLR

IOCLR0 |= (1 << 10) | (1 << 11);

IOPIN

Regardless of whether your GPIO pin's direction is set to Input or Output, you can use the IOPIN register to read the current 'state' of every GPIO pin in the pin block (the collection of all 32 pins in GPIO0). A 1 value means that the pin is currently 'high', and a 0 value means the pin is currently 'low'.

You could read the IOPIN register, for example, to see if your LED was currently turned on or off, where IOPIN would return 0 for pin 10 (LED1) if it was currently turned on (since setting the GPIO pin to low turns the LED on) and 1 if the LED was off (since setting the GPIO pin high turns our LED off). Here's a simple method showing how to do this, including one way to read the value of a single pin from the 32-bit value returned by IOPIN. This method will return '1' if the supplied pin is currently 'high', and '0' if it is currently low:

```
int get_Pin_State (int pin_Number)
{
   int pin_Block_State = IOPIN0;      //Read the current state of all pins in GPIO block 0
                              // Read the value of 'pin_Number

   int pin_State = (pin Block State & (1 << pin Number)) ? 1: 0

   return (pin_State);                    // Return the value of pin_State
}
```

Example 1: sequential accesses to IOSET and IOCLR affecting the same GPIO pin/bit

State of the output configured GPIO pin is determined by writes into the pin's port IOSET and IOCLR registers. Last of these accesses to the IOSET/IOCLR register will determine the final output of a pin.

In case of a code:

```
IODIR0 = 0x00000080; pin P0.7 configured as output
IOCLR0 = 0x00000080; P0.7 goes LOW
IOSET0 = 0x00000080 ;P0.7 goes HIGH
IOCLR0 = 0x00000080 ;P0.7 goes LOW
```

pin P0.7 is configured as an output (write to IO0DIR register). After this, P0.7 output is set to low (first write to IO0CLR register). Short high pulse follows on P0.7 (write access to IO0SET), and the final write to IO0CLR register sets pin P0.7 back to low level.

Writing to IOSET/IOCLR .vs. IOPIN

Write to the IOSET/IOCLR register allows easy change of the port's selected output pin(s) to high/low level at a time. Only pin/bit(s) in the IOSET/IOCLR written with 1 will be set to high/low level, while those written as 0 will remain unaffected. However, by just writing to either IOSET or IOCLR register it is not possible to instantaneously output arbitrary binary data containing mixture of 0s and 1s on a GPIO port. Write to the IOPIN register enables instantaneous output of a desired content on the parallel GPIO. Binary data written into the IOPIN register will affect all output configured pins of that parallel port: 0s in the IOPIN will produce low level pin outputs and 1s in IOPIN will produce high level pin outputs. In order to change output of only a group of port's pins, application must logically AND readout from the IOPIN with mask containing 0s in bits corresponding to pins that will be changed, and 1s for all others. Finally, this result has to be logically ORred with the desired content and stored back into the IOPIN register. Example 2 from above illustrates output of 0xA5 on PORT0 pins 15 to 8 while preserving all other PORT0 output pins as they were before.

End of chapter 4

Cranes Varsity

CHAPTER – 5

PHASE LOCKED LOOP (PLL)

The PLL accepts an input clock frequency in the range of 10 MHz to 25 MHz only. The input frequency is multiplied up the range of 10 MHz to 75 MHz for the CCLK clock using a Current Controlled Oscillators (CCO). The multiplier can be an integer value from 1 to 32 (in practice, the multiplier value cannot be higher than 7 on the LPC21xx/LPC22xx due to the upper frequency limit of the CPU). The CCO operates in the range of 156 MHz to 320 MHz, so there is an additional divider in the loop to keep the CCO within its frequency range while the PLL is providing the desired output frequency. The output divider may be set to divide by 2, 4, 8, or 16 to produce the output clock. Since the minimum output divider value is 2, it is insured that the PLL output has a 50% duty cycle. A block diagram of the PLL is shown in Figure.

PLL activation is controlled via the PLLCON register. The PLL multiplier and divider values are controlled by the PLLCFG register. These two registers are protected in order to prevent accidental alteration of PLL parameters or deactivation of the PLL. Since all chip operations, including the Watchdog Timer, are dependent on the PLL when it is providing the chip clock, accidental changes to the PLL setup could result in unexpected behavior of the microcontroller. The protection is accomplished by a feed sequence

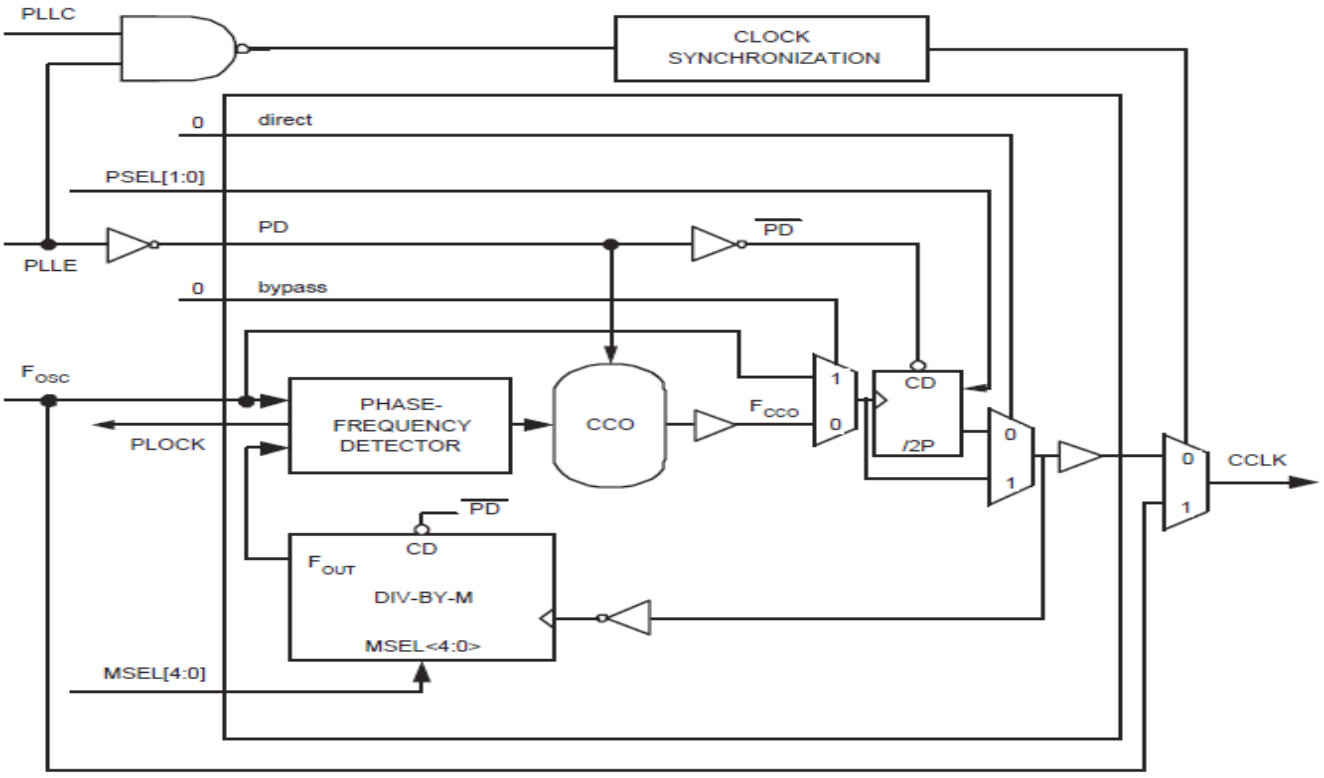
Similar to that of the Watchdog Timer. Details are provided in the description of the PLLFEED register. The PLL is turned off and bypassed following a chip reset and when by entering Power-down mode. The PLL is enabled by software only. The program must configure and activate the PLL, wait for the PLL to Lock, then connect to the PLL as a clock source.

The PLL is controlled by the registers shown in Table. More detailed descriptions follow

Generic Name	Description	Access	Reset value	System clock (PLL)Address & Name
PLLCON	PLL Control Register. Holding register for updating PLL control bits. Values written to this register do not take effect until a valid PLL feed sequence has taken place.	R/W	0	0xE01F C080 PLL0CON
PLLCFG	PLL Configuration Register. Holding register for updating PLL configuration values. Values written to this register do not take effect until a valid PLL feed sequence has taken place.	R/W	0	0xE01F C084 PLL0CFG
PLLSTAT	PLL Status Register. Read-back register for PLL control and configuration information. If PLLCON or PLLCFG have been written to, but a PLL feed sequence has not yet occurred, they will not reflect the current PLL state. Reading this register provides the actual values controlling the PLL, as well as the status of the PLL	RO	0	0xE01F C088 PLL0STAT
PLLFEED	PLL Feed Register. This register enables loading of the PLL control and configuration			

	information from the PLLCON and PLLCFG registers into the shadow registers that actually affect PLL operation	WO	NA	0xE01F C08C PLL0FEED

BLOCK DIAGRAM



PLL Control Register (PLLCON - 0xE01FC080)

The PLLCON register contains the bits that enable and connect the PLL. Enabling the PLL allows it to attempt to lock to the current settings of the multiplier and divider values. Connecting the PLL causes the processor and all chip functions to run from the PLL output clock. Changes to the PLLCON register do not take effect until a correct PLL feed sequence has been given in PLL Feed register and PLL Configuration register.

PLL Control Register (PLLCON - 0xE01FC080)bit description

Bit	Symbol	Description	Reset Value
0	PLLE	PLL Enable. When one, and after a valid PLL feed, this bit will activate the PLL and allow it to lock to the requested frequency. See PLLSTAT register.	0
1	PLLC	PLL Connect. When PLLC and PLLE are both set to one, and after a valid PLL feed, connects the PLL as the clock source for the microcontroller. Otherwise, the oscillator clock is used directly by the microcontroller. See PLLSTAT register	0
7:2	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined	NA

The PLL must be set up, enabled, and Lock established before it may be used as a clock source. When switching from the oscillator clock to the PLL output or vice versa, internal circuitry synchronizes the operation in order to ensure that glitches are not generated. Hardware does not insure that the PLL is locked before it is connected or automatically disconnect the PLL if lock is lost during operation. In the event of loss of PLL lock, it is likely that the oscillator clock has become unstable and disconnecting the PLL will not remedy the situation.

PLL Configuration Register (PLLCFG - 0xE01FC084)
The PLLCFG register contains the PLL multiplier and divider values. Changes to the PLLCFG register do not take effect until a correct PLL feed sequence has been given in PLL Feed register. Calculations for the PLL frequency, and multiplier and divider values are found in the PLL Frequency Calculation section.

PLL Configuration Register (PLLCFG - 0xE01FC084) bit description

Bit	Symbol	Description	Reset Value
4:0	MSEL	PLL Multiplier value. Supplies the value "M" in the PLL frequency calculations **Note:** For details on selecting the right value for MSEL see in "PLL frequency Calculation"	0
6:5	PSEL	PLL Divider value. Supplies the value "P" in the PLL frequency calculations **Note:** For details on selecting the right value for PSEL see in "PLL frequency Calculation"	0
7	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined	NA

PLL Status Register (PLLSTAT - 0xE01FC088)
The read-only PLLSTAT register provides the actual PLL parameters that are in effect at the time it is read, as well as the PLL status. PLLSTAT may disagree with values found in PLLCON and PLLCFG because changes to those registers do not take effect until a proper PLL feed has occurred see in "PLL Feed register"

Cranes Varsity

PLL Status Register (PLLSTAT - 0xE01FC088) bit description

Bit	Symbol	Description	Reset value
4:0	MSEL	Read-back for the PLL Multiplier value. This is the value currently used by the PLL.	0
6:5	PSEL	Read-back for the PLL Divider value. This is the value currently used by the PLL.	0
7	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
8	PLLE	Read-back for the PLL Enable bit. When one, the PLL is currently activated. When zero, the PLL is turned off. This bit is automatically cleared when Power-down mode is activated.	0
9	PLLC	Read-back for the PLL Connect bit. When PLLC and PLLE are both one, the PLL is connected as the clock source for the microcontroller. When either PLLC or PLLE is zero, the PLL is bypassed and the oscillator clock is used directly by the microcontroller. This bit is automatically cleared when Power-down mode is activated.	0
10	PLOCK	Reflects the PLL Lock status. When zero, the PLL is not locked. When one, the PLL is locked onto the requested frequency.	0
15:11	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

PLL Interrupt

The PLOCK bit in the PLLSTAT register is connected to the interrupt controller. This allows for software to turn on the PLL and continue with other functions without having to wait for the PLL to achieve lock. When the interrupt occurs (PLOCK = 1), the PLL may be connected, and the interrupt disabled. For details on how to enable and disable the PLL interrupt, see in “Interrupt Enable register (VICIntEnable - 0xFFFF F010)”and “Interrupt Enable Clear register (VICIntEnClear -0xFFFF F014)”.

PLL interrupt is available only in PLL0, i.e. the PLL that generates the CCLK. USB dedicated PLL1 does not have this capability.

PLL Modes

The combinations of PLLE and PLLC are shown in Table

PLLC	PLLE	PLL Function
0	0	PLL is turned off and disconnected. The CCLK equals the unmodified clock input. This combination can not be used in case of the PLL1 since there will be no 48 MHz clock and the USB can not operate.
0	1	The PLL is active, but not yet connected. The PLL can be connected after PLOCK is asserted.
1	0	Same as 00 combination. This prevents the possibility of the PLL being connected without also being enabled.
1	1	The PLL is active and has been connected. CCLK/system clock is sourced from the PLL0 and the USB clock is sourced from the PLL1.

PLL Control bit combinations

PLL Feed Register (PLLFEED - 0xE01FC08C)

A correct feed sequence must be written to the PLLFEED register in order for changes to the PLLCON and PLLCFG registers to take effect. The feed sequence is:

- Write the value 0xAA to PLLFEED.
- Write the value 0x55 to PLLFEED.

The two writes must be in the correct sequence, and must be consecutive APB bus cycles. The latter requirement implies that interrupts must be disabled for the duration of the PLL feed operation. If either of the feed values is incorrect, or one of the previously mentioned conditions is not met, any changes to the PLLCON or PLLCFG register will not become effective.

PLL Feed Register (PLLFEED - 0xE01FC08C)bit description

Bit	Symbol	Description	Reset value
7:0	PLLFEED	The PLL feed sequence must be written to this register in order for PLL configuration and control register changes to take effect.	0x00

PLL and Power-down mode

Power-down mode automatically turns off and disconnects activated PLL(s). Wakeup from Power-down mode does not automatically restore the PLL settings, this must be done in software. Typically, a routine to activate the PLL, wait for lock, and then connect the PLL can be called at the beginning of any interrupt service routine that might be called due to the wakeup. It is important not to attempt to restart the PLL by simply feeding it when execution resumes after a wakeup from Power-down mode. This would enable and connect the PLL at the same time, before PLL lock is established.

If activity on the USB data lines is not selected to wake up the microcontroller from Power-down mode , both the system and the USB PLL will be automatically be turned off and disconnected when Power-down mode is invoked, as described above. However, in case USBWAKE = 1 and USB_need_clock = 1 it is not possible to go into Power-down mode and any attempt to set the PD bit will fail, leaving the PLLs in the current state.

PLL frequency calculation

The PLL equations use the following parameters:

Elements determining PLL's frequency

Element	Description
F_{OSC}	the frequency from the crystal oscillator/external oscillator
F_{CCO}	the frequency of the PLL current controlled oscillator
CCLK	the PLL output frequency (also the processor clock frequency)
M	PLL Multiplier value from the MSEL bits in the PLLCFG register
P	PLL Divider value from the PSEL bits in the PLLCFG register

The PLL output frequency (when the PLL is both active and connected) is given by:
CCLK = M x FOSC or CCLK = FCCO / (2 x P)
The CCO frequency can be computed as:
FCCO = CCLK x 2 x P or FCCO = FOSC x M x 2 x P
The PLL inputs and settings must meet the following:

- FOSC is in the range of 10 MHz to 25 MHz
- CCLK is in the range of 10 MHz to Fmax (the maximum allowed frequency for the microcontroller - determined by the system microcontroller is embedded in).
- FCCO is in the range of 156 MHz to 320 MHz

Procedure for determining PLL settings

If a particular application uses the PLL0, its configuration may be determined as follows:

1. Choose the desired processor operating frequency (CCLK). This may be based on processor throughput requirements, need to support a specific set of UART baud rates, etc. Bear in mind that peripheral devices may be running from a lower clock than the processor.

2. Choose an oscillator frequency (FOSC). CCLK must be the whole (non-fractional) multiple of FOSC.

3. Calculate the value of M to configure the MSEL bits. M = CCLK / FOSC. M must be in the range of 1 to 32. The value written to the MSEL bits in PLLCFG is M x1

4. Find a value for P to configure the PSEL bits, such that FCCO is within its defined frequency limits. FCCO is calculated using the equation given above. P must have one of the values 1, 2, 4, or 8. The value written to the PSEL bits in PLLCFG is 00 for P = 1; 01 for P = 2; 10 for P = 4; 11 for P = 8

Remark: if a particular application is using the USB peripheral, the PLL1 must be configured since this is the only available source of the 48 MHz clock required by the USB. This limits the selection of FOSC to either 12 MHz, 16 MHz or 24 MHz.

PLL Divider values

PSEL Bits (PLLCFG bits [6:5])	Value of P
00	1
01	2
10	4
11	8

PLL Multiplier values

MSEL Bits (PLLCFG bits [4:0])	Value of M
00000	1
00001	2
00010	3
00011	4
...	...
11110	31
11111	32

PLL configuring examples

Example 1: an application not using the USB - configuring the PLL0
System design asks for FOSC= 10 MHz and requires CCLK = 60 MHz.
Based on these specifications, M = CCLK / Fosc = 60 MHz / 10 MHz = 6. Consequently,
M - 1 = 5 will be written as PLLCFG [4:0].

Value for P can be derived from P = FCCO / (CCLK x 2), using condition that FCCO must be in range of 156 MHz to 320 MHz Assuming the lowest allowed frequency for FCCO = 156 MHz, P = 156 MHz / (2 x 60 MHz) = 1.3. The highest FCCO frequency criteria produces P = 2.67. The only solution for P that satisfies both of these requirements and P = 2. Therefore, PLLCFG [6:5] = 1 will be used.

Cranes Varsity

CHAPTER - 6

VECTORED INTERRUPT CONTROLLER (VIC)

Features

- ARM PrimeCell Vectored Interrupt Controller
- 32 interrupt request inputs.
- 16 vectored IRQ interrupts.
- 16 priority levels dynamically assigned to interrupt requests
- Software interrupt generation

An interrupt is the occurrence of an event that causes a temporary suspension of a program while the condition is serviced by another program.
Allow a system to respond to an event and deal with the event while another program is executing. An interrupt driven system gives the illusion of doing many things simultaneously.
Of course, the CPU cannot execute more than one instruction at a time.
It can temporarily suspend execution of one program, execute another, then return to the first program.

The other program known as ISR's are like subroutines. Except that one does not know when the interrupt code will be executed. The Vectored Interrupt Controller (VIC) takes 32 interrupt request inputs and programmable assigns them into 3 categories, FIQ, vectored IRQ, and non-vectored IRQ. The programmable assignment scheme means that priorities of interrupts from the various peripherals can be dynamically assigned and adjusted. Fast Interrupt reQuest (FIQ) requests have the highest priority. If more than one request is assigned to FIQ, the VIC ORs the requests to produce the FIQ signal to the ARM processor. The fastest possible FIQ latency is achieved when only one request is classified as FIQ, because then the FIQ service routine can simply start dealing with that device. But if more than one request is assigned to the FIQ class, the FIQ service routine can read a word from the VIC that identifies which FIQ source(s) is (are) requesting an interrupt. Vectored IRQs have the middle priority, but only 16 of the 32 requests can be assigned to this category. Any of the 32 requests can be assigned to any of the 16 vectored IRQ slots, among which slot 0 has the highest priority and slot 15 has the lowest. Non-vectored IRQs have the lowest priority.The VIC ORs the requests from all the vectored and non-vectored IRQs to produce the IRQ signal to the ARM processor. The IRQ service routine can start by reading a register from the VIC and jumping there. If any of the vectored IRQs are requesting, the VIC provides the address of the highest-priority requesting IRQs service routine, otherwise it provides the address of a default routine that is shared by all the non-vectored IRQs. The default routine can read another VIC register to see what IRQs are active. All registers in the VIC are word registers. Byte and half word reads and write are not supported.

VIC registers

The following section describes the VIC registers in the order in which they are used in the VIC logic, from those closest to the interrupt request inputs to those most abstracted for use by software. For most people, this is also the best order to read about the registers when learning the VIC.

Interrupt Enable register (VICIntEnable - 0xFFFF F010)

This is a read/write accessible register. This register controls which of the 32 interrupt requests and software interrupts contribute to FIQ or IRQ.

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	CAN4 RX	CAN3 RX	CAN2 RX	CAN1 RX	FULLCAN	-
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	23	22	21	20	19	18	17	16
Symbol	CAN4 TX	CAN3 TX	CAN2 TX	CAN1 TX	CAN Common	ADC	EINT3	EINT2
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	15	14	13	12	11	10	9	8
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C	PWM
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	7	6	5	4	3	2	1	0
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	WO	WO	WO	WO	WO	WO	WO	WO

Interrupt Enable Clear register (VICIntEnClear - 0xFFFF F014)

This is a write only register. This register allows software to clear one or more bits in the Interrupt Enable register.

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	CAN4 RX	CAN3 RX	CAN2 RX	CAN1 RX	FULLCAN	-
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	23	22	21	20	19	18	17	16
Symbol	CAN4 TX	CAN3 TX	CAN2 TX	CAN1 TX	CAN Common	ADC	EINT3	EINT2
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	15	14	13	12	11	10	9	8
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C	PWM
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	7	6	5	4	3	2	1	0
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	WO	WO	WO	WO	WO	WO	WO	WO

Interrupt Select register (VICIntSelect - 0xFFFF F00C)

This is a read/write accessible register. This register classifies each of the 32 interrupt requests as contributing to FIQ or IRQ.

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	CAN4 RX	CAN3 RX	CAN2 RX	CAN1 RX	FULLCAN	-
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	23	22	21	20	19	18	17	16
Symbol	CAN4 TX	CAN3 TX	CAN2 TX	CAN1 TX	CAN Common	ADC	EINT3	EINT2
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	15	14	13	12	11	10	9	8
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C	PWM
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	7	6	5	4	3	2	1	0
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	WO	WO	WO	WO	WO	WO	WO	WO

IRQ Status register (VICIRQStatus - 0xFFFF F000)

This is a read only register. This register reads out the state of those interrupt requests that are enabled and classified as IRQ. It does not differentiate between vectored and non-vectored IRQs.

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	CAN4 RX	CAN3 RX	CAN2 RX	CAN1 RX	FULLCAN	-
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	23	22	21	20	19	18	17	16
Symbol	CAN4 TX	CAN3 TX	CAN2 TX	CAN1 TX	CAN Common	ADC	EINT3	EINT2
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	15	14	13	12	11	10	9	8
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C	PWM
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	7	6	5	4	3	2	1	0
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	WO	WO	WO	WO	WO	WO	WO	WO

FIQ Status register (VICFIQStatus - 0xFFFF F004)

This is a read only register. This register reads out the state of those interrupt requests that are enabled and classified as FIQ. If more than one request is classified as FIQ, the FIQ service routine can read this register to see which request(s) is (are) active.

Bit	31	30	29	28	27	26	25	24
Symbol	-	-	CAN4 RX	CAN3 RX	CAN2 RX	CAN1 RX	FULLCAN	-
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	23	22	21	20	19	18	17	16
Symbol	CAN4 TX	CAN3 TX	CAN2 TX	CAN1 TX	CAN Common	ADC	EINT3	EINT2
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	15	14	13	12	11	10	9	8
Symbol	EINT1	EINT0	RTC	PLL	SPI1/SSP	SPI0	I2C	PWM
Access	WO	WO	WO	WO	WO	WO	WO	WO
Bit	7	6	5	4	3	2	1	0
Symbol	UART1	UART0	TIMER1	TIMER0	ARMCore1	ARMCore0	-	WDT
Access	WO	WO	WO	WO	WO	WO	WO	WO

Vector Control registers 0-15 (VICvectCntl0-15 - 0xFFFF F200-23C)

These are a read/write accessible registers. Each of these registers controls one of the 16 vectored IRQ slots. Slot 0 has the highest priority and slot 15 the lowest. Note that disabling a vectored IRQ slot in one of the VICVectCntl registers does not disable the interrupt itself, the interrupt is simply changed to the non-vectored form.

Table 46. Vector Control registers (VICVectCntl0-15 - addresses 0xFFFF F200-23C) bit description

VICVectCntl0-15	Description	Reset value
4:0	The number of the interrupt request or software interrupt assigned to this vectored IRQ slot. As a matter of good programming practice, software should not assign the same interrupt number to more than one enabled vectored IRQ slot. But if this does occur, the lower numbered slot will be used when the interrupt request or software interrupt is enabled, classified as IRQ, and asserted.	0
5	1: this vectored IRQ slot is enabled, and can produce a unique ISR address when its assigned interrupt request or software interrupt is enabled, classified as IRQ, and asserted.	0
31:6	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

For example, the following two lines assign slot 0 to SPI0 IRQ interrupt request(s) and slot 1 to TIMER0 IRQ interrupt request(s):

```
VICVectCntl0 = 0x20 | 10;
VICVectCntl1 = 0x20 | 4;
```

Vector Address registers 0-15 (VICVectAddr0-15 - 0xFFFF F100-13C)

These are a read/write accessible registers. These registers hold the addresses of the Interrupt Service routines (ISRs) for the 16 vectored IRQ slots.

Table 47. Vector Address registers (VICVectAddr0-15 - addresses 0xFFFF F100-13C) bit description

VICVectAddr0-15	Description	Reset value
31:0	When one or more interrupt request or software interrupt is (are) enabled, classified as IRQ, asserted, and assigned to an enabled vectored IRQ slot, the value from this register for the highest-priority such slot will be provided when the IRQ service routine reads the Vector Address register -VICVectAddr (Section 5.5.10).	0

Default Vector Address register (VICDefVectAddr - 0xFFFF F034)

This is a read/write accessible register. This register holds the address of the Interrupt Service routine (ISR) for non-vectored IRQs.

VICDefVectAddr	Description	Reset value
31:0	When an IRQ service routine reads the Vector Address register (VICVectAddr), and no IRQ slot responds as described above, this address is returned.	0

Vector Address register (VICVectAddr - 0xFFFF F030)

This is a read/write accessible register. When an IRQ interrupt occurs, the IRQ service routine can read this register and jump to the value read.

Table 49. Vector Address register (VICVectAddr - address 0xFFFF F030) bit description

VICVectAddr	Description	Reset value
31:0	If any of the interrupt requests or software interrupts that are assigned to a vectored IRQ slot is (are) enabled, classified as IRQ, and asserted, reading from this register returns the address in the Vector Address Register for the highest-priority such slot (lowest-numbered) such slot. Otherwise it returns the address in the Default Vector Address Register. Writing to this register does not set the value for future reads from it. Rather, this register should be written near the end of an ISR, to update the priority hardware.	0

External interrupt registers

EXTINT : Corresponding bit is set when an external interrupt (0-3) occurs.

-	-	-	-	EINT3	EINT2	EINT1	EINT0

EXTMODE

This register is used to select whether the interrupt is level sensitive or edge sensitive. Writing logical 1 means the interrupt is edge sensitive and '0' means level sensitive.

-	-	-	-	EXTMODE3	EXTMODE2	EXTMODE1	EXTMODE0

EXTPOLAR

In level-sensitive mode, the bits in this register select whether the corresponding pin is high- or low-active. In edge-sensitive mode, they select whether the pin is rising- or falling-edge sensitive. 1 on this bit means high active or rising edge and 0 means low active or falling edge.

-	-	-	-	EXTPOLAR3	EXTPOLAR2	EXTPOLAR1	EXTPOLAR0

End of chapter 6

Cranes Varsity

CHAPTER – 7

TIMER

Features

- A 32-bit Timer/Counter with a programmable 32-bit Prescaler.
- Counter or Timer operation
- Up to four 32-bit capture channels per timer, that can take a snapshot of the timer value when an input signal transitions. A capture event may also optionally generate an interrupt.
- Four 32-bit match registers that allow:
  – Continuous operation with optional interrupt generation on match.
  – Stop timer on match with optional interrupt generation.
  – Reset timer on match with optional interrupt generation.
- Up to four external outputs corresponding to match registers, with the following capabilities:
  – Set low on match.
  – Set high on match.
  – Toggle on match.
  – Do nothing on match.

Applications

- Interval Timer for counting internal events.
- Pulse Width Demodulator via Capture inputs.
- Free running timer.

Description

The Timer/Counter is designed to count cycles of the peripheral clock (PCLK) or an externally-supplied clock, and can optionally generate interrupts or perform other actions at specified timer values, based on four match registers. It also includes four capture inputs to trap the timer value when an input signal transitions, optionally generating an interrupt.

Cranes Varsity

MAT0.3..0 MAT1.3..0	Output	External Match Output 0/1- When a match register 0/1 (MR3:0) equals the timer counter (TC) this output can either toggle, go low, go high, or do nothing. The External Match Register (EMR) controls the functionality of this output. Match Output functionality can be selected on a number of pins in parallel. It is also possible for example, to have 2 pins selected at the same time so that they provide MAT1.3 function in parallel. Here is the list of all MATCH signals, together with pins on where they can be selected: • MAT0.0 (2 pins): P0.3 and P0.22 • MAT0.1 (2 pins): P0.5 and P0.27 • MAT0.2 (2 pin): P0.16 and P0.28 • MAT0.3 (1 pin): P0.29 • MAT1.0 (1 pin): P0.12 • MAT1.1 (1 pin): P0.13 • MAT1.2 (2 pins): P0.17 and P0.19 • MAT1.3 (2 pins): P0.18 and P0.20

Pin	Type	Description
CAP0.3..0 CAP1.3..0	Input	Capture Signals- A transition on a capture pin can be configured to load one of the Capture Registers with the value in the Timer Count and optionally generate an interrupt. Capture functionality can be selected from a number of pins. When more than one pin is selecte for a Capture input on a single TIMER0/1 channel, the pin with the lowest Port number is used. If for example pins 30 (P0.6) and 46 (P0.16) are selected for CAP0.2, only pin 30 will be used by TIMER0 perform CAP0.2 function. Here is the list of all CAPTURE signals, together with pins on where they can be selected: • CAP0.0 (3 pins): P0.2, P0.22 and P0.30 • CAP0.1 (2 pins): P0.4 and P0.27 • CAP0.2 (3 pin): P0.6, P0.16 and P0.28 • CAP0.3 (1 pin): P0.29 • CAP1.0 (1 pin): P0.10 • CAP1.1 (1 pin): P0.11 • CAP1.2 (2 pins): P0.17 and P0.19 • CAP1.3 (2 pins): P0.18 and P0.21 Timer/Counter block can select a capture signal as a clock source

Pin Type Description

CAP0.3...0 & CAP1.3..0

Input Capture Signals- A transition on a capture pin can be configured to load one of the Capture Registers with the value in the Timer Counter and optionally generate an interrupt. Capture functionality can be selected from a number of pins. When more than one pin is selected for a Capture input on a single TIMER0/1 channel, the pin with the lowest Port number is used. If for example pins 30 (P0.6) and

46 (P0.16) are selected for CAP0.2, only pin 30 will be used by TIMER0 to perform CAP0.2 function. Here is the list of all CAPTURE signals, together with pins on where they can be selected:

- CAP0.0 (3 pins): P0.2, P0.22 and P0.30
- CAP0.1 (2 pins): P0.4 and P0.27
- CAP0.2 (3 pin): P0.6, P0.16 and P0.28
- CAP0.3 (1 pin): P0.29
- CAP1.0 (1 pin): P0.10
- CAP1.1 (1 pin): P0.11
- CAP1.2 (2 pins): P0.17 and P0.19
- CAP1.3 (2 pins): P0.18 and P0.21

Timer/Counter block can select a capture signal as a clock source Instead of the PCLK derived clock. For more details

Register description

MAT0.3..0 & MAT1.3..0

Output External Match Output 0/1- When a match register 0/1 (MR3:0) equals the timer counter (TC) this output can either toggle, go low, go high, or do nothing. The External Match Register (EMR) controls the functionality of this output. Match Output functionality can be selected on a number of pins in parallel. It is also possible for example, to have 2 pins selected at the same time so that they provide MAT1.3 function in parallel.

Here is the list of all MATCH signals, together with pins on where they can be selected:

- MAT0.0 (2 pins): P0.3 and P0.22
- MAT0.1 (2 pins): P0.5 and P0.27
- MAT0.2 (2 pin): P0.16 and P0.28
- MAT0.3 (1 pin): P0.29
- MAT1.0 (1 pin): P0.12
- MAT1.1 (1 pin): P0.13
- MAT1.2 (2 pins): P0.17 and P0.19
- MAT1.3 (2 pins): P0.18 and P0.20

TIMER/COUNTER0 and TIMER/COUNTER1 register map

Cranes Varsity

Generic Name	Description	Acc
IR	Interrupt Register. The IR can be written to clear interrupts. The IR can be read to identify which of eight possible interrupt sources are pending.	R/W
TCR	Timer Control Register. The TCR is used to control the Timer Counter functions. The Timer Counter can be disabled or reset through the TCR.	R/W
TC	Timer Counter. The 32-bit TC is incremented every PR+1 cycles of PCLK. The TC is controlled through the TCR.	R/W
PR	Prescale Register. The Prescale Counter (below) is equal to this value, the next clock increments the TC and clears the PC.	R/W
PC	Prescale Counter. The 32-bit PC is a counter which is incremented to the value stored in PR. When the value in PR is reached, the TC is incremented and the PC is cleared. The PC is observable and controllable through the bus interface.	R/W
MCR	Match Control Register. The MCR is used to control if an interrupt is generated and if the TC is reset when a Match occurs.	R/W
MR0	Match Register 0. MR0 can be enabled through the MCR to reset the TC, stop both the TC and PC, and/or generate an interrupt every time MR0 matches the TC.	R/W
MR1	Match Register 1. See MR0 description.	R/W
MR2	Match Register 2. See MR0 description.	R/W
MR3	Match Register 3. See MR0 description.	R/W
CCR	Capture Control Register. The CCR controls which edges of the capture inputs are used to load the Capture Registers and whether or not an interrupt is generated when a capture takes place.	R/W
CR0	Capture Register 0. CR0 is loaded with the value of TC when there is an event on the CAPn.0(CAP0.0 or CAP1.0 respectively) input.	RO
CR1	Capture Register 1. See CR0 description.	RO
CR2	Capture Register 2. See CR0 description.	RO
CR3	Capture Register 3. See CR0 description.	RO
EMR	External Match Register. The EMR controls the external match pins MATn.0-3 (MAT0.0-3 and MAT1.0-3 respectively).	R/W
CTCR	Count Control Register. The CTCR selects between Timer and Counter mode, and in Counter mode selects the signal and edge(s) for counting.	R/W

Interrupt Register (IR, TIMER0: T0IR - 0xE000 4000 and TIMER1: T1IR- 0xE000 8000)

The Interrupt Register consists of four bits for the match interrupts and four bits for the capture interrupts. If an interrupt is generated then the corresponding bit in the IR will be high. Otherwise, the bit will be low. Writing a logic one to the corresponding IR bit will reset the interrupt. Writing a zero has no effect.

Bit	Symbol	Description	Reset value
0	MR0 Interrupt	Interrupt flag for match channel 0.	0
1	MR1 Interrupt	Interrupt flag for match channel 1.	0
2	MR2 Interrupt	Interrupt flag for match channel 2.	0
3	MR3 Interrupt	Interrupt flag for match channel 3.	0
4	CR0 Interrupt	Interrupt flag for capture channel 0 event.	0
5	CR1 Interrupt	Interrupt flag for capture channel 1 event.	0
6	CR2 Interrupt	Interrupt flag for capture channel 2 event.	0
7	CR3 Interrupt	Interrupt flag for capture channel 3 event.	0

Timer Control Register (TCR, TIMER0: T0TCR - 0xE000 4004 and TIMER1: T1TCR - 0xE000 8004)

The Timer Control Register (TCR) is used to control the operation of the Timer/Counter.

Bit	Symbol	Description	Reset value
0	Counter Enable	When one, the Timer Counter and Prescale Counter are enabled for counting. When zero, the counters are disabled.	0
1	Counter Reset	When one, the Timer Counter and the Prescale Counter are synchronously reset on the next positive edge of PCLK. The counters remain reset until TCR[1] is returned to zero.	0
7:2	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Count Control Register (CTCR, TIMER0: T0CTCR - 0xE000 4070 and TIMER1: T1TCR - 0xE000 8070)

The Count Control Register (CTCR) is used to select between Timer and Counter mode, and in Counter mode to select the pin and edge(s) for counting. When Counter Mode is chosen as a mode of operation, the CAP input (selected by the CTCR bits 3:2) is sampled on every rising edge of the PCLK clock. After comparing two consecutive samples of this CAP input, one of the following four events is recognized: rising edge, falling edge, either of edges or no changes in the level of the selected CAP input. Only if the identified event corresponds to the one selected by bits 1:0 in the CTCR register, the Timer Counter register will be incremented. Effective processing of the externally supplied clock to the counter has some limitations. Since two successive rising edges of the PCLK clock are used to identify only one edge on the CAP selected input, the frequency of the CAP input cannot exceed one half of the PCLK clock. Consequently, duration of the high/low levels on the same CAP input in this case cannot be shorter than 1/PCLK.

Bit	Symbol	Value	Description	Reset value
1:0	Counter/ Timer Mode		This field selects which rising PCLK edges can increment Timer's Prescale Counter (PC), or clear PC and increment Timer Counter (TC).	00
		00	Timer Mode: every rising PCLK edge	
		01	Counter Mode: TC is incremented on rising edges on the CAP input selected by bits 3:2.	
		10	Counter Mode: TC is incremented on falling edges on the CAP input selected by bits 3:2.	
		11	Counter Mode: TC is incremented on both edges on the CAP input selected by bits 3:2.	
3:2	Count Input Select		When bits 1:0 in this register are not 00, these bits select which CAP pin is sampled for clocking:	00
		00	CAPn.0 (CAP0.0 for TIMER0 and CAP1.0 for TIMER1)	
		01	CAPn.1 (CAP0.1 for TIMER0 and CAP1.1 for TIMER1)	
		10	CAPn.2 (CAP0.2 for TIMER0 and CAP1.2 for TIMER1)	
		11	CAPn.3 (CAP0.3 for TIMER0 and CAP1.3 for TIMER1)	
			Note: If Counter mode is selected for a particular CAPn input in the TnCTCR, the 3 bits for that input in the Capture Control Register (TnCCR) must be programmed as 000. However, capture and/or interrupt can be selected for the other 3 CAPn inputs in the same timer.	
7:4	-	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Timer Counter (TC, TIMER0: T0TC - 0xE000 400C and TIMER1: T1TC- 0xE000 8008

The 32-bit Timer Counter is incremented when the Prescale Counter reaches its terminal count. Unless it is reset before reaching its upper limit, the TC will count up through the value 0xFFFF FFFF and then wrap back to the value 0x0000 0000. This event does not cause an interrupt, but a Match register can be used to detect an overflow if needed.

Prescale Register (PR, TIMER0: T0PR - 0xE000 400C and TIMER1: T1PR - 0xE000 800C)

The 32-bit Prescale Register specifies the maximum value for the Prescale Counter.

Prescale Counter Register (PC, TIMER0: T0PC - 0xE000 4010 and TIMER1: T1PC - 0xE000 8010)

The 32-bit Prescale Counter controls division of PCLK by some constant value before it is applied to the Timer Counter. This allows control of the relationship of the resolution of the timer versus the maximum time before the timer overflows. The Prescale Counter is incremented on every PCLK. When it reaches the value stored in the Prescale Register, the Timer Counter is incremented and the Prescale Counter is reset on the next PCLK. This causes the TC to increment on every PCLK when PR = 0, every 2 PCLKs when PR = 1, etc.

Match Registers (MR0 - MR3)

The Match register values are continuously compared to the Timer Counter value. When the two values are equal, actions can be triggered automatically. The action possibilities are to generate an interrupt, reset the Timer Counter, or stop the timer. Actions are controlled by the settings in the MCR register.

Match Control Registers (MCR, TIMER0: T0MCR - 0xE000 4014 and TIMER1: T1MCR - 0xE000 8014)

The Match Control Register is used to control what operations are performed when one of the Match Registers matches the Timer Counter.

Bit	Symbol	Value	Description	Reset value
0	MR0I	1	Interrupt on MR0: an interrupt is generated when MR0 matches the value in the TC.	0
		0	This interrupt is disabled	
1	MR0R	1	Reset on MR0: the TC will be reset if MR0 matches it.	0
		0	Feature disabled.	
2	MR0S	1	Stop on MR0: the TC and PC will be stopped and TCR[0] will be set to 0 if MR0 matches the TC.	0
		0	Feature disabled.	
3	MR1I	1	Interrupt on MR1: an interrupt is generated when MR1 matches the value in the TC.	0
		0	This interrupt is disabled	
4	MR1R	1	Reset on MR1: the TC will be reset if MR1 matches it.	0
		0	Feature disabled.	
5	MR1S	1	Stop on MR1: the TC and PC will be stopped and TCR[0] will be set to 0 if MR1 matches the TC.	0
		0	Feature disabled.	
6	MR2I	1	Interrupt on MR2: an interrupt is generated when MR2 matches the value in the TC.	0
		0	This interrupt is disabled	
7	MR2R	1	Reset on MR2: the TC will be reset if MR2 matches it.	0
		0	Feature disabled.	
8	MR2S	1	Stop on MR2: the TC and PC will be stopped and TCR[0] will be set to 0 if MR2 matches the TC.	0
		0	Feature disabled.	
9	MR3I	1	Interrupt on MR3: an interrupt is generated when MR3 matches the value in the TC.	0
		0	This interrupt is disabled	
10	MR3R	1	Reset on MR3: the TC will be reset if MR3 matches it.	0
		0	Feature disabled.	
11	MR3S	1	Stop on MR3: the TC and PC will be stopped and TCR[0] will be set to 0 if MR3 matches the TC.	0
		0	Feature disabled.	
15:12	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Capture Registers (CR0 - CR3)

Each Capture register is associated with a device pin and may be loaded with the Timer Counter value when a specified event occurs on that pin. The settings in the Capture Control Register register determine whether the capture function is enabled, and whether a capture event happens on the rising edge of the associated pin, the falling edge, or on both edges.

Capture Control Register (CCR, TIMER0: T0CCR - 0xE000 4028 and TIMER1: T1CCR - 0xE000 8028)

The Capture Control Register is used to control whether one of the four Capture Registers is loaded with the value in the Timer Counter when the capture event occurs, and whether an interrupt is generated by the capture event. Setting both the rising and falling bits at the same time is a valid configuration, resulting in a capture event for both edges. In the description below, "n" represents the Timer number, 0 or 1.

Capture Control Register (CCR, TIMER0: T0CCR - address 0xE000 4028 and TIMER1: T1CCR – address 0xE000 8028) bit description

Bit	Symbol	Value	Description
0	CAP0RE	1	Capture on CAPn.0 rising edge: a sequence of 0 then 1 on CAPn.0 will cause CR0 to be loaded with the contents of TC.
		0	This feature is disabled.
1	CAP0FE	1	Capture on CAPn.0 falling edge: a sequence of 1 then 0 on CAPn.0 will cause CR0 to be loaded with the contents of TC.
		0	This feature is disabled.
2	CAP0I	1	Interrupt on CAPn.0 event: a CR0 load due to a CAPn.0 event will generate an interrupt.
		0	This feature is disabled.
3	CAP1RE	1	Capture on CAPn.1 rising edge: a sequence of 0 then 1 on CAPn.1 will cause CR1 to be loaded with the contents of TC.
		0	This feature is disabled.
4	CAP1FE	1	Capture on CAPn.1 falling edge: a sequence of 1 then 0 on CAPn.1 will cause CR1 to be loaded with the contents of TC.
		0	This feature is disabled.
5	CAP1I	1	Interrupt on CAPn.1 event: a CR1 load due to a CAPn.1 event will generate an interrupt.
		0	This feature is disabled.
6	CAP2RE	1	Capture on CAPn.2 rising edge: A sequence of 0 then 1 on CAPn.2 will cause CR2 to be loaded with the contents of TC.
		0	This feature is disabled.
7	CAP2FE	1	Capture on CAPn.2 falling edge: a sequence of 1 then 0 on CAPn.2 will cause CR2 to be loaded with the contents of TC.
		0	This feature is disabled.
8	CAP2I	1	Interrupt on CAPn.2 event: a CR2 load due to a CAPn.2 event will generate an interrupt.
		0	This feature is disabled.
9	CAP3RE	1	Capture on CAPn.3 rising edge: a sequence of 0 then 1 on CAPn.3 will cause CR3 to be loaded with the contents of TC.
		0	This feature is disabled.

10	CAP3FE	1	Capture on CAPn.3 falling edge: a sequence of 1 then 0 on CAPn.3 will cause CR3 to be loaded with the contents of TC
		0	This feature is disabled.
11	CAP3I	1	Interrupt on CAPn.3 event: a CR3 load due to a CAPn.3 event will generate an interrupt.
		0	This feature is disabled.
15:12	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.

External Match Register (EMR, TIMER0: T0EMR - 0xE000 403C; and TIMER1: T1EMR - 0xE000 803C)

The External Match Register provides both control and status of the external match pins MAT(0-3).

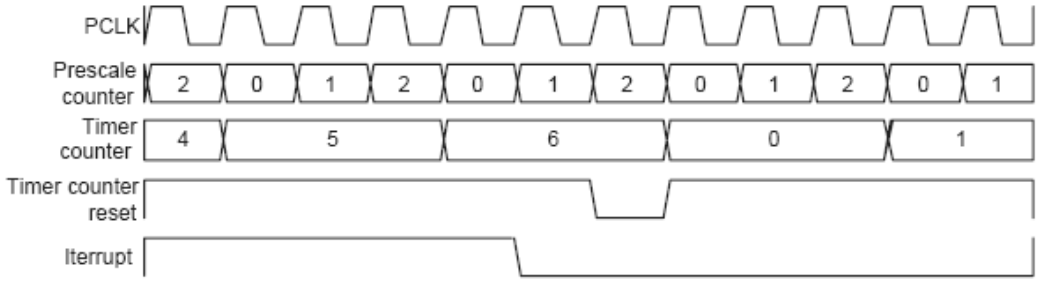
Bit	Symbol	Description
0	EM0	External Match 0. This bit reflects the state of output MAT0.0/MAT1.0, whether or not this output is connected to its pin. When a match occurs between the TC and MR0, this output of the timer can either toggle, go low, go high, or do nothing. Bits EMR[5:4] control the functionality of this output.
1	EM1	External Match 1. This bit reflects the state of output MAT0.1/MAT1.1, whether or not this output is connected to its pin. When a match occurs between the TC and MR1, this output of the timer can either toggle, go low, go high, or do nothing. Bits EMR[7:6] control the functionality of this output.
2	EM2	External Match 2. This bit reflects the state of output MAT0.2/MAT1.2, whether or not this output is connected to its pin. When a match occurs between the TC and MR2, this output of the timer can either toggle, go low, go high, or do nothing. Bits EMR[9:8] control the functionality of this output.
3	EM3	External Match 3. This bit reflects the state of output MAT0.3/MAT1.3, whether or not this output is connected to its pin. When a match occurs between the TC and MR3, this output of the timer can either toggle, go low, go high, or do nothing. Bits EMR[11:10] control the functionality of this output.
5:4	EMC0	External Match Control 0. Determines the functionality of External Match 0. Table 244 shows the encoding of these bits.
7:6	EMC1	External Match Control 1. Determines the functionality of External Match 1. Table 244 shows the encoding of these bits.
9:8	EMC2	External Match Control 2. Determines the functionality of External Match 2. Table 244 shows the encoding of these bits.
11:10	EMC3	External Match Control 3. Determines the functionality of External Match 3. Table 244 shows the encoding of these bits.
15:12	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.

External Match Control Register

EMR[11:10], EMR[9:8], EMR[7:6], or EMR[5:4]	Function
00	Do Nothing.
01	Clear the corresponding External Match bit/output to 0 (MATn.m pin is LOW if pinned out).
10	Set the corresponding External Match bit/output to 1 (MATn.m pin is HIGH if pinned out).
11	Toggle the corresponding External Match bit/output.

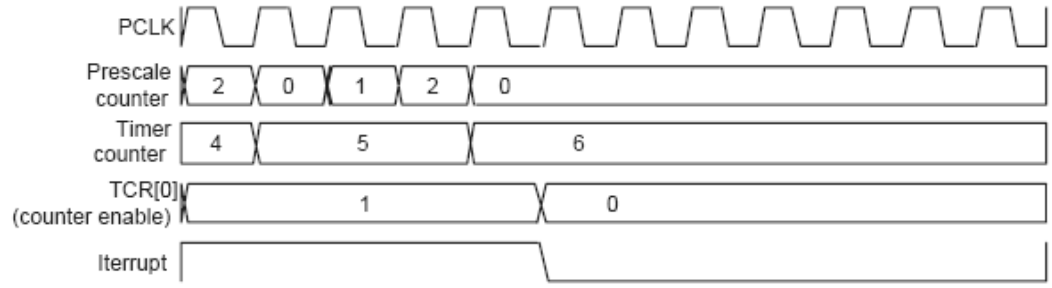
Example timer operation

Figure shows a timer configured to reset the count and generate an interrupt on match. The prescaler is set to 2 and the match register set to 6. At the end of the timer cycle where the match occurs, the timer count is reset. This gives a full length cycle to the match value. The interrupt indicating that a match occurred is generated in the next clock after the timer reached the match value.



A timer cycle in which PR=2, MRx=6, and both interrupt and reset on match are enabled

Below Figure shows a timer configured to stop and generate an interrupt on match. The prescaler is again set to 2 and the match register set to 6. In the next clock after the timer reaches the match value, the timer enable bit in TCR is cleared, and the interrupt indicating that a match occurred is generated.



. A timer cycle in which PR=2, MRx=6, and both interrupt and stop on match are enabled

CHAPTER – 8

ANALOG TO DIGITAL CONVERTOR (ADC)

How to receive and interpret analog data (i.e., 'interacting with the Real world').In our previous tutorial (on GPIO) we learned how to turn an LED on or off and check whether a button is currently pressed or not. As useful as that is, it's probably going to get dull pretty quickly since you can only do so many things with a simple button or LED.

The reality is that a great many of the devices that you will want to interact with aren't going to be digital, or consist of only two possible 'states' (On/Off, True/False, etc.). Almost every dial or knob on any modern electronic device, for example, is probably analog. Since our microcontrollers are digital, what that means is that we need to find a way to convert those analog signals into something 'digital' that our microcontroller can actually understand. That's where Analog to Digital Converters (ADCs) comes in, and thankfully for us the LPC2129 has two of them built in.

ADCs essentially act as a bridge (or a 'translator') between the messy outside analog world and the cozy, black and white (green and white?) digital world our microcontrollers live in. They work by converting voltage to a numeric value that the microcontroller can understand. For example, with an internal voltage of 3.3V (which is the Vref on the LPC2129) and your ADC set to return the maximum 10-bit data (meaning you have possible values between 0 and 1023), 0.0V would return 0, 3.3V (or higher) would return 1023, and 1.65V would return ~512.

They only work in one direction (reading data from outside and sending it 'into' the chip), but life without them would be a lot more challenging ... or at the very least a lot more expensive (analog devices are often much cheaper than their digital counterparts). In this lesson, we're going to show you how to use several analog devices to accomplish some common tasks with an analog to digital converter:

1. Determine the current position on a simple rotary dial (aka 'potentiometer')
2. Determine where a user is currently touching a touch screen
3. Measure the distance between two points with an ultrasonic range finder

How does the ADC convert a signal?

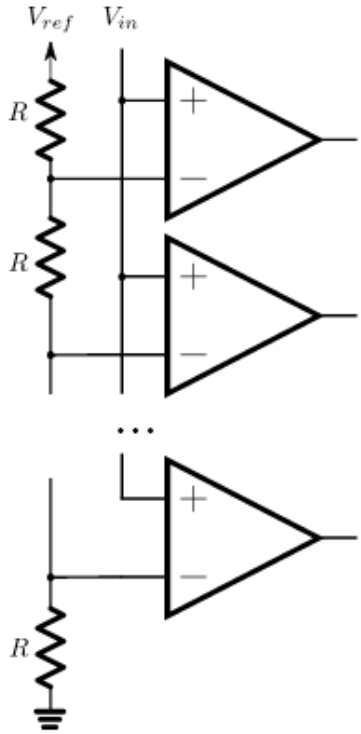
Many ways have been developed to convert an analog signal, each with its strengths and weaknesses. The choice of the ADC for a given application is usually defined by the requirements you have: if you need speed, use a fast ADC; if you need precision, use an accurate ADC; if you are constrained in space, use a compact ADC.
All ADCs work under the same principle: they need to convert a signal to a certain number of bits N. The sequence of bits represents the number and each bit has the double of the weight of the next, starting from the Most Significant Bit (MSB) up to the Least Significant Bit (LSB). In a nutshell, we want to find the sequence of bits bN−1, bN−2... b0 that represents the analog value Vin as
Vin=∑n=0N−1bn2nVref2N.
The MSB has weight Vref/2, the next Vref/4, etc., and the LSB has weight Vref/2N. Therefore, more bits leads to more precision in the digital representation. Here we simplify the range to be between 0 and Vref, although the range may be between any two values.

Let's talk about the following ADCs (although there are more):

- Flash or Direct Conversion
- Ramp Compare
- Successive-Approximations Register (SAR)

Flash ADCs

Flash converters have a resistive ladder that divides the reference voltage in 2N equal parts. For each part, a comparator compares the input signal with the voltage supplied by that part of the resistive ladder. The output of all the comparators is like a thermometer: the higher the input value, more comparators have their outputs high from bottom to top. A dedicated component called "Priority Encoder" translates this gauge into a binary code, which corresponds to the position of the last comparator with high output, counting from the bottom up.



Ramp up

First, a voltage ramps up with slope proportional to the input voltage Vin for a fixed period of time. This can be achieved, for example, with a current source proportional to the input voltage charging a capacitor. The voltage at the end of that integration time is:
Vout(tint)=Vinτtint+Vinitial,

where tint is the integration time, 1/τ is the slope proportionality factor and Vinitial is the initial voltage, say zero.

Ramp down

Second, the output voltage ramps down with slope proportional to a fixed voltage Vref. Note that in the first phase, the slope is variable and the integration time is fixed. Now, the slope is fixed and the integration time is variable: the voltage ramps down until it reaches zero, which, by intuition, should take a period of time proportional to the input voltage. Let's see if intuition is right. The time it takes for the voltage to reach zero is:

$0 = -V_{ref}\tau t + V_{out}(t_{int})$

$0 = -V_{ref}\tau t + V_{in}\tau t_{int}$

$t = \frac{V_{in}}{V_{ref}} t_{int}$

Where we have assumed that Vinitial=0. As you can see, the time it takes the voltage to reach zero is indeed proportional to Vin, while the other terms are known. During ramp down, a counter counts the number of clocks until the output voltage reaches zero. The number in the counter is then proportional to the input voltage.

The slider below controls the input voltage of the ADC. When you change the input voltage, you can see in the plot above that a) the slope of the ramp up changes with Vin and b) Vout reaches 0 in a period of time proportional to Vin.

Strengths

- Very precise. The sources of errors are only the comparison with zero and the clock period.

Weaknesses

- Slow. The ADC needs time to ramp up and down the output voltage and doubles with each bit added to the representation, for a fixed clock period.

Successive Approximation Register (SAR) converters

A Successive Approximation Register converter evaluates each bit at a time, from the most to the least significant bits. They successively approach the output of a digital-analog converter (DAC) in them to the input voltage. The input of the DAC is stored in a N bit register, which is also the output of the ADC. Let's see the flow of this ADC with the aid of the picture below. First, the analog signal is sampled and kept fixed. If the input value is changed during the conversion, the result can be completely wrong. Then, the bit $N-1$ of the register is set to 1 and every other bit to 0. Since the reference voltage of the DAC is Vref, its output is set to Vref/2. The output of the comparator bk is latched to the MSB $bN-1$, i.e., if $V_{in} < V_{ref}/2$, then $bN-1$ is reset to 0, otherwise it stays 1. By successively setting the next bit to 1, comparing the output of the DAC with the input voltage and latching the result in the same bit, the **converter is generating a signal from the register that is successively approximating the input value** (hence its name).

SAR ADC architecture

In a nutshell, a SAR follows these two steps for each bit, from most to least significant bit, after resetting the register value to 0:

- **Comparison:** Set bit to 1 and compare the output of the DAC with the input voltage
- **Latching:** Latch the result of the comparator to the same bit in the register

The interactive plot below shows the input signal and the output of the DAC during the comparison phase.
The slider below controls the input voltage of an 8-bit (8 cycles) SAR ADC. When you change the input voltage, you can see in the plot above that the output of the DAC tends to get closer to the input signal as more bits are defined. To see if a particular bit was set or cleared during the latching phase, you have to see if the output of the DAC at the next cycle is above or below the value of the previous cycle.

Strengths

- It uses only one comparator
- Low power consumption

Weaknesses

- The DAC grows with the number of bits
- They take as many cycles to convert the signal as the number of bits
- The component mismatch in the DAC limits its linearity (and therefore of the ADC) to around 12bits

Features

• 10 bit successive approximation analog to digital converter.
• Input multiplexing among 4 pins (LPC2119)
• Power down mode
• Measurement range 0 to 3 V
• 10 bit conversion time >= 2.44 uS
• Burst conversion mode for single or multiple inputs

• Optional conversion on transition on input pin or Timer Match signal

Before we jump into the examples above, though, let's get started with the absolute basics. In the case of ADC, we're actually going to work backwards: we'll start with a simple working example of how to use ADC, and then explain what's going on in this code. The reason for this is that ADC -- and most other peripherals on the LPC2129 -- requires a little bit more effort to use than GPIO, since it needs to be properly 'configured' and enabled before it can be used. If you're just getting started, though, it's more motivating to see something working before diving into all the little details of how to make it work.

Steps involved in initializing ADC unit:

```
AD0CR  = 0x00000000;
PINSEL1      = 0x10000000;      // Enable the 2nd functionality of P0.30 pin.
```

```
AD0CR  = 0x00250208;                    // Enable ADC
AD0DR  = 0;
   AD0CR              |= 1 << 24;
```

Testing it out

The easiest way to test the ADC code above is with a 'potentiometer' (often referred to simply as a 'pot'). A potentiometer is a simple device that changes its' 'resistance' as you adjust it. It may have 10.0K Ohms resistance at one end, for example, and 0 Ohms at the other. Since this 'resistance' will affect the amount of current available on the ADC line, we can quickly determine where the dial is currently positioned between the two extremes.

Conveniently, there is a potentiometer hooked up to AD0.3 on the Cranes LPC-P2148 development board (which we can see by looking at the schematic in the very top Left-hand corner [in Blue Color]). To test out the code above, simply run it and monitor the value of 'results' in the main method (either by setting a breakpoint, or by creating a 'watch'). If you are using Cross works, you should also be able to see the results of the A/D conversion by adjusting your code in 'main' as follows:

```
while (1)
{
        results = adcRead0_3 ();
     debug_printf ("%d\n", results);
}
```

As you turn the potentiometer with a small screw-driver, you should see the current position of the dial getting 'converted' from analog to digital and receive a value between 0 (at one end) and 1023 at the other end. And presto ... you have a fully working 10-bit analog to digital converter! Now that we've got that out of the way, we can get on with boring you with all the dull details of how to make it happen!

Step 1: *Configuring the ADC (ADCR)*

Looking at the example code above, you can see that before we can read any results from the ADC we need to configure it (using "adcInit0_3"). In this particular case, we're using AD0.3, which means we are using A/D Converter 0 (the LPC2148 has two ADCs named AD0 and AD1), and channel 3 (out of a possible 8 channels). We chose this particular device and channel because the Cranes LPC-P2148 development board already has a 10K potentiometer on AD0.3 and the pin is conveniently 'broken-out' (you'll find a pin labeled AD03 towards the bottom of the prototyping area on the board).

1. Select the pin function (PINSEL)

One of the interesting features of ARM microcontrollers is that each pin can perform up to 4 different 'functions' ... though only one at a time! If you're not sure how this works, or what the advantages of this are, we discuss this in more detail in our description of PINSEL (a set of registers that we can use to indicate exactly which function we would like each pin to perform). In the case of AD0.3 (located on pin 15 of the lpc2148), the pin can be configuring to perform any of the following functions (see page 71 of the LPC2148 User's Manual):

1. **P0.30 -** General Purpose Input/output 0.30
2. **AD0.3** - Analog/Digital Converter 0, Input 3
3. **EINT3** - External Interrupt 3
4. **CAP0.0-** Capture input for Timer 0, Channel 0

This 'pin selection' is accomplished with the following line of code

Force pin 0.30 to function as AD0.3

PCB_PINSEL1 = (PCB_PINSEL1 & ~PCB_PINSEL1_P030_MASK) | PCB_PINSEL1_P030_AD03;

2. Make sure ADC0 is powered-on (PCONP) It's always good practice to make sure that a peripheral is powered on (using the PCONP register) before trying to use it. This can be accomplished for ADC0 with the following line of code:

SCB_PCONP |= SCB_PCONP_PCAD0; // Enable power for ADC0

3. Configure ADC0 (ADCR) Configuring the A/D Converter is probably the most complicated part, since you need to know certain things about the way your microcontroller is currently set up (primarily, the value of PCLK, which determines the 'speed' at which your microcontroller is running). You also need to do a little bit of math to set everything up properly. It isn't complicated once you understand it, but it can be intimidating at first to know what all these values mean, and which ones you should be using in your specific situation. To configure the A/D Converter, we need to pass a specific 32-bit value to the appropriate **ADCR**, or **"Analog/Digital Control Register"** (refer LPC2148 User Manual for more details). This 'control register' (defined in lpc214x.h as **'AD0_CR'** and **'AD1_CR'**) manages the configuration of our A/D converter, and determines a variety of things, including:

1. **SEL** - Which channel should be used (0..7)
2. **CLKDIV** - A value to divide PCLK by to determine which speed the A/D Converter should operate at (up to a maximum of 4.5MHz)
3. **CLKS** - How precise the conversion results should be (between 3 and 10 bits)
4. **PDN** Whether the A/D Converter is currently active (1) or sleeping (0)

The 32-bit Analog/Digital Control Register has the following format:

Function	-	EDGE	START	-	**PDN**	-	**CLKS**	BURST	**CLKDIV**	**SEL**
ADCR Bit(s)	31..28	27	26..24	23.22	21	20	19..17	16	15..8	7..0

Unfortunately, that may not make very much sense to you, but without going into every little detail of every possible configuration option (see p.270-272 for the User's Manual for full details), we'll try to explain the register values that are being used in the example at the beginning of this tutorial: **SEL, CLKDIV, CLKS** and **PDN.**

SEL This set of 8 bits corresponds to the 8 different 'channels' available on either A/D converter. You can indicate which channel you wish to use by setting its appropriate bit to '1'. For example, since we are using AD0.3 in this case (channel 3), we would pass the following value to AD0_CR:

	-	EDGE	START	-	PDN	-	CLKS	BURST	CLKDIV	SEL
AD0_CR	****	*	***	**	*	*	***	*	********	00001000

CLKDIV

The A/D Converters on the LPC2149 are able to run at a maximum speed of 4.5MHz. The conversion speed is selectable by the user, but the only catch is that to arrive at a number equal to or less than 4.5MHz, we need to 'divide' our PCLK (the speed at which our microprocessor is running) by a fixed number, which we provide (in binary format) using the 8 **CLKDIV** bits.

The default PCLK for your microcontroller is 12MHz (since the Cranes LPC-P2149 has a 12.0MHz crystal installed on it), but this can be 'multiplied' up to 60MHz, so you need to know what 'speed' you have set to your mcu to before providing a value in CLKDIV.

We are going to assume that we are running at 12.0MHz (since we haven't covered how to adjust the mcu speed yet!). In order to stay below the ADC's maximum speed of 4.5MHz, we would need to divide our PCLK (12MHz) by 3, which will give us 4MHz (the closest value we can have with a 12MHz clock since we need to divide by a whole number). In order to avoid a 'divide by 0' error, though, the A/D control register will add one to whatever value you supply.

This means that if we want to divide the PCLK by 3, we actually need to provide '2', which will be adjusted up one to '3' by the control register. (This is one of the little details that can cause your software to malfunction if you don't read the user manual and pay close attention to how to configure your peripherals.) This means that if we were running at 12.0MHz, we could achieve a 4.0MHz A/D conversion speed by setting the following CLKDIV bits in AD0_CR ("00000010" being the binary equivalent of 2:

	-	EDGE	START	-	PDN	-	CLKS	BURST	CLKDIV	SEL
AD0_CR	****	*	***	**	*	*	***	*	00000010	********

And what if we were running at 48.0MHz, for example? We would be able to achieve a maximum conversion speed of 4.36MHz by dividing 48.0MHz by 11, so we would provide '10' to the CLKDIV (10 + 1 = 11), as follows ('00001010' being the binary equivalent of 10):

	-	EDGE	START	-	PDN	-	CLKS	BURST	CLKDIV	SEL
AD0_CR	****	*	***	**	*	*	***	*	00001010	********

CLKS

These three bits are used to indicate the range of values used when converting analog data. You can set the 'precision' of the results from 3-bits (values from 0-7) up to 10-bits (values from 0-1023), depending on your requirements. You simply need to provide one of the following values to indicate how many 'bits' to use for the conversion:

000 = 10-bits (0..1023)
001 = 9-bits (0..511)
010 = 8-bits (0..255)
011 = 7-bits (0..127)
100 = 6-bits (0..63)
101 = 5-bits (0..31)
110 = 4-bits (0..15)
111 = 3-bits (0..7)

For example, to get the maximum 10-bit 'range', we would provide the following value to the CLKS field:

	-	EDGE	START	-	PDN	-	CLKS	BURST	CLKDIV	SEL

AD0_CR	****	*	***	**	*	*	000	*	********	********

PDN PDN (short for 'Power-Down') indicates whether the ADC should be in 'Power-Down' mode (0), or actively converting data (1). Since we want to convert data right away, we could tell the ADC to go out of power-down mode (its default value) by setting this bit to 1 as follows:

	-	EDGE	START	-	PDN	-	CLKS	BURST	CLKDIV	SEL
AD0_CR	****	*	***	**	1	*	***	*	********	********

How does all this relate to our sample code? If we take another look at the configuration values we are passing to AD0_CR we can see that we are setting all four of the fields described above as follows:

```
// Initialize ADC converter

// (10-bit accuracy, ADC active, 4.0MHz clock, channel 3 selected)

AD0_CR = AD_CR_CLKS10 | AD_CR_PDN | ((3 - 1) << AD_CR_CLKDIVSHIFT) | AD_CR_SEL3;
```

This code is using a number of 'aliases' that are defined in lpc214x.h (take a look in the header file to see the corresponding values). We could just as easily have passed the hexadecimal equivalent of these configuration settings to AD0_CR:

```
AD0_CR = 0x00200208; // = 0000 0 000 00 1 0 000 0 00000010 00001000
```

The two lines of code would be identical when compiled. The difference is that the first line (the one used in our example) is a little bit easier to understand and to modify later if we need to do so, which is good. It's beyond the scope of this (already lengthy!) tutorial to start describing aliases and bit operators, etc., but a quick summary of the aliases we are using here might be helpful:

1. **AD_CR_CLKS10:** Sets the bit-accuracy of the converted values to the maximum 10-bit (values between 0 and 1023)
2. **AD_CR_PDN:** Exits 'powered-down' mode, activating the ADC
3. **((3 - 1) << AD_CR_CLKDIVSHIFT):** Set the CLK divider (15MHz/3) and shifts it to the left to it's appropriate position
4. **AD_CR_SEL3:** Selects AD0 channel 3

Step 2: Reading the Conversion Results

Each ADC channel has its own dedicated data register (ADDR0...7) That we can use to 'read' the results of the analog to digital conversion, as well to check whether the current conversion is complete or not. The 32-bit value has the following format:

Cranes Varsity

A/D Data Register (ADDR - 0xE0034004)

ADDR	Name	Description	Reset Value
31	DONE	This bit is set to 1 when an A/D conversion completes. It is cleared when this register is read and when the ADCR is written. If the ADCR is written while a conversion is still in progress, this bit is set and a new conversion is started.	0
30	OVERUN	This bit is 1 in burst mode if the results of one or more conversions was (were) lost and overwritten before the conversion that produced the result in the LS bits. In non-FIFO operation, this bit is cleared by reading this register.	0
29:27		These bits always read as zeroes. They could be used for expansion of the CHN field in future compatible A/D converters that can convert more channels.	0
26:24	CHN	These bits contain the channel from which the LS bits were converted.	X
23:16		These bits always read as zeroes. They allow accumulation of successive A/D values without AND-masking, for at least 256 values without overflow into the CHN field.	0
15:6	V/V_{3A}	When DONE is 1, this field contains a binary fraction representing the voltage on the Ain pin selected by the SEL field, divided by the voltage on the VddA pin. Zero in the field indicates that the voltage on the Ain pin was less than, equal to, or close to that on V_{SSA}, while 0x3FF indicates that the voltage on Ain was close to, equal to, or greater than that on V_{3A}. For testing, data written to this field is captured in a shift register that is clocked by the A/D converter clock. The MS bit of this register sources the DINSERI input of the A/D converter, which is used only when TEST1:0 are 10.	X
5:0		These bits always read as zeroes. They provide compatible expansion room for future, higher-resolution A/D converters.	0

Table 175: A/D Data Register (ADDR - 0xE0034004)

DONE (Bit 31)
This bit is set to 1 when an A/D conversion is complete. For accurate results, you need to wait until this value is 1 before reading the RESULT bits. (Please note that this value is cleared when you read this register.)

OVERRUN (Bit 30)
While not relevant to the examples used in this tutorial, this value with be 1 if the results of one or more conversions were lost when converting in BURST mode. See the User's Manual for further details. (As with DONE, this bit will be cleared when you read this register.)

RESULTS (Bits 15...6)
If DONE is 1 (meaning the conversion is complete), these 10 bits will contain a binary number representing the results of our analog to digital conversion. It works by measuring the voltage on the analog input pin divided by the voltage on the Vref pin. Zero means that the voltage on the analog input pin was less than, equal to or close to GND (Vssa), and 0x3FF or (0011 1111 1111) indicates that the voltage on the analog input pin was close to, equal to or greater than the voltage on the Vref pin. Anything value between these two extremes will be returned as a 10-bit number (between 0 and 1023).

Before we can read the results from the A/D Data Register, we first need to perform the following steps (to start the actual conversion process by re-configuring ADCR):

1. Stop any current conversions and deselect all channels
This can be accomplished by modifying the A/D Control Register (see ADCR above) as follows:
AD0_CR &= ~ (AD_CR_START_MASK | AD_CR_SELMASK);
What you are doing here is setting the START bits in ADCR to 000 (stopping any conversions) and disabling any channels that may have been previously selected. This essentially 'resets' our ADC to a blank, inactive state.

2. Select the channel you wish to use in your conversion

Since we have previously deselected all channels, we need to re-enable the specific channel that we wish to use in this conversion. In this particular case, we can enable channel 3 with the following line of code:

```
AD0_CR |= (AD_CR_START_NONE | AD_CR_SEL3);
```

3. Manually tell the ADC to start converting
The ADC can be configured to start in two ways: Manually (as we will do here), or when some sort of internal hardware event occurs such as an external interrupt (for example when a button connected to an EINT pin is pressed), or some other form of interrupt. (For details on starting the conversion when an interrupt occurs, please see chapter 17 of the LPC2129 User's Manual. Interrupts will also be covered in a later tutorial.)

In our case, the easiest solution is to simply tell the ADC to start converting manually, which is as simple as setting the appropriate START bits (26...24) in the ADC Control Register (see ADCR above). To start converting now, we need to send 001 to the three-bit START block in ADCR, which can be done with the following line of code (AD_CR_START_NOW is defined as '001' in LPC21xx.h):

```
AD0_CR |=   AD_CR_START_NOW;
```

4. Wait for the conversion to complete
Now that the analog to digital converter is properly configured and has been started (on channel 3 in this case), we can wait for the results on the appropriate AD Data Register ... ADDR3 in this particular case. As we mentioned above, we know when a conversion is complete because bit 31 (DONE) of ADDR0...7 will be set to 1. As such, we simply need to wait until we encounter a 1 value on this bit, which can be done with the following line of code:

```
while (! (AD0_DR3 & AD_DR_DONE));
```

This will cause the code to endlessly loop until the conversion is complete and then move on to the next line once the conversion is finished.

Read the 10-bit conversion results The last step (now that we know the conversion is complete) is simply to read the 10-bit value stored in the appropriate ADDR register, and do whatever we need to do with it. This last step can be accomplished by reading bits 6 through 15 (15..6) of ADDR3 (since we are using Channel 3 in this example), and then 'shifting' the results to the right 6 places to give a normal 10-bit value (AD_DR_RESULTSHIFT is defined as 6 in lpc214x.h since we need to shift the results 6 positions to the right):

```
unsigned int results = ((AD0_DR3 & AD_DR_RESULTMASK) >> AD_DR_RESULTSHIFT);
return results;
```

That's all that's involved in manually starting an A/D conversion and reading the results. If you wish to continually read the results of the ADC, you can simply place the appropriate code in a method/function and continually call it, as we are doing in the example presented at the very beginning of this tutorial.

To make sure that this all make sense to you, try going back to the full example we gave you at the beginning of this article. Reading over the code line by line, can you understand what we're doing and why? If not, simply look at the line(s) you don't understand, and try to find the part of this article that explains that particular step, and why it's necessary or why are we using the values that we assigned.

OPERATION

Hardware-Triggered Conversion

If the BURST bit in the ADCR is 0 and the START field contains 010-111, the A/D converter will start a conversion when a transition occurs on a selected pin or Timer Match signal . The choices include conversion on a specified edge of any of 4 Match signals, or conversion on a specified edge of either of 2 Capture/Match pins. The pin state from the selected pad or the selected
Match signal, XORed with ADCR bit 27, is used in the edge detection logic.

Clock Generation

It is highly desirable that the clock divider for the 4.5 MHz conversion clock be held in a Reset state when the A/D converter is idle, so that the sampling clock can begin immediately when 01 is written to the START field of the ADCR, or the selected edge occurs on the selected signal. This feature also saves power, particularly if the A/D converter is used infrequently.

Interrupts

An interrupt request is asserted to the Vectored Interrupt Controller (VIC) when the DONE bit is 1. Software can use the Interrupt Enable bit for the A/D Converter in the VIC to control whether this assertion results in an interrupt. DONE is negated when the ADDR is read.

End of chapter 8

Cranes Varsity

CHAPTER – 9

WATCH DOG TIMER

A watchdog timer (WDT; sometimes called a computer operating properly or COP timer, or simply a watchdog) is an electronic timer that is used to detect and recover from computer malfunctions. During normal operation, the computer regularly restarts the watchdog timer to prevent it from elapsing, or "timing out". If, due to a hardware fault or program error, the computer fails to restart the watchdog, the timer will elapse and generate a timeout signal. The timeout signal is used to initiate corrective action or actions. The corrective actions typically include placing the computer system in a safe state and restoring normal system operation.

Watchdog timers are commonly found in embedded systems and other computer-controlled equipment where humans cannot easily access the equipment or would be unable to react to faults in a timely manner. In such systems, the computer cannot depend on a human to reboot it if it hangs; it must be self-reliant. For example, remote embedded systems such as space probes are not physically accessible to human operators; these could become permanently disabled if they were unable to autonomously recover from faults. A watchdog timer is usually employed in cases like these. Watchdog timers may also be used when running untrusted code in a sandbox, to limit the CPU time available to the code and thus prevent some types of denial-of-service attacks.

FEATURES

- Internally resets chip if not periodically reloaded
- Debug mode
- Enabled by software but requires a hardware reset or a Watchdog reset/interrupt to be disabled
- Incorrect/Incomplete feed sequence causes reset/interrupt if enabled
- Flag to indicate Watchdog reset
- Programmable 32-bit timer with internal pre-scaler
- Selectable time period from (tpclk x 256 x 4) to (tpclk x 232 x 4) in multiples of tpclk x 4

Architecture and operation

Watchdog restart

The act of restarting a watchdog timer is commonly referred to as "kicking the dog" or other similar terms; this is typically done by writing to a watchdog control port. Alternatively, in microcontrollers that have an integrated watchdog timer, the watchdog is sometimes kicked by executing a special machine language instruction. An example of this is the CLRWDT (clear watchdog timer) instruction found in the instruction set of some PIC microcontrollers.

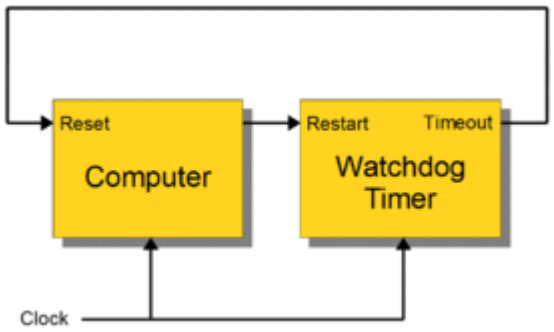
In computers that are running operating systems, watchdog resets are usually invoked through a device driver. For example, in the Linux operating system, a user space program will kick the watchdog by interacting with the watchdog device driver, typically by writing a zero character to /dev/watchdog.

Cranes Varsity

The device driver, which serves to abstract the watchdog hardware from user space programs, is also used to configure the time-out period and start and stop the timer.

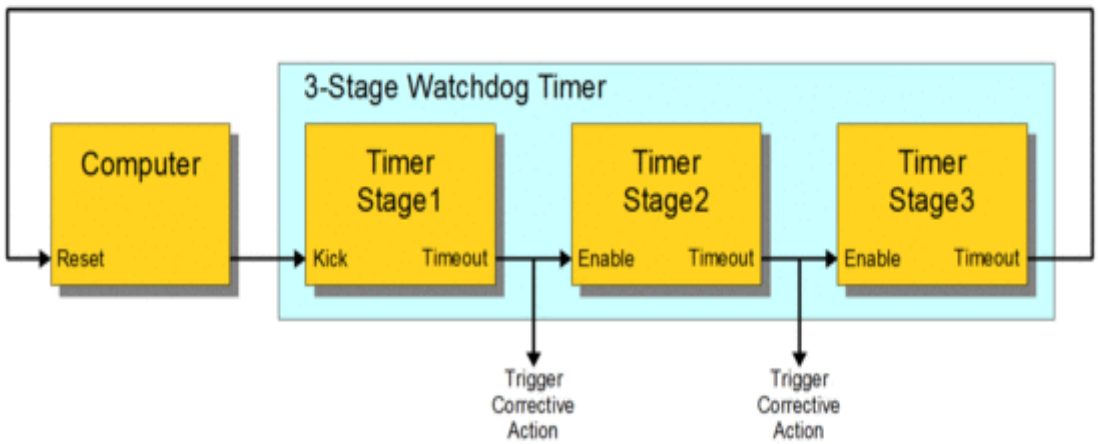
Single-stage watchdog

Watchdog timers come in many configurations, and many allow their configurations to be altered. Microcontrollers often include an integrated, on-chip watchdog. In other computers the watchdog may reside in a nearby chip that connects directly to the CPU, or it may be located on an external expansion card in the computer's chassis. The watchdog and CPU may share a common clock signal, as shown in the block diagram below, or they may have independent clock signals.



Multistage watchdog

Two or more timers are sometimes cascaded to form a *multistage watchdog timer*, where each timer is referred to as a *timer stage*, or simply a *stage*. For example, the block diagram below shows a three-stage watchdog. In a multistage watchdog, only the first stage is kicked by the processor. Upon first stage timeout, a corrective action is initiated and the next stage in the cascade is started. As each subsequent stage times out, it triggers a corrective action and starts the next stage. Upon final stage timeout, a corrective action is initiated, but no other stage is started because the end of the cascade has been reached. Typically, single-stage watchdog timers are used to simply restart the computer, whereas multistage watchdog timers will sequentially trigger a series of corrective actions, with the final stage triggering a computer restart.



Time intervals

Watchdog timers may have either fixed or programmable time intervals. Some watchdog timers allow the time interval to be programmed by selecting from among a few selectable, discrete values. In others, the interval can be programmed to arbitrary values. Typically, watchdog time intervals range from ten milliseconds to a minute or more. In a multistage watchdog, each timer may have its own, unique time interval.

Fault detection

A computer system is typically designed so that its watchdog timer will be kicked only if the computer deems the system functional. The computer determines whether the system is functional by conducting one or more fault detection tests and it will kick the watchdog only if all tests have passed. In computers that are running an operating system and multiple processes, a single, simple test may be insufficient to guarantee normal operation, as it could fail to detect a subtle fault condition and therefore allow the watchdog to be kicked even though a fault condition exists.

For example, in the case of the Linux operating system, a user-space watchdog daemon may simply kick the watchdog periodically without performing any tests. As long as the daemon runs normally, the system will be protected against serious system crashes such as a kernel panic. To detect less severe faults, the daemon can be configured to perform tests that cover resource availability (e.g., sufficient memory and file handles, reasonable CPU time), evidence of expected process activity (e.g., system daemons running, specific files being present or updated), overheating, and network activity, and system-specific test scripts or programs may also be run.[5]

Upon discovery of a failed test, the Linux watchdog daemon may attempt to perform a software-initiated restart, which can be preferable to a hardware reset as the file systems will be safely uncounted and fault information will be logged. However it is essential to have the insurance of the hardware timer as a software restart can fail under a number of fault conditions. In effect, this is a dual-stage watchdog with the software restart comprising the first stage and the hardware reset the second stage.

Programming with WDT

The watchdog consists of a divide by 4 fixed pre-scaler and a 32-bit counter. The clock is fed to the timer via a pre-scaler. The timer decrements when clocked. The minimum value from which the counter decrements is 0xFF. Setting a value lower than 0xFF causes 0xFF to be loaded in the counter. Hence the minimum watchdog interval is (TPCLK x 256 x 4) and the maximum watchdog interval is (TPCLK x 232 x 4) in multiples of (TPCLK x 4). The watchdog should be used in the following manner:

- Set the watchdog timer constant reload value in WDTC register.
- Setup mode in WDMOD register.
- Start the watchdog by writing 0xAA followed by 0x55 to the WDFEED register.
- Watchdog should be fed again before the watchdog counter underflows to prevent

reset/interrupt.

When the Watchdog counter underflows, the program counter will start from 0x0000 0000 as in the case of external reset. The Watchdog Time-Out Flag (WDTOF) can be examined to determine if the watchdog has caused the reset condition. The WDTOF flag must be cleared by software.

Register Description ;

Name	Description	Access	Reset Value*	Address
WDMOD	Watchdog mode register. This register contains the basic mode and status of the Watchdog Timer.	Read/Set	0	0xE0000000
WDTC	Watchdog timer constant register. This register determines the time-out value.	Read/Write	0xFF	0xE0000004
WDFEED	Watchdog feed sequence register. Writing AAh followed by 55h to this register reloads the Watchdog timer to its preset value.	Write Only	NA	0xE0000008
WDTV	Watchdog timer value register. This register reads out the current value of the Watchdog timer.	Read Only	0xFF	0xE000000C

Watchdog Mode Register

Watchdog Mode Register (WDMOD - 0xE0000000)

The WDMOD register controls the operation of the Watchdog as per the combination of WDEN and RESET bits.

WDEN	WDRESET	
0	X	Debug/Operate without the Watchdog running
1	0	Debug with the Watchdog interrupt but no WDRESET
1	1	Operate with the Watchdog interrupt and WDRESET

Once the WDEN and/or WDRESET bits are set they can not be cleared by software. Both flags are cleared by an external reset or a Watchdog timer underflow.

WDTOF The Watchdog time-out flag is set when the Watchdog times out. This flag is cleared by software.

WDINT The Watchdog interrupt flag is set when the Watchdog times out. This flag is cleared when any reset occurs.

WDMOD	Function	Description	Reset Value
0	WDEN	Watchdog interrupt enable bit (Set only)	0
1	WDRESET	Watchdog reset enable bit (Set Only)	0
2	WDTOF	Watchdog time-out flag	0 (Only after external reset)
3	WDINT	Watchdog interrupt flag (Read Only)	0
7:4	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Watchdog Timer Constant Register

The WDTC register determines the time-out value. Every time a feed sequence occurs the WDTC content is reloaded in to the watchdog timer. It's a 32-bit register with 8 LSB set to 1 on reset. Writing values below 0xFF will cause 0xFF to be loaded to the WDTC. Thus the minimum time-out interval is TPCLK × 256 × 4.

WDTC	Function	Description	Reset Value
31:0	Count	Watchdog time-out interval	0xFF

Watchdog Feed Register

Writing 0xAA followed by 0x55 to this register will reload the watchdog timer to the WDTC value. This operation will also start the watchdog if it is enabled via the WDMOD register. Setting the WDEN bit in the WDMOD register is not sufficient to enable the watchdog. A valid feed sequence must first be completed before the Watchdog is capable of generating an interrupt/reset. Until then, the watchdog will ignore feed errors. Once 0xAA is written to the WDFEED register the next operation in the Watchdog register space should be a **WRITE** (0x55) to the WDFFED register otherwise the watchdog is triggered.
The interrupt/reset will be generated during the second PCLK following an incorrect access to a watchdog timer register during a feed sequence.

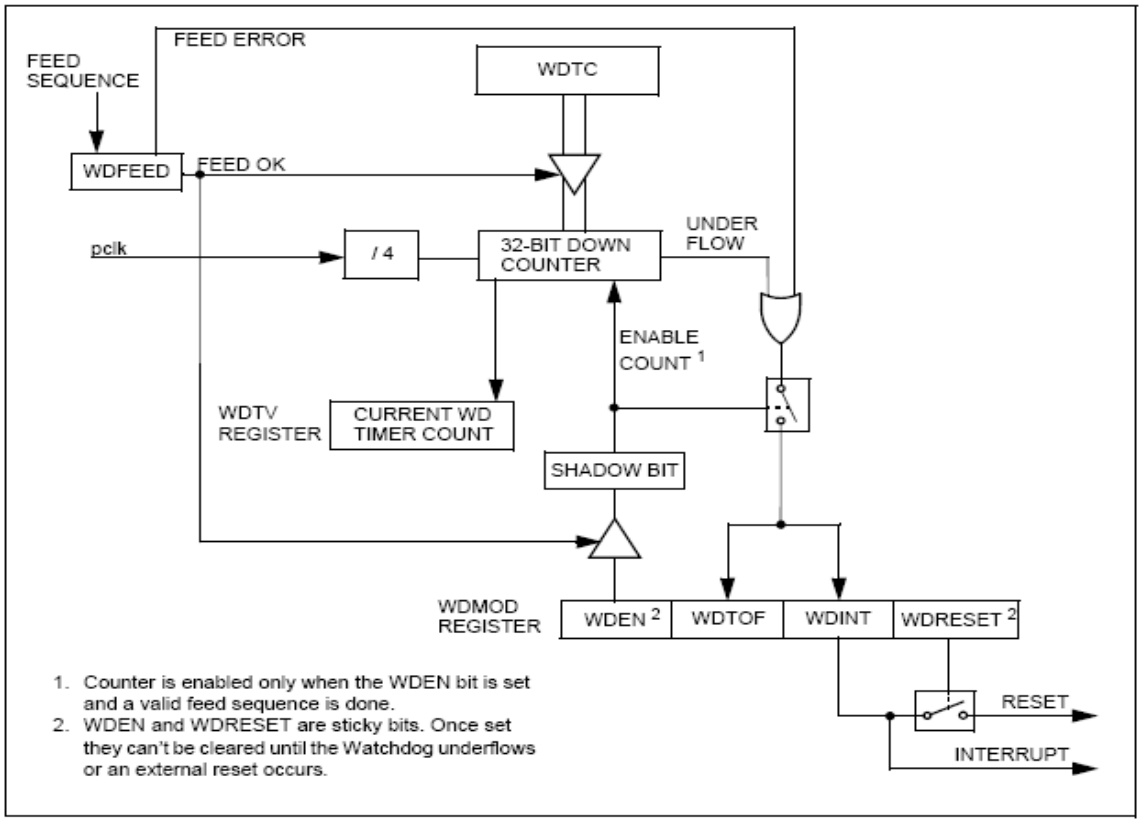
WDFEED	Function	Description	Reset Value
7:0	Feed	Feed value should be 0xAA followed by 0x55	undefined

Watchdog Timer Value Register

This above register is used to read the current value of the watchdog timer.

WDTV	Function	Description	Reset Value
31:0	Count	Current timer value	0xFF

Block Diagram



End of chapter 9

Cranes Varsity

CHAPTER-10

PULSE WIDTH MODULATION

Pulse Width Modulation (**PWM)** or **pulse-duration modulation** (**PDM**),

Is a technique used to encode a message into a pulsing signal. It is a type of modulation. Although this modulation technique can be used to encode information for transmission, its main use is to allow the control of the power supplied to electrical devices, especially to inertial loads such as motors. In addition, PWM is one of the two principal algorithms used in photovoltaic solar battery chargers the other being MPPT.

The average value of voltage (and current) fed to the load is controlled by turning the switch between supply and load on and off at a fast rate. The longer the switch is on compared to the off periods, the higher the total power supplied to the load.

The PWM switching frequency has to be much higher than what would affect the load (the device that uses the power), which is to say that the resultant waveform perceived by the load must be as smooth as possible. Typically switching has to be done several times a minute in an electric stove, 120 Hz in a lamp dimmer, from few kilohertz (kHz) to tens of kHz for a motor drive and well into the tens or hundreds of kHz in audio amplifiers and computer power supplies.

The term *duty cycle* describes the proportion of 'on' time to the regular interval or 'period' of time; a low duty cycle corresponds to low power, because the power is off for most of the time. Duty cycle is expressed in percent, 100% being fully on.

The main advantage of PWM is that power loss in the switching devices is very low. When a switch is off there is practically no current, and when it is on and power is being transferred to the load, there is almost no voltage drop across the switch. Power loss, being the product of voltage and current, is thus in both cases close to zero. PWM also works well with digital controls, which, because of their on/off nature, can easily set the needed duty cycle.

PWM has also been used in certain communication systems where its duty cycle has been used to convey information over a communications channel.

Delta

In the use of delta modulation for PWM control, the output signal is integrated, and the result is compared with limits, which correspond to a Reference signal offset by a constant. Every time the integral of the output signal reaches one of the limits, the PWM signal changes state.

Delta-sigma

In delta-sigma modulation as a PWM control method, the output signal is subtracted from a reference signal to form an error signal. This error is integrated, and when the integral of the error exceeds the limits, the output changes state.

Space vector modulation

Space vector modulation is a PWM control algorithm for multi-phase AC generation, in which the reference signal is sampled regularly; after each sample, non-zero active switching vectors adjacent to the reference vector and one or more of the zero switching vectors are selected for the appropriate fraction of the sampling period in order to synthesize the reference signal as the average of the used vectors.

Direct torque control (DTC)

Direct torque control is a method used to control AC motors. It is closely related with the delta modulation (see above). Motor torque and magnetic flux are estimated and these are controlled to stay within their hysteresis bands by turning on new combination of the device's semiconductor switches each time either of the signal tries to deviate out of the band.

Features of PWM in LPC2129:

• Seven match registers allow up to 6 single edge controlled or 3 double edge controlled PWM outputs, or a mix of both types.

The match registers also allow:

- Continuous operation with optional interrupt generation on match.
- Stop timer on match with optional interrupt generation.
- Reset timer on match with optional interrupt generation.

• An external output for each match register with the following capabilities:

- Set low on match.
- Set high on match.
- Toggle on match.
- Do nothing on match.

• Supports single edge controlled and/or double edge controlled PWM outputs. Single edge controlled PWM outputs all go high at the beginning of each cycle unless the output is a constant low. Double edge controlled PWM outputs can have either edge occur at any position within a cycle. This allows for both positive going and negative going pulses.

• Pulse period and width can be any number of timer counts. This allows complete flexibility in the trade-off between resolution and repetition rate. All PWM outputs will occur at the same repetition rate.

• Double edge controlled PWM outputs can be programmed to be either positive going or negative going pulses.

• Match register updates are synchronized with pulse outputs to prevent generation of erroneous pulses. Software must "release" new match values before they can become effective.

• May be used as a standard timer if the PWM mode is not enabled.

• A 32-bit Timer/Counter with a programmable 32-bit Prescaler.

• Four 32-bit capture channels take a snapshot of the timer value when an input signal transitions. A capture event may also optionally generate an interrupt

Description:

The PWM is based on the standard Timer block and inherits all of its features, although only the PWM function is pinned out on the LPC2129. The Timer is designed to count cycles of the peripheral clock (PCLK) and optionally generate interrupts or perform other actions when specified timer values occur, based on seven match registers. It also includes four capture inputs to save the timer value when an input signal transitions, and optionally generate an interrupt when those events occur. The PWM function is in addition to these features, and is based on match register events.

The ability to separately control rising and falling edge locations allows the PWM to be used for more applications. For instance, multi-phase motor control typically requires three non-overlapping PWM outputs with individual control of all three pulse widths and positions.

Cranes Varsity

Two match registers can be used to provide a single edge controlled PWM output. One match register (PWMMR0) controls the PWM cycle rate, by resetting the count upon match. The other match register controls the PWM edge position. Additional single edge controlled PWM outputs require only one match register each, since the repetition rate is the same for all PWM outputs. Multiple single edge controlled PWM outputs will all have a rising edge at the beginning of each PWM cycle, when an PWMMR0 match occurs.

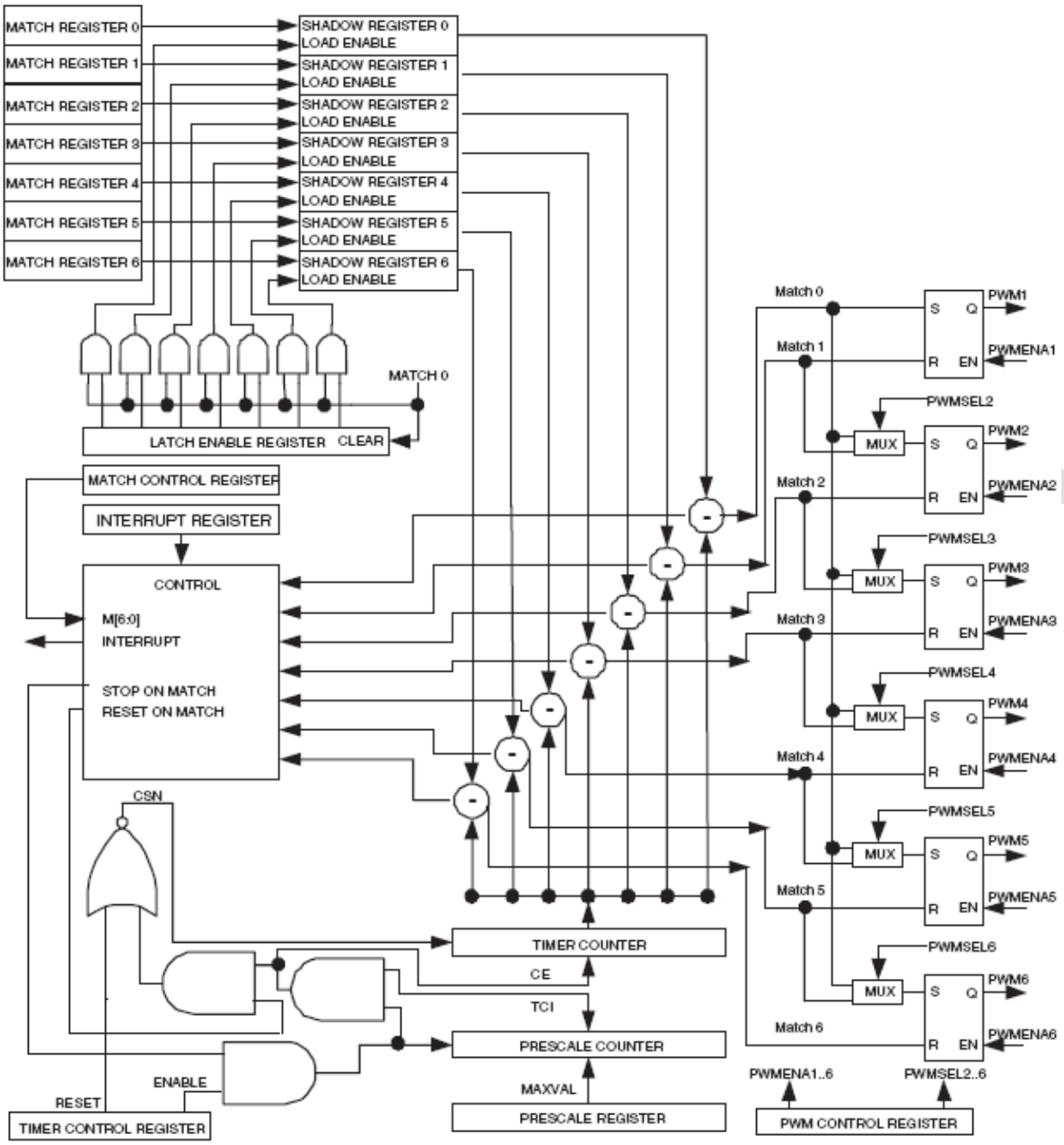
Three match registers can be used to provide a PWM output with both edges controlled. Again, the PWMMR0 match register controls the PWM cycle rate. The other match registers control the two PWM edge positions. Additional double edge controlled PWM outputs require only two match registers each, since the repetition rate is the same for all PWM outputs.

With double edge controlled PWM outputs, specific match registers control the rising and falling edge of the output. This allows both positive going PWM pulses (when the rising edge occurs prior to the falling edge), and negative going PWM pulses (when the falling edge occurs prior to the rising edge).

PWM Channel	Single Edge PWM (PWMSELn = 0)		Double Edge PWM (PWMSELn = 1)	
	Set by	Reset by	Set by	Reset by
1	Match 0	Match 1	Match 0 [1]	Match 1 [1]
2	Match 0	Match 2	Match 1	Match 2
3	Match 0	Match 3	Match 2 [2]	Match 3 [2]
4	Match 0	Match 4	Match 3	Match 4
5	Match 0	Match 5	Match 4 [2]	Match 5 [2]
6	Match 0	Match 6	Match 5	Match 6

Table : Set and reset inputs for PWM Flip-Flops

Block Diagram:



Rules for single edge controlled PWM outputs:

- All single edge controlled PWM outputs go high at the beginning of a PWM cycle unless their match value is equal to 0.

- Each PWM output will go low when its match value is reached. If no match occurs (i.e. the match value is greater than the PWM rate), the PWM output remains continuously high.

Rules for double edge controlled PWM outputs:

Five rules are used to determine the next value of a PWM output when a new cycle is about to begin:

- The match values for the **next** PWM cycle are used at the end of a PWM cycle (a time point which is coincident with the beginning of the next PWM cycle), except as noted in rule 3.
- A match value equal to 0 or the current PWM rate (the same as the Match channel 0 value) has the same effect, except as noted in rule 3. For example, a request for a falling edge at the beginning of the PWM cycle has the same effect as a request for a falling edge at the end of a PWM cycle.
- When match values are changing, if one of the "old" match values is equal to the PWM rate, it is used again once if the neither of the new match values are equal to 0 or the PWM rate, nor there was no old match value equal to 0.
- If both a set and a clear of a PWM output are requested at the same time, clear takes precedence. This can occur when the set and clear match values are the same as in, or when the set or clear value equals 0 and the other value equals the PWM rate.
- If a match value is out of range (i.e. greater than the PWM rate value), no match event occurs and that match channel has no effect on the output. This means that the PWM output will remain always in one state, allowing always low, always high, or "no change" outputs.

Pin description :

Pin name	Pin direction	Pin Description
PWM1	**Output**	**Output from PWM1 channel 1**
PWM2	**Output**	**Output from PWM2 channel 2**
PWM3	**Output**	**Output from PWM3 channel 3**
PWM4	**Output**	**Output from PWM4 channel 4**
PWM5	**Output**	**Output from PWM5 channel 5**
PWM6	**Output**	**Output from PWM6 channel 6**

Register description:

Cranes Varsity

Name	Description	Access	Reset Value*	Address
PWMIR	PWM Interrupt Register. The IR can be written to clear interrupts. The IR can be read to identify which of the possible interrupt sources are pending.	R/W	0	0xE0014000
PWMTCR	PWM Timer Control Register. The TCR is used to control the Timer Counter functions. The Timer Counter can be disabled or reset through the TCR.	R/W	0	0xE0014004
PWMTC	PWM Timer Counter. The 32-bit TC is incremented every PR+1 cycles of pclk. The TC is controlled through the TCR.	RW	0	0xE0014008
PWMPR	PWM Prescale Register. The TC is incremented every PR+1 cycles of pclk.	R/W	0	0xE001400C
PWMPC	PWM Prescale Counter. The 32-bit PC is a counter which is incremented to the value stored in PR. When the value in PR is reached, the TC is incremented.	R/W	0	0xE0014010
PWMMCR	PWM Match Control Register. The MCR is used to control if an interrupt is generated and if the TC is reset when a Match occurs.	R/W	0	0xE0014014
PWMMR0	PWM Match Register 0. MR0 can be enabled through MCR to reset the TC, stop both the TC and PC, and/or generate an interrupt when it matches the TC. In addition, a match between MR0 and the TC sets all PWM outputs that are in single-edge mode, and sets PWM1 if it is in double-edge mode.	R/W	0	0xE0014018
PWMMR1	PWM Match Register 1. MR1 can be enabled through MCR to reset the TC, stop both the TC and PC, and/or generate an interrupt when it matches the TC. In addition, a match between MR1 and the TC clears PWM1 in either single-edge mode or double-edge mode, and sets PWM2 if it is in double-edge mode.	R/W	0	0xE001401C
PWMMR2	PWM Match Register 2. MR2 can be enabled through MCR to reset the TC, stop both the TC and PC, and/or generate an interrupt when it matches the TC. In addition, a match between MR2 and the TC clears PWM2 in either single-edge mode or double-edge mode, and sets PWM3 if it is in double-edge mode.	R/W	0	0xE0014020
PWMMR3	PWM Match Register 3. MR3 can be enabled through MCR to reset the TC, stop both the TC and PC, and/or generate an interrupt when it matches the TC. In addition, a match between MR3 and the TC clears PWM3 in either single-edge mode or double-edge mode, and sets PWM4 if it is in double-edge mode.	R/W	0	0xE0014024
PWMMR4	PWM Match Register 4. MR4 can be enabled through MCR to reset the TC, stop both the TC and PC, and/or generate an interrupt when it matches the TC. In addition, a match between MR4 and the TC clears PWM4 in either single-edge mode or double-edge mode, and sets PWM5 if it is in double-edge mode.	R/W	0	0xE0014040

Name	Description	Access	Reset Value*	Address
PWMMR5	PWM Match Register 5. MR5 can be enabled through MCR to reset the TC, stop both the TC and PC, and/or generate an interrupt when it matches the TC. In addition, a match between MR5 and the TC clears PWM5 in either single-edge mode or double-edge mode, and sets PWM6 if it is in double-edge mode.	R/W	0	0xE0014044
PWMMR6	PWM Match Register 6. MR6 can be enabled through MCR to reset the TC, stop both the TC and PC, and/or generate an interrupt when it matches the TC. In addition, a match between MR6 and the TC clears PWM6 in either single-edge mode or double-edge mode.	R/W	0	0xE0014048
PWMPCR	PWM Control Register. Enables PWM outputs and selects PWM channel types as either single edge or double edge controlled.	R/W	0	0xE001404C
PWMLER	PWM Latch Enable Register. Enables use of new PWM match values.	R/W	0	0xE0014050

PWM Interrupt Register:

The PWM Interrupt Register consists of eleven bits seven for the match interrupts and four reserved for the future use. If an interrupt is generated then the corresponding bit in the PWMIR will be high. Otherwise, the bit will be low. Writing a logic one to the corresponding IR bit will reset the interrupt. Writing a zero has no effect.

PWMIR	Function	Description	Reset Value
0	PWMMR0 Interrupt	Interrupt flag for PWM match channel 0.	0
1	PWMMR1 Interrupt	Interrupt flag for PWM match channel 1.	0
2	PWMMR2 Interrupt	Interrupt flag for PWM match channel 2.	0
3	MR3 Interrupt	Interrupt flag for PWM match channel 3.	0
4	Reserved.	Application must not write 1 to this bit.	0
5	Reserved.	Application must not write 1 to this bit.	0
6	Reserved.	Application must not write 1 to this bit.	0
7	Reserved.	Application must not write 1 to this bit.	0
8	PWMMR4 Interrupt	Interrupt flag for PWM match channel 4.	0
9	PWMMR5 Interrupt	Interrupt flag for PWM match channel 5.	0
10	PWMMR6 Interrupt	Interrupt flag for PWM match channel 6.	0

PWM Timer Control Register:

The PWM Timer Control Register (PWMTCR) is used to control the operation of the PWM Timer Counter.

PWMTCR	Function	Description	Reset Value
0	Counter Enable	When one, the PWM Timer Counter and PWM Prescale Counter are enabled for counting. When zero, the counters are disabled.	0
1	Counter Reset	When one, the PWM Timer Counter and the PWM Prescale Counter are synchronously reset on the next positive edge of pclk. The counters remain reset until TCR[1] is returned to zero.	0
2	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
3	PWM Enable	When one, PWM mode is enabled. PWM mode causes shadow registers to operate in connection with the Match registers. A program write to a Match register will not have an effect on the Match result until the corresponding bit in PWMLER has been set, followed by the occurrence of a PWM Match 0 event. Note that the PWM Match register that determines the PWM rate (PWM Match 0) must be set up prior to the PWM being enabled. Otherwise a Match event will not occur to cause shadow register contents to become effective.	0

PWM Timer Counter:

The 32-bit PWM Timer Counter is incremented when the Prescale Counter reaches its terminal count. Unless it is reset before reaching its upper limit, the PWMTC will count up through the value 0xFFFF FFFF and then wrap back to the value 0x0000 0000. This event does not cause an interrupt, but a Match register can be used to detect an overflow if needed.

PWM Prescale Register:

The 32-bit PWM Prescale Register specifies the maximum value for the PWM Prescale Counter.

PWM Prescale Counter Register:

The 32-bit PWM Prescale Counter controls division of PCLK by some constant value before it is applied to the PWM Timer Counter. This allows control of the relationship of the resolution of the timer versus the maximum time before the timer overflows. The PWM Prescale Counter is incremented on every PCLK. When it reaches the value stored in the PWM Prescale Register, the PWM Timer Counter is incremented and the PWM Prescale Counter is reset on the next PCLK. This causes the PWM TC to increment on every PCLK when PWMPR = 0, every 2 PCLKs when PWMPR = 1, etc.

PWM Match Registers:

The 32-bit PWM Match register values are continuously compared to the PWM Timer Counter value. When the two values are equal, actions can be triggered automatically. The action possibilities are to generate an interrupt, reset the PWM Timer Counter, or stop the timer. Actions are controlled by the settings in the PWMMCR register.

PWM Match Control Register:

The PWM Match Control Register is used to control what operations are performed when one of the PWM Match Registers matches the PWM Timer Counter.

Cranes Varsity

PWMMCR	Function	Description	Reset Value
0	Interrupt on PWMMR0	When one, an interrupt is generated when PWMMR0 matches the value in the PWMTC. When zero this interrupt is disabled.	0
1	Reset on PWMMR0	When one, the PWMTC will be reset if PWMMR0 matches it. When zero this feature is disabled.	0
2	Stop on PWMMR0	When one, the PWMTC and PWMPC will be stopped and PWMTCR[0] will be set to 0 if PWMMR0 matches the PWMTC. When zero this feature is disabled.	0
3	Interrupt on PWMMR1	When one, an interrupt is generated when PWMMR1 matches the value in the PWMTC. When zero this interrupt is disabled.	0
4	Reset on PWMMR1	When one, the PWMTC will be reset if PWMMR1 matches it. When zero this feature is disabled.	0
5	Stop on PWMMR1	When one, the PWMTC and PWMPC will be stopped and PWMTCR[0] will be set to 0 if PWMMR1 matches the PWMTC. When zero this feature is disabled	0
6	Interrupt on PWMMR2	When one, an interrupt is generated when PWMMR2 matches the value in the PWMTC. When zero this interrupt is disabled.	0
7	Reset on PWMMR2	When one, the PWMTC will be reset if PWMMR2 matches it. When zero this feature is disabled.	0
8	Stop on PWMMR2	When one, the PWMTC and PWMPC will be stopped and PWMTCR[0] will be set to 0 if PWMMR2 matches the PWMTC. When zero this feature is disabled	0
9	Interrupt on PWMMR3	When one, an interrupt is generated when PWMMR3 matches the value in the PWMTC. When zero this interrupt is disabled	0
10	Reset on PWMMR3	When one, the PWMTC will be reset if PWMMR3 matches it. When zero this feature is disabled.	0
11	Stop on PWMMR3	When one, the PWMTC and PWMPC will be stopped and PWMTCR[0] will be set to 0 if PWMMR3 matches the PWMTC. When zero this feature is disabled	0
12	Interrupt on PWMMR4	When one, an interrupt is generated when PWMMR4 matches the value in the PWMTC. When zero this interrupt is disabled.	0
13	Reset on PWMMR4	When one, the PWMTC will be reset if PWMMR4 matches it. When zero this feature is disabled.	0
14	Stop on PWMMR4	When one, the PWMTC and PWMPC will be stopped and PWMTCR[0] will be set to 0 if PWMMR4 matches the PWMTC. When zero this feature is disabled	0
15	Interrupt on PWMMR5	When one, an interrupt is generated when PWMMR5 matches the value in the PWMTC. When zero this interrupt is disabled.	0
16	Reset on PWMMR5	When one, the PWMTC will be reset if PWMMR5 matches it. When zero this feature is disabled.	0

PWMMCR	Function	Description	Reset Value
17	Stop on PWMMR5	When one, the PWMTC and PWMPC will be stopped and PWMTCR[0] will be set to 0 if PWMMR5 matches the PWMTC. When zero this feature is disabled	0
18	Interrupt on PWMMR6	When one, an interrupt is generated when PWMMR6 matches the value in the PWMTC. When zero this interrupt is disabled.	0
19	Reset on PWMMR6	When one, the PWMTC will be reset if PWMMR6 matches it. When zero this feature is disabled.	0
20	Stop on PWMMR6	When one, the PWMTC and PWMPC will be stopped and PWMTCR[0] will be set to 0 if PWMMR6 matches the PWMTC. When zero this feature is disabled	0

PWM Control Register:

The PWM Control Register is used to enable and select the type of each PWM channel.

PWMPCR	Function	Description	Reset Value
1:0	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
2	PWMSEL2	When zero, selects single edge controlled mode for PWM2. When one, selects double edge controlled mode for the PWM2 output.	0
3	PWMSEL3	When zero, selects single edge controlled mode for PWM3. When one, selects double edge controlled mode for the PWM3 output.	0
4	PWMSEL4	When zero, selects single edge controlled mode for PWM4. When one, selects double edge controlled mode for the PWM4 output.	0
5	PWMSEL5	When zero, selects single edge controlled mode for PWM5. When one, selects double edge controlled mode for the PWM5 output.	0
6	PWMSEL6	When zero, selects single edge controlled mode for PWM6. When one, selects double edge controlled mode for the PWM6 output.	0
8:7	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
9	PWMENA1	When one, enables the PWM1 output. When zero, disables the PWM1 output.	0
10	PWMENA2	When one, enables the PWM2 output. When zero, disables the PWM2 output.	0
11	PWMENA3	When one, enables the PWM3 output. When zero, disables the PWM3 output.	0
12	PWMENA4	When one, enables the PWM4 output. When zero, disables the PWM4 output.	0
13	PWMENA5	When one, enables the PWM5 output. When zero, disables the PWM5 output.	0
14	PWMENA6	When one, enables the PWM6 output. When zero, disables the PWM6 output.	0
15	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

PWM Latch Enable Register:

The PWM Latch Enable Register is used to control the update of the PWM Match registers when they are used for PWM generation. When software writes to the location of a PWM Match register while the Timer is in PWM mode, the value is held in a shadow register. When a PWM Match 0 event occurs (normally also resetting the timer in PWM mode), the contents of shadow registers will be transferred to the actual Match registers if the corresponding bit in the Latch Enable Register has been set. At that point, the new values will take effect and determine the course of the next PWM cycle. Once the transfer of new values has taken place, all bits of the LER are automatically cleared. Until the corresponding bit in the PWMLER is set and a PWM Match 0 event occurs, any value written to the PWM Match registers has no effect on PWM operation.

For example, if PWM2 is configured for double edge operation and is currently running, a typical sequence of events for changing the timing would be:

- Write a new value to the PWM Match1 register.
- Write a new value to the PWM Match2 register.
- Write to the PWMLER, setting bits 1 and 2 at the same time.
- The altered values will become effective at the next reset of the timer (when a PWM Match 0 event occurs).

The order of writing the two PWM Match registers is not important, since neither value will be used until after the write to PWMLER. This insures that both values go into effect at the same time, if that is required. A single value may be altered in the same way if needed.

PWMLER	Function	Description	Reset Value
0	Enable PWM Match 0 Latch	Writing a one to this bit allows the last value written to the PWM Match 0 register to be become effective when the timer is next reset by a PWM Match event. See the description of the PWM Match Control Register (PWMMCR).	0
1	Enable PWM Match 1 Latch	Writing a one to this bit allows the last value written to the PWM Match 1 register to be become effective when the timer is next reset by a PWM Match event. See the description of the PWM Match Control Register (PWMMCR).	0
2	Enable PWM Match 2 Latch	Writing a one to this bit allows the last value written to the PWM Match 2 register to be become effective when the timer is next reset by a PWM Match event. See the description of the PWM Match Control Register (PWMMCR).	0
3	Enable PWM Match 3 Latch	Writing a one to this bit allows the last value written to the PWM Match 3 register to be become effective when the timer is next reset by a PWM Match event. See the description of the PWM Match Control Register (PWMMCR).	0
4	Enable PWM Match 4 Latch	Writing a one to this bit allows the last value written to the PWM Match 4 register to be become effective when the timer is next reset by a PWM Match event. See the description of the PWM Match Control Register (PWMMCR).	0
5	Enable PWM Match 5 Latch	Writing a one to this bit allows the last value written to the PWM Match 5 register to be become effective when the timer is next reset by a PWM Match event. See the description of the PWM Match Control Register (PWMMCR).	0
6	Enable PWM Match 6 Latch	Writing a one to this bit allows the last value written to the PWM Match 6 register to be become effective when the timer is next reset by a PWM Match event. See the description of the PWM Match Control Register (PWMMCR).	0
7	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Cranes

CHAPTER – 11

REAL TIME CLOCK

A real-time clock (RTC) is a computer clock (most often in the form of an integrated circuit) that keeps track of the current time. Although the term often refers to the devices in personal computers, servers and embedded systems, RTCs are present in almost any electronic device which needs to keep accurate time.

The term is used to avoid confusion with ordinary hardware clocks which are only signals that govern digital electronics, and do not count time in human units. RTC should not be confused with real-time computing, which shares its three-letter acronym but does not directly relate to time of day.

Purpose

Although keeping time can be done without an RTC,using one has benefits:

- Low power consumption(important when running from alternate power)
- Frees the main system for time-critical tasks
- Sometimes more accurate than other methods

Power source

RTCs often have an alternate source of power, so they can continue to keep time while the primary source of power is off or unavailable. This alternate source of power is normally a lithium battery in older systems, but some newer systems use a super capacitor, because they are rechargeable and can be soldered. The alternate power source can also supply power to battery backed RAM.

Timing

Most RTCs use a crystal oscillator, but some use the power line frequency. In many cases, the oscillator's frequency is 32.768 kHz. This is the same frequency used in quartz clocks and watches, and for the same reasons, namely that the frequency is exactly 215 cycles per second, which is a convenient rate to use with simple binary counter circuits.

Examples

This chip, labeled ODIN, is a generic equivalent to a particular Dallas RTC.

Many integrated circuit manufacturers make RTCs, including Epson, Intersil, IDT, Maxim, NXP Semiconductors, Texas Instruments and STMicroelectronics. The RTC was introduced to PC compatibles by the IBM PC/AT in 1984, which used a Motorola MC146818 RTC. Later, Dallas Semiconductor made compatible RTCs, which were often used in older personal computers, and are easily found on motherboards because of their distinctive black battery cap and silkscreened logo. In newer systems, the RTC is integrated into the south bridge chip.

Some microcontrollers have a real-time clock built in, generally only the ones with many other features and peripherals.

Features of RTC in LPC2129 :

- Measures the passage of time to maintain a calendar and clock.

Cranes Varsity

- Ultra Low Power design to support battery powered systems.
- Provides Seconds, Minutes, Hours, Day of Month, Month, Year, Day of Week, and Day of Year.

Description:

The Real Time Clock (RTC) is a set of counters for measuring time when system power is on, and optionally when it is off. It uses little power in Power-down mode. On the LPC2129, the RTC can be clocked by a separate 32.768 KHz oscillator, or by a programmable prescale divider based on the APB clock. Also, the RTC is powered by its own power supply pin, VBAT, which can be connected to a battery or to the same 3.3 V supply used by the rest of the device.

Architecture:

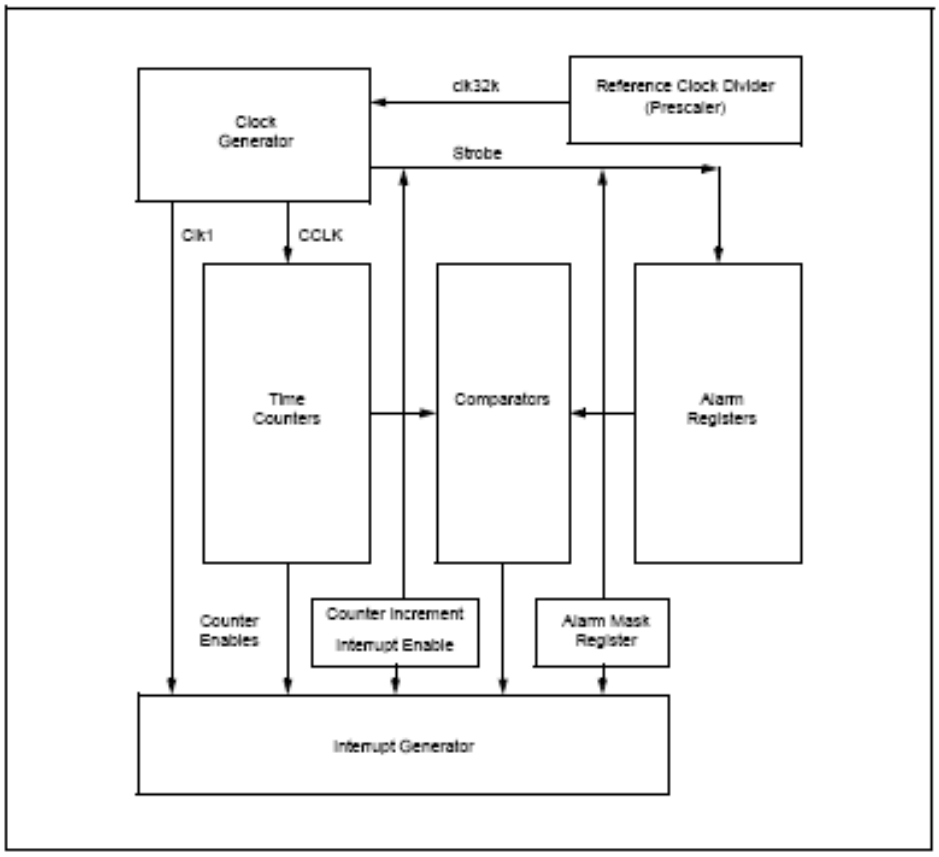


Figure 44: RTC block diagram

Register description :

Name	Size	Description	Access	Reset Value	Address
ILR	2	Interrupt Location Register	R/W	*	0xE0024000
CTC	15	Clock Tick Counter.	RO	*	0xE0024004
CCR	4	Clock Control Register	R/W	*	0xE0024008
CIIR	8	Counter Increment Interrupt Register	R/W	*	0xE002400C
AMR	8	Alarm Mask Register	R/W	*	0xE0024010
CTIME0	(32)	Consolidated Time Register 0	RO	*	0xE0024014
CTIME1	(32)	Consolidated Time Register 1	RO	*	0xE0024018
CTIME2	(32)	Consolidated Time Register 2	RO	*	0xE002401C
SEC	6	Seconds Register	R/W	*	0xE0024020
MIN	6	Minutes Register	R/W	*	0xE0024024
HOUR	5	Hours Register	R/W	*	0xE0024028
DOM	5	Day of Month Register	R/W	*	0xE002402C
DOW	3	Day of Week Register	R/W	*	0xE0024030
DOY	9	Day of Year Register	R/W	*	0xE0024034
MONTH	4	Months Register	R/W	*	0xE0024038
YEAR	12	Years Register	R/W	*	0xE002403C
ALSEC	6	Alarm value for Seconds	R/W	*	0xE0024060
ALMIN	6	Alarm value for Minutes	R/W	*	0xE0024064
ALHOUR	5	Alarm value for Hours	R/W	*	0xE0024068
ALDOM	5	Alarm value for Day of Month	R/W	*	0xE002406C
ALDOW	3	Alarm value for Day of Week	R/W	*	0xE0024070
ALDOY	9	Alarm value for Day of Year	R/W	*	0xE0024074
ALMON	4	Alarm value for Months	R/W	*	0xE0024078
ALYEAR	12	Alarm value for Year	R/W	*	0xE002407C
PREINT	13	Prescale value, integer portion	R/W	0	0xE0024080
PREFRAC	15	Prescale value, fractional portion	R/W	0	0xE0024084

RTC INTERRUPTS

Interrupt generation is controlled through the Interrupt Location Register (ILR), Counter Increment Interrupt Register (CIIR), the alarm registers, and the Alarm Mask Register (AMR). Interrupts are generated only by the transition into the interrupt state. The ILR separately enables CIIR and AMR interrupts. Each bit in CIIR corresponds to one of the time counters. If CIIR is enabled for a particular counter, then every time the counter is incremented an interrupt is generated. The alarm registers allow the user to specify a date and time for an interrupt to be generated. The AMR provides a mechanism to

mask alarm compares. If all non masked alarm registers match the value in their corresponding time counter, then an interrupt is generated.

MISCELLANEOUS REGISTER GROUP

Address	Name	Size	Description	Access
0xE0024000	ILR	2	Interrupt Location. Reading this location indicates the source of an interrupt. Writing a one to the appropriate bit at this location clears the associated interrupt.	RW
0xE0024004	CTC	15	Clock Tick Counter. Value from the clock divider.	RO
0xE0024008	CCR	4	Clock Control Register. Controls the function of the clock divider.	RW
0xE002400C	CIIR	8	Counter Increment Interrupt. Selects which counters will generate an interrupt when they are incremented.	RW
0xE0024010	AMR	8	Alarm Mask Register. Controls which of the alarm registers are masked.	RW
0xE0024014	CTIME0	32	Consolidated Time Register 0	RO
0xE0024018	CTIME1	32	Consolidated Time Register 1	RO
0xE002401C	CTIME2	32	Consolidated Time Register 2	RO

Interrupt Location Register:

The Interrupt Location Register is a 2-bit register that specifies which blocks are generating an interrupt. Writing a one to the appropriate bit clears the corresponding interrupt. Writing a zero has no effect. This allows the programmer to read this register and write back the same value to clear only the interrupt that is detected by the read.

ILR	Function	Description
0	RTCCIF	When one, the Counter Increment Interrupt block generated an interrupt. Writing a one to this bit location clears the counter increment interrupt.
1	RTCALF	When one, the alarm registers generated an interrupt. Writing a one to this bit location clears the alarm interrupt.

Clock Tick Counter Register:

The Clock Tick Counter is read only. It can be reset to zero through the Clock Control Register (CCR). The CTC consists of the bits of the clock divider counter.

CTC	Function	Description
0	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.
15:1	Clock Tick Counter	Prior to the Seconds counter, the CTC counts 32,768 clocks per second. Due to the RTC Prescaler, these 32,768 time increments may not all be of the same duration. Refer to the Reference Clock Divider (Prescaler) description for details.

Clock Control Register :

The clock register is a 5-bit register that controls the operation of the clock divide circuit.

CCR	Function	Description
0	CLKEN	Clock Enable. When this bit is a one the time counters are enabled. When it is a zero, they are disabled so that they may be initialized.
1	CTCRST	CTC Reset. When one, the elements in the Clock Tick Counter are reset. The elements remain reset until CCR[1] is changed to zero.
3:2	CTTEST	Test Enable. These bits should always be zero during normal operation.

Counter Increment Interrupt Register :

The Counter Increment Interrupt Register (CIIR) gives the ability to generate an interrupt every time a counter is incremented. This interrupt remains valid until cleared by writing a one to bit zero of the Interrupt Location Register (ILR[0]).

CIIR	Function	Description
0	IMSEC	When one, an increment of the Second value generates an interrupt.
1	IMMIN	When one, an increment of the Minute value generates an interrupt.
2	IMHOUR	When one, an increment of the Hour value generates an interrupt.
3	IMDOM	When one, an increment of the Day of Month value generates an interrupt.
4	IMDOW	When one, an increment of the Day of Week value generates an interrupt.
5	IMDOY	When one, an increment of the Day of Year value generates an interrupt.
6	IMMON	When one, an increment of the Month value generates an interrupt.
7	IMYEAR	When one, an increment of the Year value generates an interrupt.

Alarm Mask Register :

The Alarm Mask Register (AMR) allows the user to mask any of the alarm registers. For the alarm function, every non-masked alarm register must match the corresponding time counter for an interrupt to be generated. The interrupt is generated only when the counter comparison first changes from no match to match. The interrupt is removed when a one is written to the appropriate bit of the Interrupt Location Register (ILR). If all mask bits are set, then the alarm is disabled.

AMR	Function	Description
0	AMRSEC	When one, the Second value is not compared for the alarm.
1	AMRMIN	When one, the Minutes value is not compared for the alarm.
2	AMRHOUR	When one, the Hour value is not compared for the alarm.
3	AMRDOM	When one, the Day of Month value is not compared for the alarm.
4	AMRDOW	When one, the Day of Week value is not compared for the alarm.
5	AMRDOY	When one, the Day of Year value is not compared for the alarm.
6	AMRMON	When one, the Month value is not compared for the alarm.
7	AMRYEAR	When one, the Year value is not compared for the alarm.

Consolidated Time register 0 :

The Consolidated Time Register 0 contains the low order time values: Seconds, Minutes, Hours, and Day of Week.

CTIME0	Function	Description
31:27	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.
26:24	Day of Week	Day of week value in the range of 0 to 6.
23:21	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.
20:16	Hours	Hours value in the range of 0 to 23.
15:14	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.
13:8	Minutes	Minutes value in the range of 0 to 59.
7:6	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.
5:0	Seconds	Seconds value in the range of 0 to 59.

Consolidated Time register 1:

The Consolidate Time register 1 contains the Day of Month, Month, and Year values.

CTIME1	Function	Description
31:28	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.
27:16	Year	Year value in the range of 0 to 4095.
15:12	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.
11:8	Month	Month value in the range of 1 to 12.
7:5	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.
4:0	Day of Month	Day of month value in the range of 1 to 28, 29, 30, or 31 (depending on the month and whether it is a leap year).

Consolidated Time register 2 :

The Consolidate Time register 2 contains just the Day of Year value.

CTIME2	Function	Description
11:0	Day of Year	Day of year value in the range of 1 to 365 (366 for leap years).
31:12	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.

REFERENCE CLOCK DIVIDER (PRESCALER)

The reference clock divider (hereafter referred to as the Prescaler) allows generation of a 32.768 kHz reference clock from any peripheral clock frequency greater than or equal to 65.536 kHz (2 x 32.768 kHz). This permits the RTC to always run at the proper rate regardless of the peripheral clock rate. Basically, the Prescaler divides the peripheral clock (pclk) by a value which contains both an

integer portion and a fractional portion. The result is not a continuous output at a constant frequency, some clock periods will be one pclk longer than others. However, the overall result can always be 32,768 counts per second.The reference clock divider consist of a 13-bit integer counter and a 15-bit fractional counter.

The reasons for these counter sizes are as follows:

1. For frequencies that are expected to be supported by the LPC2119/2129/2194/2292/2294, a 13-bit integer counter is required. This can be calculated as 160 MHz divided by 32,768 minus 1 = 4881 with a remainder of 26,624. Thirteen bits are needed to hold the value 4881, but actually supports frequencies up to 268.4 MHz (32,768 x 8192).
2. The remainder value could be as large as 32,767, which requires 15 bits.

Address	Name	Size	Description	Access
0xE0024080	PREINT	13	Prescale Value, integer portion	R/W
0xE0024084	PREFRAC	15	Prescale Value, fractional portion	R/W

Prescaler Integer Register (PREINT - 0xE0024080)

This is the integer portion of the prescale value, calculated as:
PREINT = int (pclk / 32768) - 1. The value of PREINT must be greater than or equal to 1.

PREINT	Function	Description	Reset Value
15:13	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
12:0	Prescaler Integer	Contains the integer portion of the RTC prescaler value.	0

Prescaler Fraction Register (PREFRAC - 0xE0024084)

This is the fractional portion of the prescale value, and may be calculated as:
PREFRAC = pclk - ((PREINT +1) x 32768).

PREFRAC	Function	Description	Reset Value
15	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
14:0	Prescaler Fraction	Contains the fractional portion of the RTC prescaler value.	0

Example of Prescaler Usage

In a simplistic case, the pclk frequency is 65.537 kHz. So:
PREINT = int (pclk / 32768) - 1 = 1 and PREFRAC = pclk - ((PREINT +1) x 32768) = 1
With this prescaler setting, exactly 32,768 clocks per second will be provided to the RTC by counting 2 pclks 32,767 times, and 3 pclks once.

In a more realistic case, the pclk frequency is 10 MHz. Then,
PREINT = int (pclk / 32768) - 1 = 304 and PREFRAC = pclk - ((PREINT +1) x 32768) = 5,760.
In this case, 5,760 of the prescaler output clocks will be 306 (305+1) pclks long, the rest will be 305 pclks long.

In a similar manner, any pclk rate greater than 65.536 kHz (as long as it is an even number of cycles per second) may be turned into a 32 kHz reference clock for the RTC. The only caveat is that if PREFRAC does not contain a zero, then not all of the 32,768 per second clocks are of the same length. Some of the clocks are one pclk longer than others. While the longer pulses are distributed as evenly as possible

among the remaining pulses, this "jitter" could possibly be of concern in an application that wishes to observe the contents of the Clock Tick Counter (CTC) directly.

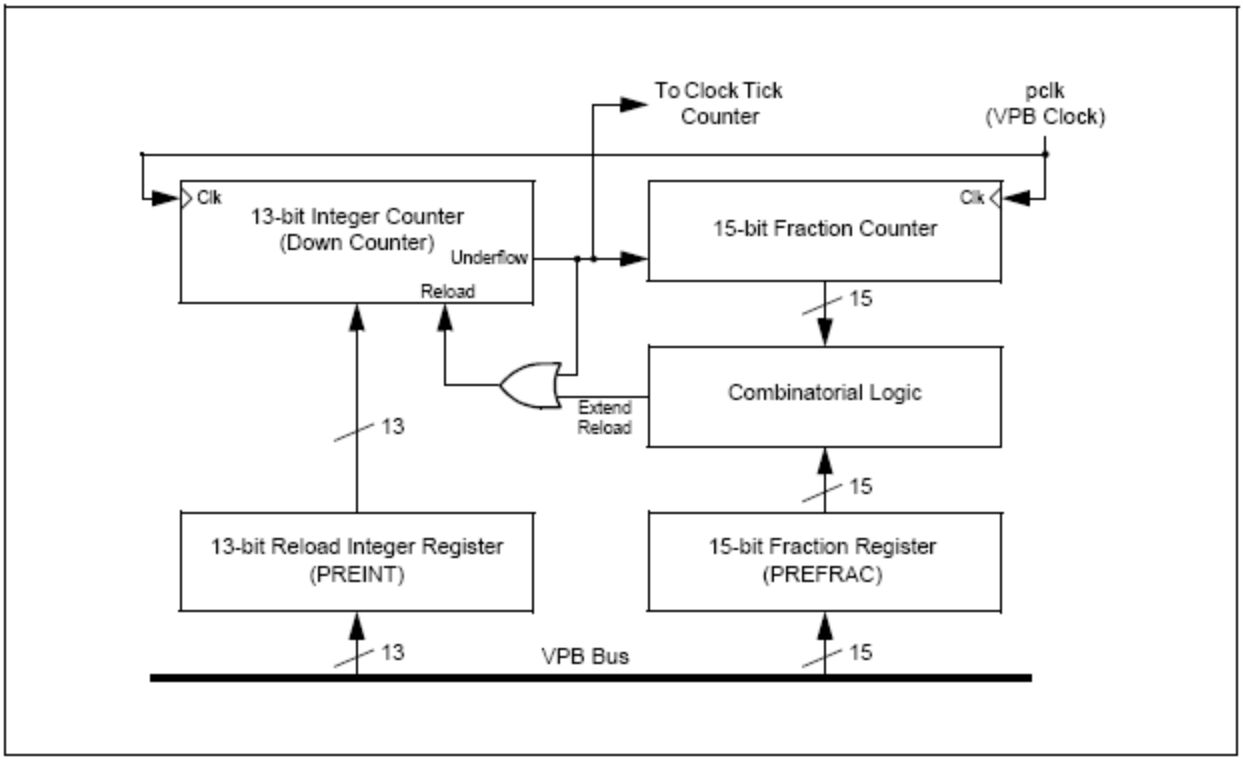


Figure 45: RTC Prescaler block diagram

Prescaler Operation

The Prescaler block labelled "Combination Logic" in Figure 45 determines when the decrement of the 13-bit PREINT counter is extended by one pclk. In order to both insert the correct number of longer cycles, and to distribute them evenly, the Combinatorial Logic associates each bit in PREFRAC with a combination in the 15-bit Fraction Counter.
For example, if PREFRAC bit 14 is a one (representing the fraction 1/2), then half of the cycles counted by the 13-bit counter need to be longer. When there is a 1 in the LSB of the Fraction Counter, the logic causes every alternate count (whenever the LSB of the Fraction Counter=1) to be extended by one pclk, evenly distributing the pulse widths. Similarly, a one in PREFRAC bit 13 (representing the fraction 1/4) will cause every fourth cycle (whenever the two LSBs of the Fraction Counter=10) counted by the 13-bit counter to be longer.

Fraction Counter	PREFRAC Bit															
	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
--- ---- ---- ---1	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-
--- ---- ---- --10	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-
--- ---- ---- -100	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-
--- ---- ---- 1000	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-
--- ---- ---1 0000	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-
--- ---- --10 0000	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-
--- ---- -100 0000	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-
--- ---- 1000 0000	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-
--- ---1 0000 0000	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-
--- --10 0000 0000	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-
--- -100 0000 0000	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-
--- 1000 0000 0000	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-
--1 0000 0000 0000	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-
-10 0000 0000 0000	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-
100 0000 0000 0000	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1

Leap year calculation :

The RTC does a simple bit comparison to see if the two lowest order bits of the year counter are zero. If true, then the RTC considers that year a leap year. The RTC considers all years evenly divisible by 4 as leap years. This algorithm is accurate from the year 1901 through the year 2099, but fails for the year 2100, which is not a leap year. The only effect of leap year on the RTC is to alter the length of the month of February for the month, day of month, and year counters.

End of chapter 11

Cranes Varsity

CHAPTER – 12

UNIVERSAL ASYNCHRONOUS RECEIVER/TRANSMITTER

Features

- 16 byte Receive and Transmit FIFOs
- Register locations conform to '550 industry standard.
- Receiver FIFO trigger points at 1, 4, 8, and 14 bytes.
- Built-in fractional baud rate generator with autobauding capabilities.
- Mechanism that enables software and hardware flow control implementation.

Pin description:

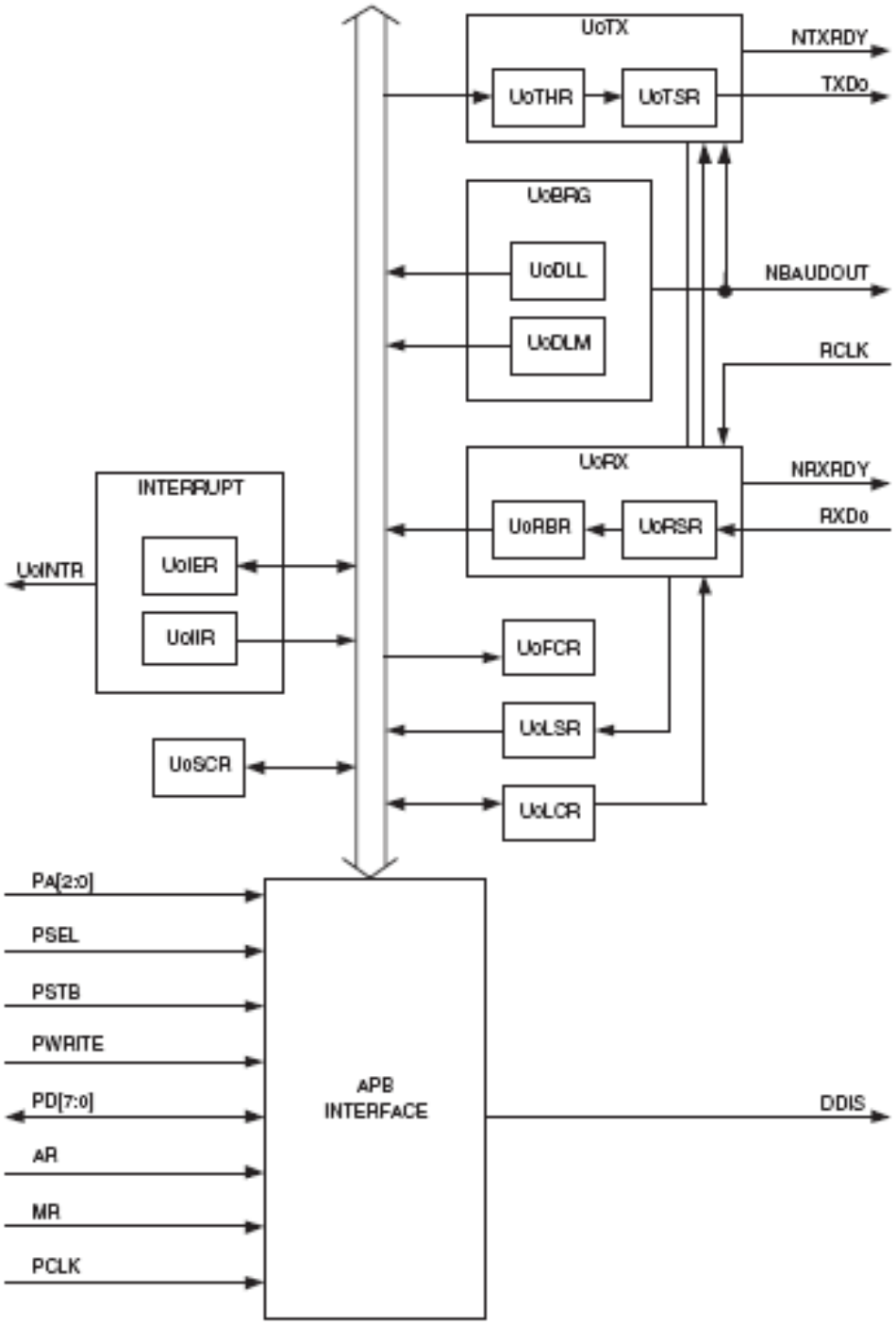
Pin Name	Type	Description
RxD0	Input	**Serial Input**. Serial receive data.
TxD0	Output	**Serial Output**. Serial transmit data.

Architecture

The architecture of the UART0 is shown below in the block diagram. The APB interface provides a communications link between the CPU or host and the UART0. The UART0 receiver block, U0RX, monitors the serial input line, RXD0, for valid input. The UART0 RX Shift Register (U0RSR) accepts valid characters via RXD0. After a valid character is assembled in the U0RSR, it is passed to the UART0 RX Buffer Register FIFO to await access by the CPU or host via the generic host interface.

The UART0 transmitter block, U0TX, accepts data written by the CPU or host and buffers the data in the UART0 TX Holding Register FIFO (U0THR). The UART0 TX Shift Register (U0TSR) reads the data stored in the U0THR and assembles the data to transmit via the serial output pin, TXD0. The UART0 Baud Rate Generator block, U0BRG, generates the timing enables used by the UART0 TX block. The U0BRG clock input source is the APB clock (PCLK). The main clock is divided down per the divisor specified in the U0DLL and U0DLM registers. This divided down clock is a 16x oversample clock, NBAUDOUT. The interrupt interface contains registers U0IER and U0IIR. The interrupt interface receives several one clock wide enables from the U0TX and U0RX blocks. Status information from the U0TX and U0RX is stored in the U0LSR. Control information for the U0TX and U0RX is stored in the U0LCR.

UART0 Block Diagram



Register description

UART0 contains registers organized as shown in Table. The Divisor Latch Access Bit (DLAB) is contained in U0LCR[7] and enables access to the Divisor Latches.

Name	Description	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	Access	Reset Value*	Address
U0RBR	Receiver Buffer Register	MSB			READ DATA				LSB	RO	un-defined	0xE000C000 DLAB = 0
U0THR	Transmit Holding Register	MSB			WRITE DATA				LSB	WO	NA	0xE000C000 DLAB = 0
U0IER	Interrupt Enable Register	0	0	0	0	0	Enable Rx Line Status Interrupt	Enable THRE Interrupt	Enable Rx Data Available Interrupt	R/W	0	0xE000C004 DLAB = 0
U0IIR	Interrupt ID Register	FIFOs Enabled		0	0	IIR3	IIR2	IIR1	IIR0	RO	0x01	0xE000C008
U0FCR	FIFO Control Register	Rx Trigger		Reserved		-	Tx FIFO Reset	Rx FIFO Reset	FIFO Enable	WO	0	0xE000C008
U0LCR	Line Control Register	DLAB	Set Break	Stick Parity	Even Parity Select	Parity Enable	Number of Stop Bits	Word Length Select		R/W	0	0xE000C00C
U0LSR	Line Status Register	Rx FIFO Error	TEMT	THRE	BI	FE	PE	OE	DR	RO	0x60	0xE000C014
U0SCR	Scratch Pad Register	MSB							LSB	R/W	0	0xE000C01C
U0DLL	Divisor Latch LSB	MSB							LSB	R/W	0x01	0xE000C000 DLAB = 1
U0DLM	Divisor Latch MSB	MSB							LSB	R/W	0	0xE000C004 DLAB = 1

UART0 Receiver Buffer Register (U0RBR - 0xE000 C000, when DLAB = 0, Read Only)

The U0RBR is the top byte of the UART0 Rx FIFO. The top byte of the Rx FIFO contains the oldest character received and can be read via the bus interface. The LSB (bit 0) represents the "oldest" received data bit. If the character received is less than 8 bits, the unused MSBs are padded with zeroes. The Divisor Latch Access Bit (DLAB) in U0LCR must be zero in order to access the U0RBR. The U0RBR is always Read Only. Since PE, FE and BI bits correspond to the byte sitting on the top of the RBR FIFO (i.e. the one that will be read in the next read from the RBR), the right approach for fetching the valid pair of received byte and its status bits is first to read the content of the U0LSR register, and then to read a byte from the U0RBR.

U0RBR	Function	Description	Reset Value
7:0	Receiver Buffer Register	The UART0 Receiver Buffer Register contains the oldest received byte in the UART0 Rx FIFO.	un-defined

UART0 Transmit Holding Register (U0THR - 0xE000 C000, when DLAB = 0, Write Only)

The U0THR is the top byte of the UART0 TX FIFO. The top byte is the newest character in the TX FIFO and can be written via the bus interface. The LSB represents the first bit to transmit. The Divisor Latch Access Bit (DLAB) in U0LCR must be zero in order to access the U0THR. The U0THR is always Write Only.

U0THR	Function	Description	Reset Value
7:0	Transmit Holding Register	Writing to the UART0 Transmit Holding Register causes the data to be stored in the UART0 transmit FIFO. The byte will be sent when it reaches the bottom of the FIFO and the transmitter is available.	N/A

UART0 Line Control Register (U0LCR - 0xE000 C00C)

The U0LCR determines the format of the data character that is to be transmitted or received.

Table 144: UART0 Line Control Register (U0LCR - address 0xE000 C00C) bit description

Bit	Symbol	Value	Description	Reset value
1:0	Word Length Select	00	5 bit character length	0
		01	6 bit character length	
		10	7 bit character length	
		11	8 bit character length	
2	Stop Bit Select	0	1 stop bit.	0
		1	2 stop bits (1.5 if U0LCR[1:0]=00).	
3	Parity Enable	0	Disable parity generation and checking.	0
		1	Enable parity generation and checking.	
5:4	Parity Select	00	Odd parity. Number of 1s in the transmitted character and the attached parity bit will be odd.	0
		01	Even Parity. Number of 1s in the transmitted character and the attached parity bit will be even.	
		10	Forced "1" stick parity.	
		11	Forced "0" stick parity.	
6	Break Control	0	Disable break transmission.	0
		1	Enable break transmission. Output pin UART0 TXD is forced to logic 0 when U0LCR[6] is active HIGH.	
7	Divisor Latch Access Bit (DLAB)	0	Disable access to Divisor Latches.	0
		1	Enable access to Divisor Latches.	

UART0 Fractional Divider Register (U0FDR - address 0xE000 C028) bit description

Bit	Function	Description	Reset value
3:0	DIVADDVAL	Baudrate generation pre-scaler divisor value. If this field is 0, fractional baudrate generator will not impact the UART0 baudrate.	0
7:4	MULVAL	Baudrate pre-scaler multiplier value. This field must be greater or equal 1 for UART0 to operate properly, regardless of whether the fractional baudrate generator is used or not.	1
31:8	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

This register controls the clock pre-scaler for the baud rate generation. The reset value of the register keeps the fractional capabilities of UART0 disabled making sure that UART0 is fully software and hardware compatible with UARTs not equipped with this feature. UART0 baud rate can be calculated as:

$$UART0_{baudrate} = \frac{PCLK}{16 \times (256 \times U0DLM + U0DLL) \times \left(1 + \frac{DivAddVal}{MulVal}\right)}$$

Where PCLK is the peripheral clock, U0DLM and U0DLL are the standard UART0 baud rate divider registers, and DIVADDVAL and MULVAL are UART0 fractional baudrate generator specific parameters. The value of MULVAL and DIVADDVAL should comply to the following conditions:

0 < MULVAL ≤ 15
2. 0 ≤ DIVADDVAL ≤ 15

If the U0FDR register value does not comply to these two requests then the fractional divider output is undefined. If DIVADDVAL is zero then the fractional divider is disabled and the clock will not be divided. The value of the U0FDR should not be modified while transmitting/receiving data or data may be lost or corrupted. Usage Note: For practical purposes, UART0 baudrate formula can be written in a way that identifies the part of a UART baudrate generated without the fractional baud rate generator, and the correction factor that this module adds:

$$UART0_{baudrate} = \frac{PCLK}{16 \times (256 \times U0DLM + U0DLL)} \times \frac{MulVal}{(MulVal + DivAddVal)}$$

Based on this representation, fractional baudrate generator contribution can also be described as a prescaling with a factor of

MULVAL / (MULVAL + DIVADDVAL).

UART0 baudrate calculation:

Example 1: Using UART0baudrate formula from above, it can be determined that system with PCLK = 20 MHz, U0DL = 130 (U0DLM = 0x00 and U0DLL = 0x82), DIVADDVAL = 0 and MULVAL = 1 will enable UART0 with UART0baudrate = 9615 bauds.

Example 2: Using UART0baudrate formula from above, it can be determined that system with PCLK = 20 MHz, U0DL = 93 (U0DLM = 0x00 and

U0DLL = 0x5D), DIVADDVAL = 2 and

MULVAL = 5 will enable UART0 with UART0baudrate = 9600 bauds.

Baudrates available when using 20 MHz peripheral clock (PCLK = 20 MHz)

Desired baudrate	MULVAL = 0 DIVADDVAL = 0			Optimal MULVAL & DIVADDVAL		
	U0DLM:U0DLL hex[2]	dec[1]	% error[3]	U0DLM:U0DLL dec[1]	Fractional pre-scaler value MULDIV / (MULDIV + DIVADDVAL)	% error[3]
50	61A8	25000	0.0000	25000	1/(1+0)	0.0000
75	411B	16667	0.0020	12500	3/(3+1)	0.0000
110	2C64	11364	0.0032	6250	11/(11+9)	0.0000
134.5	244E	9294	0.0034	3983	3/(3+4)	0.0001
150	208D	8333	0.0040	6250	3/(3+1)	0.0000
300	1047	4167	0.0080	3125	3/(3+1)	0.0000
600	0823	2083	0.0160	1250	3/(3+2)	0.0000
1200	0412	1042	0.0320	625	3/(3+2)	0.0000
1800	02B6	694	0.0640	625	9/(9+1)	0.0000
2000	0271	625	0.0000	625	1/(1+0)	0.0000
2400	0209	521	0.0320	250	12/(12+13)	0.0000

Baudrates available when using 20 MHz peripheral clock (PCLK = 20 MHz)

Desired baudrate	MULVAL = 0 DIVADDVAL = 0			Optimal MULVAL & DIVADDVAL		
	U0DLM:U0DLL		% error[3]	U0DLM:U0DLL	Fractional pre-scaler value	% error[3]
	hex[2]	dec[1]		dec[1]	MULDIV / MULDIV + DIVADDVAL	
3600	015B	347	0.0640	248	5/(5+2)	0.0064
4800	0104	260	0.1600	125	12/(12+13)	0.0000
7200	00AE	174	0.2240	124	5/(5+2)	0.0064
9600	0082	130	0.1600	93	5/(5+2)	0.0064
19200	0041	65	0.1600	31	10/(10+11)	0.0064
38400	0021	33	1.3760	12	7/(7+12)	0.0594
56000	0021	22	1.4400	13	7/(7+5)	0.0160
57600	0016	22	1.3760	19	7/(7+1)	0.0594
112000	000B	11	1.4400	6	7/(7+6)	0.1600
115200	000B	11	1.3760	4	7/(7+12)	0.0594
224000	0006	6	7.5200	3	7/(7+6)	0.1600
448000	0003	3	7.5200	2	5/(5+2)	0.3520

[1] Values in the row represent decimal equivalent of a 16 bit long content (DLM:DLL).

[2] Values in the row represent hex equivalent of a 16 bit long content (DLM:DLL).

[3] Refers to the percent error between desired and actual baudrate.

AUTO-BAUD

The UART0 auto-baud function can be used to measure the incoming baud-rate based on the ”AT" protocol (Hayes command). If enabled the auto-baud feature will measure the bit time of the receive data stream and set the divisor latch registers U0DLM and U0DLL accordingly. Auto-baud is started by setting the U0ACR Start bit. Auto-baud can be stopped by clearing the U0ACR Start bit. The Start bit will clear once auto-baud has finished and reading the bit will return the status of auto-baud (pending/finished).

Two auto-baud measuring modes are available which can be selected by the U0ACR Mode bit. In mode 0 the baud-rate is measured on two subsequent falling edges of the UART0 Rx pin (the falling edge of the start bit and the falling edge of the least significant bit). In mode 1 the baud-rate is measured between the falling edge and the subsequent rising edge of the UART0 Rx pin (the length of the start bit).

The U0ACR AutoRestart bit can be used to automatically restart baud-rate measurement if a time-out occurs (the rate measurement counter overflows). If this bit is set the rate measurement will restart at the next falling edge of the UART0 Rx pin.

The auto-baud function can generate two interrupts.

• The U0IIR ABTOInt interrupt will get set if the interrupt is enabled (U0IER ABToIntEn is set and the auto-baud rate measurement counter overflows).

• The U0IIR ABEOInt interrupt will get set if the interrupt is enabled (U0IER ABEOIntEn is set and the auto-baud has completed successfully). The auto-baud interrupts have to be cleared by setting the corresponding U0ACR, ABTOIntClr and ABEOIntEn bits.
Typically the fractional baud-rate generator is disabled (DIVADDVAL = 0) during auto-baud. However, if the fractional baud-rate generator is enabled (DIVADDVAL > 0), it is going to impact the measuring of UART0 Rx pin baud-rate, but the value of the U0FDR register is not going to be modified after rate measurement. Also, when auto-baud is used, any write to U0DLM and U0DLL registers should be done before U0ACR register write. The minimum and the maximum baudrates supported by UART0 are function of PCLK, number of data bits, stop-bits and parity bits.

$$ratemin = \frac{2 \times PCLK}{16 \times 2^{15}} \leq UART0_{baudrate} \leq \frac{PCLK}{16 \times (2 + databits + paritybits + stopbits)} = ratemax$$

Auto-baud Modes:

When the software is expecting an ”AT" command, it configures the UART0 with the expected character format and sets the U0ACR Start bit. The initial values in the divisor latches U0DLM and U0DLM don‘t care. Because of the ”A" or ”a" ASCII coding (”A" = 0x41, ”a" = 0x61), the UART0 Rx pin sensed start bit and the LSB of the expected character are delimited by two falling edges. When the U0ACR Start bit is set, the auto-baud protocol will execute the following phases:

1. On U0ACR Start bit setting, the baud-rate measurement counter is reset and the UART0 U0RSR is reset. The U0RSR baud rate is switch to the highest rate.

2. A falling edge on UART0 Rx pin triggers the beginning of the start bit. The rate measuring counter will start counting PCLK cycles optionally pre-scaled by the fractional baud-rate generator.

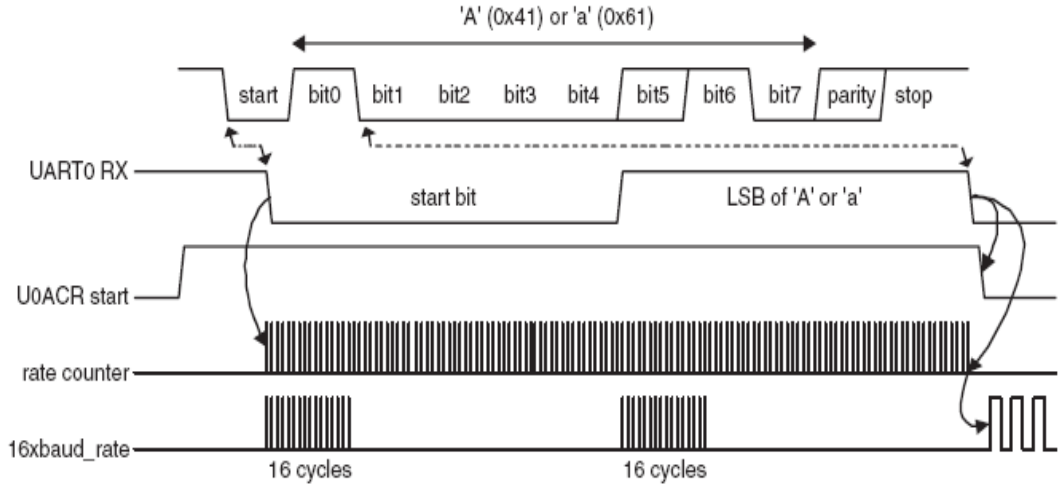
3. During the receipt of the start bit, 16 pulses are generated on the RSR baud input with the frequency of the (fractional baud-rate pre-scaled) UART0 input clock, guaranteeing the start bit is stored in the U0RSR.

4. During the receipt of the start bit (and the character LSB for mode = 0) the rate counter will continue incrementing with the pre-scaled UART0 input clock (PCLK).

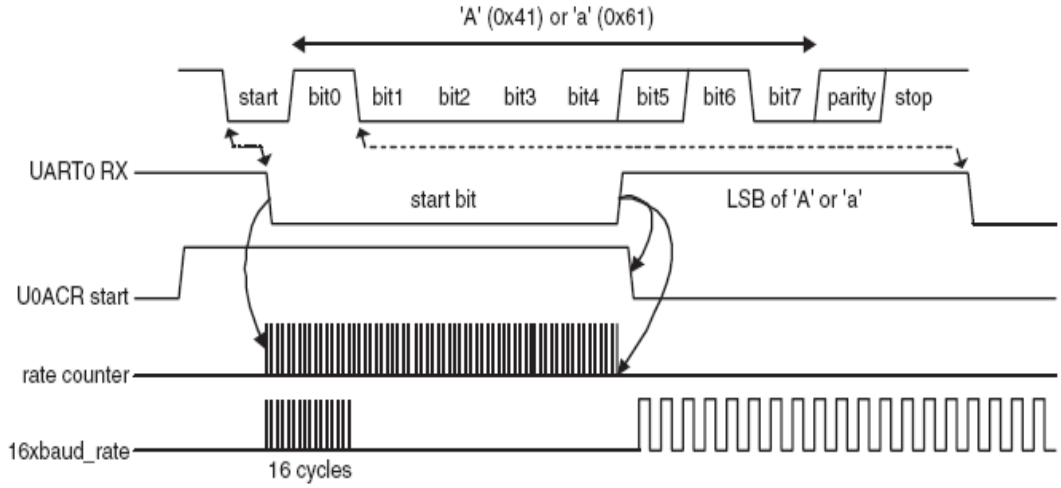
5. If Mode = 0 then the rate counter will stop on next falling edge of the UART0 Rx pin. If Mode = 1 then the rate counter will stop on the next rising edge of the UART0 Rx pin.

6. The rate counter is loaded into U0DLM/U0DLL and the baud-rate will be switched to normal operation. After setting the U0DLM/U0DLL the end of auto-baud interrupt U0IIR ABEOInt will be set, if enabled. The U0RSR will now continue receiving the remaining bits of the ”A/a" character.

Autobaud a) mode 0 and b) mode 1 waveform.



a. Mode 0 (start bit and LSB are used for auto-baud)



b. Mode 1 (only start bit is used for auto-baud)

End of chapter 12

CHAPTER – 13

SPI INTERFACES

Features

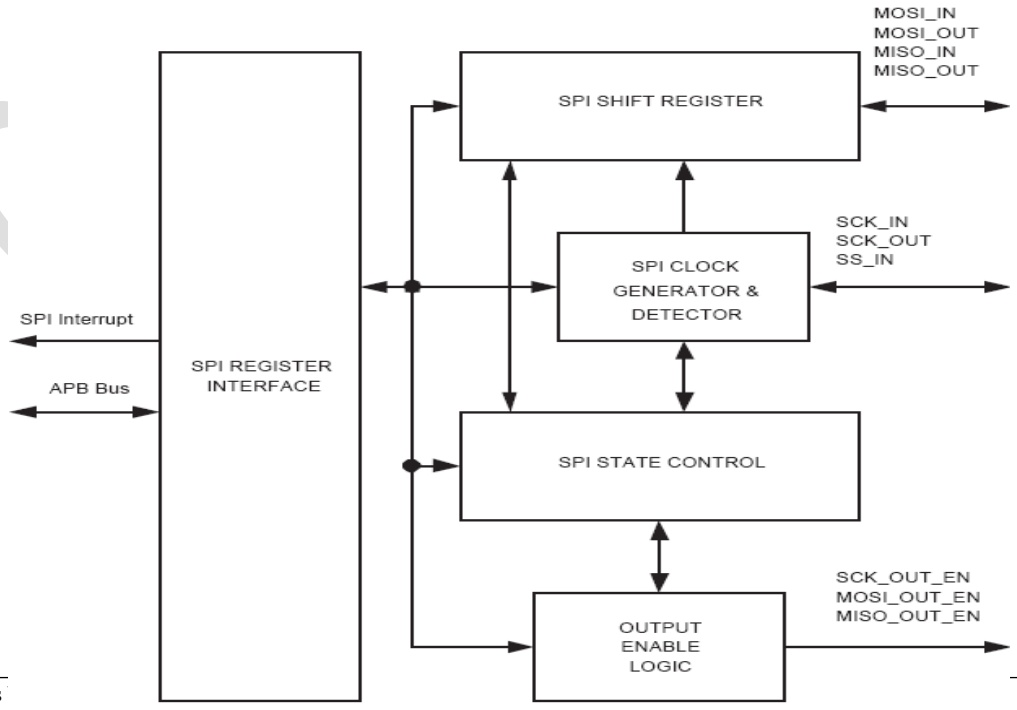
• Single complete and independent SPI controller.
• Compliant with Serial Peripheral Interface (SPI) specification.
• Synchronous, Serial, Full Duplex Communication.
• Combined SPI master and slave.
• Maximum data bit rate of one eighth of the input clock rate. 8 to 16 bits per transfer

SPI overview

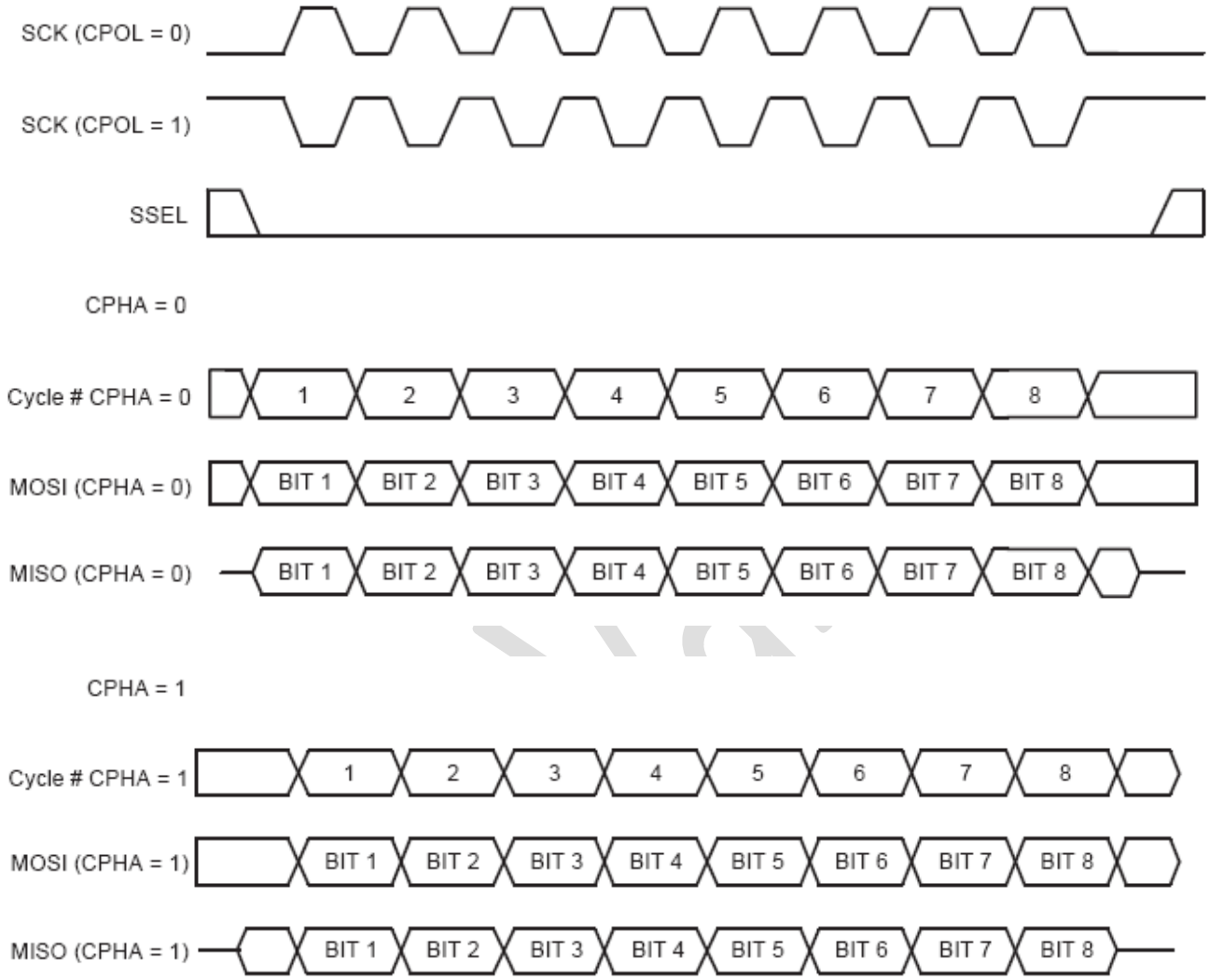
SPI is a full duplex serial interfaces. It can handle multiple masters and slaves being connected to a given bus. Only a single master and a single slave can communicate on the interface during a given data transfer. During a data transfer the master always sends 8 to 16 bits of data to the slave, and the slave always sends a byte of data to the master.

SPI data transfers

Figure is a timing diagram that illustrates the four different data transfer formats that are available with the SPI. This timing diagram illustrates a single 8 bit data transfer. The first thing you should notice in this timing diagram is that it is divided into three horizontal parts. The first part describes the SCK and SSEL signals. The second part describes the MOSI and MISO signals when the CPHA variable is 0. The third part describes the MOSI and MISO signals when the CPHA variable is 1. In the first part of the timing diagram, note two points. First, the SPI is illustrated with CPOL set to both 0 and 1. The second point to note is the activation and de-activation of the SSEL signal. When CPHA = 0, the SSEL signal will always go inactive between data transfers. This is not guaranteed when CPHA = 1 (the signal can remain active).



SPI data transfer format



The data and clock phase relationships are summarized in Table This table summarizes the following for each setting of CPOL and CPHA.

- When the first data bit is driven
- When all other data bits are driven
- When data is sampled

Table SPI data to clock phase relationship

CPOL	CPHA	Firsta data driven	Other data driven	Data sampled
0	0	Prior to first SCK rising edge	SCK falling edge	SCK rising edge
0	1	First SCK rising edge	SCK rising edge	SCK falling edge
1	0	Prior to first SCK falling edge	SCK rising edge	SCK falling edge
1	1	First SCK falling edge	SCK falling edge	SCK rising edge

The definition of when an 8 bit transfer starts and stops is dependent on whether a device

is a master or a slave, and the setting of the CPHA variable. When a device is a master, the start of a transfer is indicated by the master having a byte of data that is ready to be transmitted. At this point, the master can activate the clock, and begin the transfer. The transfer ends when the last clock cycle of the transfer is complete. When a device is a slave, and CPHA is set to 0, the transfer starts when the SSEL signal goes active, and ends when SSEL goes inactive. When a device is a slave, and CPHA is set to 1, the transfer starts on the first clock edge when the slave is selected, and ends on the last clock edge where data is sampled.

Master operation

The following sequence describes how one should process a data transfer with the SPI block when it is set up to be the master. This process assumes that any prior data transfer has already completed.

1. Set the SPI clock counter register to the desired clock rate.
2. Set the SPI control register to the desired settings.
3. Write the data to transmitted to the SPI data register This write starts the SPI data transfer.
4. Wait for the SPIF bit in the SPI status register to be set to 1. The SPIF bit will be set after the last cycle of the SPI data transfer.
5. Read the SPI status register.
6. Read the received data from the SPI data register (optional).
7. Go to step 3 if more data is required to transmit. Note that a read or write of the SPI data register is required in order to clear the SPIF status bit. Therefore, if the optional read of the SPI data register does not take place, a write to this register is required in order to clear the SPIF status bit.

Slave operation

The following sequence describes how one should process a data transfer with the SPI block when it is set up to be a slave. This process assumes that any prior data transfer has already completed. It is required that the system clock driving the SPI logic be at least 8X faster than the SPI.

1. Set the SPI control register to the desired settings.
2. Write the data to transmitted to the SPI data register (optional). Note that this can only be done when a slave SPI transfer is not in progress.
3. Wait for the SPIF bit in the SPI status register to be set to 1. The SPIF bit will be set after the last sampling clock edge of the SPI data transfer.
4. Read the SPI status register.
5. Read the received data from the SPI data register (optional).
6. Go to step 2 if more data is required to transmit. Note that a read or write of the SPI data register is required in order to clear the SPIF status bit. Therefore, at least one of the optional reads or writes of the SPI data register must take place, in order to clear the SPIF status bit.

Exception conditions

Read Overrun

A read overrun occurs when the SPI block internal read buffer contains data that has not been read by the processor, and a new transfer has completed. The read buffer containing valid data is indicated by the SPIF bit in the status register being active. When a transfer completes, the SPI block needs to move the

received data to the read buffer. If the SPIF bit is active (the read buffer is full), the new receive data will be lost, and the read overrun (ROVR) bit in the status register will be activated.

Write Collision

As stated previously, there is no write buffer between the SPI block bus interface, and the internal shift register. As a result, data must not be written to the SPI data register when a SPI data transfer is currently in progress. The time frame where data cannot be written to the SPI data register is from when the transfer starts, until after the status register has been read when the SPIF status is active. If the SPI data register is written in this time frame, the write data will be lost, and the write collision (WCOL) bit in the status register will be activated.

Mode Fault

The SSEL signal must always be inactive when the SPI block is a master. If the SSEL signal goes active, when the SPI block is a master, this indicates another master has selected the device to be a slave. This condition is known as a mode fault. When a mode fault is detected, the mode fault (MODF) bit in the status register will be activated, the SPI signal drivers will be de-activated, and the SPI mode will be changed to be a slave.

Slave Abort

A slave transfer is considered to be aborted, if the SSEL signal goes inactive before the transfer is complete. In the event of a slave abort, the transmit and receive data for the transfer that was in progress are lost, and the slave abort (ABRT) bit in the status register will be activated.

Pin Description

Cranes Varsity

SPI pin description

Pin Name	Type	Pin Description
SCK0	Input/Output	**Serial Clock.** The SPI is a clock signal used to synchronize the transfer of data across the SPI interface. The SPI is always driven by the master and received by the slave. The clock is programmable to be active high or active low. The SPI is only active during a data transfer. Any other time, it is either in its inactive state, or tri-stated.
SSEL0	Input	**Slave Select.** The SPI slave select signal is an active low signal that indicates which slave is currently selected to participate in a data transfer. Each slave has its own unique slave select signal input. The SSEL must be low before data transactions begin and normally stays low for the duration of the transaction. If the SSEL signal goes high any time during a data transfer, the transfer is considered to be aborted. In this event, the slave returns to idle, and any data that was received is thrown away. There are no other indications of this exception. This signal is not directly driven by the master. It could be driven by a simple general purpose I/O under software control. On the LPC2141/2/4/6/8 (unlike earlier Philips ARM devices) the SSEL0 pin can be used for a different function when the SPI0 interface is only used in Master mode. For example, pin hosting the SSEL0 function can be configured as an output digital GPIO pin and used to select one of the SPI0 slaves.
MISO0	Input/Output	**Master In Slave Out.** The MISO signal is a unidirectional signal used to transfer serial data from the slave to the master. When a device is a slave, serial data is output on this signal. When a device is a master, serial data is input on this signal. When a slave device is not selected, the slave drives the signal high impedance.
MOSI0	Input/Output	**Master Out Slave In.** The MOSI signal is a unidirectional signal used to transfer serial data from the master to the slave. When a device is a master, serial data is output on this signal. When a device is a slave, serial data is input on this signal.

Register description:

Name	Description	Access	Reset value[1]	Address
S0SPCR	SPI Control Register. This register controls the operation of the SPI.	R/W	0x00	0xE002 0000
S0SPSR	SPI Status Register. This register shows the status of the SPI.	RO	0x00	0xE002 0004
S0SPDR	SPI Data Register. This bi-directional register provides the transmit and receive data for the SPI. Transmit data is provided to the SPI0 by writing to this register. Data received by the SPI0 can be read from this register.	R/W	0x00	0xE002 0008
S0SPCCR	SPI Clock Counter Register. This register controls the frequency of a master's SCK0.	R/W	0x00	0xE002 000C
S0SPINT	SPI Interrupt Flag. This register contains the interrupt flag for the SPI interface.	R/W	0x00	0xE002 001C

SPI Control Register (S0SPCR - 0xE002 0000 and S1SPCR - 0xE003 0000)

The SPCR register controls the operation of the SPI as per the configuration bits setting.

Table 195. SPI Control Register (S0SPCR - address 0xE002 0000 and S1SPCR - address 0xE003 0000) bit description

Bit	Symbol	Value	Description	Reset value
1:0	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
2	BitEnable[1]	0	The SPI controller sends and receives 8 bits of data per transfer.	0
		1	The SPI controller sends and receives the number of bits selected by bits 11:8.	
3	CPHA	0	Clock phase control determines the relationship between the data and the clock on SPI transfers, and controls when a slave transfer is defined as starting and ending. Data is sampled on the first clock edge of SCK. A transfer starts and ends with activation and deactivation of the SSEL signal.	0
		1	Data is sampled on the second clock edge of the SCK. A transfer starts with the first clock edge, and ends with the last sampling edge when the SSEL signal is active.	
4	CPOL	0	Clock polarity control. SCK is active high.	0
		1	SCK is active low.	
5	MSTR	0	Master mode select. The SPI operates in Slave mode.	0
		1	The SPI operates in Master mode.	
6	LSBF	0	LSB First controls which direction each byte is shifted when transferred. SPI data is transferred MSB (bit 7) first.	0
		1	SPI data is transferred LSB (bit 0) first.	
7	SPIE	0	Serial peripheral interrupt enable. SPI interrupts are inhibited.	0
		1	A hardware interrupt is generated each time the SPIF or MODF bits are activated.	
11:8	BITS[1]		When bit 2 of this register is 1, this field controls the number of bits per transfer:	0000
		1000	8 bits per transfer	
		1001	9 bits per transfer	
		1010	10 bits per transfer	
		1011	11 bits per transfer	
		1100	12 bits per transfer	
		1101	13 bits per transfer	
		1110	14 bits per transfer	
		1111	15 bits per transfer	
		0000	16 bits per transfer	
15:12	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

The S0SPSR register controls the operation of the SPI0 as per the configuration bits setting.

Table 196. SPI Status Register (S0SPSR - address 0xE002 0004 and S1SPSR - address 0xE003 0004) bit description

Bit	Symbol	Description	Reset value
2:0	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
3	ABRT	Slave abort. When 1, this bit indicates that a slave abort has occurred. This bit is cleared by reading this register.	0
4	MODF	Mode fault. when 1, this bit indicates that a Mode fault error has occurred. This bit is cleared by reading this register, then writing the SPI control register.	0
5	ROVR	Read overrun. When 1, this bit indicates that a read overrun has occurred. This bit is cleared by reading this register.	0
6	WCOL	Write collision. When 1, this bit indicates that a write collision has occurred. This bit is cleared by reading this register, then accessing the SPI data register.	0
7	SPIF	SPI transfer complete flag. When 1, this bit indicates when a SPI data transfer is complete. When a master, this bit is set at the end of the last cycle of the transfer. When a slave, this bit is set on the last data sampling edge of the SCK. This bit is cleared by first reading this register, then accessing the SPI data register. **Note:** This is not the SPI interrupt flag. This flag is found in the SPINT register.	0

SPI Data Register (S0SPDR - 0xE0020008, S1SPDR - 0xE0030008)

This bi-directional data register provides the transmit and receive data for the SPI. Transmit data is provided to the SPI by writing to this register. Data received by the SPI can be read from this register. When a master, a write to this register will start a SPI data transfer. Writes to this register will be blocked from when a data transfer starts to when the SPIF status bit is set, and the status register has not been read.

Table 118: SPI Data Register (S0SPDR - 0xE0020008, S1SPDR - 0xE0030008)

SPDR	Function	Description	Reset Value
7:0	Data	SPI Bi-directional data port	0

SPI Clock Counter Register (S0SPCCR - 0xE002 000C) This register controls the frequency of a master's SCK. The register indicates the number of PCLK cycles that make up an SPI clock. The value of this register must always be an even number. As a result, bit 0 must always be 0. The value of the register must also always be greater than or equal to 8. Violations of this can result in unpredictable behavior.

SPI Interrupt register (S0SPINT - address 0xE002 001C) bit description.

Bit	Symbol	Description	Reset value
0	SPI Interrupt Flag	SPI interrupt flag. Set by the SPI interface to generate an interrupt. Cleared by writing a 1 to this bit. **Note:** this bit will be set once when SPIE = 1 and at least one of SPIF and MODF bits changes from 0 to 1. However, only when the SPI Interrupt bit is set and SPI Interrupt is enabled in the VIC, SPI based interrupt can be processed by interrupt handling software.	0
7:1	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Cranes Varsity

I²C INTERFACES

Features:

- Standard I2C compliant bus interfaces may be configured as Master, Slave, or Master/Slave.
- Arbitration is handled between simultaneously transmitting masters without corruption of serial data on the bus.
- Programmable clock allows adjustment of I2C transfer rates.
- Data transfer is bidirectional between masters and slaves.
- Serial clock synchronization allows devices with different bit rates to communicate via one serial bus.
- Serial clock synchronization is used as a handshake mechanism to suspend and resume serial transfer.
- I²C-bus can be used for test and diagnostic purposes.

Applications:

Interfaces to external I2C standard parts, such as serial RAMs, LCDs, tone generators, etc.

I²C Operating Modes:

In a given application, the I2C block may operate as a master, a slave, or both. In the slave mode, the I2C hardware looks for its own slave address and the general call address. If one of these addresses is detected, an interrupt is requested. If the processor wishes to become the bus master, the hardware waits until the bus is free before the master mode is entered so that a possible slave operation is not interrupted. If bus arbitration is lost in the master mode, the I2C block switches to the slave mode immediately and can detect its own slave address in the same serial transfer.

Master Transmitter Mode:

In this mode data is transmitted from master to slave. Before the master transmitter mode can be entered, the I2CONSET register must be initialized as shown in Table 11–133. I2EN must be set to 1 to enable the I2C function. If the AA bit is 0, the I2C interface will not acknowledge any address when another device is master of the bus, so it can not enter slave mode. The STA, STO and SI bits must be 0. The SI Bit is cleared by writing 1 to the SIC bit in the I2CONCLR register The first byte transmitted contains the slave address of the receiving device (7 bits) and the data direction bit. In this mode the data direction bit (R/W) should be 0 which means Write. The first byte transmitted contains the slave address and Write bit. Data is transmitted 8 bits at a time. After each byte is transmitted, an acknowledge bit is received.

START and STOP conditions are output to indicate the beginning and the end of a serial transfer.

The I2C interface will enter master transmitter mode when software sets the STA bit. The I2C logic will send the START condition as soon as the bus is free. After the START condition is transmitted, the SI bit is set, and the status code in the I2STAT register is 0x08. This status code is used to vector to a state service routine which will load the slave address and Write bit to the I2DAT register, and then clear the SI bit. SI is cleared by writing a 1 to the SIC bit in the I2CONCLR register. When the slave address and R/W bit have been transmitted and an acknowledgment bit has been received, the SI bit is set again, and

the possible status codes now are 0x18, 0x20, or 0x38 for the master mode, or 0x68, 0x78, or 0xB0 if the slave mode was enabled (by setting AA to 1).

Master Receiver Mode:

In the master receiver mode, data is received from a slave transmitter. The transfer is initiated in the same way as in the master transmitter mode. When the START condition has been transmitted, the interrupt service routine must load the slave address and the data direction bit to the I2C Data register (I2DAT), and then clear the SI bit. In this case, the data direction bit (R/W) should be 1 to indicate a read.

When the slave address and data direction bit have been transmitted and an acknowledge bit has been received, the SI bit is set, and the Status Register will show the status code. For master mode, the possible status codes are 0x40, 0x48, or 0x38. For slave mode, the possible status codes are 0x68, 0x78, or 0xB0 After a repeated START condition, I2C may switch to the master transmitter mode.

Slave Receiver Mode:

In the slave receiver mode, data bytes are received from a master transmitter. To initialize the slave receiver mode, user write the Slave Address register (I2ADR) and write the I2C Control Set register (I2CONSET) I2EN must be set to 1 to enable the I2C function. AA bit must be set to 1 to acknowledge its own slave address or the general call address. The STA, STO and SI bits are set to 0. After I2ADR and I2CONSET are initialized, the I2C interface waits until it is addressed by its own address or general address followed by the data direction bit. If the direction bit is 0 (W), it enters slave receiver mode. If the direction bit is 1 (R), it enters slave transmitter mode. After the address and direction bit have been received, the SI bit is set and a valid status code can be read from the Status register (I2STAT). Refer to Table 11–150 for the status codes and actions.

Slave Transmitter Mode:

The first byte is received and handled as in the slave receiver mode. However, in this mode, the direction bit will be 1, indicating a read operation. Serial data is transmitted via SDA while the serial clock is input through SCL. START and STOP conditions are recognized as the beginning and end of a serial transfer. In a given application, I2C may operate as a master and as a slave. In the slave mode, the I2C hardware looks for its own slave address and the general call address. If one of these addresses is detected, an interrupt is requested. When the microcontrollers wish to become the bus master, the hardware waits until the bus is free before the master mode is entered so that a possible slave action is not interrupted. If bus arbitration is lost in the master mode, the I2C interface switches to the slave mode immediately and can detect its own slave address in the same serial transfer.

Cranes Varsity

I2C Register Description:

REGISTER DESCRIPTION

The I^2C interface contains 7 registers as shown in Table 102. below.

Table 102: I^2C Register Map

Name	Description	Access	Reset Value*	Address
I2CONSET	I^2C Control Set Register	Read/Set	0	0xE001C000
I2STAT	I^2C Status Register	Read Only	0xF8	0xE001C004
I2DAT	I^2C Data Register	Read/Write	0	0xE001C008
I2ADR	I^2C Slave Address Register	Read/Write	0	0xE001C00C
I2SCLH	SCL Duty Cycle Register High Half Word	Read/Write	0x04	0xE001C010
I2SCLL	SCL Duty Cycle Register Low Half Word	Read/Write	0x04	0xE001C014
I2CONCLR	I^2C Control Clear Register	Clear Only	NA	0xE001C018

I2C Control Set Register (I2CONSET)

Table 103: I^2C Control Set Register (I2CONSET - 0xE001C000)

I2CONSET	Function	Description	Reset Value
0	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
1	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
2	AA	Assert acknowledge flag	0
3	SI	I^2C interrupt flag	0
4	STO	STOP flag	0
5	STA	START flag	0
6	I2EN	I^2C interface enable	0
7	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

The I2CONSET registers control setting of bits in the I2CON register that controls operation of the I2C interface. Writing a one to a bit of this register causes the corresponding bit in the I2C control register to be set. Writing a zero has no effect.

I2EN I2C Interface Enable. When I2EN is 1, the I2C interface is enabled. I2EN can be cleared by writing 1 to the I2ENC bit in the I2CONCLR register. When I2EN is 0, the I2C interface is disabled. When I2EN is “0”, the SDA and SCL input signals are ignored, the I2C block is in the “not addressed” slave state, and the STO bit is forced to “0”. I2EN should not be used to temporarily release the I2C-bus since, when I2EN is reset; the I2C-bus status is lost. The AA flag should be used instead.

STA is the START flag. Setting this bit causes the I2C interface to enter master mode and transmit a START condition or transmit a repeated START condition if it is already in master mode. When STA is

1 and the I2C interface is not already in master mode, it enters master mode, checks the bus and generates a START condition if the bus is free. If the bus is not free, it waits for a STOP condition (which will free the bus) and generates a START condition after a delay of a half clock period of the internal clock generator. If the I2C interface is already in master mode and data has been transmitted or received, it transmits a repeated START condition. STA may be set at any time, including when the I2C interface is in an addressed slave mode.

STA can be cleared by writing 1 to the STAC bit in the I2CONCLR register. When STA is 0, no START condition or repeated START condition will be generated. If STA and STO are both set, then a STOP condition is transmitted on the I2C-bus if it the interface is in master mode, and transmits a START condition thereafter. If the I2C interface is in slave mode, an internal STOP condition is generated, but is not transmitted on the bus.

STO is the STOP flag. Setting this bit causes the I2C interface to transmit a STOP condition in master mode, or recover from an error condition in slave mode. When STO is 1 in master mode, a STOP condition is transmitted on the I2C-bus. When the bus detects the STOP condition, STO is cleared automatically. In slave mode, setting this bit can recover from an error condition. In this case, no STOP condition is transmitted to the bus. The hardware behaves as if a STOP condition has been received and it switches to “not addressed” slave receiver mode. The STO flag is cleared by hardware automatically.

SI is the I2C Interrupt Flag. This bit is set when the I2C state changes. However, entering state F8 does not set SI since there is nothing for an interrupt service routine to do in that case. While SI is set, the low period of the serial clock on the SCL line is stretched, and the serial transfer is suspended. When SCL is HIGH, it is unaffected by the state of the SI flag. SI must be reset by software, by writing a 1 to the SIC bit in I2CONCLR register.

AA is the Assert Acknowledge Flag. When set to 1, an acknowledge (low level to SDA) will be returned during the acknowledge clock pulse on the SCL line on the following situations:

1. The address in the Slave Address Register has been received.

2. The general call address has been received while the general call bit (GC) in I2ADR is set.

3. A data byte has been received while the I2C is in the master receiver mode.

4. A data byte has been received while the I2C is in the addressed slave receiver mode

The AA bit can be cleared by writing 1 to the AAC bit in the I2CONCLR register. When AA is 0, a not acknowledge (HIGH level to SDA) will be returned during the acknowledge clock pulse on the SCL line on the following situations:

1. A data byte has been received while the I2C is in the master receiver mode.

2. A data byte has been received while the I2C is in the addressed slave receiver mode.

I2C Control Clear Register (I2CONCLR)

I^2C Control Clear Register (I2CONCLR - 0xE001C018)

Table 104: I^2C Control Clear Register (I2CONCLR - 0xE001C018)

I2CONCLR	Function	Description	Reset Value
0	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
1	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
2	AAC	Assert Acknowledge Clear bit. Writing a 1 to this bit clears the AA bit in the I2CONSET register. Writing 0 has no effect.	NA
3	SIC	I^2C Interrupt Clear Bit. Writing a 1 to this bit clears the SI bit in the I2CONSET register. Writing 0 has no effect.	NA
4	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
5	STAC	Start flag clear bit. Writing a 1 to this bit clears the STA bit in the I2CONSET register. Writing 0 has no effect.	NA
6	I2ENC	I^2C interface disable. Writing a 1 to this bit clears the I2EN bit in the I2CONSET register.Writing 0 has no effect.	NA
7	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

The I2CONCLR registers control clearing of bits in the I2CON register that controls operation of the I2C interface. Writing a one to a bit of this register causes the corresponding bit in the I2C control register to be cleared. Writing a zero has no effect.

AAC is the Assert Acknowledge Clear bit. Writing a 1 to this bit clears the AA bit in the I2CONSET register. Writing 0 has no effect.
SIC is the I2C Interrupt Clear bit. Writing a 1 to this bit clears the SI bit in the I2CONSET register. Writing 0 has no effect.
STAC is the Start flag clear bit. Writing a 1 to this bit clears the STA bit in the I2CONSET register. Writing 0 has no effect.
I**2ENC** is the I2C Interface Disable bit. Writing a 1 to this bit clears the I2EN bit in the I2CONSET register. Writing 0 has no effect.

I2C Status Register (I2STAT)

Bit	Symbol	Description	Reset value
2:0	-	These bits are unused and are always 0.	0
7:3	Status	These bits give the actual status information about the I^2C interface.	0x1F

Each I2C Status register reflects the condition of the corresponding I2C interface. The I2C Status register is Read-Only. The three least significant bits are always 0. Taken as a byte, the status register contents represent a status code. There are 26 possible status codes. When the status code is 0xF8, there is no relevant information available and the SI bit is not set. All other 25 status codes correspond to defined I2C states. When any of these states entered, the SI bit will be set.

I2C Data Register (I2DAT: I2C0DAT, I2C1DAT)

Bit	Symbol	Description	Reset value
7:0	Data	This register holds data values that have been received, or are to be transmitted.	0

This register contains the data to be transmitted or the data just received. The CPU can read and write to this register only while it is not in the process of shifting a byte, when the SI bit is set. Data in I2DAT remains stable as long as the SI bit is set. Data in I2DAT is always shifted from right to left: the first bit to be transmitted is the MSB (bit 7), and after a byte has been received, the first bit of received data is located at the MSB of I2DAT.

I2C Salve Address Register (I2ADR: I2C0ADR, I2C1ADR)

Bit	Symbol	Description	Reset value
0	GC	General Call enable bit.	0
7:1	Address	The I^2C device address for slave mode.	0x00

These registers are readable and writable, and is only used when an I2C interface is set to slave mode. In master mode, this register has no effect. The LSB of I2ADR is the general call bit. When this bit is set, the general call address (0x00) is recognized.

I2C SCL High Duty Cycle Register (I2SCLH: I2C0SCLH, I2C1SCLH)

Bit	Symbol	Description	Reset value
15:0	SCLH	Count for SCL HIGH time period selection.	0x0004

I2C SCL Low Duty Cycle Register (I2SCLL: I2C0SCLL, I2C1SCLL)

Bit	Symbol	Description	Reset value
15:0	SCLL	Count for SCL low time period selection.	0x0004

Selecting the appropriate I2C data rate and duty cycle:

Software must set values for the registers I2SCLH and I2SCLL to select the appropriate data rate and duty cycle. I2SCLH defines the number of PCLK cycles for the SCL HIGH time, I2SCLL defines the number of PCLK cycles for the SCL low time. The frequency is determined by the following formula (PCLK is the frequency of the peripheral bus APB):

The values for I2SCLL and I2SCLH should not necessarily be the same. Software can set different duty cycles on SCL by setting these two registers.

$$I^2C_{bitfrequency} = \frac{PCLK}{I2CSCLH + I2CSCLL}$$

For example, the I2C-bus specification defines the SCL low time and high time at different values for a 400 kHz I2C rate. The value of the register must ensure that the data rate is in the I2C data rate range of 0 through 400 kHz. Each register value must be greater than or equal to 4.
Here are given with some examples of I2C-bus rates based on PCLK frequency and I2SCLL and I2SCLH values.

I2SCLL + I2SCLH	I²C Bit Frequency (kHz) at PCLK (MHz)						
	1	5	10	16	20	40	60
8	125						
10	100						
25	40	200	400				
50	20	100	200	320	400		
100	10	50	100	160	200	400	
160	6.25	31.25	62.5	100	125	250	375
200	5	25	50	80	100	200	300
400	2.5	12.5	25	40	50	100	150
800	1.25	6.25	12.5	20	25	50	75

Details of I2C Operationg Modes:
Master Transmitter mode

Status Code (I2CSTAT)	Status of the I²C-bus and hardware	Application software response					Next action taken by I²C hardware
		To/From I2DAT	To I2CON				
			STA	STO	SI	AA	
0x08	A START condition has been transmitted.	Load SLA+W	X	0	0	X	SLA+W will be transmitted; ACK bit will be received.
0x10	A repeated START condition has been transmitted.	Load SLA+W or	X	0	0	X	As above.
		Load SLA+R	X	0	0	X	SLA+W will be transmitted; the I²C block will be switched to MST/REC mode.
0x18	SLA+W has been transmitted; ACK has been received.	Load data byte or	0	0	0	X	Data byte will be transmitted; ACK bit will be received.
		No I2DAT action or	1	0	0	X	Repeated START will be transmitted.
		No I2DAT action or	0	1	0	X	STOP condition will be transmitted; STO flag will be reset.
		No I2DAT action	1	1	0	X	STOP condition followed by a START condition will be transmitted; STO flag will be reset.
0x20	SLA+W has been transmitted; NOT ACK has been received.	Load data byte or	0	0	0	X	Data byte will be transmitted; ACK bit will be received.
		No I2DAT action or	1	0	0	X	Repeated START will be transmitted.
		No I2DAT action or	0	1	0	X	STOP condition will be transmitted; STO flag will be reset.
		No I2DAT action	1	1	0	X	STOP condition followed by a START condition will be transmitted; STO flag will be reset.
0x28	Data byte in I2DAT has been transmitted; ACK has been received.	Load data byte or	0	0	0	X	Data byte will be transmitted; ACK bit will be received.
		No I2DAT action or	1	0	0	X	Repeated START will be transmitted.
		No I2DAT action or	0	1	0	X	STOP condition will be transmitted; STO flag will be reset.
		No I2DAT action	1	1	0	X	STOP condition followed by a START condition will be transmitted; STO flag will be reset.
0x30	Data byte in I2DAT has been transmitted; NOT ACK has been received.	Load data byte or	0	0	0	X	Data byte will be transmitted; ACK bit will be received.
		No I2DAT action or	1	0	0	X	Repeated START will be transmitted.
		No I2DAT action or	0	1	0	X	STOP condition will be transmitted; STO flag will be reset.
		No I2DAT action	1	1	0	X	STOP condition followed by a START condition will be transmitted; STO flag will be reset.
0x38	Arbitration lost in SLA+R/W or Data bytes.	No I2DAT action or	0	0	0	X	I²C-bus will be released; not addressed slave will be entered.
		No I2DAT action	1	0	0	X	A START condition will be transmitted when the bus becomes free.

Cranes

Master Receiver mode

Status Code (I2CSTAT)	Status of the I^2C-bus and hardware	Application software response					Next action taken by I^2C hardware
		To/From I2DAT	To I2CON				
			STA	STO	SI	AA	
0x08	A START condition has been transmitted.	Load SLA+R	X	0	0	X	SLA+R will be transmitted; ACK bit will be received.
0x10	A repeated START condition has been transmitted.	Load SLA+R or	X	0	0	X	As above.
		Load SLA+W	X	0	0	X	SLA+W will be transmitted; the I^2C block will be switched to MST/TRX mode.
0x38	Arbitration lost in NOT ACK bit.	No I2DAT action or	0	0	0	X	I^2C-bus will be released; the I^2C block will enter a slave mode.
		No I2DAT action	1	0	0	X	A START condition will be transmitted when the bus becomes free.
0x40	SLA+R has been transmitted; ACK has been received.	No I2DAT action or	0	0	0	0	Data byte will be received; NOT ACK bit will be returned.
		No I2DAT action	0	0	0	1	Data byte will be received; ACK bit will be returned.
0x48	SLA+R has been transmitted; NOT ACK has been received.	No I2DAT action or	1	0	0	X	Repeated START condition will be transmitted.
		No I2DAT action or	0	1	0	X	STOP condition will be transmitted; STO flag will be reset.
		No I2DAT action	1	1	0	X	STOP condition followed by a START condition will be transmitted; STO flag will be reset.
0x50	Data byte has been received; ACK has been returned.	Read data byte or	0	0	0	0	Data byte will be received; NOT ACK bit will be returned.
		Read data byte	0	0	0	1	Data byte will be received; ACK bit will be returned.
0x58	Data byte has been received; NOT ACK has been returned.	Read data byte or	1	0	0	X	Repeated START condition will be transmitted.
		Read data byte or	0	1	0	X	STOP condition will be transmitted; STO flag will be reset.
		Read data byte	1	1	0	X	STOP condition followed by a START condition will be transmitted; STO flag will be reset.

Slave Receiver mode

Status Code (I2CSTAT)	Status of the I^2C-bus and hardware	Application software response					Next action taken by I^2C hardware
		To/From I2DAT	To I2CON				
			STA	STO	SI	AA	
0x60	Own SLA+W has been received; ACK has been returned.	No I2DAT action or	X	0	0	0	Data byte will be received and NOT ACK will be returned.
		No I2DAT action	X	0	0	1	Data byte will be received and ACK will be returned.
0x68	Arbitration lost in SLA+R/W as master; Own SLA+W has been received, ACK returned.	No I2DAT action or	X	0	0	0	Data byte will be received and NOT ACK will be returned.
		No I2DAT action	X	0	0	1	Data byte will be received and ACK will be returned.
0x70	General call address (0x00) has been received; ACK has been returned.	No I2DAT action or	X	0	0	0	Data byte will be received and NOT ACK will be returned.
		No I2DAT action	X	0	0	1	Data byte will be received and ACK will be returned.
0x78	Arbitration lost in SLA+R/W as master; General call address has been received, ACK has been returned.	No I2DAT action or	X	0	0	0	Data byte will be received and NOT ACK will be returned.
		No I2DAT action	X	0	0	1	Data byte will be received and ACK will be returned.
0x80	Previously addressed with own SLV address; DATA has been received; ACK has been returned.	Read data byte or	X	0	0	0	Data byte will be received and NOT ACK will be returned.
		Read data byte	X	0	0	1	Data byte will be received and ACK will be returned.
0x88	Previously addressed with own SLA; DATA byte has been received; NOT ACK has been returned.	Read data byte or	0	0	0	0	Switched to not addressed SLV mode; no recognition of own SLA or General call address.
		Read data byte or	0	0	0	1	Switched to not addressed SLV mode; Own SLA will be recognized; General call address will be recognized if I2ADR[0] = logic 1.
		Read data byte or	1	0	0	0	Switched to not addressed SLV mode; no recognition of own SLA or General call address. A START condition will be transmitted when the bus becomes free.
		Read data byte	1	0	0	1	Switched to not addressed SLV mode; Own SLA will be recognized; General call address will be recognized if I2ADR[0] = logic 1. A START condition will be transmitted when the bus becomes free.
0x90	Previously addressed with General Call; DATA byte has been received; ACK has been returned.	Read data byte or	X	0	0	0	Data byte will be received and NOT ACK will be returned.
		Read data byte	X	0	0	1	Data byte will be received and ACK will be returned.

0x98	Previously addressed with General Call; DATA byte has been received; NOT ACK has been returned.	Read data byte or	0	0	0	0	Switched to not addressed SLV mode; no recognition of own SLA or General call address.
		Read data byte or	0	0	0	1	Switched to not addressed SLV mode; Own SLA will be recognized; General call address will be recognized if I2ADR[0] = logic 1.
		Read data byte or	1	0	0	0	Switched to not addressed SLV mode; no recognition of own SLA or General call address. A START condition will be transmitted when the bus becomes free.
		Read data byte	1	0	0	1	Switched to not addressed SLV mode; Own SLA will be recognized; General call address will be recognized if I2ADR[0] = logic 1. A START condition will be transmitted when the bus becomes free.
0xA0	A STOP condition or repeated START condition has been received while still addressed as SLV/REC or SLV/TRX.	No STDAT action or	0	0	0	0	Switched to not addressed SLV mode; no recognition of own SLA or General call address.
		No STDAT action or	0	0	0	1	Switched to not addressed SLV mode; Own SLA will be recognized; General call address will be recognized if I2ADR[0] = logic 1.
		No STDAT action or	1	0	0	0	Switched to not addressed SLV mode; no recognition of own SLA or General call address. A START condition will be transmitted when the bus becomes free.
		No STDAT action	1	0	0	1	Switched to not addressed SLV mode; Own SLA will be recognized; General call address will be recognized if I2ADR[0] = logic 1. A START condition will be transmitted when the bus becomes free.

Cranes

Slave Transmitter mode

Status Code (I2CSTAT)	Status of the I²C-bus and hardware	Application software response					Next action taken by I²C hardware
		To/From I2DAT	To I2CON				
			STA	STO	SI	AA	
0xA8	Own SLA+R has been received; ACK has been returned.	Load data byte or	X	0	0	0	Last data byte will be transmitted and ACK bit will be received.
		Load data byte	X	0	0	1	Data byte will be transmitted; ACK will be received.
0xB0	Arbitration lost in SLA+R/W as master; Own SLA+R has been received, ACK has been returned.	Load data byte or	X	0	0	0	Last data byte will be transmitted and ACK bit will be received.
		Load data byte	X	0	0	1	Data byte will be transmitted; ACK bit will be received.
0xB8	Data byte in I2DAT has been transmitted; ACK has been received.	Load data byte or	X	0	0	0	Last data byte will be transmitted and ACK bit will be received.
		Load data byte	X	0	0	1	Data byte will be transmitted; ACK bit will be received.
0xC0	Data byte in I2DAT has been transmitted; NOT ACK has been received.	No I2DAT action or	0	0	0	0	Switched to not addressed SLV mode; no recognition of own SLA or General call address.
		No I2DAT action or	0	0	0	1	Switched to not addressed SLV mode; Own SLA will be recognized; General call address will be recognized if I2ADR[0] = logic 1.
		No I2DAT action or	1	0	0	0	Switched to not addressed SLV mode; no recognition of own SLA or General call address. A START condition will be transmitted when the bus becomes free.
		No I2DAT action	1	0	0	1	Switched to not addressed SLV mode; Own SLA will be recognized; General call address will be recognized if I2ADR[0] = logic 1. A START condition will be transmitted when the bus becomes free.
0xC8	Last data byte in I2DAT has been transmitted (AA = 0); ACK has been received.	No I2DAT action or	0	0	0	0	Switched to not addressed SLV mode; no recognition of own SLA or General call address.
		No I2DAT action or	0	0	0	1	Switched to not addressed SLV mode; Own SLA will be recognized; General call address will be recognized if I2ADR[0] = logic 1.
		No I2DAT action or	1	0	0	0	Switched to not addressed SLV mode; no recognition of own SLA or General call address. A START condition will be transmitted when the bus becomes free.
		No I2DAT action	1	0	0	01	Switched to not addressed SLV mode; Own SLA will be recognized; General call address will be recognized if I2ADR.0 = logic 1. A START condition will be transmitted when the bus becomes free.

Software Examples:

Initialization routine:

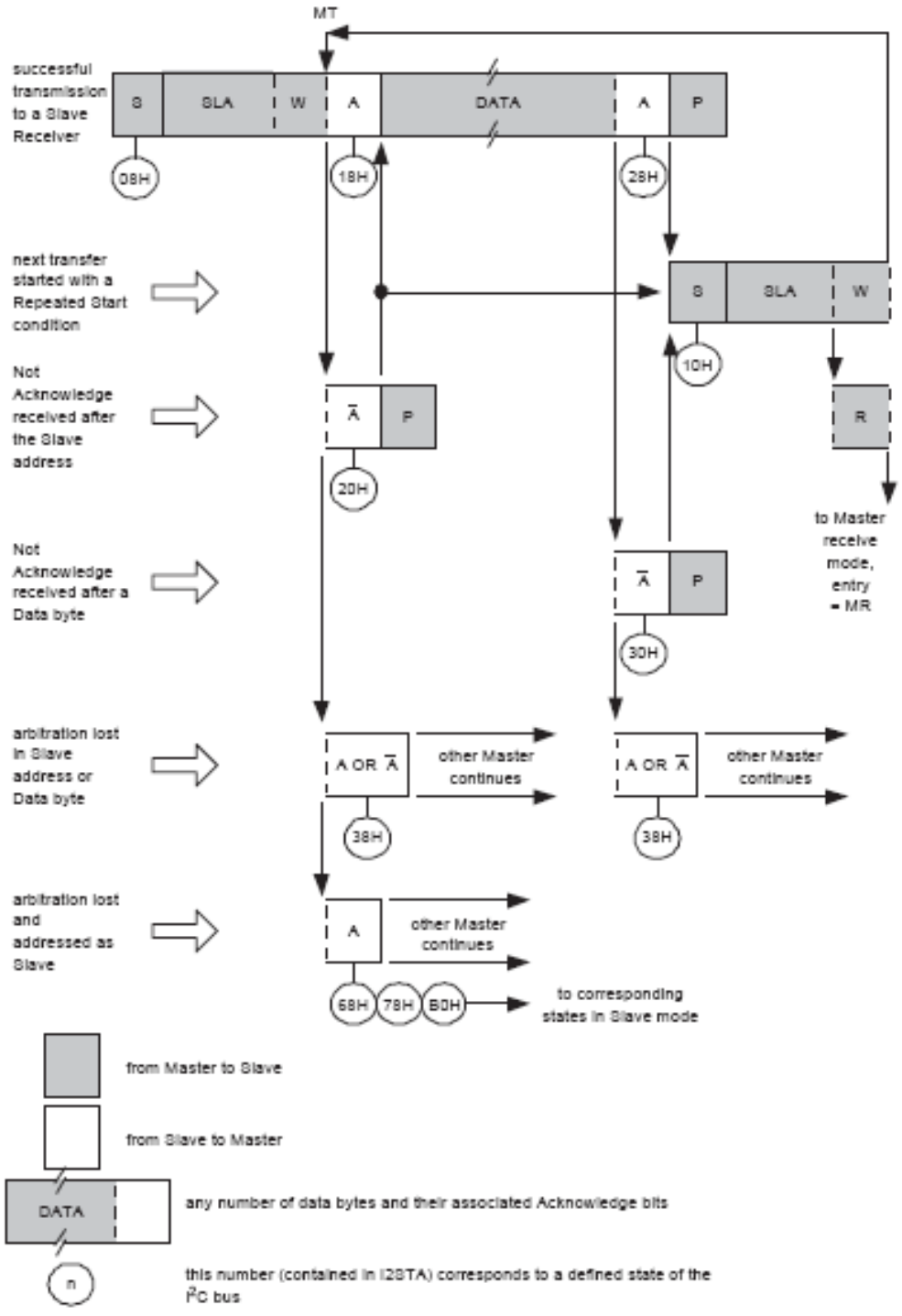
Example to initialize I²C Interface as a Slave and/or Master.
1. Load I2ADR with own Slave Address, enable general call recognition if needed.
2. Enable I2C interrupt.
3. Write 0x44 to I2CONSET to set the I2EN and AA bits, enabling Slave functions. For master only functions, write 0x40 to I2CONSET.

Start master receive function:

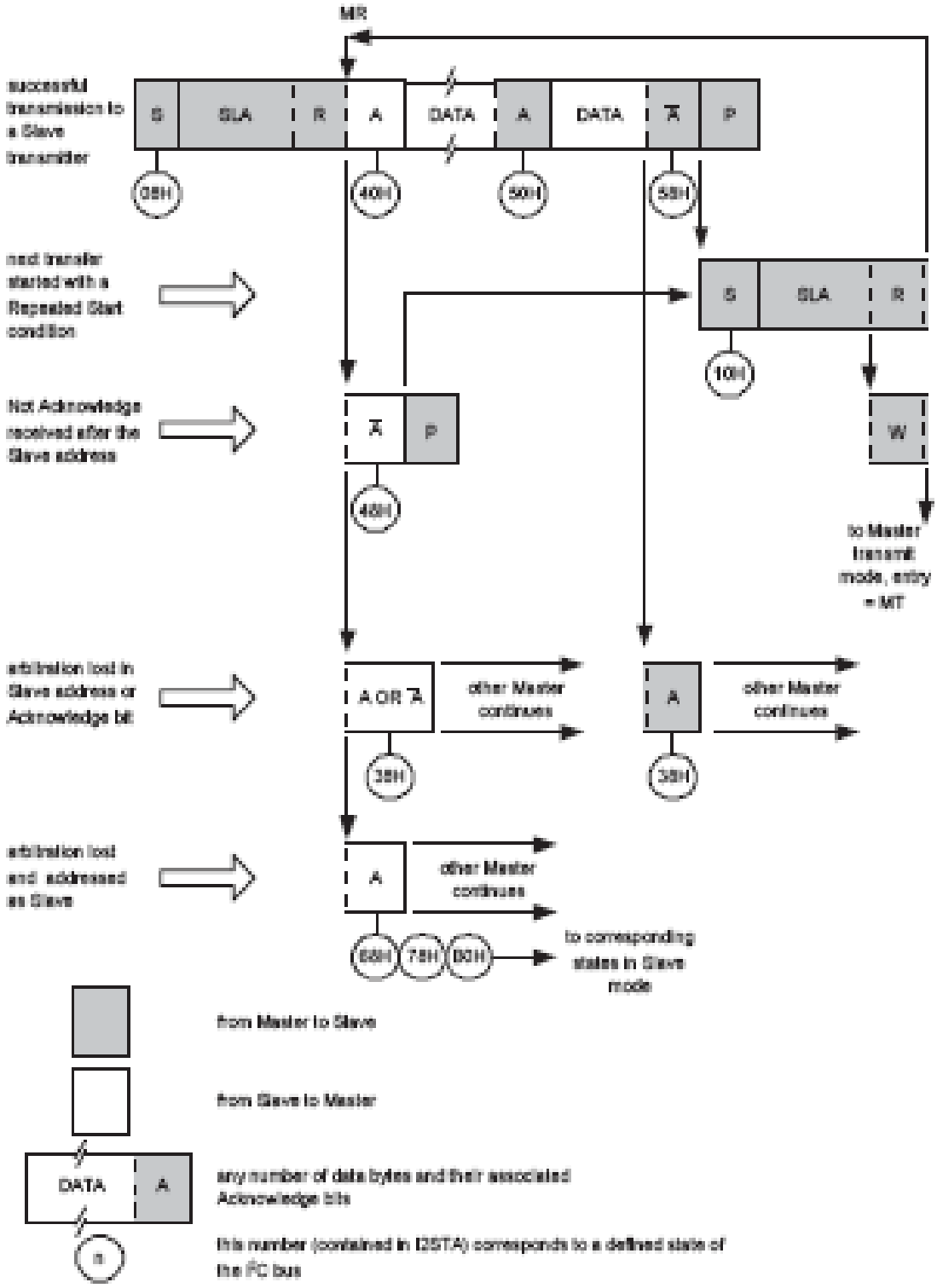
Begin a Master Transmit operation by setting up the buffer, pointer, and data count, then initiating a Start.
1. Initialize Master data counter.
2. Set up the Slave Address to which data will be transmitted, and add the Write bit.
3. Write 0x20 to I2CONSET to set the STA bit.
4. Set up data to be transmitted in Master Transmit buffer.
5. Initialize the Master data counter to match the length of the message being sent.
6. Exit

Cranes Varsity

Format and States in the Master Transmitter mode



Format and States in the Master Receiver mode


MR
successful transmission to a Slave transmitter
S
SLA
R
A
DATA
A
DATA
A
P
08H
40H
50H
58H
next transfer started with a Repeated Start condition
S
SLA
R
10H
Not Acknowledge received after the Slave address
A
P
W
48H
to Master transmit mode, entry = MT
arbitration lost in Slave address or Acknowledge bit
A OR A
other Master continues
A
other Master continues
38H
38H
arbitration lost and addressed as Slave
A
other Master continues
68H
78H
B0H
to corresponding states in Slave mode
from Master to Slave
from Slave to Master
DATA
A
any number of data bytes and their associated Acknowledge bits
n
this number (contained in I2STA) corresponds to a defined state of the I²C bus

CHAPTER-15

INTRODUCTION TO ARM CORTEX

The ARM® Cortex® series of cores encompasses a very wide range of scalable performance options offering designers a great deal of choice and the opportunity to use the best-fit core for their application without being forced into a one-size-fits-all solution. The cortex family of processors provides ARM partners with a range of solutions optimized around specific market applications across the full performance spectrum. This underlines ARM's strategy of aligning technology around specific market applications and requirements.

The Cortex portfolio is split broadly into three main categories:

• Cortex-Ax -- application processor cores for a performance-intensive systems. The Cortex-A8 and Cortex-A9 processors support the ARM, Thumb and Thumb-2 instruction sets.

• Cortex-Rx – high-performance cores for real-time applications.

• Cortex-Mx– microcontroller cores for a wide range of embedded applications. These processors support the Thumb-2 instruction set only. This family comprises the Cortex-M3, Cortex-M3, the Cortex-M1 and the Cortex-M0 processors

The **x** indicates that after the letter (A, R, M) is a number that identifies in detail the core.

Cortex-A processors

Provide a range of solutions for devices that make use of a rich operating system such as Linux or Android and are used in a wide range of applications from low-cost handsets to smart phones, tablet computers, set-top boxes and also enterprise networking equipment. The first range of Cortex-A processors (A5, A7, A8, A9, A12, A15 and A17) is based on the ARMv7-architecture. Each core shares a common feature set including items such as the NEON media processing engine, Trust zone for security extensions, and single- and double-precision floating point support along with support for several instruction sets (ARM, Thumb-2, Thumb, Jazelle and DSP). Together this group of processors offers design flexibility by providing the required peak performance points while delivering the desired power efficiency.

Cortex Ax Features:

- Instruction set support – ARM, Thumb, Thumb2, DSP
- Advanced single and double precision floating point support.
- Virtualization
- Large physical address extension (LPAE) addressing up to 1TB of physical memory.
- Single to quad core implementation for performance oriented applications.
- Supports symmetric and asymmetric OS implementations.
- High performance 32bit and 64 bit.
- Advanced branch prediction.

- Cores range from 8 stage pipeline up to 15+ stage pipeline

Cortex-R

Moving on from Cortex-A, the Cortex-R series is the smallest ARM processor offering in terms of derivatives and possibly the least well known. The Cortex-R processors target high-performance real-time applications such as hard disk controllers (or solid state drive controllers), networking equipment and printers in the enterprise segment, consumer devices such as Blu-ray players and media players, and also automotive applications such as airbags, braking systems and engine management. The Cortex-R series is similar in some respects to a high-end microcontroller (MCU) but targets larger systems than you would typically use a standard MCU.

Cortex Rx Features:

1. Low energy micro architecture.
2. MP core interrupt controller.
3. Floating point instructions including double precision.
4. ARM or Thumb2 code for high code density.
5. Data I/O coherency
6. Target from low power/small die area to high performance
7. Synthesis feature configuration allowing designers to select features of the processor for a precise match with application requirements.
8. Any instructions that could delay interrupt response by more than a few cycles can stop and restart.
9. These options enable Cortex-R series processors to address a wide range of embedded applications and designers can trade off features and performance against power consumption, area and cost of the final device.

Cortex-M

Finally we come to the Cortex-M series, designed specifically to target the already very crowded MCU market. The Cortex-M series is built on the ARMv7-M architecture (used for Cortex-M3 and Cortex-M4), and the smaller Cortex-M0+ is built on the ARMv6-M architecture. The first Cortex-M processor was released in 2004, and it quickly gained popularity when a few mainstream MCU vendors picked up the core and started producing MCU devices. It is safe to say that the Cortex-M has become for the 32-bit world what the 8051 is for the 8-bit – an industry-standard core supplied by many vendors, each of which dip the core in their own special sauce to provide differentiation in the market. The Cortex-M series can be implemented as a soft core in an FPGA, for example, but it is much more common to find them implemented as MCU with integrated memories, clocks and peripherals. Some are optimized for energy efficiency, some for high performance and some are tailored to a specific market segment such as smart metering.

Cranes Varsity

Cortex Mx Features:

- Power efficient 32 bit processors
- Support for sleep modes
- Low power design with further optimization packs available.
- Low power consumption enables longer battery life.
- Instructions to support sleep modes.
- High code density reduces memory size.
- Smaller area reduces die cost and chip packaging size.
- Powerful instruction set to enable high performance systems.
- Bit field processing instructions for I/O control and communication applications.
- Powerful debug and trace features using small number of connection pins
- Support for debug multiple processors
- Multiple choices of debug communication protocols(JTAG and Serial wire debug)

ARM family comparison

ARM family attribute comparison.

	ARM7	ARM9	ARM10	ARM11
Pipeline depth	three-stage	five-stage	six-stage	eight-stage
Typical MHz	80	150	260	335
mW/MHz[a]	0.06 mW/MHz	0.19 mW/MHz (+ cache)	0.5 mW/MHz (+ cache)	0.4 mW/MHz (+ cache)
MIPS[b]/MHz	0.97	1.1	1.3	1.2
Architecture	Von Neumann	Harvard	Harvard	Harvard
Multiplier	8 × 32	8 × 32	16 × 32	16 × 32

[a] Watts/MHz on the same 0.13 micron process.
[b] MIPS are Dhrystone VAX MIPS.

	ARM9E	ARM11	Cortex-R4	Cortex-A8 w/NEON*
Typical clock rate*	265 MHz (130 nm)	335 MHz (130 nm)	375 MHz (90 nm)	450 MHz – 1100 MHz (65 nm)
Instruction sets	ARMv5E, Thumb	ARMv6, Thumb, Thumb2	ARMv7, Thumb, Thumb2	ARMv7, Thumb, Thumb2, NEON
Issue width	Single issue	Single issue	Dual issue (superscalar)	Dual issue (superscalar)
Pipeline stages	5	8	8	13 + 10 (NEON)
DSP/media instructions	Minor	Minor	Minor	Extensive (NEON)
Per-cycle multiply-accumulate throughput (fixed-point)	1 × 32-bit 1 × 16-bit	1 × 32-bit 2 × 16-bit	1 × 32-bit 2 × 16-bit	2 × 32-bit 4 × 16-bit 8 × 8-bit Float: 2 × 32-bit
Data bus	32-bit	64-bit	64-bit	64-/128-bit
Branch prediction	No	Yes	Yes	Yes

Cranes Varsity

Cortex-M3 processor (LPC1768)

The Cortex-M3 processor is a high performance 32-bit processor designed for the micro-controller market. It offers significant benefits to developers, including:

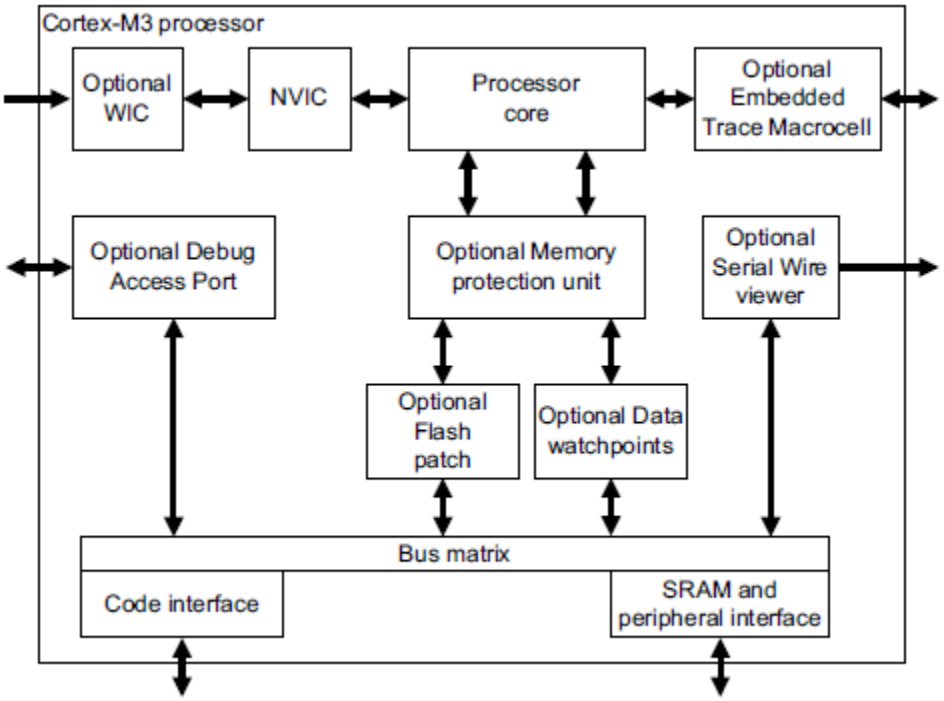
• outstanding processing performance combined with fast interrupt handling

• enhanced system debug with extensive break-point and trace capabilities

• efficient processor core, system and memories

• ultra-low power consumption with integrated sleep mode and an optional deep sleep mode

• platform security robustness, with an optional integrated Memory Protection Unit (MPU).



The Cortex-M3 processor is built on a high-performance processor core, with a 3-stage pipeline Harvard architecture, making it ideal for demanding embedded applications. The processor delivers exceptional power efficiency through an efficient instruction set and extensively optimized design, providing high-end processing hardware including optional IEEE754-compliant single-precision floating-point computation, a range of single-cycle and SIMD multiplication and multiply-with-accumulate capabilities, saturating arithmetic and dedicated hardware division.

To facilitate the design of cost-sensitive devices, the Cortex-M3 processor implements tightly-coupled system components that reduce processor area while significantly improving interrupt handling and system debug capabilities. The Cortex-M3 processor implements a version of the Thumb® instruction set based on Thumb-2 technology, ensuring high code density and reduced program memory

requirements. The Cortex-M3 instruction set provides the exceptional performance expected of a modern 32-bit architecture, with the high code density of 8-bit and 16-bit microcontrollers.

The Cortex-M3 processor closely integrates a configurable NVIC, to deliver industry-leading interrupt performance. The NVIC includes a Non-Mask-able Interrupt (NMI) that can provide up to 256 interrupt priority levels. The tight integration of the processor core and NVIC provides fast execution of Interrupt Service Routines (ISRs), dramatically reducing the interrupt latency.

This is achieved through the hardware stacking of registers, and the ability to suspend

load-multiple and store-multiple operations. Interrupt handlers do not require wrapping in assembler code, removing any code overhead from the ISRs. A tail-chain optimization also significantly reduces the overhead when switching from one ISR to another.

To optimize low-power designs, the NVIC integrates with the sleep modes, which can include an optional deep sleep function. This enables the entire device to be rapidly powered down while still retaining program state.

System-level interface

The Cortex-M3 processor provides multiple interfaces using AMBA® technology to provide high speed, low latency memory accesses. It supports unaligned data accesses and implements atomic bit manipulation that enables faster peripheral controls, system spin locks and thread-safe Boolean data handling. The Cortex-M3 processor has an optional Memory Protection Unit (MPU) that permits control of individual regions in memory, enabling applications to utilize multiple privilege levels, separating and protecting code, data and stack on a task-by-task basis. Such requirements are becoming critical in many embedded applications such as automotive.

Optional integrated configurable debug

The Cortex-M3 processor can implement a complete hardware debug solution. This provides high system visibility of the processor and memory through either a traditional JTAG port or a 2-pin Serial Wire Debug (SWD) port that is ideal for microcontrollers and other small package devices.

For system trace the processor integrates an Instrumentation Trace Macrocell (ITM) alongside data watchpoints and a profiling unit. To enable simple and cost-effective profiling of the system events these generate, a Serial Wire Viewer (SWV) can export a stream of software-generated messages, data trace, and profiling information through a single pin.

The optional Embedded Trace Macrocell™ (ETM) delivers unrivalled instruction trace capture in an area far smaller than traditional trace units, enabling many low cost MCUs to implement full instruction trace for the first time.

The optional Flash Patch and Breakpoint Unit (FPB) provides up to eight hardware breakpoint comparators that debuggers can use. The comparators in the FPB also provide remap functions of up to eight words in the program code in the CODE memory region. This enables applications stored on a non-erasable, ROM-based microcontroller to be patched if a small programmable memory, for example flash, is available in the device. During initialization, the application in ROM detects, from the programmable memory, whether a patch is required. If a patch is required, the application programs the

FPB to remap a number of addresses. When those addresses are accessed, the accesses are redirected to a remap table specified in the FPB configuration, which means the program in the non-modifiable ROM can be patched.

Cortex-M3 processor features and benefits

• tight integration of system peripherals reduces area and development costs

• Thumb instruction set combines high code density with 32-bit performance

• code-patch ability for ROM system updates

• power control optimization of system components

• integrated sleep modes for low power consumption

• fast code execution permits slower processor clock or increases sleep mode time

• hardware division and fast digital-signal-processing orientated multiply accumulate

• deterministic, high-performance interrupt handling for time-critical applications

• optional MPU for safety-critical applications

• extensive implementation-defined debug and trace capabilities:

— Serial Wire Debug and Serial Wire Trace reduce the number of pins required for

debugging, tracing, and code profiling.

Cortex-M3 core peripherals

Nested Vectored Interrupt Controller

The NVIC is an embedded interrupt controller that supports low latency interrupt processing.

System Control Block

The System Control Block (SCB) is the programmer's model interface to the processor. It provides system implementation information and system control, including configuration, control, and reporting of system exceptions.

System timer

The system timer, SysTick, is a 24-bit count-down timer. Use this as a Real Time Operating System (RTOS) tick timer or as a simple counter.

Memory Protection Unit

The MPU improves system reliability by defining the memory attributes for different memory regions. It provides up to eight different regions, and an optional predefined background region.

Programmers model

Cranes Varsity

This section describes the Cortex-M3 programmers model. In addition to the individual core register descriptions, it contains information about the processor modes and privilege levels for software execution and stacks.

Processor mode and privilege levels for software execution

The processor modes are:

Thread mode Used to execute application software. The processor enters Thread mode when it comes out of reset.

Handler mode Used to handle exceptions. The processor returns to Thread mode when it has finished all exception processing.

The privilege levels for software execution are:

Unprivileged The software:

- has limited access to the MSR and MRS instructions, and cannot use the CPS instruction
- cannot access the system timer, NVIC, or system control block
- might have restricted access to memory or peripherals.

Unprivileged software executes at the unprivileged level.

Privileged

The software can use all the instructions and has access to all resources. Privileged software executes at the privileged level.

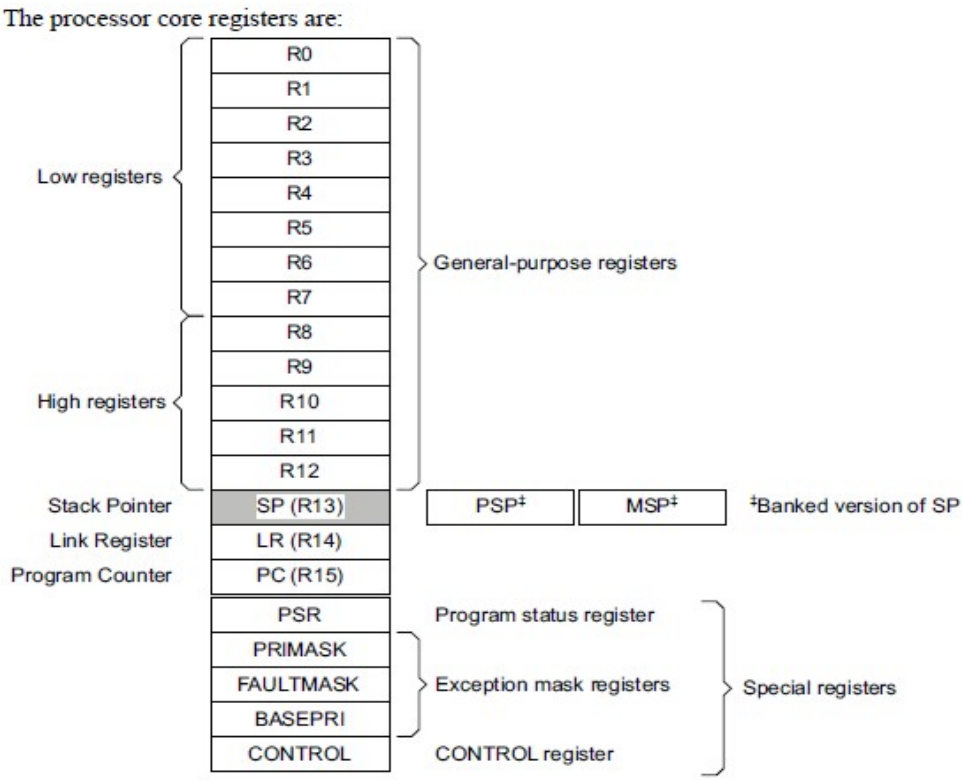
In Thread mode, the CONTROL register controls whether software execution is privileged or unprivileged. In Handler mode, software execution is always privileged.

Only privileged software can write to the CONTROL register to change the privilege level for software execution in Thread mode. Unprivileged software can use the SVC instruction to make a supervisor call to transfer control to privileged software.

Table 2-1 Summary of processor mode, execution privilege level, and stack use options

Processor mode	Used to execute	Privilege level for software execution	Stack used
Thread	Applications	Privileged or unprivileged[a]	Main stack or process stack[a]
Handler	Exception handlers	Always privileged	Main stack

Core registers



Name	Type[a]	Required privilege[b]	Reset value
R0-R12	RW	Either	Unknown
MSP	RW	Privileged	See description
PSP	RW	Either	Unknown
LR	RW	Either	0xFFFFFFFF
PC	RW	Either	See description
PSR	RW	Privileged	0x01000000
ASPR	RW	Either	Unknown
IPSR	RO	Privileged	0x00000000
EPSR	RO	Privileged	0x01000000
PRIMASK	RW	Privileged	0x00000000
FAULTMASK	RW	Privileged	0x00000000
BASEPRI	RW	Privileged	0x00000000
CONTROL	RW	Privileged	0x00000000

General-purpose registers

R0-R12 are 32-bit general-purpose registers for data operations.

Stack Pointer

The Stack Pointer (SP) is register R13. In Thread mode, bit[1] of the CONTROL register indicates the stack pointer to use:

• 0 = Main Stack Pointer (MSP). This is the reset value.

• 1 = Process Stack Pointer (PSP).

On reset, the processor loads the MSP with the value from address 0x00000000.

Link Register

The Link Register (LR) is register R14. It stores the return information for subroutines, function calls, and exceptions. On reset, the processor sets the LR value to 0xFFFFFFFF.

Program Counter

The Program Counter (PC) is register R15. It contains the current program address. On reset, the processor loads the PC with the value of the reset vector, which is at address 0x00000004. Bit[0] of the value is loaded into the EPSR T-bit at reset and must be 1.

Program Status Register

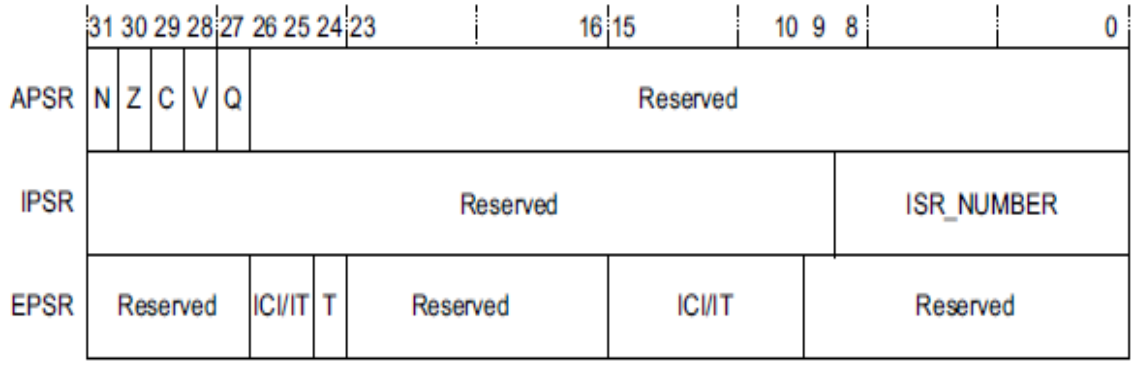
The Program Status Register (PSR) combines:

• Application Program Status Register (APSR)

• Interrupt Program Status Register (IPSR)

• Execution Program Status Register (EPSR).

These registers are mutually exclusive bitfields in the 32-bit PSR. The bit assignments are:



Access these registers individually or as a combination of any two or all three registers, using the register name as an argument to the MSR or MRS instructions. For example:

• read all of the registers using PSR with the MRS instruction

• write to the APSR N, Z, C, V, and Q bits using APSR_nzcvq with the MSR instruction.

Application Program Status Register

The APSR contains the current state of the condition flags from previous instruction executions.

The bit assignments are:

Table 2-4 APSR bit assignments

Bits	Name	Function
[31]	N	Negative flag
[30]	Z	Zero flag
[29]	C	Carry or borrow flag
[28]	V	Overflow flag
[27]	Q	Saturation flag
[26:0]	-	Reserved

Interrupt Program Status Register

The IPSR contains the exception type number of the current Interrupt Service Routine (ISR). The bit assignments are:

Cranes Varsity

Bits	Name	Function
[31:9]	-	Reserved.
[8:0]	ISR_NUMBER	This is the number of the current exception: 0 = Thread mode 1 = Reserved 2 = NMI 3 = HardFault 4 = MemManage 5 = BusFault 6 = UsageFault 7-10 = Reserved 11 = SVCall 12 = Reserved for Debug 13 = Reserved 14 = PendSV 15 = SysTick 16 = IRQ0 . . . n+15 = IRQ(n-1)[a].

Execution Program Status Register

The EPSR contains the Thumb state bit, and the execution state bits for either the:

• If-Then (IT) instruction

• Interruptible-Continuable Instruction (ICI) field for an interrupted load multiple or store multiple instruction.

The bit assignments are:

Bits	Name	Function
[31:27]	-	Reserved.
[26:25], [15:10]	ICI/IT	Indicates the interrupted position of a continuable instruction, see *Interruptible-continuable instructions* on page 2-7, or the execution state of an IT instruction, see *IT* on page 3-64.
[24]	T	Thumb state bit, see *Thumb state*.
[23:16]	-	Reserved.
[9:0]	-	Reserved.

Attempts to read the EPSR directly through application software using the MSR instruction always return zero. Attempts to write the EPSR using the MSR instruction in application software are ignored.

Interruptible-Continuable instructions

When an interrupt occurs during the execution of an LDM, STM, PUSH, or POP instruction, the processor:

• stops the load multiple or store multiple instruction operation temporarily

• stores the next register operand in the multiple operation to EPSR bits[15:12].

After servicing the interrupt, the processor:

• returns to the register pointed to by bits[15:12]

• resumes execution of the multiple load or store instruction.

When the EPSR holds ICI execution state, bits[26:25,11:10] are zero.

If-Then block

The If-Then block contains up to four instructions following an IT instruction. Each instruction in the block is conditional. The conditions for the instructions are either all the same, or some can be the inverse of others. See IT on page 3-64 for more information.

Thumb state

The Cortex-M3 processor only supports execution of instructions in Thumb state. The following can clear the T bit to 0:

• instructions BLX, BX and POP{PC}

• restoration from the stacked xPSR value on an exception return

• bit[0] of the vector value on an exception entry or reset.

Attempting to execute instructions when the T bit is 0 results in a fault or lockup.

Exception mask registers

The exception mask registers disable the handling of exceptions by the processor. Disable exceptions where they might impact on timing critical tasks.

To access the exception mask registers use the MSR and MRS instructions, or the CPS instruction to change the value of PRIMASK or FAULTMASK.

Priority Mask Register

The PRIMASK register prevents activation of all exceptions with configurable priority. The bit assignments are:

Cranes Varsity

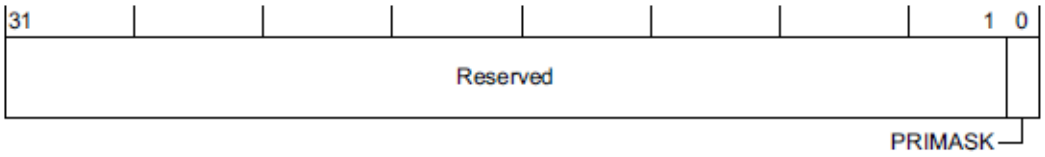


Table 2-7 PRIMASK register bit assignments

Bits	Name	Function
[31:1]	-	Reserved
[0]	PRIMASK	0 = no effect 1 = prevents the activation of all exceptions with configurable priority.

Fault Mask Register

The FAULTMASK register prevents activation of all exceptions except for Non-Maskable Interrupt (NMI). The bit assignments are

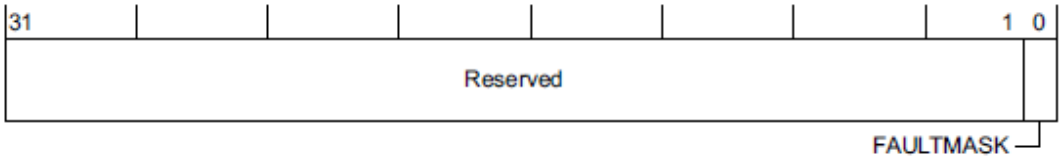


Table 2-8 FAULTMASK register bit assignments

Bits	Name	Function
[31:1]	-	Reserved
[0]	FAULTMASK	0 = no effect 1 = prevents the activation of all exceptions except for NMI.

The processor clears the FAULTMASK bit to 0 on exit from any exception handler except the NMI handler.

Base Priority Mask Register

The BASEPRI register defines the minimum priority for exception processing. When BASEPRI is set to a nonzero value, it prevents the activation of all exceptions with the same or lower priority level as the BASEPRI value. The bit assignments are:

Table 2-9 BASEPRI register bit assignments

Bits	Name	Function
[31:8]	-	Reserved
[7:0]	BASEPRI[a]	Priority mask bits: 0x00 = no effect Nonzero = defines the base priority for exception processing. The processor does not process any exception with a priority value greater than or equal to BASEPRI.

CONTROL register

The CONTROL register controls the stack used and the privilege level for software execution when the processor is in Thread mode. The bit assignments are:

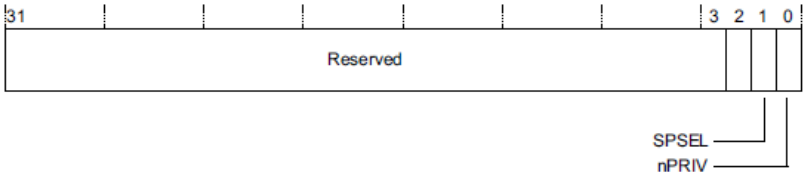


Table 2-10 CONTROL register bit assignments

Bits	Name	Function
[31:2]	-	Reserved.
[1]	SPSEL	Defines the currently active stack pointer: In Handler mode this bit reads as zero and ignores writes. The Cortex-M3 updates this bit automatically on exception return. 0 = MSP is the current stack pointer 1 = PSP is the current stack pointer.
[0]	nPRIV	Defines the Thread mode privilege level: 0 = Privileged 1 = Unprivileged.

Handler mode always uses the MSP, so the processor ignores explicit writes to the active stack pointer bit of the CONTROL register when in Handler mode. The exception entry and return mechanisms automatically update the CONTROL register based on the EXC_RETURN value.

In an OS environment, ARM recommends that threads running in Thread mode use the process stack and the kernel and exception handlers use the main stack.

By default, Thread mode uses the MSP. To switch the stack pointer used in Thread mode to the PSP, either:

• use the MSR instruction to set the Active stack pointer bit to 1.

• perform an exception return to Thread mode with the appropriate EXC_RETURN value.

Note

When changing the stack pointer, software must use an ISB instruction immediately after the MSR instruction. This ensures that instructions after the ISB instruction execute using the new stack pointer.

The Cortex Microcontroller Software Interface Standard

For a Cortex-M3 microcontroller system, the Cortex Microcontroller Software Interface Standard (CMSIS) defines:

• a common way to:

— access peripheral registers

— define exception vectors

• the names of:

— the registers of the core peripherals

— the core exception vectors

• a device-independent interface for RTOS kernels, including a debug channel.

The CMSIS includes address definitions and data structures for the core peripherals in the Cortex-M3 processor.

CMSIS simplifies software development by enabling the reuse of template code and the combination of CMSIS-compliant software components from various middleware vendors. Software vendors can expand the CMSIS to include their peripheral definitions and access functions for those peripherals.

Cranes Varsity

CHAPTER-17

Introduction to LPC1768

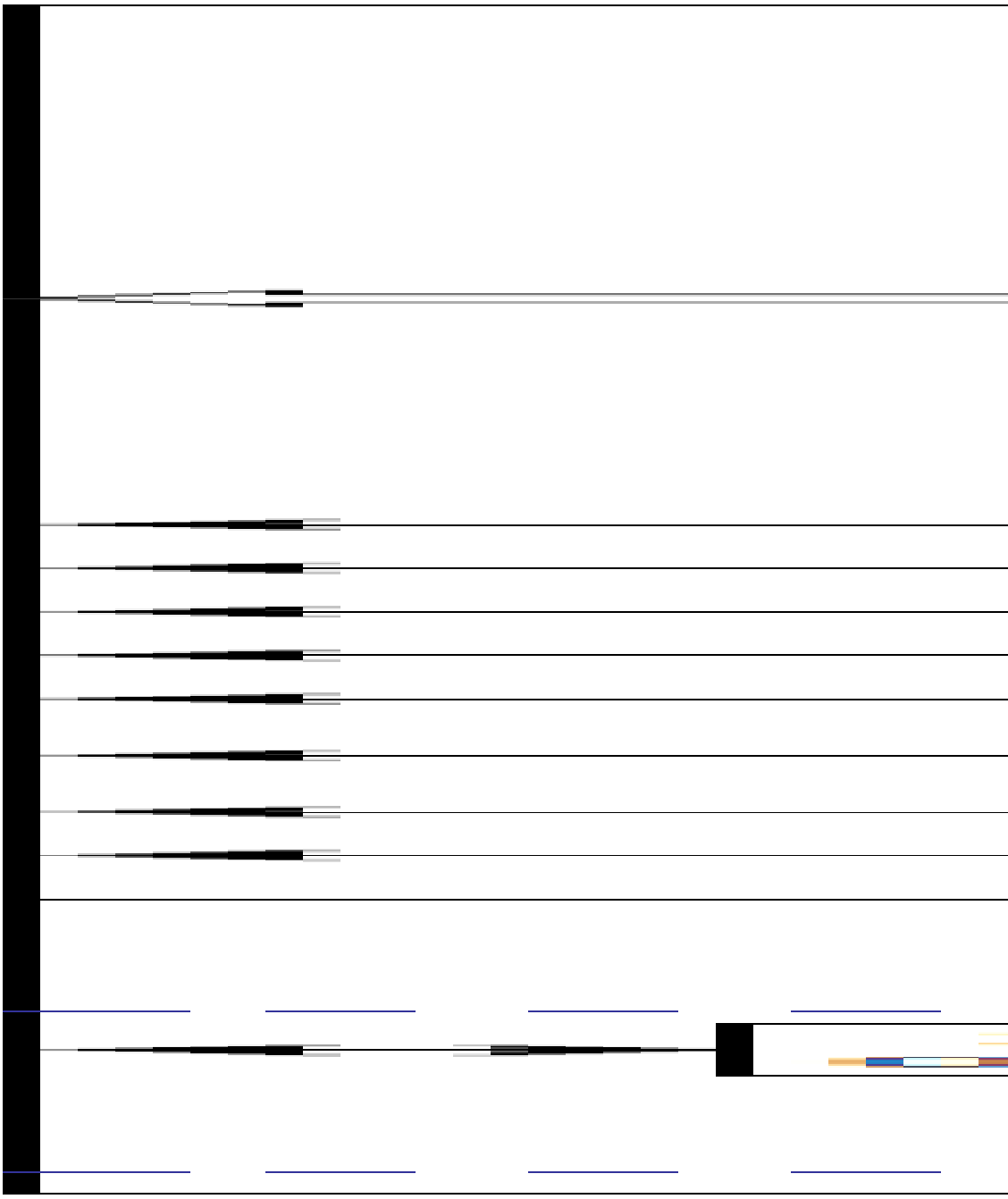
Features

- Arm® Cortex-M3 processor, running at frequencies of up to 100 MHz
- Arm Cortex-M3 built-in Nested Vectored Interrupt Controller (NVIC)
- Up to 512 kB on-chip flash programming memory
- Up to 64 kB On-chip SRAM
- In-System Programming (ISP) and In-Application Programming (IAP)
- Eight channel General Purpose DMA controller (GPDMA)
- Ethernet MAC with RMII interface and dedicated DMA controller
- USB 2.0 full-speed device/Host/OTG controller
- Four UARTs with fractional baud rate generation, internal FIFO, and DMA support
- CAN 2.0B controller with two channels
- SPI controller with synchronous, serial, full duplex communication
- Two SSP controllers with FIFO and multi-protocol capabilities
- Three enhanced I2C bus interfaces
- I2S (Inter-IC Sound) interface
- 70 General Purpose I/O (GPIO) pins with configurable pull-up/down resistors
- 12-bit/8-ch Analog/Digital Converter (ADC) with conversion rates up to 200 kHz
- 10-bit Digital/Analog Converter (DAC) with dedicated conversion timer and DMA
- Four general purpose timers/counters
- One motor control PWM with support for three-phase motor control
- Quadrature encoder interface that can monitor one external quadrature encoder
- One standard PWM/timer block with external count input
- Low power RTC with a separate power domain and dedicated oscillator
- WatchDog Timer (WDT)

Cranes Varsity

- Arm Cortex-M3 system tick timer, including an external clock input option
- Repetitive interrupt timer provides programmable and repeating timed interrupts
- Each peripheral has its own clock divider for further power savings
- Standard JTAG test/debug interface for compatibility with existing tools
- Integrated PMU (Power Management Unit)
- Four reduced power modes: Sleep, Deep-sleep, Power-down, and Deep power-down
- Single 3.3 V power supply (2.4 V to 3.6 V)
- Four external interrupt inputs configurable as edge/level sensitive
- Non-maskable Interrupt (NMI) input
- Wake-up Interrupt Controller (WIC)
- Processor wake-up from Power-down mode via any interrupt
- Brownout detect with separate threshold for interrupt and forced reset
- Power-On Reset (POR)
- Crystal oscillator with an operating range of 1 MHz to 25 MHz
- 4 MHz internal RC oscillator trimmed to 1 % accuracy
- Code Read Protection (CRP) with different security levels
- Unique device serial number for identification purposes

Cranes Varsity



NXP LPC176X

Pin connect block

Description

The pin connect block allows most pins of the microcontroller to have more than one

potential function. Configuration registers control the multiplexers to allow connection between the pin and the on chip peripherals. Peripherals should be connected to the appropriate pins prior to being activated and prior to any related interrupt(s) being enabled. Activity of any enabled peripheral function that is not mapped to a related pin should be considered undefined. Selection of a single function on a port pin excludes other peripheral functions available on the same pin. However, the GPIO input stays connected and may be read by software or used to contribute to the GPIO interrupt feature.

Pin mode select register values

The PINMODE registers control the input mode of all ports. This includes the use of the on-chip pull-up/pull-down resistor feature and a special open drain operating mode. The on-chip pull-up/pull-down resistor can be selected for every port pin regardless of the function on this pin with the exception of the I2C pins for the I2C0 interface and the USB pins. Three bits are used to control the mode of a port pin, two in a PINMODE register, and an additional one in a PINMODE_OD register. Bits are reserved for unused pins as in the PINSEL registers.

PINMODE0 to PINMODE9 Values	Function	Value after Reset
00	Pin has an on-chip pull-up resistor enabled.	00
01	Repeater mode (see text below).	
10	Pin has neither pull-up nor pull-down resistor enabled.	
11	Pin has an on-chip pull-down resistor enabled.	

Repeater mode enables the pull-up resistor if the pin is at a logic high and enables the pull-down resistor if the pin is at a logic low. This causes the pin to retain its last known state if it is configured as an input and is not driven externally. The state retention is not applicable to the Deep Power-down mode. Repeater mode may typically be used to prevent a pin from floating (and potentially using significant power if it floats to an indeterminate state) if it is temporarily not driven.

The PINMODE_OD registers control the open drain mode for ports. The open drain mode causes the pin to be pulled low normally if it is configured as an output and the data value is 0. If the data value is 1, the output drive of the pin is turned off, equivalent to changing the pin direction. This combination simulates an open drain output.

PINMODE_OD0 to PINMODE_OD4 Values	Function	Value after Reset
0	Pin is in the normal (not open drain) mode.	00
1	Pin is in the open drain mode.	

Function of PINMODE in open drain mode

Normally the value of PINMODE applies to a pin only when it is in the input mode. When a pin is in the open drain mode, caused by a 1 in the corresponding bit of one of the PINMODE_OD registers, the input mode still does not apply when the pin is outputting a 0. However, when the pin value is 1, PINMODE applies since this state turns off the pin's output driver. For example, this allows for the possibility of configuring a pin to be open drain with an on-chip pullup. A pullup in this case which is only on when the pin is not being pulled low by the pin's own output.

Register description

Name	Description	Access	Reset Value[1]	Address
PINSEL0	Pin function select register 0.	R/W	0	0x4002 C000
PINSEL1	Pin function select register 1.	R/W	0	0x4002 C004
PINSEL2	Pin function select register 2.	R/W	0	0x4002 C008
PINSEL3	Pin function select register 3.	R/W	0	0x4002 C00C
PINSEL4	Pin function select register 4	R/W	0	0x4002 C010
PINSEL7	Pin function select register 7	R/W	0	0x4002 C01C
PINSEL8	Pin function select register 8	R/W	0	0x4002 C020
PINSEL9	Pin function select register 9	R/W	0	0x4002 C024
PINSEL10	Pin function select register 10	R/W	0	0x4002 C028
PINMODE0	Pin mode select register 0	R/W	0	0x4002 C040
PINMODE1	Pin mode select register 1	R/W	0	0x4002 C044
PINMODE2	Pin mode select register 2	R/W	0	0x4002 C048
PINMODE3	Pin mode select register 3.	R/W	0	0x4002 C04C
PINMODE4	Pin mode select register 4	R/W	0	0x4002 C050
PINMODE5	Pin mode select register 5	R/W	0	0x4002 C054
PINMODE6	Pin mode select register 6	R/W	0	0x4002 C058
PINMODE7	Pin mode select register 7	R/W	0	0x4002 C05C
PINMODE9	Pin mode select register 9	R/W	0	0x4002 C064
PINMODE_OD0	Open drain mode control register 0	R/W	0	0x4002 C068
PINMODE_OD1	Open drain mode control register 1	R/W	0	0x4002 C06C
PINMODE_OD2	Open drain mode control register 2	R/W	0	0x4002 C070
PINMODE_OD3	Open drain mode control register 3	R/W	0	0x4002 C074
PINMODE_OD4	Open drain mode control register 4	R/W	0	0x4002 C078
I2CPADCFG	I^2C Pin Configuration register	R/W	0	0x4002 C07C

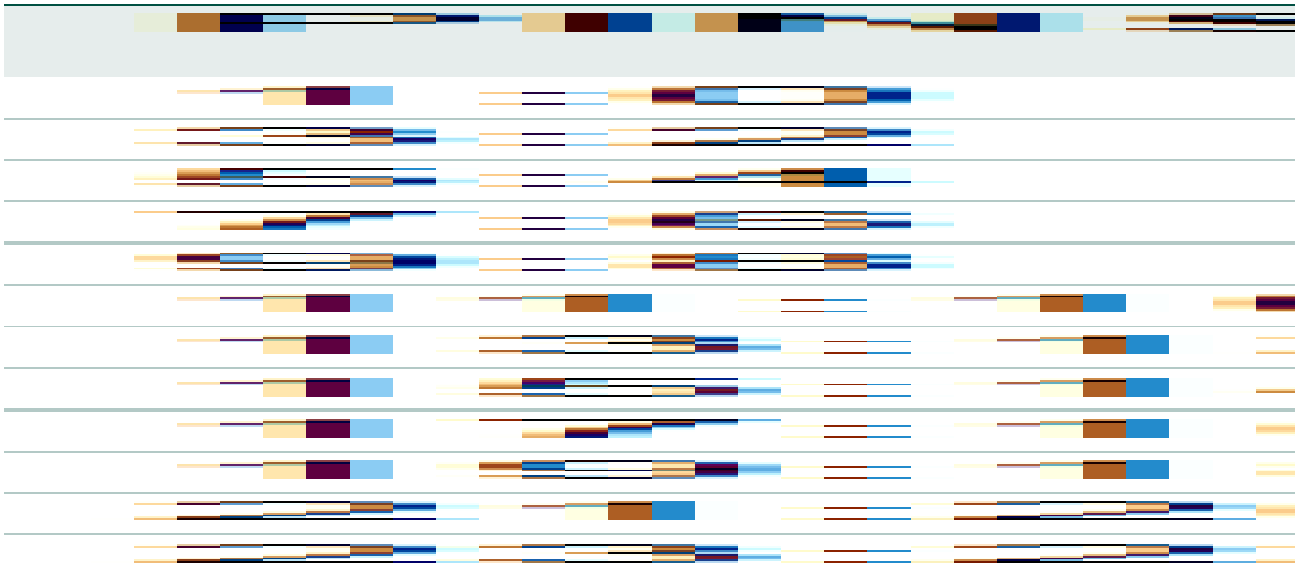
Pin Function Select register 0 (PINSEL0 - 0x4002 C000)

The PINSEL0 register controls the functions of the lower half of Port 0. The direction control bit in FIO0DIR register is effective only when the GPIO function is selected for a pin. For other functions, the direction is controlled automatically.

PINSEL0	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
1:0	P0.0	GPIO Port 0.0	RD1	TXD3	SDA1	00
3:2	P0.1	GPIO Port 0.1	TD1	RXD3	SCL1	00
5:4	P0.2	GPIO Port 0.2	TXD0	AD0.7	Reserved	00
7:6	P0.3	GPIO Port 0.3	RXD0	AD0.6	Reserved	00
9:8	P0.4[1]	GPIO Port 0.4	I2SRX_CLK	RD2	CAP2.0	00
11:10	P0.5[1]	GPIO Port 0.5	I2SRX_WS	TD2	CAP2.1	00
13:12	P0.6	GPIO Port 0.6	I2SRX_SDA	SSEL1	MAT2.0	00
15:14	P0.7	GPIO Port 0.7	I2STX_CLK	SCK1	MAT2.1	00
17:16	P0.8	GPIO Port 0.8	I2STX_WS	MISO1	MAT2.2	00
19:18	P0.9	GPIO Port 0.9	I2STX_SDA	MOSI1	MAT2.3	00
21:20	P0.10	GPIO Port 0.10	TXD2	SDA2	MAT3.0	00
23:22	P0.11	GPIO Port 0.11	RXD2	SCL2	MAT3.1	00
29:24	-	Reserved	Reserved	Reserved	Reserved	0
31:30	P0.15	GPIO Port 0.15	TXD1	SCK0	SCK	00

Pin Function Select Register 1 (PINSEL1 - 0x4002 C004)

The PINSEL1 register controls the functions of the upper half of Port 0. The direction control bit in the FIO0DIR register is effective only when the GPIO function is selected for a pin. For other functions the direction is controlled automatically.



25:24	P0.28[1][2]	GPIO Port 0.28	SCL0	USB_SCL	Reserved	00
27:26	P0.29	GPIO Port 0.29	USB_D+	Reserved	Reserved	00
29:28	P0.30	GPIO Port 0.30	USB_D–	Reserved	Reserved	00
31:30	-	Reserved	Reserved	Reserved	Reserved	00

Pin Function Select register 2 (PINSEL2 - 0x4002 C008)

The PINSEL2 register controls the functions of the lower half of Port 1, which contains the Ethernet related pins. The direction control bit in the FIO1DIR register is effective only when the GPIO function is selected for a pin. For other functions, the direction is controlled automatically.

PINSEL2	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
1:0	P1.0	GPIO Port 1.0	ENET_TXD0	Reserved	Reserved	00
3:2	P1.1	GPIO Port 1.1	ENET_TXD1	Reserved	Reserved	00
7:4	-	Reserved	Reserved	Reserved	Reserved	0
9:8	P1.4	GPIO Port 1.4	ENET_TX_EN	Reserved	Reserved	00
15:10	-	Reserved	Reserved	Reserved	Reserved	0
17:16	P1.8	GPIO Port 1.8	ENET_CRS	Reserved	Reserved	00
19:18	P1.9	GPIO Port 1.9	ENET_RXD0	Reserved	Reserved	00
21:20	P1.10	GPIO Port 1.10	ENET_RXD1	Reserved	Reserved	00
27:22	-	Reserved	Reserved	Reserved	Reserved	0
29:28	P1.14	GPIO Port 1.14	ENET_RX_ER	Reserved	Reserved	00
31:30	P1.15	GPIO Port 1.15	ENET_REF_CLK	Reserved	Reserved	00

Pin Function Select Register 3 (PINSEL3 - 0x4002 C00C)

The PINSEL3 register controls the functions of the upper half of Port 1. The direction control bit in the FIO1DIR register is effective only when the GPIO function is selected for a pin. For other functions, direction is controlled automatically.

PINSEL3	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
1:0	P1.16[1]	GPIO Port 1.16	ENET_MDC	Reserved	Reserved	00
3:2	P1.17[1]	GPIO Port 1.17	ENET_MDIO	Reserved	Reserved	00
5:4	P1.18	GPIO Port 1.18	USB_UP_LED	PWM1.1	CAP1.0	00
7:6	P1.19	GPIO Port 1.19	MCOA0	$\overline{\text{USB\_PPWR}}$	CAP1.1	00
9:8	P1.20	GPIO Port 1.20	MCI0	PWM1.2	SCK0	00

11:10	P1.21[1]	GPIO Port 1.21	$\overline{\text{MCABORT}}$	PWM1.3	SSEL0	00
13:12	P1.22	GPIO Port 1.22	MCOB0	USB_PWRD	MAT1.0	00
15:14	P1.23	GPIO Port 1.23	MCI1	PWM1.4	MISO0	00
17:16	P1.24	GPIO Port 1.24	MCI2	PWM1.5	MOSI0	00
19:18	P1.25	GPIO Port 1.25	MCOA1	Reserved	MAT1.1	00
21:20	P1.26	GPIO Port 1.26	MCOB1	PWM1.6	CAP0.0	00
23:22	P1.27[1]	GPIO Port 1.27	CLKOUT	$\overline{\text{USB\_OVRCR}}$	CAP0.1	00
25:24	P1.28	GPIO Port 1.28	MCOA2	PCAP1.0	MAT0.0	00
27:26	P1.29	GPIO Port 1.29	MCOB2	PCAP1.1	MAT0.1	00
29:28	P1.30	GPIO Port 1.30	Reserved	V_{BUS}	AD0.4	00
31:30	P1.31	GPIO Port 1.31	Reserved	SCK1	AD0.5	00

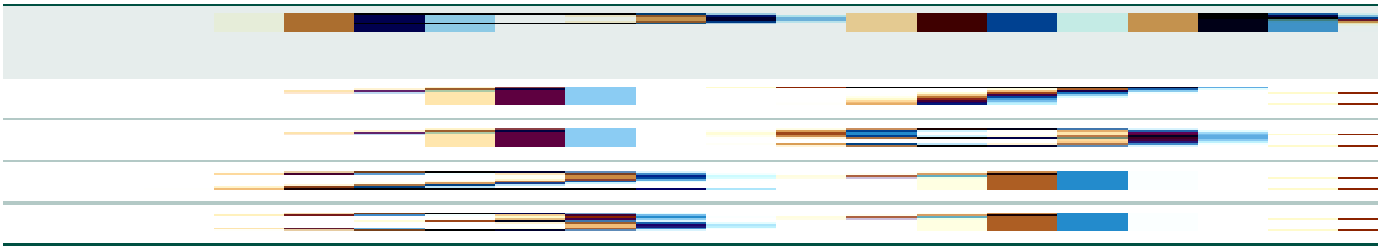
Pin Function Select Register 4 (PINSEL4 - 0x4002 C010)

The PINSEL4 register controls the functions of the lower half of Port 2. The direction control bit in the FIO2DIR register is effective only when the GPIO function is selected for a pin. For other functions, direction is controlled automatically.

PINSEL4	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
1:0	P2.0	GPIO Port 2.0	PWM1.1	TXD1	Reserved	00
3:2	P2.1	GPIO Port 2.1	PWM1.2	RXD1	Reserved	00
5:4	P2.2	GPIO Port 2.2	PWM1.3	CTS1	Reserved [2]	00
7:6	P2.3	GPIO Port 2.3	PWM1.4	DCD1	Reserved [2]	00
9:8	P2.4	GPIO Port 2.4	PWM1.5	DSR1	Reserved [2]	00
11:10	P2.5	GPIO Port 2.5	PWM1.6	DTR1	Reserved [2]	00
13:12	P2.6	GPIO Port 2.6	PCAP1.0	RI1	Reserved [2]	00
15:14	P2.7	GPIO Port 2.7	RD2	RTS1	Reserved	00
17:16	P2.8	GPIO Port 2.8	TD2	TXD2	ENET_MDC	00
19:18	P2.9	GPIO Port 2.9	USB_CONNECT	RXD2	ENET_MDIO	00
21:20	P2.10	GPIO Port 2.10	$\overline{\text{EINT0}}$	NMI	Reserved	00
23:22	P2.11[1]	GPIO Port 2.11	$\overline{\text{EINT1}}$	Reserved	I2STX_CLK	00
25:24	P2.12[1]	GPIO Port 2.12	$\overline{\text{EINT2}}$	Reserved	I2STX_WS	00
27:26	P2.13[1]	GPIO Port 2.13	$\overline{\text{EINT3}}$	Reserved	I2STX_SDA	00
31:28	-	Reserved	Reserved	Reserved	Reserved	0

Pin Function Select Register 7 (PINSEL7 - 0x4002 C01C)

The PINSEL7 register controls the functions of the upper half of Port 3. The direction control bit in the FIO3DIR register is effective only when the GPIO function is selected for a pin. For other functions, direction is controlled automatically.



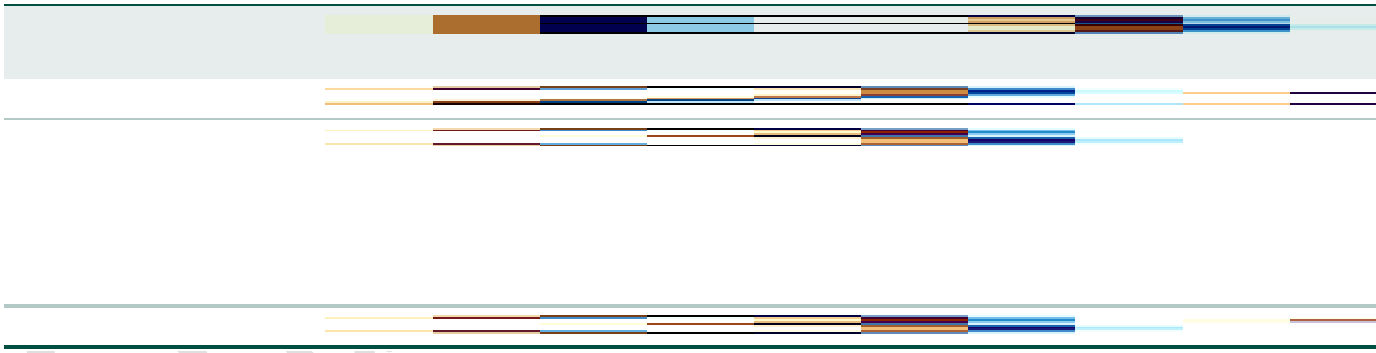
Pin Function Select Register 9 (PINSEL9 - 0x4002 C024)

The PINSEL9 register controls the functions of the upper half of Port 4. The direction control bit in the FIO4DIR register is effective only when the GPIO function is selected for a pin. For other functions, direction is controlled automatically.

PINSEL9	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
23:0	-	Reserved	Reserved	Reserved	Reserved	00
25:24	P4.28	GPIO Port 4.28	RX_MCLK	MAT2.0	TXD3	00
27:26	P4.29	GPIO Port 4.29	TX_MCLK	MAT2.1	RXD3	00
31:28	-	Reserved	Reserved	Reserved	Reserved	00

Pin Function Select Register 10 (PINSEL10 - 0x4002 C028)

Only bit 3 of this register is used to control the Trace function on pins P2.2 through P2.6.



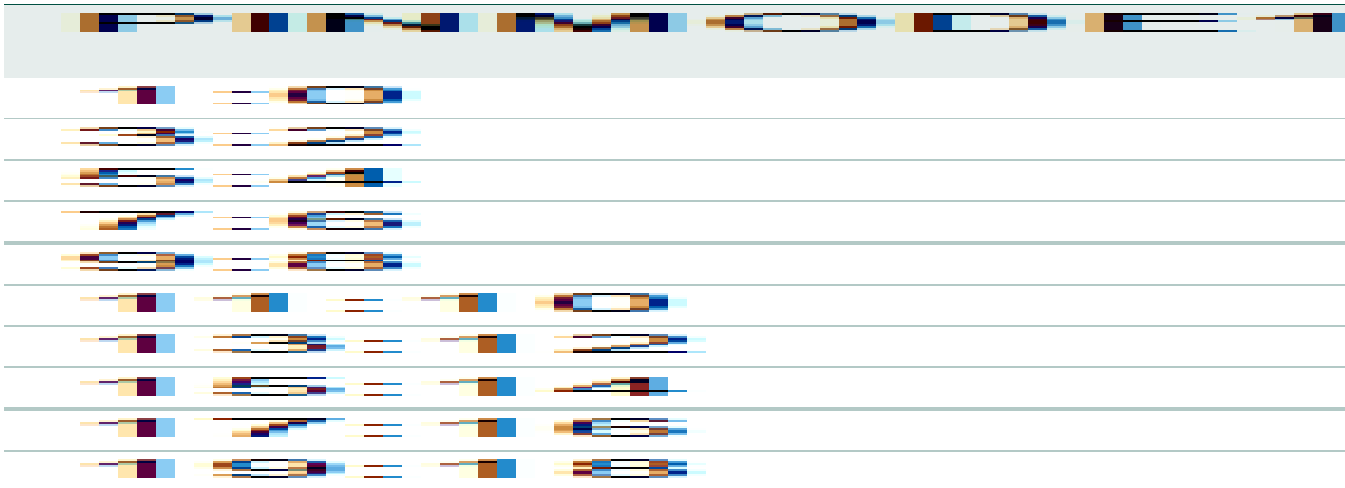
Pin Mode select register 0 (PINMODE0 - 0x4002 C040)

This register controls pull-up/pull-down resistor configuration for Port 0 pins 0 to 15.

PINMODE0	Symbol	Value	Description	Reset value
1:0	P0.00MODE		Port 0 pin 0 on-chip pull-up/down resistor control.	00
		00	P0.0 pin has a pull-up resistor enabled.	
		01	P0.0 pin has repeater mode enabled.	
		10	P0.0 pin has neither pull-up nor pull-down.	
		11	P0.0 has a pull-down resistor enabled.	
3:2	P0.01MODE		Port 0 pin 1 control, see P0.00MODE.	00
5:4	P0.02MODE		Port 0 pin 2 control, see P0.00MODE.	00
7:6	P0.03MODE		Port 0 pin 3 control, see P0.00MODE.	00
9:8	P0.04MODE[1]		Port 0 pin 4 control, see P0.00MODE.	00
11:10	P0.05MODE[1]		Port 0 pin 5 control, see P0.00MODE.	00
13:12	P0.06MODE		Port 0 pin 6 control, see P0.00MODE.	00
15:14	P0.07MODE		Port 0 pin 7 control, see P0.00MODE.	00
17:16	P0.08MODE		Port 0 pin 8 control, see P0.00MODE.	00
19:18	P0.09MODE		Port 0 pin 9control, see P0.00MODE.	00
21:20	P0.10MODE		Port 0 pin 10 control, see P0.00MODE.	00
23:22	P0.11MODE		Port 0 pin 11 control, see P0.00MODE.	00
29:24	-		Reserved.	NA
31:30	P0.15MODE		Port 0 pin 15 control, see P0.00MODE.	00

Pin Mode select register 1 (PINMODE1 - 0x4002 C044)

This register controls pull-up/pull-down resistor configuration for Port 1 pins 16 to 26.



21:20	P0.26MODE	Port 1 pin 26 control, see P0.00MODE.	00
29:22	-	Reserved. [2]	NA
31:30	-	Reserved.	NA

Pin Mode select register 2 (PINMODE2 - 0x4002 C048)

This register controls pull-up/pull-down resistor configuration for Port 1 pins 0 to 15.

PINMODE2	Symbol	Description	Reset value
1:0	P1.00MODE	Port 1 pin 0 control, see P0.00MODE.	00
3:2	P1.01MODE	Port 1 pin 1 control, see P0.00MODE.	00
7:4	-	Reserved.	NA
9:8	P1.04MODE	Port 1 pin 4 control, see P0.00MODE.	00
15:10	-	Reserved.	NA
17:16	P1.08MODE	Port 1 pin 8 control, see P0.00MODE.	00
19:18	P1.09MODE	Port 1 pin 9 control, see P0.00MODE.	00
21:20	P1.10MODE	Port 1 pin 10 control, see P0.00MODE.	00
27:22	-	Reserved.	NA
29:28	P1.14MODE	Port 1 pin 14 control, see P0.00MODE.	00
31:30	P1.15MODE	Port 1 pin 15 control, see P0.00MODE.	00

Pin Mode select register 3 (PINMODE3 - 0x4002 C04C)

This register controls pull-up/pull-down resistor configuration for Port 1 pins 16 to 31.

PINMODE3	Symbol	Description	Reset value
1:0	P1.16MODE[1]	Port 1 pin 16 control, see P0.00MODE.	00
3:2	P1.17MODE[1]	Port 1 pin 17 control, see P0.00MODE.	00
5:4	P1.18MODE	Port 1 pin 18 control, see P0.00MODE.	00
7:6	P1.19MODE	Port 1 pin 19 control, see P0.00MODE.	00
9:8	P1.20MODE	Port 1 pin 20 control, see P0.00MODE.	00
11:10	P1.21MODE[1]	Port 1 pin 21 control, see P0.00MODE.	00
13:12	P1.22MODE	Port 1 pin 22 control, see P0.00MODE.	00

15:14	P1.23MODE	Port 1 pin 23 control, see P0.00MODE.	00
17:16	P1.24MODE	Port 1 pin 24 control, see P0.00MODE.	00
19:18	P1.25MODE	Port 1 pin 25 control, see P0.00MODE.	00
21:20	P1.26MODE	Port 1 pin 26 control, see P0.00MODE.	00
23:22	P1.27MODE[1]	Port 1 pin 27 control, see P0.00MODE.	00
25:24	P1.28MODE	Port 1 pin 28 control, see P0.00MODE.	00
27:26	P1.29MODE	Port 1 pin 29 control, see P0.00MODE.	00
29:28	P1.30MODE	Port 1 pin 30 control, see P0.00MODE.	00
31:30	P1.31MODE	Port 1 pin 31 control, see P0.00MODE.	00

Pin Mode select register 4 (PINMODE4 - 0x4002 C050)

This register controls pull-up/pull-down resistor configuration for Port 2 pins 0 to 15.

PINMODE4	Symbol	Description	Reset value
1:0	P2.00MODE	Port 2 pin 0 control, see P0.00MODE.	00
3:2	P2.01MODE	Port 2 pin 1 control, see P0.00MODE.	00
5:4	P2.02MODE	Port 2 pin 2 control, see P0.00MODE.	00
7:6	P2.03MODE	Port 2 pin 3 control, see P0.00MODE.	00
9:8	P2.04MODE	Port 2 pin 4 control, see P0.00MODE.	00
11:10	P2.05MODE	Port 2 pin 5 control, see P0.00MODE.	00
13:12	P2.06MODE	Port 2 pin 6 control, see P0.00MODE.	00
15:14	P2.07MODE	Port 2 pin 7 control, see P0.00MODE.	00
17:16	P2.08MODE	Port 2 pin 8 control, see P0.00MODE.	00
19:18	P2.09MODE	Port 2 pin 9 control, see P0.00MODE.	00
21:20	P2.10MODE	Port 2 pin 10 control, see P0.00MODE.	00
23:22	P2.11MODE[1]	Port 2 pin 11 control, see P0.00MODE.	00
25:24	P2.12MODE[1]	Port 2 pin 12 control, see P0.00MODE.	00
27:26	P2.13MODE[1]	Port 2 pin 13 control, see P0.00MODE.	00
31:28	-	Reserved.	NA

Pin Mode select register 7 (PINMODE7 - 0x4002 C05C)

This register controls pull-up/pull-down resistor configuration for Port 3 pins 16 to 31.

PINMODE7	Symbol	Description	Reset value
17:0	-	Reserved	NA
19:18	P3.25MODE[1]	Port 3 pin 25 control, see P0.00MODE.	00
21:20	P3.26MODE[1]	Port 3 pin 26 control, see P0.00MODE.	00
31:22	-	Reserved.	NA

Pin Mode select register 9 (PINMODE9 - 0x4002 C064)

This register controls pull-up/pull-down resistor configuration for Port 4 pins 16 to 31.

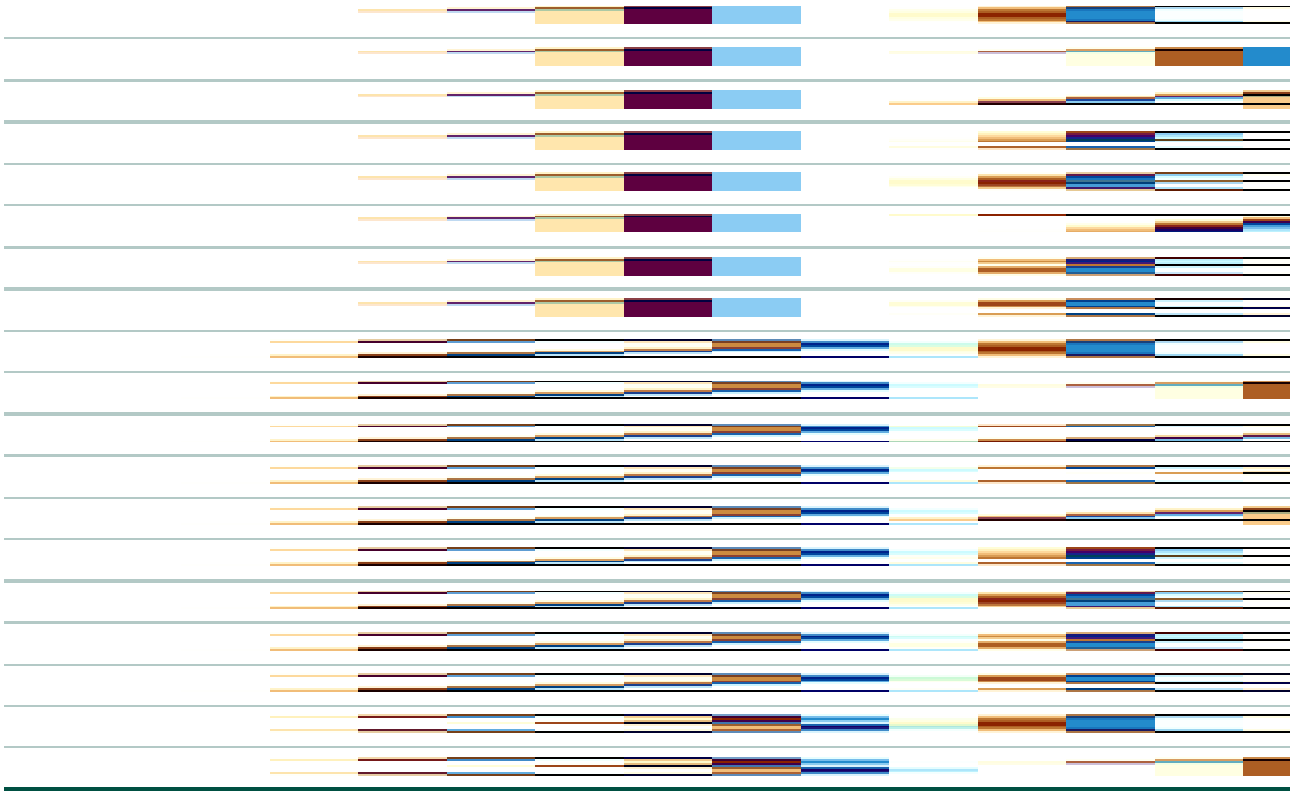
PINMODE9	Symbol	Description	Reset value
23:0	-	Reserved.	NA
25:24	P4.28MODE	Port 4 pin 28 control, see P0.00MODE.	00
27:26	P4.29MODE	Port 4 pin 29 control, see P0.00MODE.	00
31:28	-	Reserved.	NA

Open Drain Pin Mode select register 0 (PINMODE_OD0 - 0x4002 C068)

This register controls the open drain mode for Port 0 pins.

PINMODE_OD0	Symbol	Value	Description	Reset value
0	P0.00OD[3]		Port 0 pin 0 open drain mode control.	0
		0	P0.0 pin is in the normal (not open drain) mode.	
		1	P0.0 pin is in the open drain mode.	
1	P0.01OD[3]		Port 0 pin 1 open drain mode control, see P0.00OD	0
2	P0.02OD		Port 0 pin 2 open drain mode control, see P0.00OD	0
3	P0.03OD		Port 0 pin 3 open drain mode control, see P0.00OD	0
4	P0.04OD		Port 0 pin 4 open drain mode control, see P0.00OD	0
5	P0.05OD		Port 0 pin 5 open drain mode control, see P0.00OD	0
6	P0.06OD		Port 0 pin 6 open drain mode control, see P0.00OD	0
7	P0.07OD		Port 0 pin 7 open drain mode control, see P0.00OD	0
8	P0.08OD		Port 0 pin 8 open drain mode control, see P0.00OD	0
9	P0.09OD		Port 0 pin 9 open drain mode control, see P0.00OD	0

Cranes Varsity



Not available on 80-pin package.

[2] Port 0 pins 27 and 28 should be set up using the I2CPADCFG register if they are used for an I2C-bus. Bits 27 and 28 of PINMODE_OD0 do not have any affect on these pins, they are special open drain I2C-bus compatible pins.

[3] Port 0 bits 1:0, 11:10, and 20:19 may potentially be used for I2C-buses using standard port pins. If so, they should be configured for open drain mode via the related bits in PINMODE_OD0.

Open Drain Pin Mode select register 1 (PINMODE_OD1 - 0x4002 C06C)

This register controls the open drain mode for Port 1 pins.

PINMODE_OD1	Symbol	Value	Description	Reset value
0	P1.00OD		Port 1 pin 0 open drain mode control.	0
		0	P1.0 pin is in the normal (not open drain) mode.	
		1	P1.0 pin is in the open drain mode.	
1	P1.01OD		Port 1 pin 1 open drain mode control, see P1.00OD	0
3:2	-		Reserved.	NA
4	P1.04OD		Port 1 pin 4 open drain mode control, see P1.00OD	0

7:5	-	Reserved.	NA
8	P1.08OD	Port 1 pin 8 open drain mode control, see P1.00OD	0
9	P1.09OD	Port 1 pin 9 open drain mode control, see P1.00OD	0
10	P1.10OD	Port 1 pin 10 open drain mode control, see P1.00OD	0
13:11	-	Reserved.	NA
14	P1.14OD	Port 1 pin 14 open drain mode control, see P1.00OD	0
15	P1.15OD	Port 1 pin 15 open drain mode control, see P1.00OD	0
16	P1.16OD[1]	Port 1 pin 16 open drain mode control, see P1.00OD	0
17	P1.17OD[1]	Port 1 pin 17 open drain mode control, see P1.00OD	0
18	P1.18OD	Port 1 pin 18 open drain mode control, see P1.00OD	0
19	P1.19OD	Port 1 pin 19 open drain mode control, see P1.00OD	0
20	P1.20OD	Port 1 pin 20open drain mode control, see P1.00OD	0
21	P1.21OD[1]	Port 1 pin 21 open drain mode control, see P1.00OD	0
22	P1.22OD	Port 1 pin 22 open drain mode control, see P1.00OD	0
23	P1.23OD	Port 1 pin 23 open drain mode control, see P1.00OD	0
24	P1.24OD	Port 1 pin 24open drain mode control, see P1.00OD	0
25	P1.25OD	Port 1 pin 25 open drain mode control, see P1.00OD	0
26	P1.26OD	Port 1 pin 26 open drain mode control, see P1.00OD	0
27	P1.27OD[1]	Port 1 pin 27 open drain mode control, see P1.00OD	0
28	P1.28OD	Port 1 pin 28 open drain mode control, see P1.00OD	0
29	P1.29OD	Port 1 pin 29 open drain mode control, see P1.00OD	0
30	P1.30OD	Port 1 pin 30 open drain mode control, see P1.00OD	0
31	P1.31OD	Port 1 pin 31 open drain mode control.	0

Open Drain Pin Mode select register 2 (PINMODE_OD2 - 0x4002 C070)

This register controls the open drain mode for Port 2 pins.

PINMODE_OD2	Symbol	Value	Description	Reset value
0	P2.00OD		Port 2 pin 0 open drain mode control.	0
		0	P2.0 pin is in the normal (not open drain) mode.	
		1	P2.0 pin is in the open drain mode.	
1	P2.01OD		Port 2 pin 1 open drain mode control, see P2.00OD	0
2	P2.02OD		Port 2 pin 2 open drain mode control, see P2.00OD	0
3	P2.03OD		Port 2 pin 3 open drain mode control, see P2.00OD	0
4	P2.04OD		Port 2 pin 4 open drain mode control, see P2.00OD	0

PINMODE_OD2	Symbol	Value	Description	Reset value
5	P2.05OD		Port 2 pin 5 open drain mode control, see P2.00OD	0
6	P2.06OD		Port 2 pin 6 open drain mode control, see P2.00OD	0
7	P2.07OD		Port 2 pin 7 open drain mode control, see P2.00OD	0
8	P2.08OD		Port 2 pin 8 open drain mode control, see P2.00OD	0
9	P2.09OD		Port 2 pin 9 open drain mode control, see P2.00OD	0
10	P2.10OD		Port 2 pin 10 open drain mode control, see P2.00OD	0
11	P2.11OD[1]		Port 2 pin 11 open drain mode control, see P2.00OD	0
12	P2.12OD[1]		Port 2 pin 12 open drain mode control, see P2.00OD	0
13	P2.13OD[1]		Port 2 pin 13 open drain mode control, see P2.00OD	0
31:14	-		Reserved.	NA

Open Drain Pin Mode select register 3 (PINMODE_OD3 - 0x4002 C074)

This register controls the open drain mode for Port 3 pins.

PINMODE_OD3	Symbol	Value	Description	Reset value
24:0	-		Reserved.	NA
25	P3.25OD[1]		Port 3 pin 25 open drain mode control.	0
		0	P3.25 pin is in the normal (not open drain) mode.	
		1	P3.25 pin is in the open drain mode.	
26	P3.26OD[1]		Port 3 pin 26 open drain mode control, see P3.25OD	0
31:27	-		Reserved.	NA

Open Drain Pin Mode select register 4 (PINMODE_OD4 - 0x4002 C078)

This register controls the open drain mode for Port 4 pins.

PINMODE_OD4	Symbol	Value	Description	Reset value
27:0	-		Reserved.	NA
28	P4.28OD		Port 4 pin 28 open drain mode control.	0
		0	P4.28 pin is in the normal (not open drain) mode.	
		1	P4.28 pin is in the open drain mode.	
29	P4.28OD		Port 4 pin 29 open drain mode control, see P4.28OD	0
31:30	-		Reserved.	NA

General Purpose Input/Output (GPIO)

Features

• Accelerated GPIO functions:

– GPIO registers are located on a peripheral AHB bus for fast I/O timing.

– Mask registers allow treating sets of port bits as a group, leaving other bits unchanged.

– All GPIO registers are byte, half-word, and word addressable.

– Entire port value can be written in one instruction.

– GPIO registers are accessible by the GPDMA.

• Bit-level set and clear registers allow a single instruction set or clear of any number of bits in one port

. • All GPIO registers support Cortex-M3 bit-banding.

• GPIO registers are accessible by the GPDMA controller to allow DMA of data to or from GPIOs, synchronized to any DMA request.

• Direction control of individual port bits.

• All I/Os default to input with pullup after reset.

Interrupt generating digital ports

• Port 0 and Port 2 can provide a single interrupt for any combination of port pins.

• Each port pin can be programmed to generate an interrupt on a rising edge, a falling edge, or both.

• Edge detection is asynchronous, so it may operate when clocks are not present, such as during Power-down mode. With this feature, level triggered interrupts are not needed.

• Each enabled interrupt contributes to a wake-up signal that can be used to bring the part out of Power-down mode.

• Registers provide a software view of pending rising edge interrupts, pending falling edge interrupts, and overall pending GPIO interrupts.

• GPIO0 and GPIO2 interrupts share the same position in the NVIC with External Interrupt 3.

Applications

• General purpose I/O

• Driving LEDs or other indicators

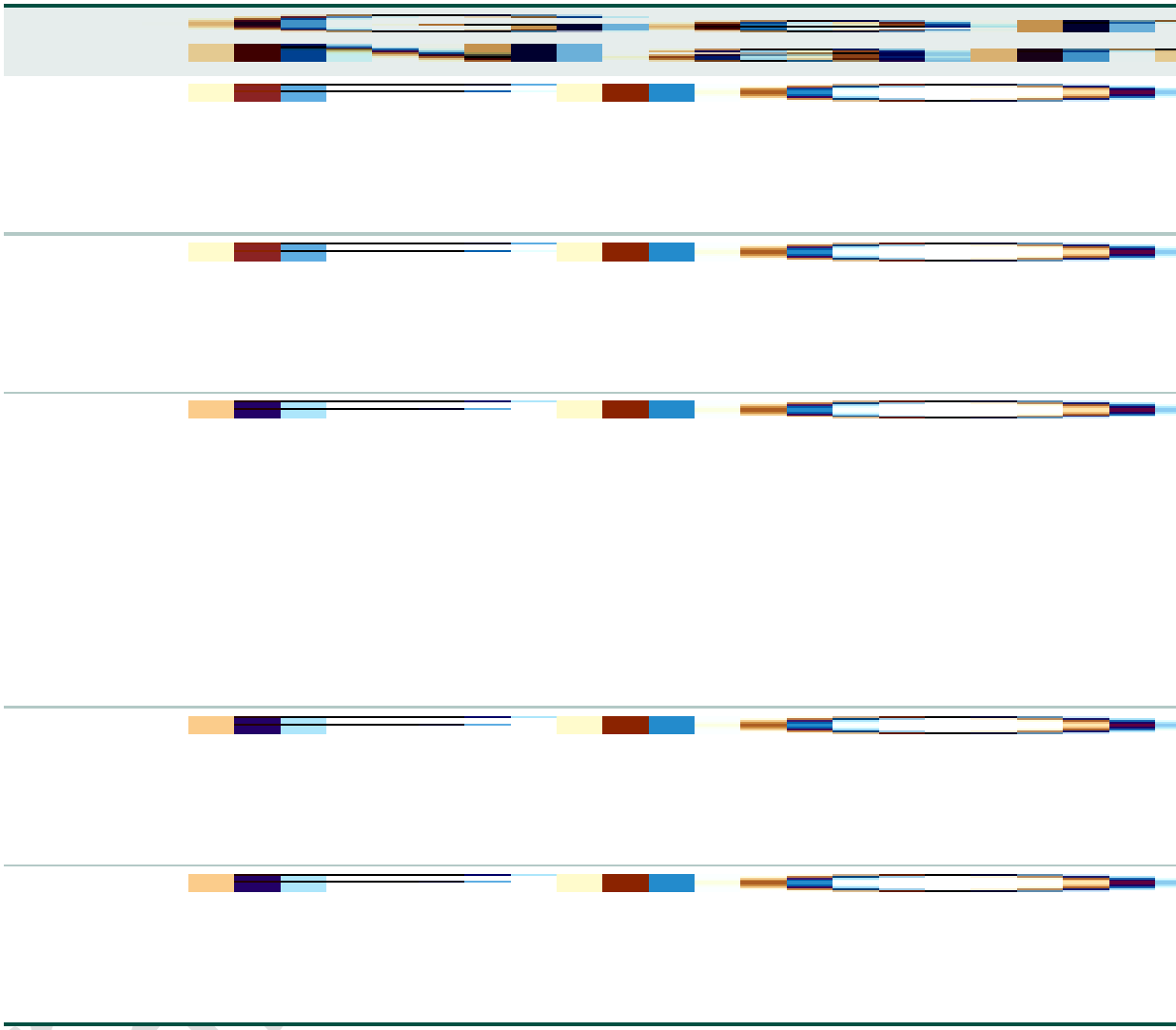
• Controlling off-chip devices

• Sensing digital inputs, detecting edges

Cranes Varsity

• Bringing the part out of Power-down mode.

Register description



GPIO interrupt register map

Generic Name	Description	Access	Reset value[1]	PORTn Register Name & Address
IntEnR	GPIO Interrupt Enable for Rising edge.	R/W	0	IO0IntEnR - 0x4002 8090 IO2IntEnR - 0x4002 80B0
IntEnF	GPIO Interrupt Enable for Falling edge.	R/W	0	IO0IntEnR - 0x4002 8094 IO2IntEnR - 0x4002 80B4
IntStatR	GPIO Interrupt Status for Rising edge.	RO	0	IO0IntStatR - 0x4002 8084 IO2IntStatR - 0x4002 80A4
IntStatF	GPIO Interrupt Status for Falling edge.	RO	0	IO0IntStatF - 0x4002 8088 IO2IntStatF - 0x4002 80A8
IntClr	GPIO Interrupt Clear.	WO	0	IO0IntClr - 0x4002 808C IO2IntClr - 0x4002 80AC
IntStatus	GPIO overall Interrupt Status.	RO	0	IOIntStatus - 0x4002 8080

GPIO port Direction register FIOxDIR (FIO0DIR to FIO4DIR- 0x2009 C000 to 0x2009 C080)

This word accessible register is used to control the direction of the pins when they are configured as GPIO port pins. Direction bit for any pin must be set according to the pin functionality.

Bit	Symbol	Value	Description	Reset value
31:0	FIO0DIR FIO1DIR FIO2DIR FIO3DIR FIO4DIR		Fast GPIO Direction PORTx control bits. Bit 0 in FIOxDIR controls pin Px.0, bit 31 in FIOxDIR controls pin Px.31.	0x0
		0	Controlled pin is input.	
		1	Controlled pin is output.	

Note that GPIO pins P0.29 and P0.30 are shared with the USB_D+ and USB_D- pins and must have the same direction. If either FIO0DIR bit 29 or 30 are configured as zero, both P0.29 and P0.30 will be inputs. If both FIO0DIR bits 29 and 30 are ones, both P0.29 and P0.30 will be outputs.

Aside from the 32-bit long and word only accessible FIODIR register, every fast GPIO port can also be controlled via several byte and half-word accessible registers listed in Table 105, too. Next to providing the same functions as the FIODIR register, these additional registers allow easier and faster access to the physical port pins.

Generic Register name	Description	Register length (bits) & access	Reset value	PORTn Register Address & Name
FIOxDIR0	Fast GPIO Port x Direction control register 0. Bit 0 in FIOxDIR0 register corresponds to pin Px.0 … bit 7 to pin Px.7.	8 (byte) R/W	0x00	FIO0DIR0 - 0x2009 C000 FIO1DIR0 - 0x2009 C020 FIO2DIR0 - 0x2009 C040 FIO3DIR0 - 0x2009 C060 FIO4DIR0 - 0x2009 C080
FIOxDIR1	Fast GPIO Port x Direction control register 1. Bit 0 in FIOxDIR1 register corresponds to pin Px.8 … bit 7 to pin Px.15.	8 (byte) R/W	0x00	FIO0DIR1 - 0x2009 C001 FIO1DIR1 - 0x2009 C021 FIO2DIR1 - 0x2009 C041 FIO3DIR1 - 0x2009 C061 FIO4DIR1 - 0x2009 C081
FIOxDIR2	Fast GPIO Port x Direction control register 2. Bit 0 in FIOxDIR2 register corresponds to pin Px.16 … bit 7 to pin Px.23.	8 (byte) R/W	0x00	FIO0DIR2 - 0x2009 C002 FIO1DIR2 - 0x2009 C022 FIO2DIR2 - 0x2009 C042 FIO3DIR2 - 0x2009 C062 FIO4DIR2 - 0x2009 C082
FIOxDIR3	Fast GPIO Port x Direction control register 3. Bit 0 in FIOxDIR3 register corresponds to pin Px.24 … bit 7 to pin Px.31.	8 (byte) R/W	0x00	FIO0DIR3 - 0x2009 C003 FIO1DIR3 - 0x2009 C023 FIO2DIR3 - 0x2009 C043 FIO3DIR3 - 0x2009 C063 FIO4DIR3 - 0x2009 C083
FIOxDIRL	Fast GPIO Port x Direction control Lower half-word register. Bit 0 in FIOxDIRL register corresponds to pin Px.0 … bit 15 to pin Px.15.	16 (half-word) R/W	0x0000	FIO0DIRL - 0x2009 C000 FIO1DIRL - 0x2009 C020 FIO2DIRL - 0x2009 C040 FIO3DIRL - 0x2009 C060 FIO4DIRL - 0x2009 C080
FIOxDIRU	Fast GPIO Port x Direction control Upper half-word register. Bit 0 in FIOxDIRU register corresponds to Px.16 … bit 15 to Px.31.	16 (half-word) R/W	0x0000	FIO0DIRU - 0x2009 C002 FIO1DIRU - 0x2009 C022 FIO2DIRU - 0x2009 C042 FIO3DIRU - 0x2009 C062 FIO4DIRU - 0x2009 C082

GPIO port output Set register FIOxSET (FIO0SET to FIO4SET - 0x2009 C018 to 0x2009 C098)

This register is used to produce a HIGH level output at the port pins configured as GPIO in an OUTPUT mode. Writing 1 produces a HIGH level at the corresponding port pins. Writing 0 has no effect. If any pin is configured as an input or a secondary function, writing 1 to the corresponding bit in the FIOxSET has no effect.

Reading the FIOxSET register returns the value of this register, as determined by previous writes to FIOxSET and FIOxCLR (or FIOxPIN as noted above). This value does not reflect the effect of any outside world influence on the I/O pins.Access to a port pin via the FIOxSET register is conditioned by the corresponding bit of the FIOxMASK register

Bit	Symbol	Value	Description	Reset value
31:0	FIO0SET FIO1SET FIO2SET FIO3SET FIO4SET		Fast GPIO output value Set bits. Bit 0 in FIOxSET controls pin Px.0, bit 31 in FIOxSET controls pin Px.31.	0x0
		0	Controlled pin output is unchanged.	
		1	Controlled pin output is set to HIGH.	

Aside from the 32-bit long and word only accessible FIOxSET register, every fast GPIO port can also be controlled via several byte and half-word accessible registers listed in Table 107, too. Next to providing the same functions as the FIOxSET register, these additional registers allow easier and faster access to the physical port pins.

Generic Register name	Description	Register length (bits) & access	Reset value	PORTn Register Address & Name
FIOxSET0	Fast GPIO Port x output Set register 0. Bit 0 in FIOxSET0 register corresponds to pin Px.0 ... bit 7 to pin Px.7.	8 (byte) R/W	0x00	FIO0SET0 - 0x2009 C018 FIO1SET0 - 0x2009 C038 FIO2SET0 - 0x2009 C058 FIO3SET0 - 0x2009 C078 FIO4SET0 - 0x2009 C098
FIOxSET1	Fast GPIO Port x output Set register 1. Bit 0 in FIOxSET1 register corresponds to pin Px.8 ... bit 7 to pin Px.15.	8 (byte) R/W	0x00	FIO0SET1 - 0x2009 C019 FIO1SET1 - 0x2009 C039 FIO2SET1 - 0x2009 C059 FIO3SET1 - 0x2009 C079 FIO4SET1 - 0x2009 C099
FIOxSET2	Fast GPIO Port x output Set register 2. Bit 0 in FIOxSET2 register corresponds to pin Px.16 ... bit 7 to pin Px.23.	8 (byte) R/W	0x00	FIO0SET2 - 0x2009 C01A FIO1SET2 - 0x2009 C03A FIO2SET2 - 0x2009 C05A FIO3SET2 - 0x2009 C07A FIO4SET2 - 0x2009 C09A
FIOxSET3	Fast GPIO Port x output Set register 3. Bit 0 in FIOxSET3 register corresponds to pin Px.24 ... bit 7 to pin Px.31.	8 (byte) R/W	0x00	FIO0SET3 - 0x2009 C01B FIO1SET3 - 0x2009 C03B FIO2SET3 - 0x2009 C05B FIO3SET3 - 0x2009 C07B FIO4SET3 - 0x2009 C09B
FIOxSETL	Fast GPIO Port x output Set Lower half-word register. Bit 0 in FIOxSETL register corresponds to pin Px.0 ... bit 15 to pin Px.15.	16 (half-word) R/W	0x0000	FIO0SETL - 0x2009 C018 FIO1SETL - 0x2009 C038 FIO2SETL - 0x2009 C058 FIO3SETL - 0x2009 C078 FIO4SETL - 0x2009 C098
FIOxSETU	Fast GPIO Port x output Set Upper half-word register. Bit 0 in FIOxSETU register corresponds to Px.16 ... bit 15 to Px.31.	16 (half-word) R/W	0x0000	FIO0SETU - 0x2009 C01A FIO1SETU - 0x2009 C03A FIO2SETU - 0x2009 C05A FIO3SETU - 0x2009 C07A FIO4SETU - 0x2009 C09A

GPIO port output Clear register FIOxCLR (FIO0CLR to FIO4CLR0x2009 C01C to 0x2009 C09C)

This register is used to produce a LOW level output at port pins configured as GPIO in an OUTPUT mode. Writing 1 produces a LOW level at the corresponding port pin and clears the corresponding bit in the FIOxSET register. Writing 0 has no effect. If any pin is configured as an input or a secondary function, writing to FIOxCLR has no effect.

Access to a port pin via the FIOxCLR register is conditioned by the corresponding bit of the FIOxMASK register.

Bit	Symbol	Value	Description	Reset value
31:0	FIO0CLR FIO1CLR FIO2CLR FIO3CLR FIO4CLR		Fast GPIO output value Clear bits. Bit 0 in FIOxCLR controls pin Px.0, bit 31 controls pin Px.31.	0x0
		0	Controlled pin output is unchanged.	
		1	Controlled pin output is set to LOW.	

Aside from the 32-bit long and word only accessible FIOxCLR register, every fast GPIO port can also be controlled via several byte and half-word accessible registers listed in Table 109, too. Next to providing the same functions as the FIOxCLR register, these additional registers allow easier and faster access to the physical port pins.

Generic Register name	Description	Register length (bits) & access	Reset value	PORTn Register Address & Name
FIOxCLR0	Fast GPIO Port x output Clear register 0. Bit 0 in FIOxCLR0 register corresponds to pin Px.0 … bit 7 to pin Px.7.	8 (byte) WO	0x00	FIO0CLR0 - 0x2009 C01C FIO1CLR0 - 0x2009 C03C FIO2CLR0 - 0x2009 C05C FIO3CLR0 - 0x2009 C07C FIO4CLR0 - 0x2009 C09C
FIOxCLR1	Fast GPIO Port x output Clear register 1. Bit 0 in FIOxCLR1 register corresponds to pin Px.8 … bit 7 to pin Px.15.	8 (byte) WO	0x00	FIO0CLR1 - 0x2009 C01D FIO1CLR1 - 0x2009 C03D FIO2CLR1 - 0x2009 C05D FIO3CLR1 - 0x2009 C07D FIO4CLR1 - 0x2009 C09D
FIOxCLR2	Fast GPIO Port x output Clear register 2. Bit 0 in FIOxCLR2 register corresponds to pin Px.16 … bit 7 to pin Px.23.	8 (byte) WO	0x00	FIO0CLR2 - 0x2009 C01E FIO1CLR2 - 0x2009 C03E FIO2CLR2 - 0x2009 C05E FIO3CLR2 - 0x2009 C07E FIO4CLR2 - 0x2009 C09E
FIOxCLR3	Fast GPIO Port x output Clear register 3. Bit 0 in FIOxCLR3 register corresponds to pin Px.24 … bit 7 to pin Px.31.	8 (byte) WO	0x00	FIO0CLR3 - 0x2009 C01F FIO1CLR3 - 0x2009 C03F FIO2CLR3 - 0x2009 C05F FIO3CLR3 - 0x2009 C07F FIO4CLR3 - 0x2009 C09F
FIOxCLRL	Fast GPIO Port x output Clear Lower half-word register. Bit 0 in FIOxCLRL register corresponds to pin Px.0 … bit 15 to pin Px.15.	16 (half-word) WO	0x0000	FIO0CLRL - 0x2009 C01C FIO1CLRL - 0x2009 C03C FIO2CLRL - 0x2009 C05C FIO3CLRL - 0x2009 C07C FIO4CLRL - 0x2009 C09C

Generic Register name	Description	Register length (bits) & access	Reset value	PORTn Register Address & Name
FIOxCLRU	Fast GPIO Port x output Clear Upper half-word register. Bit 0 in FIOxCLRU register corresponds to pin Px.16 … bit 15 to Px.31.	16 (half-word) WO	0x0000	FIO0CLRU - 0x2009 C01E FIO1CLRU - 0x2009 C03E FIO2CLRU - 0x2009 C05E FIO3CLRU - 0x2009 C07E FIO4CLRU - 0x2009 C09E

GPIO port Pin value register FIOxPIN (FIO0PIN to FIO4PIN- 0x2009 C014 to 0x2009 C094)

This register provides the value of port pins that are configured to perform only digital functions. The register will give the logic value of the pin regardless of whether the pin is configured for input or output, or as GPIO or an alternate digital function. As an example, a particular port pin may have GPIO input, GPIO output, UART receive, and PWM output as selectable functions. Any configuration of that pin will allow its current logic state to be read from the corresponding FIOxPIN register.

If a pin has an analog function as one of its options, the pin state cannot be read if the analog configuration is selected. Selecting the pin as an A/D input disconnects the digital features of the pin. In that case, the pin value read in the FIOxPIN register is not valid.

Writing to the FIOxPIN register stores the value in the port output register, bypassing the need to use both the FIOxSET and FIOxCLR registers to obtain the entire written value. This feature should be used carefully in an application since it affects the entire port. Access to a port pin via the FIOxPIN registeris conditioned by the corresponding bit of the FIOxMASK register Only pins masked with zeros in the Mask register will be correlated to the current content of the Fast GPIO port pin value register.

Bit	Symbol	Value	Description	Reset value
31:0	FIO0VAL FIO1VAL FIO2VAL FIO3VAL FIO4VAL		Fast GPIO output value bits. Bit 0 corresponds to pin Px.0, bit 31 corresponds to pin Px.31. Only bits also set to 0 in the FIOxMASK register are affected by a write or show the pin's actual logic state.	0x0
		0	Reading a 0 indicates that the port pin's current state is LOW. Writing a 0 sets the output register value to LOW.	
		1	Reading a 1 indicates that the port pin's current state is HIGH. Writing a 1 sets the output register value to HIGH.	

Aside from the 32-bit long and word only accessible FIOxPIN register, every fast GPIO port can also be controlled via several byte and half-word accessible register listed in. Next to providing the same functions as the FIOxPIN register, these additional registers allow easier and faster access to the physical port pins.

Generic Register name	Description	Register length (bits) & access	Reset value	PORTn Register Address & Name
FIOxPIN0	Fast GPIO Port x Pin value register 0. Bit 0 in FIOxPIN0 register corresponds to pin Px.0 … bit 7 to pin Px.7.	8 (byte) R/W	0x00	FIO0PIN0 - 0x2009 C014 FIO1PIN0 - 0x2009 C034 FIO2PIN0 - 0x2009 C054 FIO3PIN0 - 0x2009 C074 FIO4PIN0 - 0x2009 C094
FIOxPIN1	Fast GPIO Port x Pin value register 1. Bit 0 in FIOxPIN1 register corresponds to pin Px.8 … bit 7 to pin Px.15.	8 (byte) R/W	0x00	FIO0PIN1 - 0x2009 C015 FIO1PIN1 - 0x2009 C035 FIO2PIN1 - 0x2009 C055 FIO3PIN1 - 0x2009 C075 FIO4PIN1 - 0x2009 C095
FIOxPIN2	Fast GPIO Port x Pin value register 2. Bit 0 in FIOxPIN2 register corresponds to pin Px.16 … bit 7 to pin Px.23.	8 (byte) R/W	0x00	FIO0PIN2 - 0x2009 C016 FIO1PIN2 - 0x2009 C036 FIO2PIN2 - 0x2009 C056 FIO3PIN2 - 0x2009 C076 FIO4PIN2 - 0x2009 C096
FIOxPIN3	Fast GPIO Port x Pin value register 3. Bit 0 in FIOxPIN3 register corresponds to pin Px.24 … bit 7 to pin Px.31.	8 (byte) R/W	0x00	FIO0PIN3 - 0x2009 C017 FIO1PIN3 - 0x2009 C037 FIO2PIN3 - 0x2009 C057 FIO3PIN3 - 0x2009 C077 FIO4PIN3 - 0x2009 C097
FIOxPINL	Fast GPIO Port x Pin value Lower half-word register. Bit 0 in FIOxPINL register corresponds to pin Px.0 … bit 15 to pin Px.15.	16 (half-word) R/W	0x0000	FIO0PINL - 0x2009 C014 FIO1PINL - 0x2009 C034 FIO2PINL - 0x2009 C054 FIO3PINL - 0x2009 C074 FIO4PINL - 0x2009 C094
FIOxPINU	Fast GPIO Port x Pin value Upper half-word register. Bit 0 in FIOxPINU register corresponds to pin Px.16 … bit 15 to Px.31.	16 (half-word) R/W	0x0000	FIO0PINU - 0x2009 C016 FIO1PINU - 0x2009 C036 FIO2PINU - 0x2009 C056 FIO3PINU - 0x2009 C076 FIO4PINU - 0x2009 C096

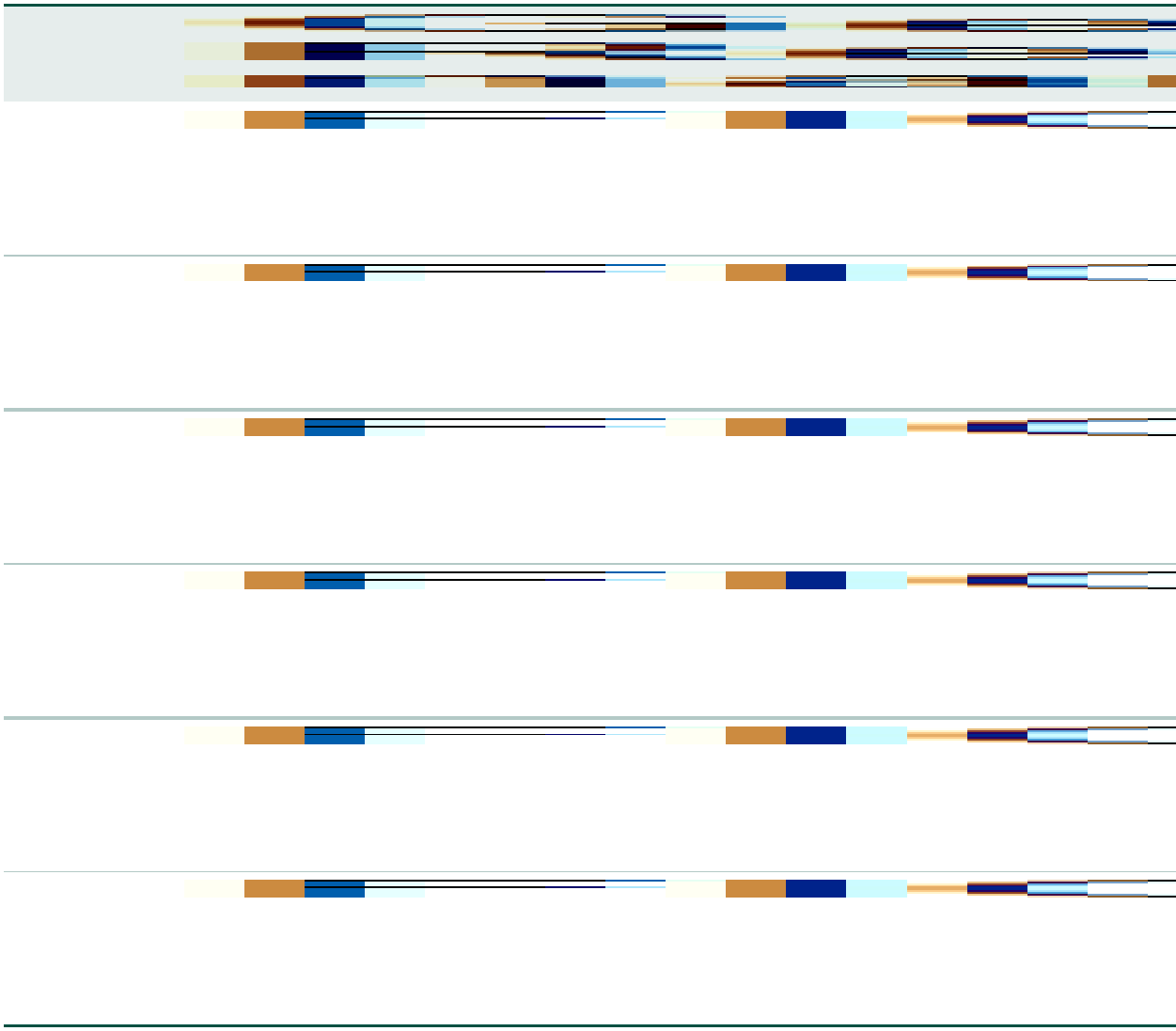
Fast GPIO port Mask register FIOxMASK (FIO0MASK to FIO4MASK - 0x2009 C010 to 0x2009 C090)

This register is used to select port pins that will and will not be affected by write accesses to the FIOxPIN, FIOxSET or FIOxCLR register. Mask register also filters out port's content when the FIOxPIN register is read.

A zero in this register's bit enables an access to the corresponding physical pin via a read or write access. If a bit in this register is one, corresponding pin will not be changed with write access and if read, will not be reflected in the updated FIOxPIN register. For software examples

Bit	Symbol	Value	Description	Reset value
31:0	FIO0MASK FIO1MASK FIO2MASK FIO3MASK FIO4MASK		Fast GPIO physical pin access control.	0x0
		0	Controlled pin is affected by writes to the port's FIOxSET, FIOxCLR, and FIOxPIN register(s). Current state of the pin can be read from the FIOxPIN register.	
		1	Controlled pin is not affected by writes into the port's FIOxSET, FIOxCLR and FIOxPIN register(s). When the FIOxPIN register is read, this bit will not be updated with the state of the physical pin.	

Aside from the 32-bit long and word only accessible FIOxMASK register, every fast GPIO port can also be controlled via several byte and half-word accessible registers listed in below Table. Next to providing the same functions as the FIOxMASK register, these additional registers allow easier and faster access to the physical port pins.



GPIO interrupt registers

The following registers configure the pins of Port 0 and Port 2 to generate interrupts.

GPIO overall Interrupt Status register (IOIntStatus - 0x4002 8080)

This read-only register indicates the presence of interrupt pending on all of the GPIO ports that support GPIO interrupts. Only status one bit per port is required.

Bit	Symbol	Value	Description	Reset value
0	P0Int		Port 0 GPIO interrupt pending.	0
		0	There are no pending interrupts on Port 0.	
		1	There is at least one pending interrupt on Port 0.	
1	-	-	Reserved. The value read from a reserved bit is not defined.	NA
2	P2Int		Port 2 GPIO interrupt pending.	0
		0	There are no pending interrupts on Port 2.	
		1	There is at least one pending interrupt on Port 2.	
31:2	-	-	Reserved. The value read from a reserved bit is not defined.	NA

GPIO Interrupt Enable for port 0 Rising Edge (IO0IntEnR - 0x4002 8090)

Each bit in these read-write registers enables the rising edge interrupt for the corresponding port 0 pin.

Bit	Symbol	Value	Description	Reset value
0	P0.0ER		Enable rising edge interrupt for P0.0.	0
		0	Rising edge interrupt is disabled on P0.0.	
		1	Rising edge interrupt is enabled on P0.0.	
1	P0.1ER		Enable rising edge interrupt for P0.1.	0
2	P0.2ER		Enable rising edge interrupt for P0.2.	0
3	P0.3ER		Enable rising edge interrupt for P0.3.	0
4	P0.4ER[1]		Enable rising edge interrupt for P0.4.	0
5	P0.5ER[1]		Enable rising edge interrupt for P0.5.	0
6	P0.6ER		Enable rising edge interrupt for P0.6.	0
7	P0.7ER		Enable rising edge interrupt for P0.7.	0
8	P0.8ER		Enable rising edge interrupt for P0.8.	0
9	P0.9ER		Enable rising edge interrupt for P0.9.	0
10	P0.10ER		Enable rising edge interrupt for P0.10.	0
11	P0.11ER		Enable rising edge interrupt for P0.11.	0
14:12	-		Reserved	NA
15	P0.15ER		Enable rising edge interrupt for P0.15.	0
16	P0.16ER		Enable rising edge interrupt for P0.16.	0

Bit	Symbol	Value	Description	Reset value
17	P0.17ER		Enable rising edge interrupt for P0.17.	0
18	P0.18ER		Enable rising edge interrupt for P0.18.	0
19	P0.19ER[1]		Enable rising edge interrupt for P0.19.	0
20	P0.20ER[1]		Enable rising edge interrupt for P0.20.	0
21	P0.21ER[1]		Enable rising edge interrupt for P0.21.	0
22	P0.22ER		Enable rising edge interrupt for P0.22.	0
23	P0.23ER[1]		Enable rising edge interrupt for P0.23.	0
24	P0.24ER[1]		Enable rising edge interrupt for P0.24.	0
25	P0.25ER		Enable rising edge interrupt for P0.25.	0
26	P0.26ER		Enable rising edge interrupt for P0.26.	0
27	P0.27ER[1]		Enable rising edge interrupt for P0.27.	0
28	P0.28ER[1]		Enable rising edge interrupt for P0.28.	0
29	P0.29ER		Enable rising edge interrupt for P0.29.	0
30	P0.30ER		Enable rising edge interrupt for P0.30.	0
31	-		Reserved.	NA

GPIO Interrupt Enable for port 2 Rising Edge (IO2IntEnR - 0x4002 80B0)

Each bit in these read-write registers enables the rising edge interrupt for the corresponding port 2 pin.

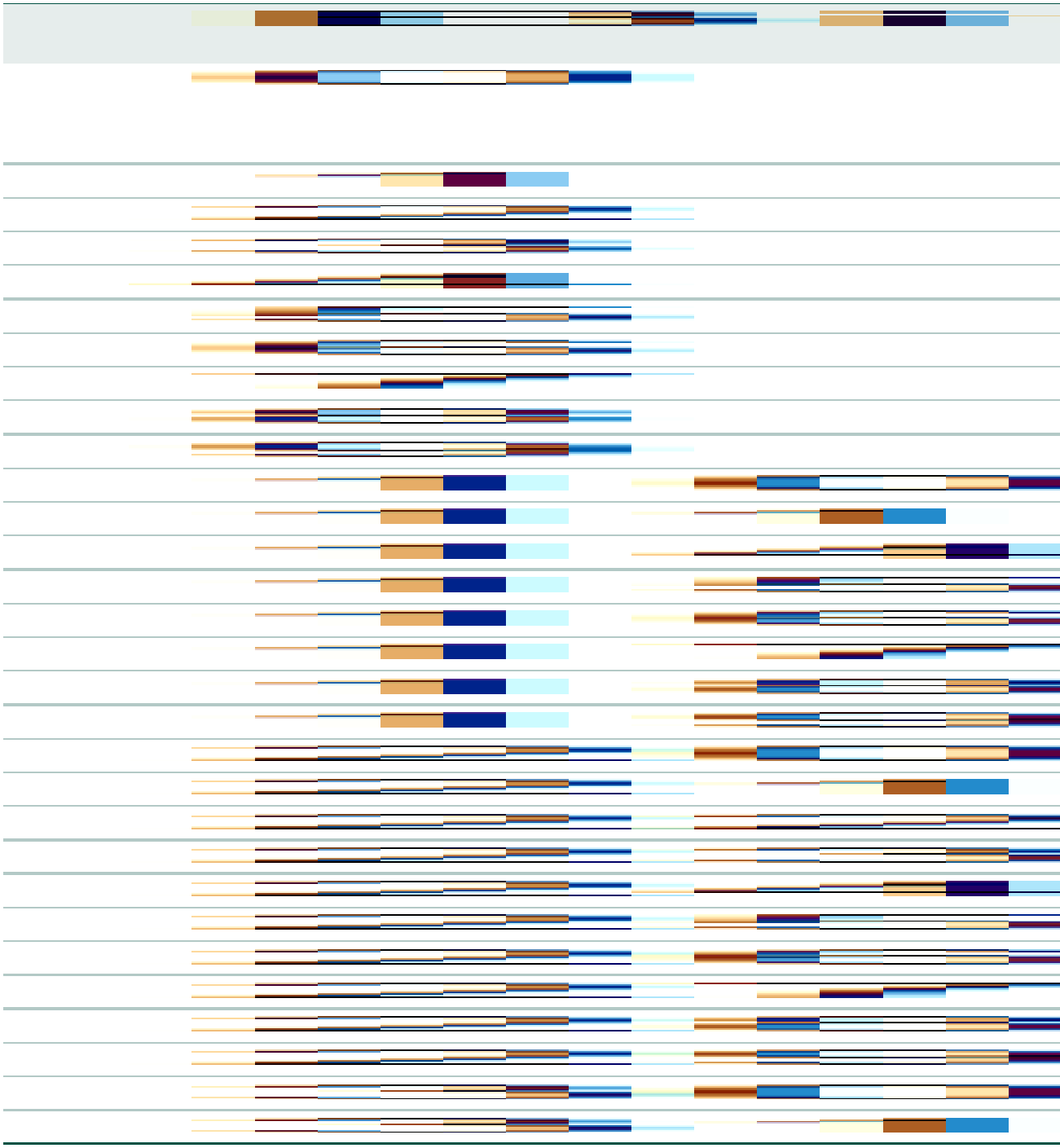
Bit	Symbol	Value	Description	Reset value
0	P2.0ER		Enable rising edge interrupt for P2.0.	0
		0	Rising edge interrupt is disabled on P2.0.	
		1	Rising edge interrupt is enabled on P2.0.	
1	P2.1ER		Enable rising edge interrupt for P2.1.	0
2	P2.2ER		Enable rising edge interrupt for P2.2.	0
3	P2.3ER		Enable rising edge interrupt for P2.3.	0
4	P2.4ER		Enable rising edge interrupt for P2.4.	0
5	P2.5ER		Enable rising edge interrupt for P2.5.	0
6	P2.6ER		Enable rising edge interrupt for P2.6.	0
7	P2.7ER		Enable rising edge interrupt for P2.7.	0
8	P2.8ER		Enable rising edge interrupt for P2.8.	0
9	P2.9ER		Enable rising edge interrupt for P2.9.	0
10	P2.10ER		Enable rising edge interrupt for P2.10.	0
11	P2.11ER[1]		Enable rising edge interrupt for P2.11.	0
12	P2.12ER[1]		Enable rising edge interrupt for P2.12.	0
13	P2.13ER[1]		Enable rising edge interrupt for P2.13.	0
31:14	-		Reserved.	NA

GPIO Interrupt Enable for port 0 Falling Edge (IO0IntEnF - 0x4002 8094) Each bit in these read-write registers enables the falling edge interrupt for the corresponding GPIO port 0 pin.

GPIO Interrupt Enable for port 2 Falling Edge (IO2IntEnF - 0x4002 80B4) Each bit in these read-write registers enables the falling edge interrupt for the corresponding GPIO port 2 pin.

Cranes Varsity

GPIO Interrupt Status for port 0 Rising Edge Interrupt (IO0IntStatR - 0x4002 8084) Each bit in these read-only registers indicates the rising edge interrupt status for port0.



GPIO Interrupt Status for port 2 Rising Edge Interrupt (IO2IntStatR - 0x4002 80A4)

Each bit in these read-only registers indicates the rising edge interrupt status for port 2.

Bit	Symbol	Value	Description	Reset value
0	P2.0REI		Status of Rising Edge Interrupt for P2.0	0
		0	A rising edge has not been detected on P2.0.	
		1	Interrupt has been generated due to a rising edge on P2.0.	
1	P2.1REI		Status of Rising Edge Interrupt for P2.1.	0
2	P2.2REI		Status of Rising Edge Interrupt for P2.2.	0
3	P2.3REI		Status of Rising Edge Interrupt for P2.3.	0
4	P2.4REI		Status of Rising Edge Interrupt for P2.4.	0
5	P2.5REI		Status of Rising Edge Interrupt for P2.5.	0
6	P2.6REI		Status of Rising Edge Interrupt for P2.6.	0
7	P2.7REI		Status of Rising Edge Interrupt for P2.7.	0
8	P2.8REI		Status of Rising Edge Interrupt for P2.8.	0
9	P2.9REI		Status of Rising Edge Interrupt for P2.9.	0
10	P2.10REI		Status of Rising Edge Interrupt for P2.10.	0
11	P2.11REI[1]		Status of Rising Edge Interrupt for P2.11.	0
12	P2.12REI[1]		Status of Rising Edge Interrupt for P2.12.	0
13	P2.13REI[1]		Status of Rising Edge Interrupt for P2.13.	0
31:14	-		Reserved.	NA

GPIO Interrupt Status for port 0 Falling Edge Interrupt (IO0IntStatF - 0x4002 8088)

Each bit in these read-only registers indicates the falling edge interrupt status for port 0.

GPIO Interrupt Status for port 2 Falling Edge Interrupt (IO2IntStatF - 0x4002 80A8)

Each bit in these read-only registers indicates the falling edge interrupt status for port 2.

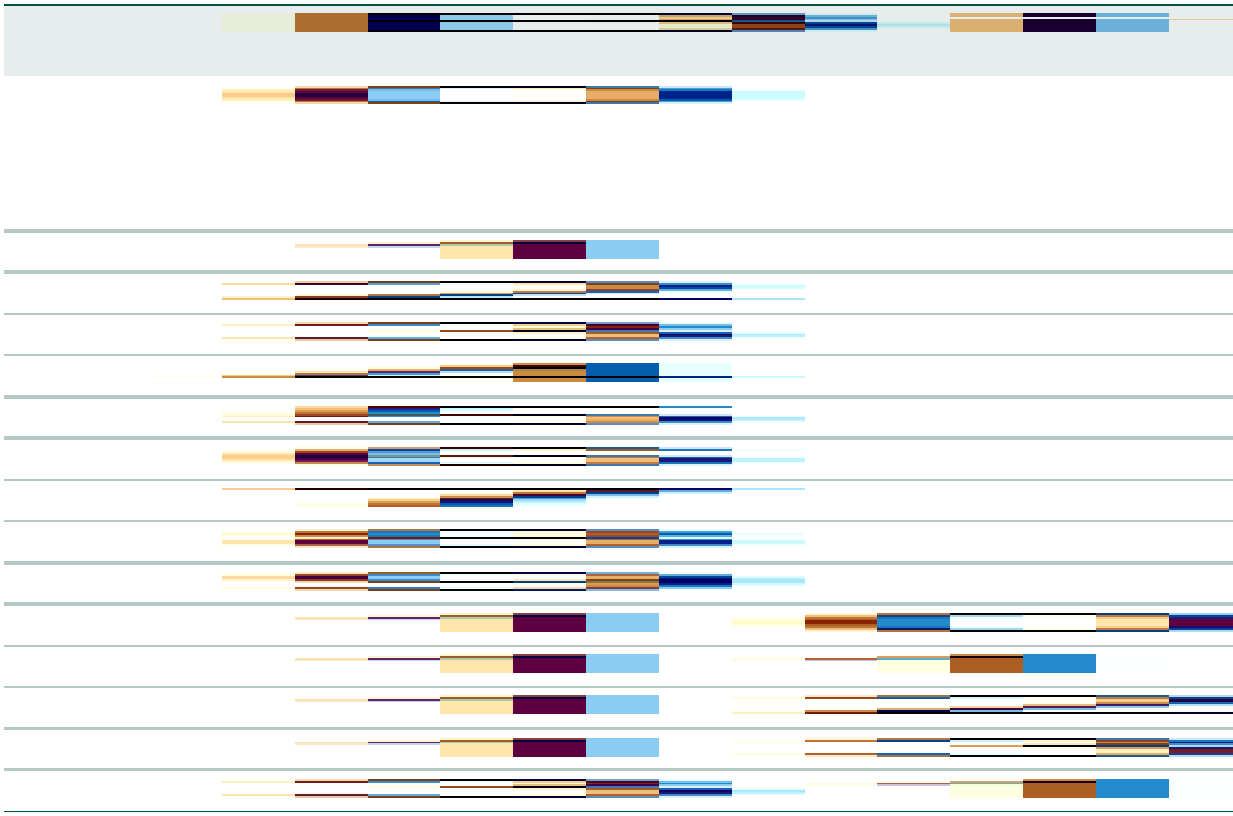
GPIO Interrupt Clear register for port 0 (IO0IntClr - 0x4002 808C)

Writing a 1 into a bit in this write-only register clears any interrupts for the corresponding port 0 pin.

Bit	Symbol	Value	Description	Reset value
0	P0.0CI		Clear GPIO port Interrupts for P0.0	0
		0	Corresponding bits in IOxIntStatR and IOxIntStatF are unchanged.	
		1	Corresponding bits in IOxIntStatR and IOxStatF are cleared.	
1	P0.1CI		Clear GPIO port Interrupts for P0.1.	0
2	P0.2CI		Clear GPIO port Interrupts for P0.2.	0
3	P0.3CI		Clear GPIO port Interrupts for P0.3.	0
4	P0.4CI[1]		Clear GPIO port Interrupts for P0.4.	0
5	P0.5CI[1]		Clear GPIO port Interrupts for P0.5.	0
6	P0.6CI		Clear GPIO port Interrupts for P0.6.	0
7	P0.7CI		Clear GPIO port Interrupts for P0.7.	0
8	P0.8CI		Clear GPIO port Interrupts for P0.8.	0
9	P0.9CI		Clear GPIO port Interrupts for P0.9.	0
10	P0.10CI		Clear GPIO port Interrupts for P0.10.	0
11	P0.11CI		Clear GPIO port Interrupts for P0.11.	0
14:12	-		Reserved.	NA
15	P0.15CI		Clear GPIO port Interrupts for P0.15.	0
16	P0.16CI		Clear GPIO port Interrupts for P0.16.	0
17	P0.17CI		Clear GPIO port Interrupts for P0.17.	0
18	P0.18CI		Clear GPIO port Interrupts for P0.18.	0
19	P0.19CI[1]		Clear GPIO port Interrupts for P0.19.	0
20	P0.20CI[1]		Clear GPIO port Interrupts for P0.20.	0
21	P0.21CI[1]		Clear GPIO port Interrupts for P0.21.	0
22	P0.22CI		Clear GPIO port Interrupts for P0.22.	0
23	P0.23CI[1]		Clear GPIO port Interrupts for P0.23.	0
24	P0.24CI[1]		Clear GPIO port Interrupts for P0.24.	0
25	P0.25CI		Clear GPIO port Interrupts for P0.25.	0
26	P0.26CI		Clear GPIO port Interrupts for P0.26.	0
27	P0.27CI[1]		Clear GPIO port Interrupts for P0.27.	0
28	P0.28CI[1]		Clear GPIO port Interrupts for P0.28.	0
29	P0.29CI		Clear GPIO port Interrupts for P0.29.	0
30	P0.30CI		Clear GPIO port Interrupts for P0.30.	0
31	-		Reserved.	NA

GPIO Interrupt Clear register for port 2 (IO2IntClr - 0x4002 80AC)

Writing a 1 into a bit in this write-only register clears any interrupts for the corresponding port 2 pin



Cranes V

CHAPTER-19

Nested Vectored Interrupt Controller (NVIC)

Features

• Nested Vectored Interrupt Controller that is an integral part of the ARM Cortex-M3

• Tightly coupled interrupt controller provides low interrupt latency

• Controls system exceptions and peripheral interrupts

• In the LPC176x/5x, the NVIC supports 35 vectored interrupts

• 32 programmable interrupt priority levels, with hardware priority level masking

• Relocatable vector table

• Non-Maskable Interrupt

• Software interrupt generation

Description

The Nested Vectored Interrupt Controller (NVIC) is an integral part of the Cortex-M3. The tight coupling to the CPU allows for low interrupt latency and efficient processing of late arriving interrupts.

Interrupt sources

Table 50 lists the interrupt sources for each peripheral function. Each peripheral device may have one or more interrupt lines to the Vectored Interrupt Controller. Each line may represent more than one interrupt source, as noted. Exception numbers relate to where entries are stored in the exception vector table. Interrupt numbers are used in some other contexts, such as software interrupts. In addition, the NVIC handles the Non-Maskable Interrupt (NMI). In order for NMI to operate from an external signal, the NMI function must be connected to the related device pin (P2.10 / EINT0n / NMI). When connected, a logic 1 on the pin will cause the NMI to be processed.

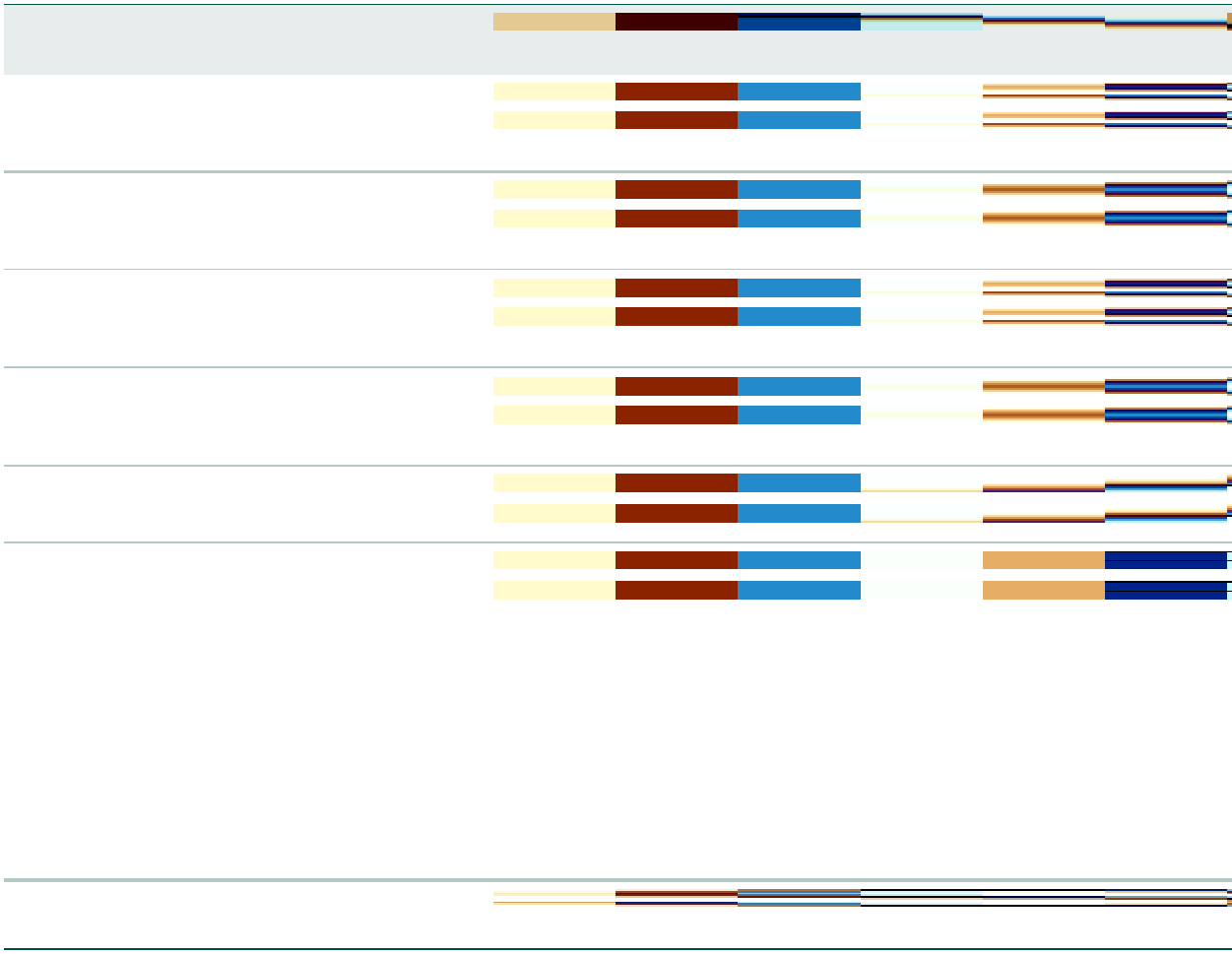
Cranes Varsity

Interrupt ID	Exception Number	Vector Offset	Function	Flag(s)
0	16	0x40	WDT	Watchdog Interrupt (WDINT)
1	17	0x44	Timer 0	Match 0 - 1 (MR0, MR1) Capture 0 - 1 (CR0, CR1)
2	18	0x48	Timer 1	Match 0 - 2 (MR0, MR1, MR2) Capture 0 - 1 (CR0, CR1)
3	19	0x4C	Timer 2	Match 0-3 Capture 0-1
4	20	0x50	Timer 3	Match 0-3 Capture 0-1
5	21	0x54	UART0	Rx Line Status (RLS) Transmit Holding Register Empty (THRE) Rx Data Available (RDA) Character Time-out Indicator (CTI) End of Auto-Baud (ABEO) Auto-Baud Time-Out (ABTO)
6	22	0x58	UART1	Rx Line Status (RLS) Transmit Holding Register Empty (THRE) Rx Data Available (RDA) Character Time-out Indicator (CTI) Modem Control Change End of Auto-Baud (ABEO) Auto-Baud Time-Out (ABTO)
7	23	0x5C	UART 2	Rx Line Status (RLS) Transmit Holding Register Empty (THRE) Rx Data Available (RDA) Character Time-out Indicator (CTI) End of Auto-Baud (ABEO) Auto-Baud Time-Out (ABTO)
8	24	0x60	UART 3	Rx Line Status (RLS) Transmit Holding Register Empty (THRE) Rx Data Available (RDA) Character Time-out Indicator (CTI) End of Auto-Baud (ABEO) Auto-Baud Time-Out (ABTO)
9	25	0x64	PWM1	Match 0 - 6 of PWM1 Capture 0-1 of PWM1
10	26	0x68	I^2C0	SI (state change)
11	27	0x6C	I^2C1	SI (state change)
12	28	0x70	I^2C2	SI (state change)
13	29	0x74	SPI	SPI Interrupt Flag (SPIF) Mode Fault (MODF)

Interrupt ID	Exception Number	Vector Offset	Function	Flag(s)
14	30	0x78	SSP0	Tx FIFO half empty of SSP0 Rx FIFO half full of SSP0 Rx Timeout of SSP0 Rx Overrun of SSP0
15	31	0x7C	SSP 1	Tx FIFO half empty Rx FIFO half full Rx Timeout Rx Overrun
16	32	0x80	PLL0 (Main PLL)	PLL0 Lock (PLOCK0)
17	33	0x84	RTC	Counter Increment (RTCCIF) Alarm (RTCALF)
18	34	0x88	External Interrupt	External Interrupt 0 (EINT0)
19	35	0x8C	External Interrupt	External Interrupt 1 (EINT1)
20	36	0x90	External Interrupt	External Interrupt 2 (EINT2)
21	37	0x94	External Interrupt	External Interrupt 3 (EINT3). **Note:** EINT3 channel is shared with GPIO interrupts
22	38	0x98	ADC	A/D Converter end of conversion
23	39	0x9C	BOD	Brown Out detect
24	40	0xA0	USB	USB_INT_REQ_LP, USB_INT_REQ_HP, USB_INT_REQ_DMA
25	41	0xA4	CAN	CAN Common, CAN 0 Tx, CAN 0 Rx, CAN 1 Tx, CAN 1 Rx
26	42	0xA8	GPDMA	IntStatus of DMA channel 0, IntStatus of DMA channel 1
27	43	0xAC	I^2S	irq, dmareq1, dmareq2
28	44	0xB0	Ethernet	WakeupInt, SoftInt, TxDoneInt, TxFinishedInt, TxErrorInt, TxUnderrunInt, RxDoneInt, RxFinishedInt, RxErrorInt, RxOverrunInt.
29	45	0xB4	Repetitive Interrupt Timer	RITINT
30	46	0xB8	Motor Control PWM	IPER[2:0], IPW[2:0], ICAP[2:0], FES
31	47	0xBC	Quadrature Encoder	INX_Int, TIM_Int, VELC_Int, DIR_Int, ERR_Int, ENCLK_Int, POS0_Int, POS1_Int, POS2_Int, REV_Int, POS0REV_Int, POS1REV_Int, POS2REV_Int
32	48	0xC0	PLL1 (USB PLL)	PLL1 Lock (PLOCK1)
33	49	0xC4	USB Activity Interrupt	USB_NEED_CLK
34	50	0xC8	CAN Activity Interrupt	CAN1WAKE, CAN2WAKE

Register description

The following table summarizes the registers in the NVIC as implemented in the LPC176x/5x. The Cortex-M3 User Guide Section 34.4.2 provides a functional description of the NVIC.



CHAPTER 20
INTRODUCTION TO REAL TIME OPERATING SYSTEMS

REAL TIME EMBEDDED SYSTEMS

In the simplest form, real-time systems can be defined as those systems that respond to external events in a timely fashion. The response time is guaranteed. A good way to understand the relationship between real-time systems and embedded systems is to view them as two intersecting circles, as shown in Figure 20.1 . It can be seen that not all embedded systems exhibit real-time behaviors nor are all real-time systems are embedded. However, the two systems are not mutually exclusive, and the area in which they overlap creates the combination of systems known as *real-time embedded systems*.

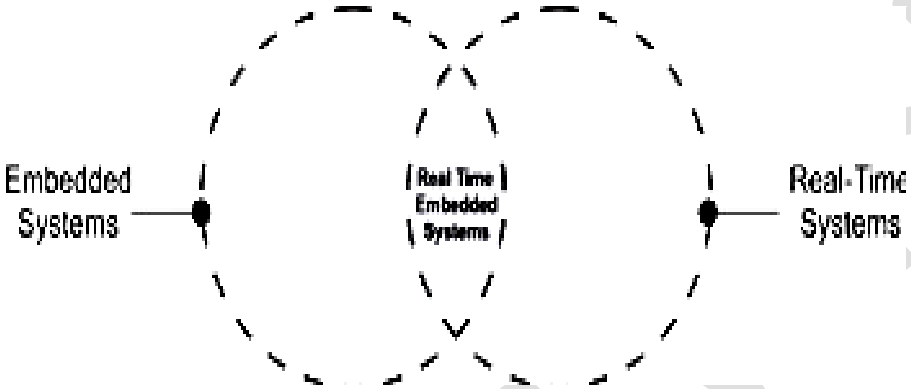


Figure 20.1: Real-time embedded systems.

Real-Time Systems

The structure of a real-time system, as shown in Figure 20.2, is a controlling system and at least one controlled system. The controlling system interacts with the controlled system in various ways.

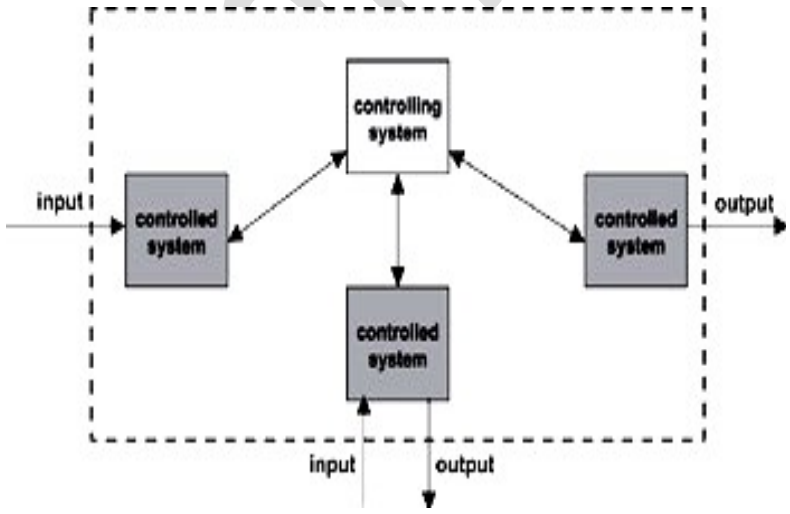


Figure 20.2: Structure of real-time systems.

First, the interaction can be *periodic*, in which communication is initiated from the controlling system to the controlled system. In this case, the communication is predictable and occurs at predefined intervals. Second, the interaction can be *aperiodic*, in which communication is initiated from the controlled system to the controlling system. In this case, the communication is unpredictable and is determined by the random occurrences of external events in the environment of the controlled system. Finally, the communication can be a combination of both types. The controlling system must process and respond to the events and information generated by the controlled system in a guaranteed time frame.

Imagine a real-time weapons defense system whose role is to protect a naval destroyer by shooting down incoming missiles. The idea is to shred an incoming missile into pieces with bullets before it reaches the ship. The weapons system is comprised of a radar system, a command-and- decision (C&D) system, and weapons firing control system. The controlling system is the C&D system, whereas the controlled systems are the radar system and the weapons firing control system.

- The radar system scans and searches for potential targets. Coordinates of a potential target are sent to the C&D system periodically with high frequency after the target is acquired.
- The C&D system must first determine the threat level by threat classification and evaluation, based on the target information provided by the radar system. If a threat is imminent, the C&D system must, at a minimum, calculate the speed and flight path or trajectory, as well as estimate the impact location. Because a missile tends to drift off its flight path with the degree of drift dependent on the precision of its guidance system, the C&D system calculates an area (a box) around the flight path.
- The C&D system then activates the weapons firing control system closest to the anticipated impact location and guides the weapons system to fire continuously within the moving area or box until the target is destroyed. The weapons firing control system is comprised of large-caliber, multi-barrel, high-muzzle velocity, high-power machine guns. In this weapons defense system example, the communication between the radar system and the C&D system is aperiodic, because the occurrence of a potential target is unpredictable and the potential target can appear at any time. The communication between the C&D system and the weapons firing control system is, however, periodic because the C&D system feeds the firing coordinates into the weapons control system periodically (with an extremely high frequency). Initial firing coordinates are based on a pre-computed flight path but are updated in real-time according to the actual location of the incoming missile.

Consider another example of a real-time system-the cruise missile guidance system. A cruise missile flies at subsonic speed. It can travel at about 10 meters above water, 30 meters above flat ground, and 100 meters above mountain terrains. A modern cruise missile can hit a target within a 50-meter range. All these capabilities are due to the high-precision, real-time guidance system built into the nose of a cruise missile. In a simplified view, the guidance system is comprised of the radar system (both forward-looking and look-down radars), the navigation system, and the divert-and-altitude-control system. The navigation system contains digital maps covering the missile flight path. The forward-looking radar scans and maps out the approaching terrains. This information is fed to the navigation system in real time. The navigation system must then recalculate flight coordinates to avoid terrain obstacles. The new coordinates are immediately fed to the divert-and-altitude-control system to adjust the flight path. The look-down radar periodically scans the ground terrain along its flight path. The scanned data is compared

with the estimated section of the pre-recorded maps. Corrective adjustments are made to the flight coordinates and sent to the divert-and-altitude-control system if data comparison indicates that the missile has drifted off the intended flight path.

In this example, the controlling system is the navigation system. The controlled systems are the radar system and the divert-and-altitude-control system. We can observe both periodic and aperiodic communications in this example. The communication between the radars and the navigation system is aperiodic. The communication between the navigation system and the diver and- altitude-control system is periodic.

Characteristics of Real-Time Systems

The C&D system in the weapons defense system must calculate the anticipated flight path of the incoming missile quickly and guide the firing system to shoot the missile down before it reaches the destroyer. Assume T1 is the time the missile takes to reach the ship and is a function of the missile's distance and velocity. Assume T2 is the time the C&D system takes to activate the weapons firing control system and includes transmitting the firing coordinates plus the firing delay. The difference between T1 and T2 is how long the computation may take. The missile would reach its intended target if the C&D system took too long in computing the flight path. The missile would still reach its target if the computation produced by the C&D system was inaccurate. The navigation system in the cruise missile must respond to the changing terrain fast enough so that it can re-compute coordinates and guide the altitude control system to a new flight path. The missile might collide with a mountain if the navigation system cannot compute new flight coordinates fast enough, or if the new coordinates do not steer the missile out of the collision course.

Therefore, we can extract two essential characteristics of real-time systems from the examples given earlier. These characteristics are that real-time systems must produce correct computational results, called *logical or functional correctness*, and that these computations must conclude within a predefined period, called *timing correctness.*

"Real-time systems are defined as those systems in which the overall correctness of the system depends on both the functional correctness and the timing correctness. The timing correctness is at least as important as the functional correctness. If the timing constraints of the system are not met, system failure is said to occur"

Types of RTS

Based on the Time constraints RTS can further categorized as:

- Soft Real Time Systems
- Hard Real Time Systems
- Firm Real Time Systems

Hard Real Time Systems

Hard real-time means that the system (i.e., the entire system including OS, middleware, application, HW, communications, etc.) must be designed to guarantee that response requirements are met. Hard Real-Time doesn't mean fast execution. Examples: Electronic Engines, Automotive and Flight Control Systems, Medical Systems, Industrial Control Systems, Robotics.

Soft Real Time Systems

Soft real-time is exactly the same as hard real-time in its infrastructure requirements, but it is not necessary for system success that every time constraint be met. Example: Telecommunication Systems, Internet Video, ATM, Online reservation system.

Firm Real Time Systems

Firm real time Systems have a combination of soft deadlines and hard deadlines. First dead line is a soft deadline. If missed, it reduces efficiency of the system. Second deadline is hard. If missed; it may cause a system failure. Example: Navigation Systems in autonomous weed cutters.

Introduction to RTOS

To most people, embedded systems are not recognizable as computers. Instead, they are hidden inside everyday objects that surround us and help us in our lives. Embedded systems typically do not interface with the outside world through familiar personal computer interface devices such as a mouse, keyboard and graphic user interfaces. Instead, they interface with the outside world through unusual interfaces such as sensors, actuators and specialized communication links.

Real-time and embedded systems operate in constrained environments in which computer memory and processing power are limited. They often need to provide their services within strict time deadlines to their users and to the surrounding world. It is these memory, speed and timing constraints that dictate the use of real-time operating systems in embedded software.

Real-time and embedded operating systems are in most respects similar to general purpose operating systems. They provide the interface between application programs and the system hardware, and they rely on basically the same set of programming primitives and concepts. But general purpose operating systems make different trade-offs in applying these primitives because they have different goals.

Defining an RTOS

An Operating System that helps to build a Real Time System.

A real-time operating system (RTOS) is a key to many embedded systems today and, provides a software platform upon which to build applications. Not all embedded systems, however, are designed with an RTOS. Some embedded systems with relatively simple hardware or a small amount of software application code might not require an RTOS. Many embedded systems, however, with moderate-to-large software applications require some form of scheduling, and these systems require an RTOS. This chapter

sets the stage for all subsequent chapters in this section. It describes the key concepts upon which most real-time operating systems are based.

Components of RTOS

A real-time operating system (RTOS) is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code. For example, in some applications, an RTOS comprises only a kernel, which is the core supervisory software that provides minimal logic, scheduling, and resource-management algorithms. Although many RTOSs can scale up or down to meet application requirements, this book focuses on the common element at the heart of all RTOSs - the kernel. Most RTOS kernels contain the following components as shown in Figure 1.3:

Scheduler-is contained within each kernel and follows a set of algorithms that determines which task executes when.

Objects-are special kernel constructs that help developers create applications for real-time embedded systems. Common kernel objects include tasks, semaphores, and message queues.

Services-are operations that the kernel performs on an object or, generally operations such as timing, interrupt handling, and resource management. Common components in an RTOS kernel include objects, the scheduler and some services. This diagram is highly simplified; remember that not all RTOS kernels conform to this exact set of objects, scheduling algorithms, and services.

Examples of RTOS……

FreeRTOS : Real Time Engineers Ltd.

VxWorks : Wind River Systems

pSOS : Integrated Systems Inc

Nucleus : Accelerated Systems

WinCE : Microsoft

eCOS, VRTX, uC/OS……..

Cranes Varsity

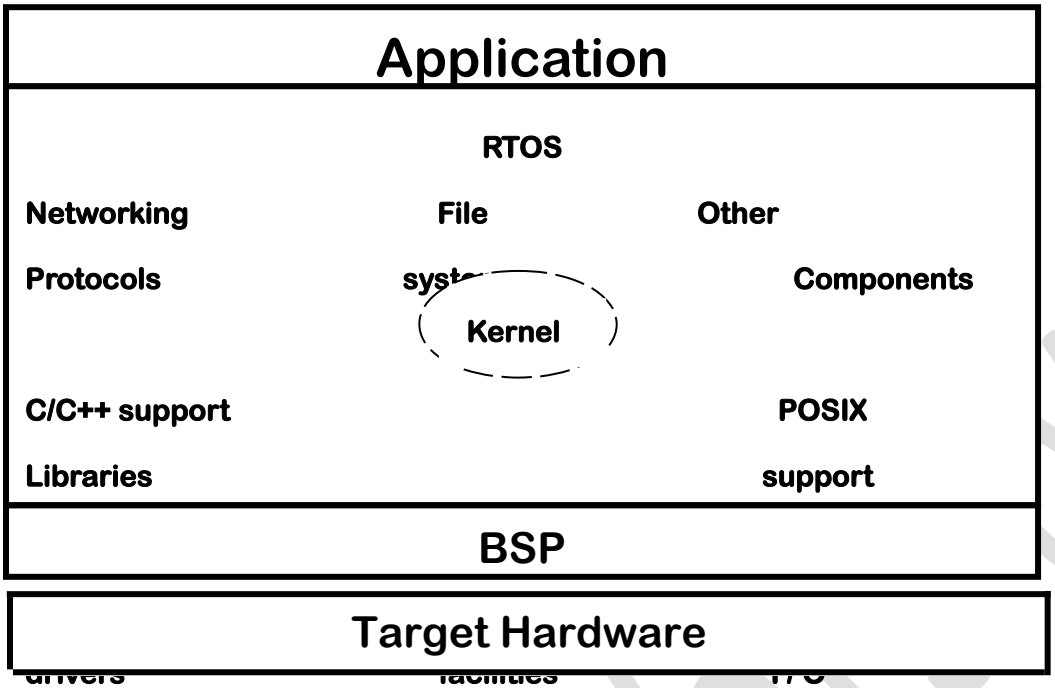


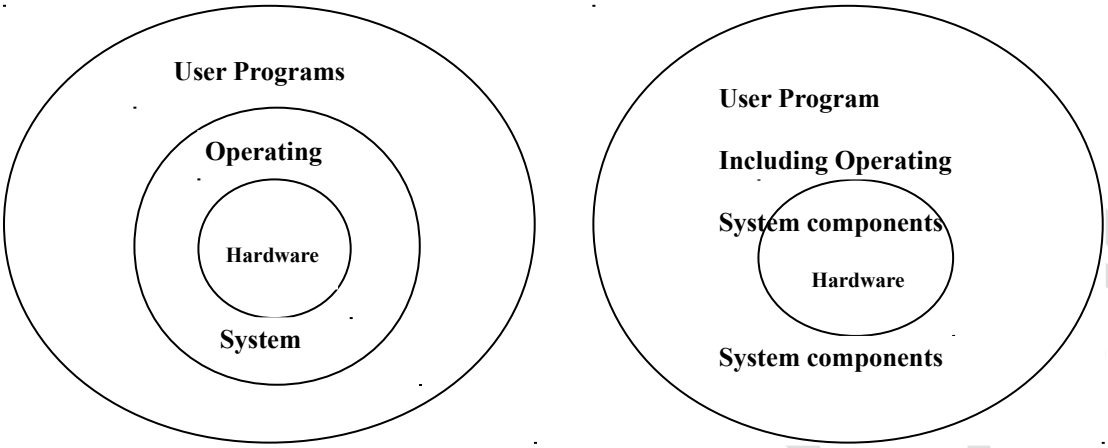
Figure 20.3 High level views of RTOS, its Kernel, and other components found in embedded system

Characteristics of Real Time Operating System

- Deterministic
- Multitasking
- Fast Context Switching
- Support for Preemptive Based Scheduling
- Support for Multiple Priority level
- Support for Inter Task Communication
- Support for Inter Task Synchronization
- Low Interrupt Latency
- Low Memory Footprint
- Scalable

Cranes Varsity

Difference between a General Purpose System Configuration and Embedded Configurations



Typical GPS Configuration **Typical Embedded Configuration.**

Figure 20.4

RTOS stands for real-time operating system. The key difference between general-computing operating systems and real-time operating systems is the need for **"deterministic"** timing behavior in the real-time operating systems. Formally, "deterministic" timing means that operating system services consume only known and expected amounts of time. In theory, these service times could be expressed as mathematical formulas.

These formulas must be strictly algebraic and not include any random timing components. Random elements in service times could cause random delays in application software and could then make the application randomly miss real-time deadlines – a scenario clearly unacceptable for a real-time embedded system. Many non-real-time operating systems also provide similar kernel services. General-computing non-real-time operating systems are often quite non-deterministic. Their services can inject random delays into application software and thus cause slow responsiveness of an application at unexpected times. If you ask the developer of a non-real-time operating system for the algebraic formula describing the timing behavior of one of its services (such as sending a message from task to task), you will invariably not get an algebraic formula. Instead the developer of the non-real-time operating system (such as Windows, UNIX or Linux) will just give you a puzzled look. Deterministic timing behavior was simply not a design goal for these general-computing operating systems.

On the other hand, real-time operating systems often go a step beyond basic determinism. For most kernel services, these operating systems offer constant load-independent timing: In other words, the algebraic formula is as simple as: T(message_send) = constant, irrespective of the length of the message to be sent, or other factors such as the numbers of tasks and queues and messages being managed by the RTOS.

There are some key functional differences that set RTOSs apart from GPOSs:

RTOS	GPOS
•Predictable	•Unpredictable
•Small Footprint –Scalability	•High Footprint
•Minimum Latency	•High Latency
•Supports a wide range of hardware platforms	• Fairness Policy

- better reliability in embedded application contexts
- the ability to scale up or down to meet application needs,
- faster performance,
- reduced memory requirements,
- scheduling policies tailored for real-time embedded systems,
- support for diskless embedded systems by allowing executables to boot and run from ROM or RAM, and
- Better portability to different hardware platforms.

Today, GPOSs target general-purpose computing and run predominantly on systems such as personal computers, workstations, and mainframes. In some cases, GPOSs run on embedded devices that have ample memory and very soft real-time requirements. GPOSs typically require a lot more memory, however, and are not well suited to real-time embedded devices with limited memory and high performance requirements.

RTOSs, on the other hand, can meet these requirements. They are reliable, compact, and scalable, and they perform well in real-time embedded systems. In addition, RTOSs can be easily tailored to use only those components required for a particular application.

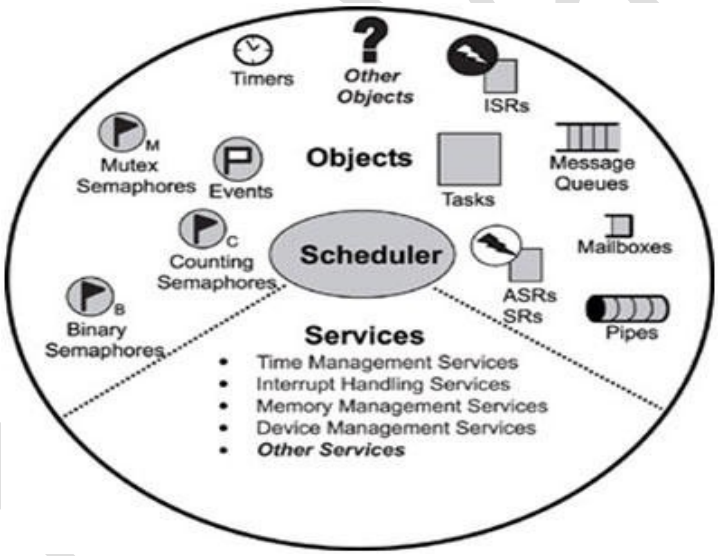
KERNEL

The kernel is the part of multitasking system responsible for the management of the tasks (i.e.., for managing the CPU's time) and communication between tasks. The fundamental services provided by the kernel is context switching. The use real-time kernel generally simplifies the design of system by allowing the application to be divided into multiple tasks managed by the kernel. A kernel adds overheads to your system because it requires extra ROM (code space) and additional RAM for the kernel data structures. But most importantly, each task requires its own stack space, which has tendency to eat up your RAM quite quickly a kernel also consume CPU time (typically 2to 5 percent).

Basic Kernel Services

In the discussion below, we will focus on the "kernel" – the part of an operating system that provides the most basic services to application software running on a processor. The "kernel" of a real-time operating system ("RTOS") provides an "abstraction layer" that hides from application software the hardware details of the processor (or set of processors) upon which the application software will run. This is shown in Figure 20.1

Figure 20.1: Basic Services Provided by a Real-Time Operating System Kernel



This Section discusses the basic responsibilities of the operating system that are relevant for this text: (i) task management and scheduling, (ii) (deferred) interrupt servicing, (iii) inter-process communication and synchronization, and (iv) memory management. General-purpose operating systems also have other responsibilities, which are beyond the horizon of a *real-time* operating system: file systems and file management, (graphical) user interaction, communication protocol stacks, disk IO, to name a few.

- **Task management and scheduling:** *Task (or "process", or "thread") management* is a primary job of the operating system: tasks must be created and deleted while the system is running; tasks can change their priority levels, their timing constraints, their memory needs; etcetera. Task management for an RTOS is a bit more dangerous than for a general purpose OS:

if a real-time task is created, it *has* to get the memory it needs without delay, and that memory *has* to be locked in main memory in order to avoid access latencies due to swapping; changing run-time priorities influences the run-time behavior of the whole system and hence also the predictability which is so important for an RTOS. So, dynamic process management is a potential headache for an RTOS.

In general, multiple tasks will be active at the same time, and the OS is responsible for sharing the available resources (CPU time, memory, etc.) over the tasks. The CPU is one of the important resources, and deciding how to share the CPU over the tasks is called "scheduling". The scheduler is at the heart of every kernel. A scheduler provides the algorithms needed to determine which task executes when. The functionalities of scheduler includes

- multitasking,
- context switching,
- dispatcher, and
- Scheduling algorithms.

General purpose and real-time operating systems differ considerably in their scheduling algorithms. They use the same basic principles, but apply them differently because they have to satisfy different performance criteria. The scheduler determines which task runs by following a scheduling algorithm (also known as scheduling policy). Most kernels today support two common scheduling algorithms:

- Priority Based Preemptive Scheduling
- Round-Robin scheduling

The RTOS manufacturer typically predefines these algorithms; however, in some cases, developers can create and define their own scheduling algorithms

- **Interrupt servicing:** An operating system must not only be able to schedule tasks according to a deterministic algorithm, but it also has to service peripheral hardware, such as timers, motors, sensors, communication devices, disks, etc. All of those can request the attention of the OS *asynchronously*, i.e., at the time that *they* want to use the OS services, the OS has to make sure it is ready to service the requests. Such a request for attention is often signaled by means of an *interrupt*. There are two kinds of interrupts:
  - ***Hardware interrupts****:* The peripheral device can put a bit on a particular hardware channel that triggers the processor(s) on which the OS runs, to signal that the device needs servicing. The result of this trigger is that the processor saves its current state, and jumps to an address in its memory space, that has been connected to the hardware interrupt at initialization time.
  - ***Software interrupts****:* Many processors have built-in software instructions with which the effect of a hardware interrupt can be generated in software. The result of a software interrupt is also a triggering of the processor, so that it jumps to a pre-specified address.
- **Communication and synchronization:** A third responsibility of an OS is commonly known under the name of *Inter-Process Communication* (IPC). ("Process" is, in this context, just another name for "task".) The general name IPC collects a large set of programming primitives that the operating system makes available to tasks that need to exchange information with other tasks, or synchronize their actions. Again, an RTOS has to make sure that this communication and synchronization take place in a deterministic way.

- **Memory management:** A fourth responsibility of the OS is *memory management*: the different tasks in the system all require part of the available memory, often to be placed on specified hardware addresses (for memory-mapped IO). The job of the OS then is (i) to give each task the memory it needs (*memory allocation*), (ii) to map the real memory onto the address ranges used in the different tasks (*memory mapping*), and (iii) to take the appropriate action when a task uses memory that it has not allocated. (Common causes are: unconnected pointers and array indexing beyond the bounds of the array.) This is the so-called *memory protection* feature of the OS.

Basic building blocks of RTOS

- Task Management (Scheduling, Dispatching, TCB, …)
- Inter – task Communication (Pipes, Message passing, …)
- Inter – task Synchronization (Semaphore)
- Interrupt and Exception handling Mechanism
- Timer
- Memory Management

Cranes Varsity

CHAPTER 21
TASK MANAGEMENT

Program, Process, Task and Thread

The usual approach to achieve better control and flexibility in an embedded application is to divide the entire task of the system into smaller, well-defined parts. The resulting parts are often called as “Process”, “Task”, or “Threads”. The distinction between these terms is unfortunately very unclear since the terms are often used interchangeably. Further depending on the target system (general computer or embedded system), the meaning of the term also change.

- Program: A program is a passive entity which consists of set of instructions/code to be executed.
- Process: A process is an active entity. It is the program under execution. It has a Control Block (PCB) associated with it.
- Thread: A Light-weighted process. Multiple threads share the shared resource as well as state information of a single process. (Ref: Bibliography indexed b.)

In RTOS, there is no difference between Process, Task and Thread (depends on implementation)

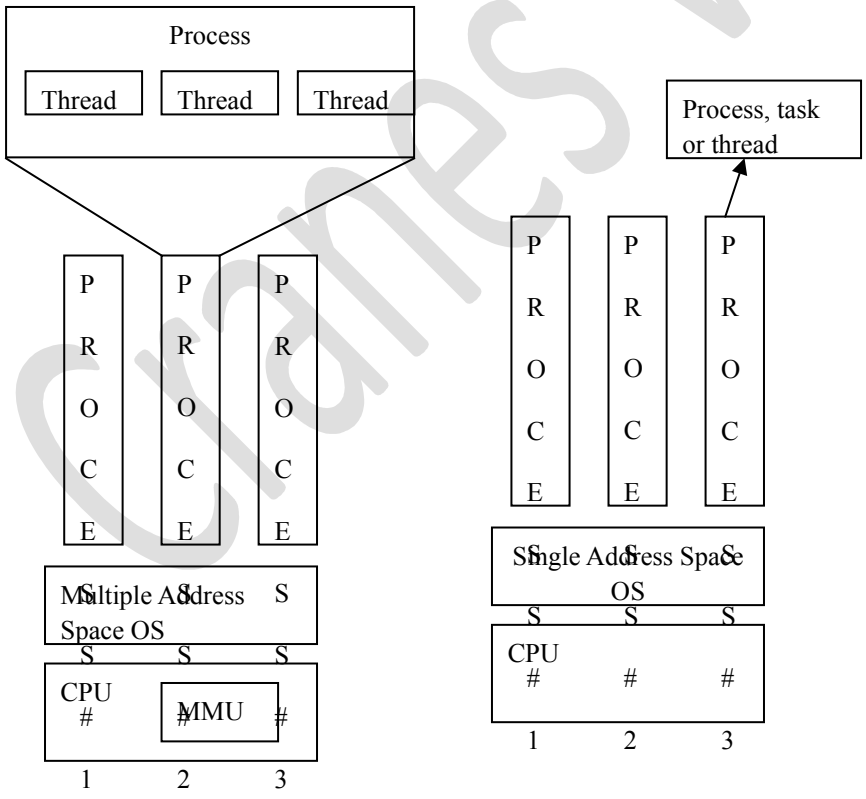


Figure 21.1

Task Internals

Three jobs performed by the kernel when there is a request for task_creation

- Set up Task Control Block(TCB)
- Set up stack
- Initialize the state of a task

Task Control Block (TCB):

Program counter
Task status
Task ID #
Content of register 0
Pointer to next TCB
: :
Contents of register n
other context

A TCB is data structure which contains everything the kernel needs to know about the task regarding.

- Name and status of the task
- Priority
- Task ID
- Remainder of time slice
- Context switching information like Task program counter, SP, CPU register, I/O assignment, and delay timer.

The size of data structure depends on the OS designer.

Stack: The *stack* is a very common data structure used in programs. By *data structure*, we mean something that is meant to *hold* data and provides certain *operations* on that data. The size of stack in RTOS is variable i.e. user has provision to mention his choice of stack size.

Task States

In RTOS, a task is usually thought of as some amount of work which is started and finished within a short period of time. Each task in an RTOS is always in one of three states as shown in Figure 4.2:

Cranes Varsity

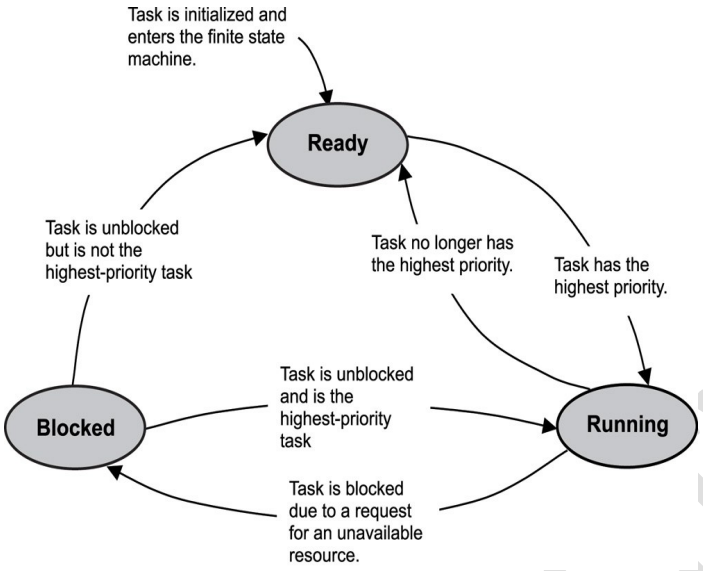


Figure 21.2: Task State Transition in RTOS

- **Ready state**-the task is ready to run but cannot because a higher priority task is executing. Any number of tasks can be in this state.
- **Blocked state**-the task has requested a resource that is not available, has requested to wait until some event occurs, or has delayed itself for some duration.
- **Running state**-the task is the highest priority task and is running.

  Transitions from one state to another are caused by system calls (suspend), by interrupts (events), and by scheduler actions (priority, time slicing).

Multitasking

Multitasking is the process of scheduling and switching the CPU between several tasks. A single CPU switches its attention between several sequential tasks. Multitasking maximize the utilization of the CPU and also provides for modular construction of applications. One of the most important aspects of multitasking is that it allows the application programmer to manage complexity inherent in real-time application. Application programs are typically easier to design and maintain if multitasking is used.

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.

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. Decomposing an application into many task leads to a concept known as Concurrent Programming. That is, simultaneous execution of tasks is known as concurrency (concurrent programming).

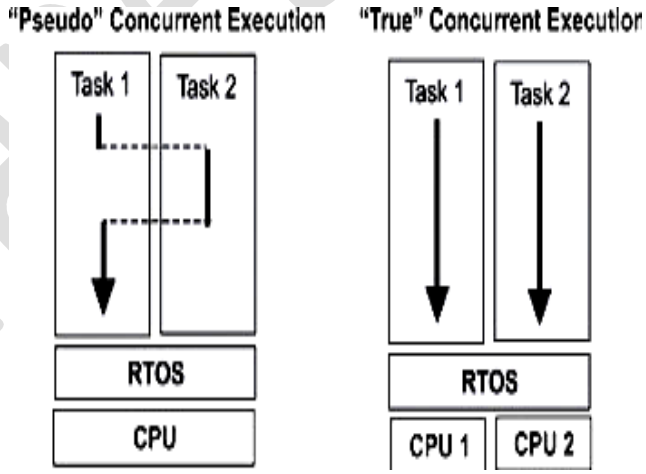
Example - The typical home computer is full of examples. In separate windows, a user runs a web browser, a text editor, and a music player all at once. The operating system interacts with each of them.

Types of Concurrency

Concurrent tasks in a real-time application can be scheduled to run on a single processor or multiple processors. Single-processor systems can achieve pseudo concurrent execution, in which an application is decomposed into multiple tasks maximizing the use of a single CPU. It is important to note that on a single-CPU system, only one *program counter* (also called an *instruction pointer)* is used and hence, only one instruction can be executed at any time. Most applications in this environment use an underlying scheduler's multitasking capabilities to interleave the execution of multiple tasks. Therefore, the term *pseudo concurrent execution* is used.

In contrast, true concurrent execution can be achieved when multiple CPUs are used in the designs of real-time embedded systems. For example, if two CPUs are used in a system, two concurrent tasks can execute in parallel at one time, as shown in Figure 22.3. This parallelism is possible because two program counters (one for each CPU) are used, which allows for two different instructions to execute simultaneously.

Figure 21.3: Multitasking (Concurrency)



Cranes Varsity

SCHEDULER

In a multitasking system, a mechanism within an OS, called a ***scheduler*** is responsible for determining the order and the duration of tasks to run on the CPU. The scheduler selects which tasks will be in what states (ready, running, or blocked), as well as loading and saving the TCB information for each task.

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. Many different kinds of task schedulers are available to software developers of embedded and real-time systems. They range from a simple cyclic executive that you can build "at home", to the many priority-based preemptive schedulers that are available commercially and beyond.

Who decides which task has to be given processor time?

Ans: Scheduler

What is a Scheduler?

It is a function which allocates the CPU time to particular task depending on the scheduling algorithm. An algorithm which determines the eligibility of a task to run is known as Scheduling Algorithm.

Hard real-time scheduling can be broadly classifies into two types: static and dynamic as shown in Figure 5.4. In static scheduling, the scheduling decisions are made at compile time. A run-time schedule is generated off-line based on the prior knowledge of task-set parameters, e.g., maximum execution times, precedence constraints, mutual exclusion constraints, and deadlines.

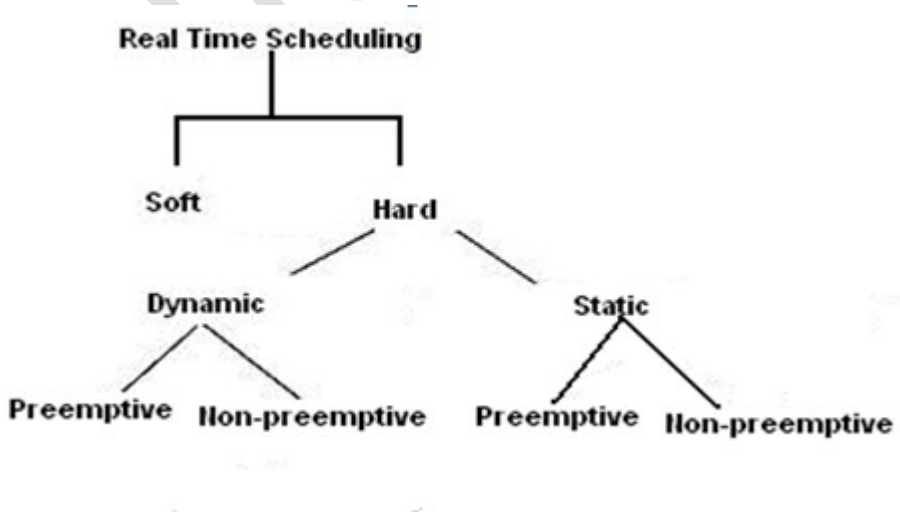


Figure 21.4

Static Scheduling Algorithms

In static scheduling, scheduling decisions are made during compile time. This assumes parameters of all the tasks is known a priori and builds a schedule based on this. Once a schedule is made, it cannot be modified online. Static scheduling is generally not recommended for dynamic systems. Applications like process control can benefit from this scheduling, where sensor data rates of all tasks are known before hand. There are no explicit static scheduling techniques except that a schedule is made to meet the deadline of the given application under known system configuration. Most often there is no notion of priority in static scheduling. Based on task arrival pattern a time line is built and embedded into the program and no change in schedules are possible during execution.

Dynamic Scheduling Algorithms

Schedulability test is often used by dynamic schedulers to determine whether a given set of ready tasks can be scheduled to meet their deadlines. Different scheduling algorithms and their schedulability criteria is explained below.

Rate Monotonic Algorithm (RMA) Rate monotonic algorithm is a dynamic preemptive algorithm based on static priorities. The rate monotonic algorithm assigns static priorities based on task periods. Here task period is the time after which the tasks repeat and inverse of period is task arrival rate. For example, a task with a period of 10ms repeats itself after every 10ms. The task with the shortest period gets the highest priority, and the task with the longest period gets the lowest static priority. At run time, the dispatcher selects the task with the highest priority for execution.

Earliest Deadline-First (EDF) ***Algorithm/*****Clock Driven Scheduling*****:** EDF algorithm is an optimal dynamic preemptive algorithm based on dynamic priorities. In this after any significant event, the task with the earliest deadline is assigned the highest dynamic priority. A significant event in a system can be blocking of a task, invocation of a task, completion of a task etc. The processor utilization can up to 100% with EDF, even when the task periods are not multiples of the smallest periods. The dispatcher operates in the same way as the dispatcher for the rate monotonic algorithm. EDF/Clock Driven algorithm schedules priorities to processes according to three parameters: ***frequency*** (number of times process is run), ***deadline*** (when processes execution needs to be completed), and ***duration*** (time it takes to execute the process).

Types of Kernel/ Schedulers

There are two types of priority based kernels,

- Non-Preemptive Kernel
- Preemptive Kernel

Cranes Varsity

Non-Preemptive Kernel

Non-preemptive kernels require that each task does something do explicitly give up control of the CPU. To maintain the illusion of concurrency; this process must be done frequently. Non-preemptive scheduling is also called cooperative multitasking. Tasks co-operate with each other to share the CPU. Asynchronous events are still handled by the ISRs. An ISR can make a higher priority task ready to run, but the ISR always returns to the interrupted task as shown in Figure 5.2. The new higher priority task will gain control of the CPU only when the current task gives up the CPU.

At the task level, non-preemptive kernel can also use non-reentrant function. Non –reentrant function can be used by each task without fear of corruption by another task. This is because each task a can run to completion before it relinquishes the CPU. Task-level response can be much lower in Non Preemptive Kernel.

Preemptive Kernel

A preemptive kernel is used when system responsiveness is of utmost important. The highest priority task ready to run is always given control of the CPU. When a task makes a higher priority task ready to run, the current tasks is preempted and the higher priority task is immediately given control of the CPU as shown in Figure 22.6. If an ISR makes a higher priority task ready, when the ISR complete the interrupted task is suspended and the new higher priority task is resumed.

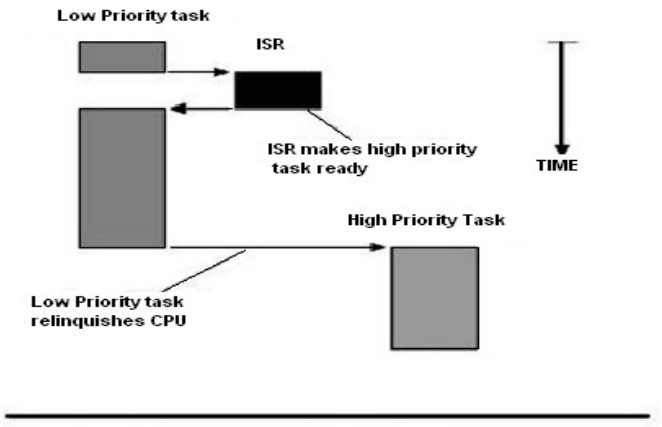


Fig 21.5: Non-Preemptive Kernel

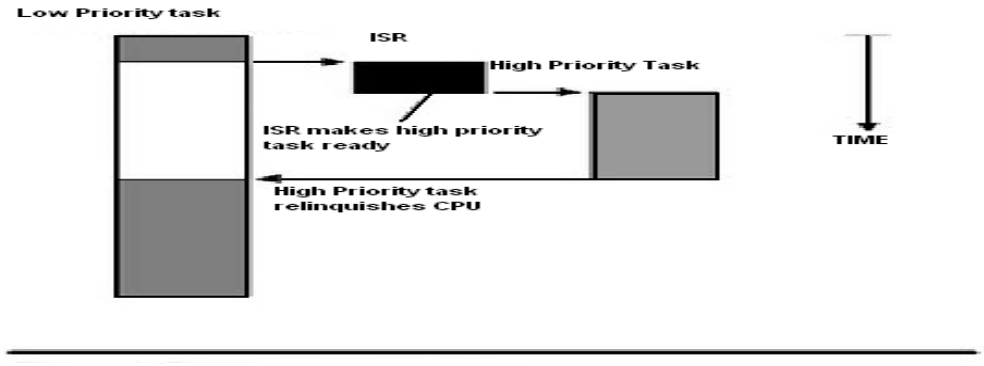


Fig 21.6: Preemptive Kernel

Most real-time kernels are preemptive. The disadvantage of preemptive kernel is that a task which can be preempted, and which can preempt others, cannot be allowed to pass information to one another by writing and reading shared data. The simple methods for passing information between tasks that worked with non-preemptive schedulers no longer work with preemptive schedulers. The problem in passing information between tasks by writing and reading shared data because if one task preempts another while the second task is in the midst of reading from a shared data table, the second task might read "old" data from part of the table and "new" data from another part of the table after the first (preempting) task writes new data into the table. This combination of old and new data might lead to erroneous results. In order to help the software developer prevent these problems, an operating system that has a preemptive scheduler should provide mechanisms for passing information between tasks (and also to/from ISRs) i. e Application code using a preemptive kernel should not use non-reentrant functions or must use an appropriate mutual exclusion method to prevent data corruption.

Getting faster response: preemptive scheduling

The schedulers which have been surveyed so far are called non-preemptive, because switching between tasks only takes place when one task has fully completed its execution and another task wants to start its execution (from its beginning). Faster response can often be obtained by going over to a "preemptive" scheduler. With a preemptive scheduler, switching between tasks can take place at any point within the execution of a task (even when the task isn't yet done executing its code).

With a preemptive scheduler, hardware devices can be either polled or interrupt-driven. Information delivered by an interrupt and acquired into software by an ISR can be immediately passed onward to a task for further processing.

Caution: preemptive schedulers bring up new issues

Preemptive schedulers offer the software developer many benefits, beyond what can be achieved with simpler "home-made" schedulers. But their sophistication brings up new issues, of which a software developer must be aware. One of these is the matter of which tasks may be preempted and which may not? The answer is to assign a priority number to each task. Tasks of higher priority can preempt tasks of lower

priority. But tasks with lower priority cannot preempt tasks of higher priority. A preemptive scheduler needs to be told the priority of each task that it can schedule.

A second issue that a software developer must consider is: tasks that can be preempted, and which can preempt others, cannot be allowed to pass information to one another by writing and reading shared data. The simple methods for passing information between tasks that worked with non-preemptive schedulers no longer work with preemptive schedulers.

If one task preempts another while the second task is in the midst of reading from a shared data table, the second task might read "old" data from part of the table and "new" data from another part of the table after the first (preempting) task writes new data into the table. This combination of old and new data might lead to erroneous results.

This is, in fact, the same problem that occurs if an ISR and a task try to communicate by writing and reading shared data, as discussed earlier. In order to help the software developer prevent these problems, an operating system that has a preemptive scheduler should provide mechanisms for passing information between tasks (and also to/from ISRs). The mechanisms provided vary in different operating systems. For example, most RTOSs provide a message queue mechanism and semaphores.

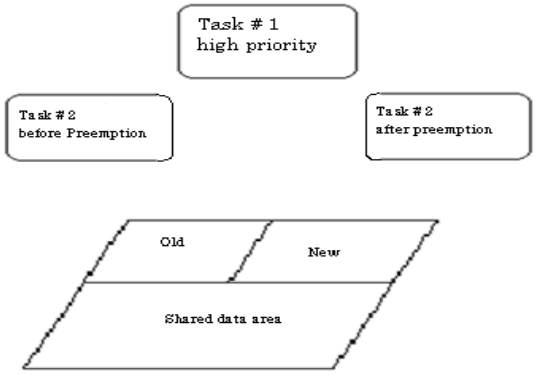


Figure 21.7: Problem in sharing data between pre-emptive tasks

Spectrum of schedulers

This has been just a short introduction to the world of task schedulers. Depending on the nature of your application and your I/O requirements, you can choose from a wide spectrum of schedulers. They range from a simple cyclic executive that you can build "at home", to the many full-featured, priority-based preemptive schedulers available commercially, and to even more sophisticated schedulers.

Context Switch

Each task has its own context, which is the state of the CPU registers required each time it is scheduled to run. A context switch occurs when the scheduler switches from one task to another. To better understand what happens during a context switch; let's examine further what a typical kernel does in this scenario.

We know that every time a new task is created, the kernel also creates and maintains an associated task control block (TCB). TCBs are system data structures that the kernel uses to maintain task-specific information. TCBs contain everything a kernel needs to know about a particular task. When a task is running, its context is highly dynamic. This dynamic context is maintained in the TCB. When the task is not running, its context is frozen within the TCB, to be restored the next time the task runs. A typical context switch scenario is illustrated in Figure 22.8

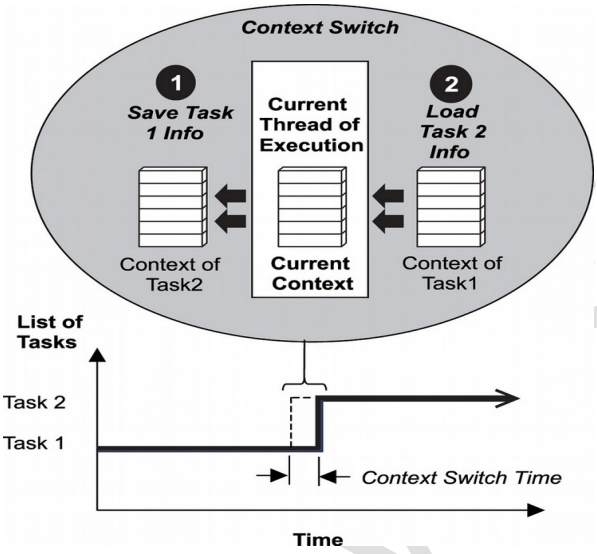


Figure 21.8: Context Switch

When the kernel's scheduler determines that it needs to stop running task 1 and start running task 2, it takes the following steps:

1. The kernel saves task 1's context information in its TCB.
2. It loads task 2's context information from its TCB, which becomes the current thread of execution.
3. The context of task 1 is frozen while task 2 executes, but if the scheduler needs to run task 1 again, task 1 continues from where it left off just before the context switch. The process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching.

The time it takes for the scheduler to switch from one task to another is the context switch time (context switch latency). If an application's design includes frequent context switching, however, the application can incur unnecessary performance overhead. Therefore, design applications in a way that does not involve excess context switching.

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. During context switching, the contents of the task are pushed into stack; basically it is the pointer to tcb (task control block) that is pushed into stack. The Task Control Block is a structure declared in kernel's header file. Some of the contents of structure are

backward link	forward link	return value from last system call	suspend value

task priority	Errno	stack frame pointer	stack pointer when suspended
stack segment	pointer to task's main function	hook_entry pointer to entry routine, if HOOKED	hook_exit pointer to exit routine, if HOOKED
stack top pointer (end stack grows toward)	stack bottom pointer (end stack starts from)	stack size	name task name (string pointer)

Critical Section (CS)\ Atomic Section

Critical section is a code segment, during the execution of which, if preempted may lead to instability of the system. E.g.: The Critical section problem occurs when the processors are in a Network. For e.g. consider that 3 processors share same database at a time, if one processor wants to update the database then no other process will get the access to that database. This point is called as critical section. Temporarily the connection between the other 2 systems will be dropped.

The scheduler is suspended/disabled while a task is executing a atomic section, to complete the execution without preemption.

Implementation of Critical Section

Use "Enter Critical/Atomic Section" API provided by the RTOS to enter into the critical/atomic section

```
E.g.: EnterCS ()
{
  DisableScheduler ();
  Return;
}
```

Use "Exit Critical/Atomic Section" API provided by the RTOS to come out of the critical section/atomic section

```
E.g.: ExitCS ()
{
  EnableScheduler ();
  Return;
}
```

Therefore, Place the code between these two constructs.

Cranes Varsity

Design constraints of Critical Section

- Keep critical/atomic sections short in execution time
- Every entry into critical/atomic section should have a corresponding exit because RTOS will never signal the end of the critical section. If there is no corresponding exit statement ,it may cause the blocking tasks to block on that critical section for ever
- Never make any scheduler dependent calls within an Atomic Section

Common Task Management Routines

- Create task
- Delete task
- Suspend task
- Resume task
- Change the priority at run time
- Stop task
- Change the scheduling policy
- Set time slice

Optimal Task Decomposition

Decomposition or splitting the application into smaller, well defined parts (tasks) is important to the overall performance of the systems. Decomposition increases the amount of concurrency and improves the overall response time. However, there is a cost associated with it. Every task has its own TCB and stack. As a result, memory consumption increases as number of tasks increase. There is also a price to be paid in terms of latency. More the number of tasks more will be the number of context switches in the system. As there is a latency associated with a context switch, this can adversely affect the response time of the application. Hence optimal task decomposition (i.e. splitting the application into correct number of tasks) is critical to system performance.

Cranes Varsity

Introduction to FreeRTOS

FreeRTOS is solely owned, developed and maintained by Real Time Engineers Ltd.

The unprecedented global success of FreeRTOS comes from its compelling value proposition; FreeRTOS is professionally developed, strictly quality controlled, robust, supported, does not contain any intellectual property ownership ambiguity, and is truly free to use in commercial applications without any requirement to expose your proprietary source code. You can take a product to market using FreeRTOS without even talking to Real Time Engineers ltd., let alone paying any fees.

FreeRTOS Features

- Pre-emptive or co-operative operation
- Very flexible task priority assignment
- Flexible, fast and light weight task notification mechanism
- Queues
- Binary semaphores
- Counting semaphores
- Mutexes
- Recursive Mutexes
- Software timers
- Event groups
- Tick hook functions
- Idle hook functions
- Stack overflow checking
- Trace recording
- Task run-time statistics gathering

FreeRTOS is a real-time kernel (or real-time scheduler) on top of which embedded applications can be built to meet their hard real-time requirements. It allows applications to be organized as a collection of independent threads of execution. On a processor that has only one core, only a single thread can be executing at any one time. The kernel decides which thread should be executing by examining the priority assigned to each thread by the application designer. In the simplest case, the application designer could assign higher priorities to threads that implement hard real-time requirements, and lower priorities to threads that implement soft real-time requirements. This would ensure that hard real-time threads are always executed ahead of soft real-time threads, but priority assignment decisions are not always that simplistic.

The Official FreeRTOS Distribution

FreeRTOS is distributed in a single zip file. The zip file contains source code for all the FreeRTOS ports, and project files for all the FreeRTOS demo applications. It also contains a selection of FreeRTOS+ ecosystem components, and a selection of FreeRTOS+ ecosystem demo applications.

FreeRTOS is ideally suited to deeply embedded real-time applications that use microcontrollers or small microprocessors. This type of application normally includes a mix of both hard and soft real-time requirements

Building FreeRTOS

Cranes Varsity

FreeRTOS can be thought of as a library that provides multi-tasking capabilities to what would otherwise be a bare metal application. FreeRTOS is supplied as a set of C source files. Some of the source files are common to all ports, while others are specific to a port. Build the source files as part of your project to make the FreeRTOS API available to your application.

FreeRTOSConfig.h

FreeRTOS is configured by a header file called FreeRTOSConfig.h. FreeRTOSConfig.h is used to tailor FreeRTOS for use in a specific application. For example, FreeRTOSConfig.h contains constants such as configUSE_PREEMPTION, the setting of which defines whether the co-operative or pre-emptive scheduling algorithm will be used1. As FreeRTOSConfig.h contains application specific definitions, it should be located in a directory that is part of the application being built, not in a directory that contains the FreeRTOS source code.A demo application is provided for every FreeRTOS port, and every demo application contains a FreeRTOSConfig.h file. It is therefore never necessary to create a FreeRTOSConfig.h file from scratch. Instead, it is recommended to start with, then adapt, the FreeRTOSConfig.h used by the demo application provided for the FreeRTOS port in use.

FreeRTOS Source Files Common to All Ports

```
FreeRTOS
    └─Source
        ├─tasks.c          FreeRTOS source file - always required
        ├─list.c           FreeRTOS source file - always required
        ├─queue.c          FreeRTOS source file - nearly always required
        ├─timers.c         FreeRTOS source file - optional
        ├─event_groups.c   FreeRTOS source file - optional
        └─croutine.c       FreeRTOS source file - optional
```

FreeRTOS Source Files Specific to a Port

Source files specific to a FreeRTOS port are contained within the

FreeRTOS/Source/portable directory.

The portable directory is arranged as a hierarchy, first by compiler, then by processor architecture.

Task Management and Scheduling in FreeRTOS

Tasks in FreeRTOS

Tasks in FreeRTOS are implemented as C functions. They return void and take a void pointer as a parameter. The prototype of task function in FreeRTOS is as shown:

```
void  aTaskFunction(void *vpParameter);
```

The task function will be typically implemented as an infinite loop and will run for ever. FreeRTOS tasks must not be allowed to return from their implementing function. They must not contain a 'return' statement and must not be allowed to execute past the end of the function. If a task is no longer required, it should instead be explicitly deleted. The structure of a typical task is shown below.

```
void  aTaskFunction(void *vpParametrs)
{
    for( ;  ;)
    {
        /*Code to implement task functionality */
    }
    vTaskDelete(NULL);   /* self delete if task exits from loop */
}
```

Task State Transitions

An application can consist of many tasks. If the processor running the application contains a single core, then only one task can be executing at any given time. This implies that a task can exist in one of two states, Running and Not Running. Fig 22.1 shows a simplified state transition diagram for FreeRTOS tasks

Cranes Varsity

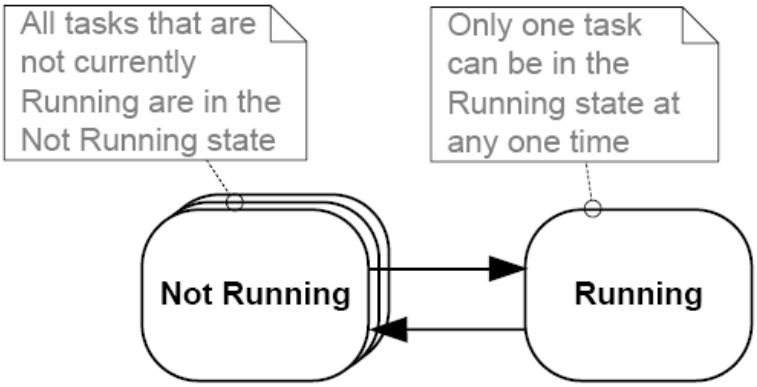


Figure 21.1: Top Level Task State Transition

The 'Not Running' state actually consists of a number of sub states. Figure 22.2 shows the 'Not Runing' state expanded into the sub states – Suspended, Ready and Blocked.

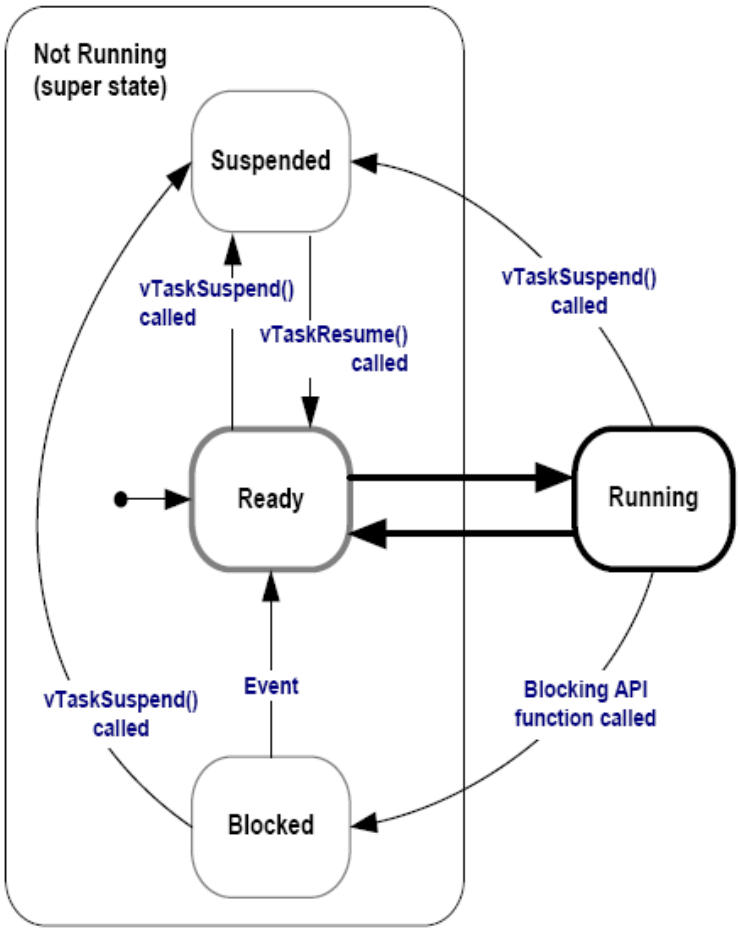


Figure 21.2 : The Full Task State Machine

The Blocked State

A task that is waiting for an event is said to be in the 'Blocked' state. Tasks can enter the Blocked state to wait for two different types of event:

1. **Temporal (time-related) events**—the event being either a delay period expiring, or an absolute time being reached. For example, a task may enter the Blocked state to wait for 10 milliseconds to pass.

2. **Synchronization events**—where the events originate from another task or interrupt. For example, a task may enter the Blocked state to wait for data to arrive on a queue. Synchronization events cover a broad range of event types.

The Suspended State Tasks in the Suspended state are not available to the scheduler. The only way into the Suspended state is through a call to the vTaskSuspend() API function, the only way out being through a call to the TaskResume() or xTaskResumeFromISR() API functions.

Most applications do not use the Suspended state.

The Ready State Tasks that are in the Not Running state but are not Blocked or Suspended are said to be in the Ready state. They are able to run, and therefore 'ready' to run, but are not currently in the Running state.

The Idle Task

There must always be **at least one task** that can enter the Running state. To ensure this is the case, an Idle task is automatically created by the scheduler at the beginning when the multitasking environment is initiated (by calling vTaskStartScheduler()). The idle task, like the other tasks is also always implemented as an infinite loop -so it is always able to run.

The idle task has the lowest possible priority (priority zero). Running at the lowest priority ensures the idle task is transitioned out of the Running state as soon as a higher priority task enters the Ready state.

Task Priorities

The maximum number of priorities available is set by the application-defined configMAX_PRIORITIES compile time configuration constant within FreeRTOSConfig.h. Low numeric priority values denote low-priority tasks, with priority 0 being the lowest priority possible. Therefore, the range of available priorities is 0 to (configMAX_PRIORITIES – 1). Any number of tasks can share the same priority—ensuring maximum design flexibility.

It is advisable to keep configMAX_PRIORITIES at the minimum necessary, as the higher its value, more the RAM consumed.

Scheduling in FreeRTOS

Cranes Varsity

The scheduling algorithm is the software routine that decides which Ready state task to transition into the Running state. The Scheduling schemes available in FreeRTOS are:

Prioritized Preemptive Scheduling with Time Slicing

Always the highest priority task in Ready state is selected for transition into running State. A running task is pre-empted if a higher priority task enters the Ready state. If the running task enters the Blocked or Suspended state, the highest priority task in Ready state is transitioned to running state. Time Slicing is used among tasks of same priority

1. **Prioritized Preemptive scheduling without Time Slicing**
   Prioritized Preemptive Scheduling without time slicing maintains the same task selection and pre-emption algorithms as described above, but does not use time slicing to share processing time between tasks of equal priority.

2. **Co-operative Scheduling**
   When the co-operative scheduler is used, a context switch will only occur when the Running state task enters the Blocked state, or the Running state task explicitly yields (manually requests a re-schedule) by calling taskYIELD(). Tasks are never pre-empted, so time slicing cannot be used.

The algorithm can be selected by setting the macros configUSE_PREEMPTION and configUSE_TIME_SLICING to appropriate values. Both constants are defined in FreeRTOSConfig.h.

Configuration parameters for scheduling algorithm.

Macros defined in FreeRTOSConfig.h ↓	**Values for (1) Prioritized preemptive scheduling with TimeSlicing**	**Values for (2) Prioritized preemptive scheduling without TimeSlicing**	**Values for (3) Co-operative Scheduling**
configUSE_PREEMPTION	**1**	**1**	**0**
configUSE_TIME_SLICING	**1**	**0**	**Any value**

Time Slicing : Pros and Cons

There are fewer task context switches when time slicing is not used than when time slicing is used. Therefore, turning time slicing off results in a reduction in the scheduler's processing overhead. However, turning time slicing off can also result in tasks of equal priority receiving greatly different amounts of processing time. For this reason, running the scheduler without time slicing is considered an advanced technique that should only be used by experienced users.

CHAPTER 22

Cranes Varsity

INTER TASK SYNCHRONIZATION

Semaphores

Semaphores are considered to be kernel synchronization objects. It synchronizes multiple threads for access to critical resource. Semaphore operations can change a task's state, fast and atomic. There are 3 types of semaphore: Binary, Mutex and Counting semaphores.

Binary Semaphores

Binary semaphores can have initial state 0 or 1 as shown in fig 22.1. Binary semaphore with 0 states is used for synchronization and with 1 state is used for providing mutual exclusion of shared resources. For *synchronization*, semaphores coordinate a task's execution with external events. For *mutual exclusion;* semaphores interlock access to shared resources. They provide mutual exclusion with finer granularity than either interrupt disabling or preemptive locks.

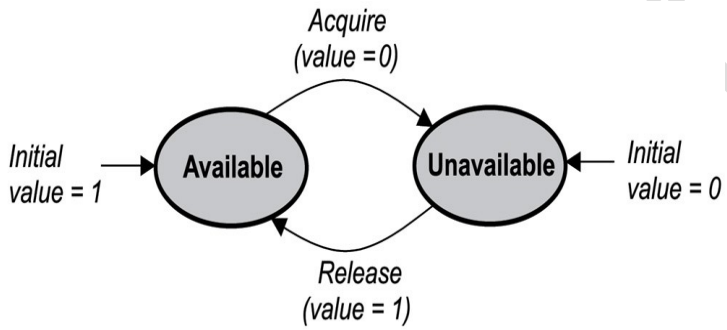


Figure 22.1: Binary Semaphore

A binary semaphore can be viewed as a flag that is available (full) or unavailable (empty). When a task takes a binary semaphore, the outcome depends on whether the semaphore is available (full) or unavailable (empty) at the time of the call. If the semaphore is available (full), the semaphore becomes unavailable (empty) and the task continues executing immediately. If the semaphore is unavailable (empty), the task is put on a queue of blocked tasks and enters a state of pending on the availability of the semaphore.

When a task gives a binary semaphore, the outcome also depends on whether the semaphore is available (full) or unavailable (empty) at the time of the call. If the semaphore is already available (full), giving the semaphore has no effect at all. If the semaphore is unavailable (empty) and no task is waiting to take it, then the semaphore becomes available (full). If the semaphore is unavailable (empty) and one or more tasks are pending on its availability, then the first task in the queue of blocked tasks is unblocked, and the semaphore is left unavailable (empty).

Synchronization

When used for task synchronization, a semaphore can represent a condition or event that a task is waiting for. Initially the semaphore is unavailable (empty). A task or ISR signals the occurrence of the

event by giving the semaphore .Another task waits for the semaphore by an attempt to lock the semaphore. The waiting task blocks until the event occurs and the semaphore is given.

Mutual exclusion

Binary semaphores interlock access to a shared resource efficiently. Unlike disabling interrupts or preemptive locks, binary semaphores limit the scope of the mutual exclusion to only the associated resource. In this technique, a semaphore is created to guard the resource. Initially the semaphore is available (full). When a task wants to access the resource, it must first take that semaphore. As long as the task keeps the semaphore, all other tasks seeking access to the resource are blocked from execution. When the task is finished with the resource, it gives back the semaphore, allowing another task to use the resource.

Mutex Semaphores

Initial state is always 1.It is a special binary semaphore in which the task who has locked the mutex can only release the lock. It can be used only for mutual exclusion.

Counting Semaphores

Counting Semaphores can be initialized with account greater than 0.They are used to restrict the access to a shared resource. Every time a semaphore is given, the count is incremented. Every time a semaphore is taken, the count is decremented. When the count reaches zero, a task that tries to take the semaphore is blocked. As with the binary semaphore, if a semaphore is given and a task is blocked, it becomes unblocked. However, unlike the binary semaphore, if a semaphore is given and no tasks are blocked, then the count is incremented. This means that a semaphore that is given twice can be taken twice without blocking. Counting semaphores are useful for guarding multiple copies of resources. For example, the use of five tape drives might be coordinated using a counting semaphore with an initial count of 5, or a ring buffer with 256 entries might be implemented using a counting semaphore with an initial count of 256.

Deadlock

Task A has locked a resource and it is blocked, waiting for a resource that is locked by task B. However, task B will be blocked waiting for the resource that is locked by task A. This situation is termed as Deadlock. Deadlock can arise during circular dependency. For example, as shown in fig 7.2, task A is waiting for task B to release a lock, task B is waiting for task C to release a lock, and task C is waiting for task A. As soon as one of these four conditions is not satisfied, a deadlock can not occur. Moreover, these conditions are not sufficient for deadlocks to occur they just describe the conditions under which it is possible to have deadlocks.

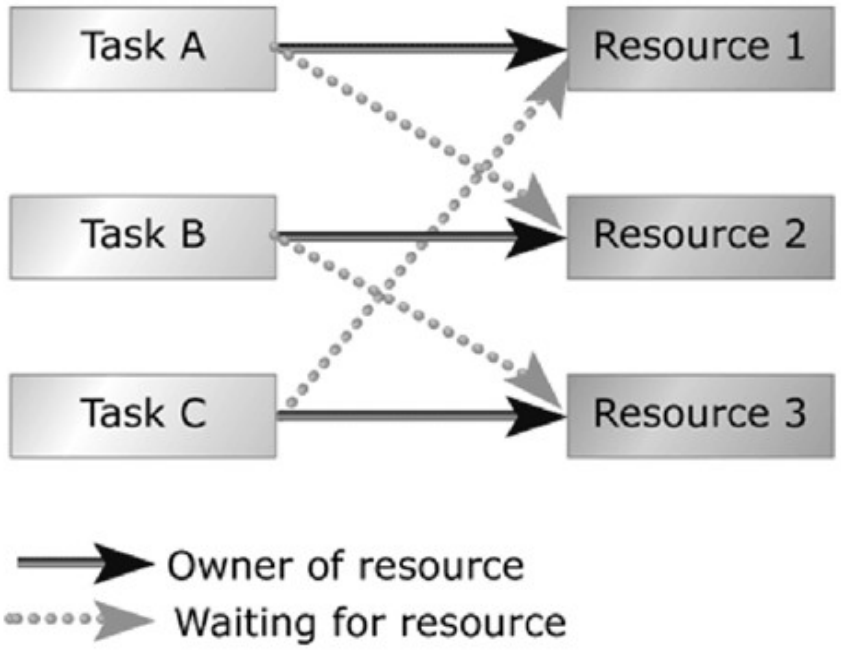


Figure 22.2 Deadlock

There are some guaranteed deadlock avoidance algorithms, which are reasonably simple to implement. For example, a deadlock cannot occur if all programs always take locks in the same order (sequential order). Other prevention algorithms use some of the following approaches: Only allow each task to hold one resource; pre-allocate resources; force release of a resource before a new request can be made; ordering all tasks and give them priority according to that order.

Priority Inversion

Priority Inversion is a situation in which the high priority task is made to wait for an indefinite amount of time to get a resource from low priority task as shown in fig 22.4.

Priority inversion may be bounded or unbounded. In bounded priority inversion, the waiting time for high priority task is limited to the time taken by low priority task to complete its execution with shared resource.

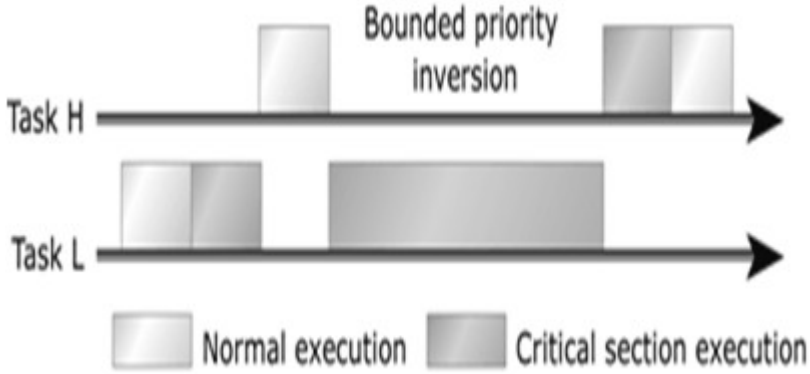


Figure 22.3 : Bounded Priority Inversion

In unbounded priority inversion, the high priority task is made to wait for indefinite amount of time to get a resource from low priority task .This occurs due to the unexpected entry of a one or more middle priority tasks which does not access the critical section as shown fig 22.4.

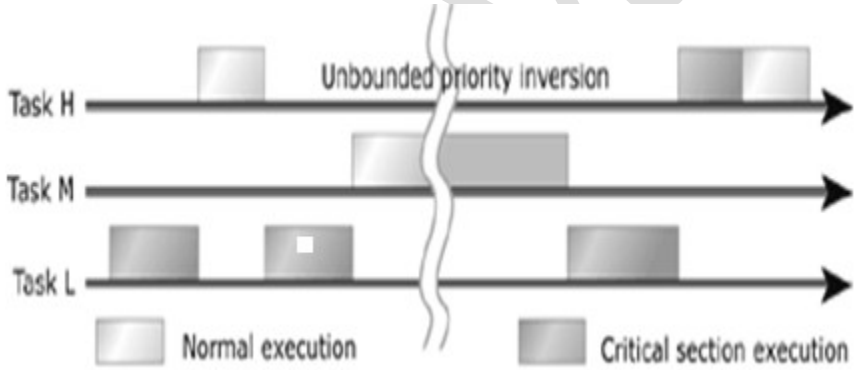


Figure 22.4 Unbounded Priority Inversion

A popular method for controlling Priority Inversion is Priority **Inheritance**

Priority Inheritance

When a job blocks one or more high priority jobs, it ignores its original priority assignment and executes its critical section at the highest priority level of all the jobs it blocks. i.e., the low priority task inherits the priority of High priority task which is blocked for the resource and executes the critical section as shown in fig 22.5

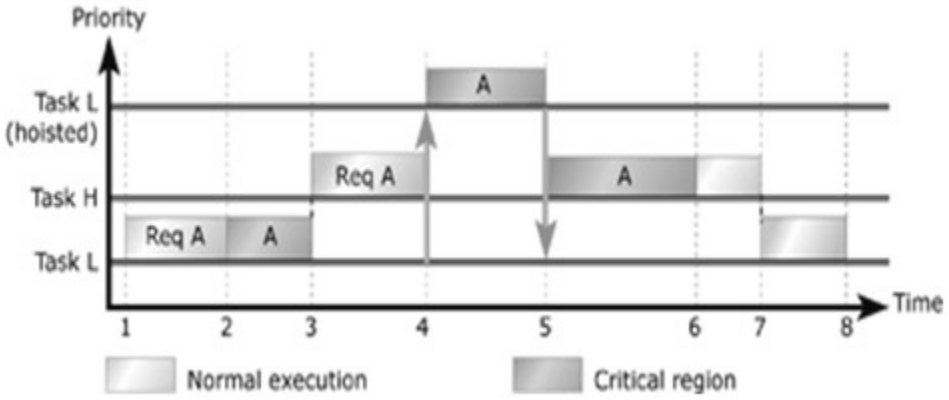


Figure 22.5 : Priority Inheritance

Semaphores in FreeRTOS

FreeRTOS provides a rich set of API for creating and handling binary, mutex and counting semaphores. Some of them are listed below.

The APIs for creating the semaphores are:

1. xSemaphoreCreateBinary(void);
2. xSemaphoreCreateCounting(UBaseType_t uxMaxCt, UBaseType_t uxInitialCt **);**
3. xSemaphoreCreateMutex(void);

The functions return a semaphore handle whose type (OS defined) is *SemaphoreHandle_t* using which the semaphore can be referenced.

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for these functions to be available.

Dynamic Allocation

Each semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a semaphore is created using the above APIs then the required RAM is automatically allocated dynamically from the FreeRTOS heap.

Static Allocation

Memory for the semaphores can also be allocated statically by using another set of API for semaphore creation.

1. xSemaphoreCreateBinaryStatic()
2. xSemaphoreCreateCountingStatic()
3. xSemaphoreCreateMutexStatic()

Cranes Varsity

Please refer to FreeRTOS user manual for more details.

Priority inheritance is actually the only difference between a binary semaphore and a mutex This mechanism helps against priority inversion phenomenon although it doesn't absolutely prevent it from happening.

Semaphores can be Taken (obtained) or Given (released) using the APIs

1. xSemaphoreTake()
2. xSemaphoreGive()

These are common to all the 3 semaphore types.

Recursive Mutexes

It is also possible for a task to deadlock with itself. This will happen if a task attempts to take the same mutex more than once, without first returning the mutex.

Consider the following scenario:

1. A task successfully obtains a mutex.
2. While holding the mutex, the task calls a library function.
3. The implementation of the library function attempts to take the same mutex, and enters the Blocked state to wait for the mutex to become available.

At the end of this scenario the task is in the Blocked state to wait for the mutex to be returned, but the task is already the mutex holder. A deadlock has occurred because the task is in the Blocked state to wait for itself. This type of deadlock can be avoided by using a recursive mutex in place of a standard mutex. A recursive mutex can be 'taken' more than once by the same task, and will be returned only after one call to 'give' the recursive mutex has been executed for every preceding call to 'take'

In FreeRTOS a recursive mutex can be created and used with the following APIs.

xSemaphoreCreateRecursiveMutex()
xSemaphoreTakeRecursive()
xSemaphoreGiveRecursive()

Cranes Varsity

CHAPTER 23
INTER TASK COMMUNICATION

There are different methods by which tasks communicate with each other. The common methods available are

- Shared memory
- Message queues
- Pipes
- Signals
- Sockets

Shared Memory

Shared-memory objects are a class of system objects that can be accessed by tasks running on different processors. They are called *shared-memory* objects because the object's data structures must reside in memory accessible by all processors. The shared data should only be accessed through certain routines. There is a chance of data corruption to occur in shared memory. So use Semaphores to synchronize access. It is the fastest method ITC as less time is spent in packet switching.

Message Queue

Message Queues are used to send “information” from one task to another. It can be implemented either as a Message Queue or as a mailbox. Mailbox is a special case message queue of length 1. Mailbox can be used as a synchronization object, if semaphores are not available in the OS.

Pipes

Pipes are virtual I/O devices and are built on top of message queues. Messages of varying lengths can be written to pipes. Tasks can use the standard I/O routines to open (), read (), and write () pipe functions, and invoke ioctl () routines. ISRs can write to a pipe, but should not read from a pipe. Select facility can be implemented using pipes.

Signals

Signal is a software analog of an interrupt. It alters the flow control of tasks by communicating asynchronous events within or between task contexts. It is sent to a task to indicate the occurrence of some asynchronous event. A task can attach a signal handler to take appropriate action on receipt of the signal. It can be raised by a task or an ISR. The task being signaled will immediately suspend its current thread of execution and invokes a task specified "signal handler" routine.

Message Passing in FreeRTOS

Message Queues are used in FreeRTOS to provide a task-to-task, task-to-interrupt, and interrupt-to-task communication mechanism.

A queue can hold a finite number of fixed size data items. The maximum number of items a queue can hold is called its ‘length’. Both the length and the size of each data item are set when the queue is created.

Cranes Varsity

Queues are normally used as First In First Out (FIFO) buffers, where data is written to the end (tail) of the queue and removed from the front (head) of the queue.

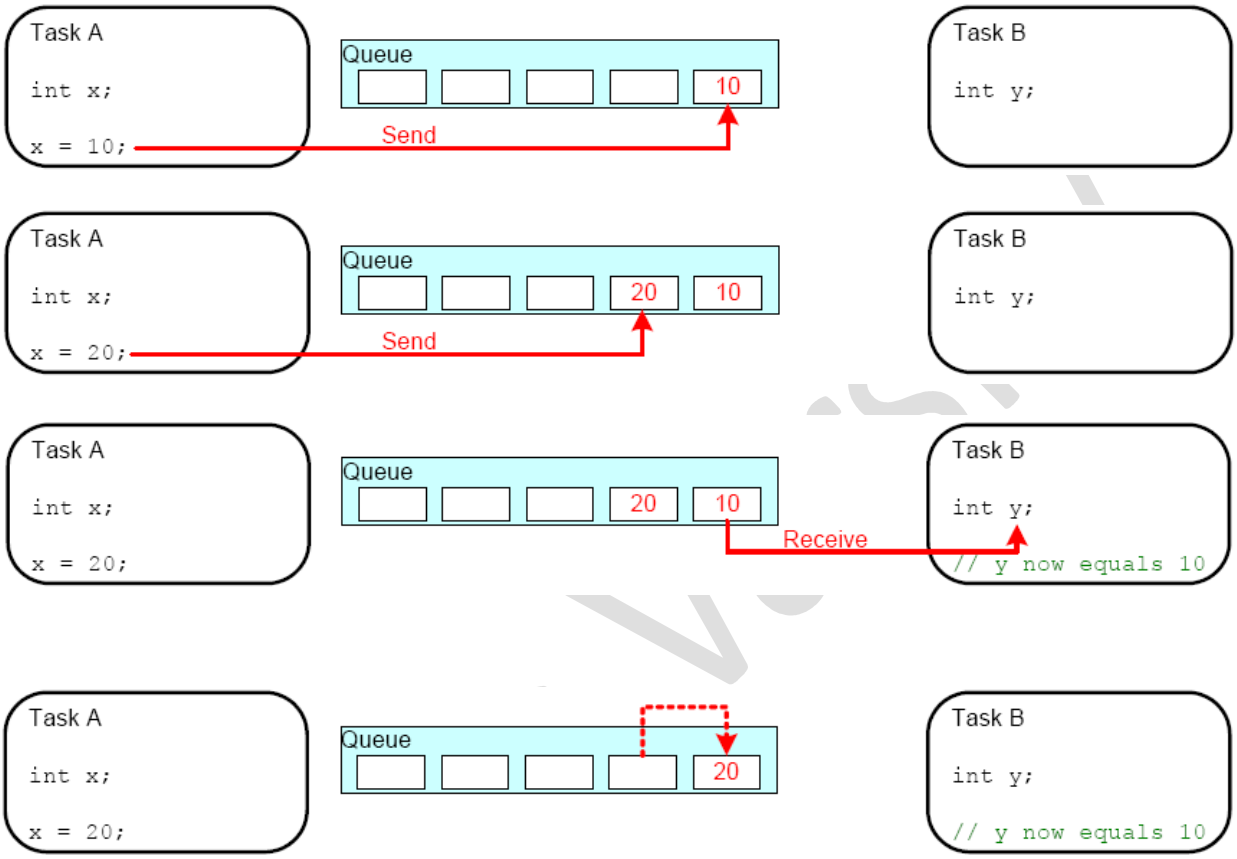


Figure 6.1 : A sequence of writes and a read happening through a queue of length 5

The function prototype for the API used to create a queue is:

QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);

Where *uxQueueLength* specifies the maximum number of messages/data-items the q can hold at a given time and *uxItemSize* specifies the maximum size of each data-item.
As can be seen, once a message is read, it is removed from the queue.

Access by Multiple Tasks

Queues are objects in their own right that can be accessed by any task or ISR that knows of their existence. Any number of tasks can write to the same queue, and any number of tasks can read from the same queue. In practice it is very common for a queue to have multiple writers, but much less common for a queue to have multiple readers.

Working with large data

Queuing Pointers

If the size of the data being stored in the queue is large, then it is preferable to use the queue to transfer pointers to the data, rather than copy the data itself. However, when queuing pointers, extreme care must be taken to ensure that:

1. The owner of the RAM being pointed to is clearly defined. When sharing memory between tasks via a pointer, it is essential to ensure that both tasks do not modify the memory contents simultaneously, or take any other action that could cause the memory contents to be invalid or inconsistent. Ideally, only the sending task should be permitted to access the memory until a pointer to the memory has been queued, and only the receiving task should be permitted to access the memory after the pointer has been received from the queue.

2. The RAM being pointed to remains valid. If the memory being pointed to be allocated dynamically, or obtained from a pool of pre-allocated buffers, then exactly one task should be responsible for freeing the memory. No tasks should attempt to access the memory after it has been freed. A pointer should never be used to access data that has been allocated on a task stack. The data will not be valid after the stack frame has changed.

Cranes Varsity

MEMORY MANAGEMENT IN RTOS

Memory

Dynamic memory allocation is important in terms of both the use of on-demand memory by application tasks and the requirements of the operating system. Application tasks use memory explicitly for example, through requests for heap memory, and implicitly through the maintenance of the run-time memory needed to support sophisticated high-order languages. The operating system (or kernel) needs to perform extensive memory management in order to keep the tasks isolated. Dangerous allocation of memory is any allocation that can preclude system determinism. Dangerous allocation can destroy event determinism; for example, by overflowing the stack or it can destroy temporal determinism by entering a deadlock situation. It is important to avoid dangerous allocation of memory while at the same time reducing the overhead incurred by memory allocation. This overhead is a standard component of the context switch time and must be minimized.

Memory is a premium resource in most embedded Systems. Most RTOS's assume a flat memory model. Memory is allocated as a fixed block to each instance of process. The memory management in a RTOS is usually simple and fast. At the same time, it can lead to a lot of fragmentation if memory usage is not properly structured

Common Address Space

This model of memory management is followed by RTOS to increase the predictability 0f operating systems. In this model, there is no distinct User Address Space and Kernel Address Space. Instead every task runs in a Common Address Space. Since every routine runs in supervisory mode, there is no Mode Switching which saves considerable amount of time in Real Time Systems. This also makes inter task communication faster and simpler. But with great powers comes great responsibility. A deviant task can corrupt the whole system.

Flat Memory Model

RTOS assumes a Flat Memory Model or Linear Memory Model rather than a Virtual Memory Model by providing a linear mapping of addresses. The address locations referred by instructions are all physical addresses (addresses in RAM) .This increases the predictability of RTES by avoiding virtual address translations, demand paging, swap in-swap out etc. This makes context switching faster by reducing the amount of context that has to be stored and restored on each context switching. In Virtual Memory Model, Page Map is also a part of Process Context.

In a general purpose OS, a process has its own memory locations for data but shares text with all other processes executing the same code. In RTOS all tasks reside in a common address space.

- **Advantages:**
  – Makes intertask communication fast and simple.

  – Makes context switch faster (does not need to save and restore virtual address contexts and swap in/out).

Cranes Varsity

- **Disadvantages:**
  – A bad task can corrupt other tasks.

  – Any non local variable is global across the tasks.

Memory Management in FreeRTOS

In FreeRTOS kernel objects are not statically allocated at compile-time, but dynamically allocated at run-time; FreeRTOS allocates RAM each time a kernel object is created, and frees RAM each time a kernel object is deleted. This policy reduces design and planning effort, simplifies the API, and minimizes the RAM footprint.

Memory can be allocated using the standard C library malloc() and free() functions, but they may not be always suitable, for one or more of the following reasons:

- They are not always available on small embedded systems.
- Their implementation can be relatively large, taking up valuable code space.
- They are rarely thread-safe.
- They are not deterministic; the amount of time taken to execute the functions will differ from call to call
- They can suffer from fragmentation.
- They can complicate the linker configuration.
- They can be the source of difficult to debug errors if the heap space is allowed to grow into memory used by other variables.

Different embedded systems have varying dynamic memory allocation and timing requirements, so a single dynamic memory allocation algorithm will not be appropriate. As a result, FreeRTOS now treats memory allocation as part of the portable layer.

Also, removing dynamic memory allocation from the core code base enables application writer's to provide their own specific implementations, when required.

When FreeRTOS requires RAM, instead of calling malloc(), it calls **pvPortMalloc().** When RAM is being freed, instead of calling free(), the kernel calls **vPortFree().**

FreeRTOS comes with five example implementations of both pvPortMalloc() and vPortFree().

The five implementations are defined in the **heap_1.c, heap_2.c, heap_3.c, heap_4.c and heap_5.c** source files respectively, located in the FreeRTOS/Source/portable/MemMang directory.

Characteristics of the 5 different Memory Allocation Schemes (Heap-1 to Heap-5)

Cranes Varsity

Heap_1 Allocation Scheme

Features:

1. Common for small dedicated embedded systems
2. Deterministic
3. No Fragmentation
4. Only pvPortMalloc() is implemented. vPortFree() is not available.
5. Memory is allocated before application starts and remains allocated for the lifetime of the application.
6. The heap_1 allocation scheme subdivides a simple array into smaller blocks, as calls to pvPortMalloc() are made
7. The total size (in bytes) of the array is set by the definition of configTOTAL_HEAP_SIZE in FreeRTOSConfig.h

Figure 23.1 demonstrates how Heap_1 allocation works.

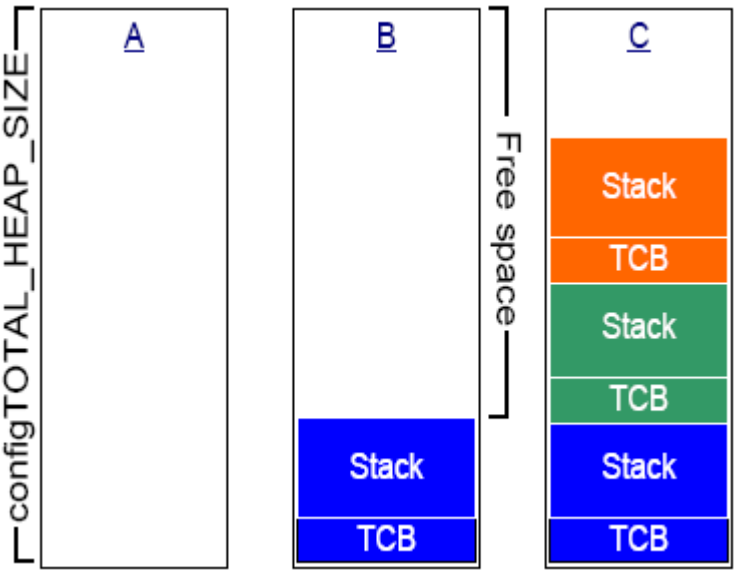


Figure 23.1 : RAM being allocated from the heap each time a task is created

Heap_2 Allocation Scheme

Heap_2 is retained in the FreeRTOS distribution for backward compatibility, but its use is not recommended for new designs. Consider using heap_4 instead of heap_2, as heap_4 provides enhanced functionality.

Features:

1. pvPortMalloc() uses Best Fit algorithm
2. vPortFree is implemented (unlike Heap_1)
3. The heap_1 allocation scheme subdivides a simple array into smaller blocks, as calls to pvPortMalloc() are made (like Heap_1)

4. The total size (in bytes) of the array is set by the definition of configTOTAL_HEAP_SIZE in FreeRTOSConfig.h (like HEAP_1)
5. Appears to consume lot of memory because of static allocation of initial array. (like Heap_1).
6. Susceptible to fragmentation as adjacent free blocks are not combined

Figure 7.2 demonstrates how HEAP_2 scheme works

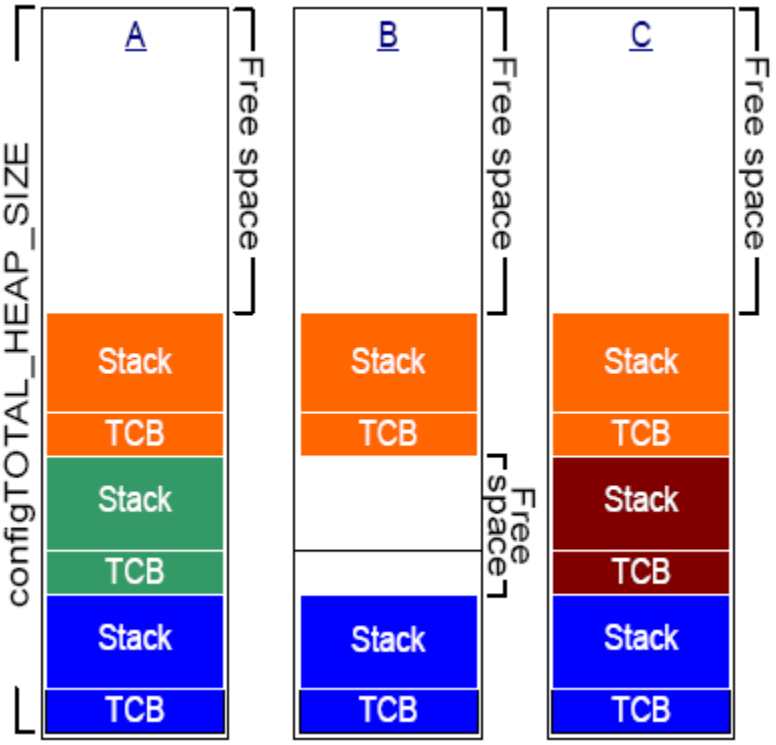


Figure 23.2: Ram being allocated and freed from Heap_2 as tasks are created and deleted.

Heap_3 Allocation Scheme

Heap_3.c uses the standard library malloc() and free() functions, so the size of the heap is defined by the linker configuration, and the configTOTAL_HEAP_SIZE setting has no affect. Heap_3 makes malloc() and free() thread-safe by temporarily suspending the FreeRTOS scheduler.

Heap_4 Allocation Scheme

Features:

1. pvPortMalloc() uses **First Fit** algorithm
2. vPortFree is implemented (unlike Heap_1)
3. The heap_1 allocation scheme subdivides a simple array into smaller blocks, as calls to pvPortMalloc() are made (like Heap_1 and Heap_2)

4. The total size (in bytes) of the array is set by the definition of configTOTAL_HEAP_SIZE in FreeRTOSConfig.h (like Heap_1 and Heap_2)
5. Appears to consume lot of memory because of static allocation of initial array. (like Heap_1).
6. **No fragmentation** as adjacent free blocks are combined (coalesced) (unlike Heap_2)

Figure 7.3 demonstrates how Heap_4 works.

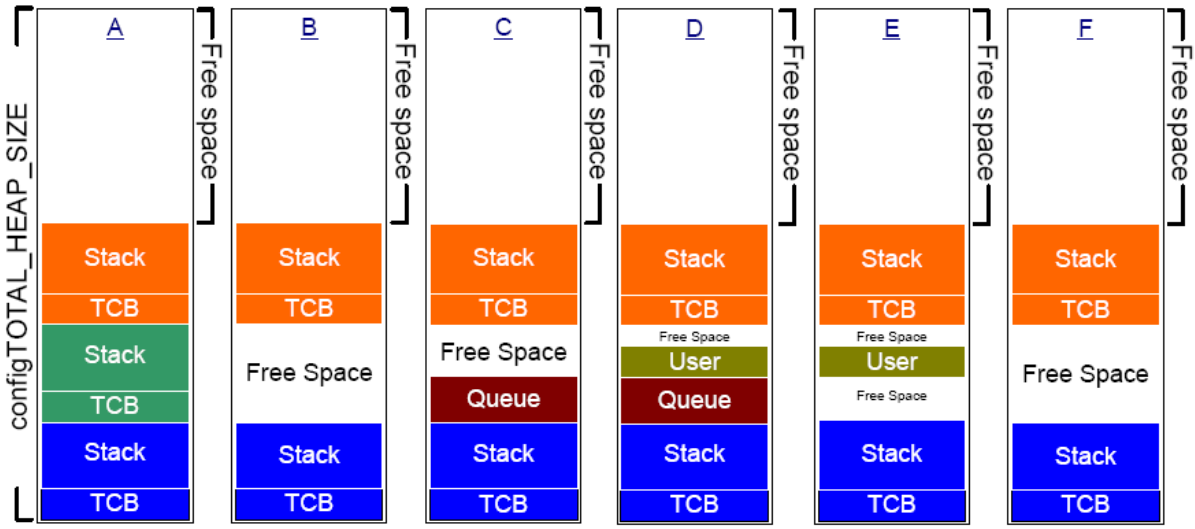


Figure 23.3 : Memory being allocated & deallocated as objects are created and destroyed in Heap_4

Heap_5 Allocation Scheme

Features:

1. Identical to that of Heap_4
2. Unlike Heap_4, heap_5 is not limited to allocating memory from a single statically declared array
3. Heap_5 can allocate memory from multiple and separated memory spaces.
4. Useful when RAM provided by the system does not appear as a single contiguous block in the system memory map.
5. Heap_5 must be initialized using vPortDefineHeapRegions() API function before pvPortMalloc() can be called (as of FreeRTOS v 9)
6. vPortDefineRegions() is used to specify start address and size of each separate memory area that makes up the total memory

CHAPTER 24
Interrupt Management

Embedded real time systems have to take action in response to external events. The software routines that execute in response to these event are known as Interrupt Service Routines(ISRs).

Tasks and ISRs

A task is a software feature that is unrelated to the hardware on which FreeRTOS is running. The priority of a task is assigned in software by the application writer, and a software algorithm (the scheduler) decides which task will be in the Running state.

1. Although written in software, an interrupt service routine is a hardware feature because the hardware controls which interrupt service routine will run, and when it will run.

2. Tasks will only run when there are no ISRs running, so the lowest priority interrupt will interrupt the highest priority task, and there is no way for a task to pre-empt an ISR.

Guidelines to ISR design in FreeRTOS

1. Keep the ISRs as small as possible . (perform deferred interrupt processing)
2. Use only interrupt safe APIs from within an ISR. i.e. APIs which have '*FromISR*' in its name. egs : xSemaphoreGiveFromISR(), xQueueSendFromISR() …

The Interrupt Safe API

Often it is necessary to use the functionality provided by a FreeRTOS API function from an interrupt service routine (ISR), but many FreeRTOS API functions perform actions that are not valid inside an ISR—the most notable of which is placing the task that called the API function into the Blocked state; if an API function is called from an ISR, then the caller(ISR) cannot be blocked. FreeRTOS solves this problem by providing two versions of some API functions; one version for use from tasks, and one version for use from ISRs. Functions intended for use from ISRs have "FromISR" appended to their name.

Deferred Interrupt Processing

An interrupt service routine must record the cause of the interrupt, and clear the interrupt. Any other processing necessitated by the interrupt can often be performed in a task, allowing the interrupt service routine to exit as quickly as is practical. This is called 'deferred interrupt processing', because the processing necessitated by the interrupt is 'deferred' from the ISR to a task.

Centralized Deferred Interrupt Processing

It is also possible to use the xTimerPendFunctionCallFromISR() API function to defer interrupt processing to the RTOS daemon task—removing the need to create a separate task for each interrupt. Deferring interrupt processing to the daemon task is called 'centralized deferred interrupt processing'.

Advantages of centralized deferred interrupt processing:

- Lower resource usage
- It removes the need to create a separate task for each deferred interrupt.
- Simplified user model

  The deferred interrupt handling function is a standard C function.

Disadvantages of centralized deferred interrupt processing include:

- Less flexibility
  It is not possible to set the priority of each deferred interrupt handling task separately. Each deferred interrupt handling function executes at the priority of the daemon task
- Less determinism

ISR to Task Communication

Binary semaphores and Queues can be used to communicate from ISR to task. Ofcourse, only the interrupt safe APIS should be used

Nesting of Interrupts

Some FreeRTOS ports allow interrupts to nest, but interrupt nesting can increase complexity and reduce predictability. The shorter an interrupt is, the less likely it is to nest.

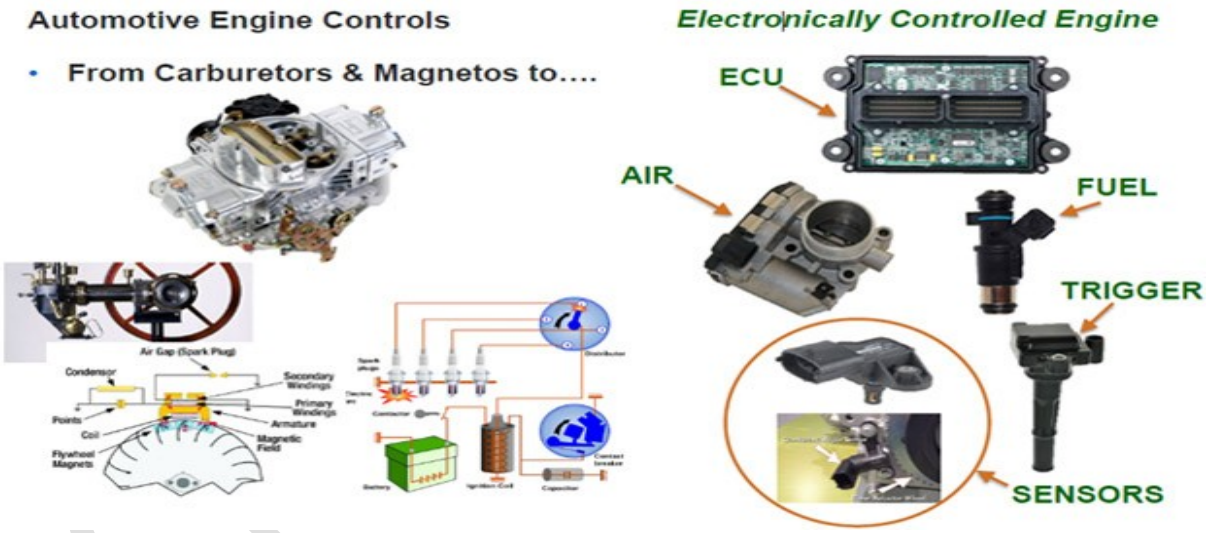
Cranes Varsity

CHAPTER - 25
INTRODUCTION TO AUTOMOTIVE EMBEDDED SYSTEMS

1.1 INTRODUCTION

Automobiles were seen as mechanical machines until the introduction of Electronics in Automotive Industry. Every component starting from engine to window, steering, brake was a mechanical component working on gears and principles of mechanics. The mechanical systems had inherent limitations and limited accuracy, which not only caused undetected failures, but also pose life threats to the consumers. These limitations meant that there was a lot of scope for innovation in automotive sector. This eventually led to the widespread introduction of electronics across components and systems within an automobile. In 1970, Automotive Electronic Control Units (ECUs) was introduced in the automotive industry and since then, it has played a fundamental role in evolution of Automobiles from being a completely mechanical to being an electronics dominant device. Modern day cars have over hundred in-built or installed ECUs in them. Luxury cars like BMW 7-series models have as many as 150 Automotive ECUs to control and regulate the functions of the car.

Understanding the Functional Difference between Mechanical and Electronics based control units:



Factors that led the Automotive OEMs' to move from Mechanical to Electronic Control Units:

The paradigm shift of Automobiles from a mechanical machine to electronic system has paved the way for innovations like power steering, cruise control, infotainment, HUD, in-car connectivity and mobility. In modern connected cars, Automotive ECUs along with LiDAR sensor technology are eventually making a self-driving autonomous car a reality. So while in the hindsight, it is a no-brainer to conclude that the electronics in automotive has indeed led to favorable results. But it would also be interesting to look at the factors that stood out as the factors driving this change in automotive industry

Driver and pedestrian safety:

Mitigation of driver distraction to ensure safety for both drivers and pedestrians has always been the top priority for Automotive OEMs' and Government Regulators. Some of the OEMs' like Volvo have also officially announced their ambitions to reduce the fatality rate due to vehicles to zero by 2020. The automotive OEMs' and Suppliers are able to walk the talk due to the capabilities of the electronics based control units within the vehicle. Automotive ECUs along with image processing algorithms, sensors and camera support a number of Advanced Driver Assistance Systems (ADAS) like adaptive cruise control, driver drowsiness detection, lane departure warning, forward collision alert, pedestrian detection and more This has been one of the major driving factors as any compromise with safety would have direct impact on the very existence of automobiles as the mode of transport.

Need for compliance with government regulations:

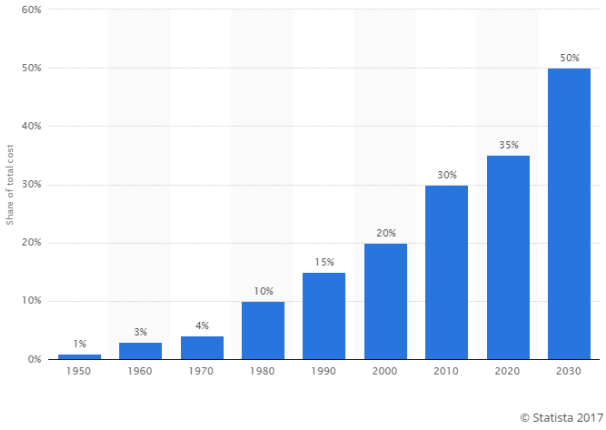
Government regulatory bodies are one of the key stakeholders of the automotive industry ecosystem. As an Automotive OEM and /or a Supplier, it is mandatory to comply with such region specific regulations and norms pertaining to emissions, energy consumption, safety and emergency responses and more implementing such mandates without the use of Electronic Control Units and software algorithms would have been a mission impossible. On the other end, due to emergence of electronic based automation and connectivity with road infrastructure, the regulators are also able to keep the malpractices in check and respond in a better way, to emergency situations. For an instance, to keep in check the frequency of road accidents due to fleet trucks and also to ensure adherence to the HOS (hours of service) policy, the U.S. Federal Motor Carrier Safety Administration (FMCSA) has issued an ELD mandate. All the fleet companies have to comply with the mandate by December 2017 by installing Electronic Logging Device (ELD) in their trucks.

Car or a mobile device on wheels :

In the last decade, since the advent of mobile phones, it became pertinent for car-makers to introduce connectivity and more electronics within the car. The generation obsessed with smart devices, web connectivity, ease of navigation, social media and consumption of information on the go, meant that car had to slowly transform into a consumer electronics device. Global OEMs' and Suppliers have been able to respond to such a change in customer preference by allowing the explosion of electronics to bolster in-car mobility and connectivity. Investments in R&D and in-vehicle infrastructure ensured that ECUs' and in-vehicle networks (Flex-Ray BUS) support multimedia systems like Infotainment and HUD (Head-up Display). These along with Telematics applications have opened up a Pandora box of new revenue opportunities for OEMs', through value-add after-sales service and remote diagnostics and maintenance support.

Cranes Varsity

Automotive Electronics timeline: The journey from Cadillac to Tesla



Numbers speak louder than words! And this graph (by 'statista) does all the justice to the influence of electronics in Automotive. It also offers a lot of insights regarding the journey of automotive electronics from 1950 to 2030. What we see here is the cost share of automotive ECU with respect to the overall cost of the car from 1950 to 2030. From the graph above, it is very evident that the presence of electronics in cars did not grow overnight. It took 3 decades of technology innovations, persistent R&D in automotive product development along with other driving factors, when finally electronics contributed 10% to overall cost in 1980s'. To be more specific the introduction of Airbags Control Unit in 1970s and the demand for fuel efficient cars, also contributed to the rapid growth of electronics during 1970-1980,1990-2010 can be considered as the best growth years for automotive electronics.

Automotive OEMs' like Toyota, Ford and Honda introduced car models with GPS, multimedia (DVD) players, advanced diagnostics systems, back-up sensor and cameras and driver assistance systems like pre-collision safety systems,and OnStar module(General motors car models). With advanced technologies like LiDAR sensor based self-driving cars, Land Rover's Invisible Car and Toyota's Hovering car, it is anticipated that, by 2030 Automotive electronics will contribute 50% of the total car cost. It is no brainer that Automotive Electronics is having a dream-run in recent years.

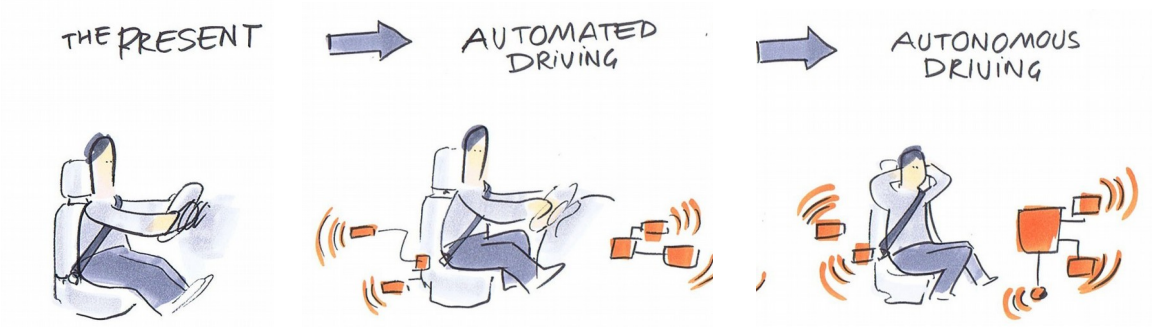
And the way this story has unfolded, all this seems destined to happen not only for better driving experience but also for the safer roads of the world!

Automotive system overview

Today's vehicles that are rolling out from the factories are basically computers with wheels. A modern vehicle consists of approximately 50-60 ECUs and the numbers are increasing for each year. Each unit is in charge of a specific functionality and they communicate with each other over some kind of bus, e.g. CAN, Flex-Ray, LIN etc. For each additional unit that is connected to the system, the system complexity increases. Applications and software must be configured for each system and with new hardware the applications must be rewritten each time to support any changes in the hardware. In order to handle this increased complexity, which would eventually become unmanageable, a handful of leading Original Equipment Manufacturers (OEM) and Tier 1 suppliers from the automotive

Cranes Varsity

industry, decided to work together to form a new standard that would serve as a platform for future vehicle applications, and their result is AUTOSAR.



Embedded technology in Automotive Industry

Recently automotive embedded system has developed highly since the advent of smart car, electric card and so on. They have various value-added system for example IPA (Intelligent Parking Assistance), BSW (Blind Spot Warning), LDWS (Lane Departure warning System), LKS (Lane Keeping System)-these are ADAS (Advanced Driver Assistance Systems). Today's automotive industry is faced with a growing demand for technical appliances and this requires an increased use of ECUs which is reflected in the complexity of the system. Traditionally, solutions have been specific for a certain platform or model and this structure is more becoming unmanageable and costly. Instead of making specific solutions a standardized future is the way to go. This increased complexity could be manageable and improved by a standardized architecture and the solution is AUTOSAR. Modern Advanced Driver Assistance Systems (ADAS) have established a standard of driving comfort and safety unknown so far. Vehicles have become increasingly "intelligent" allowing the driver to delegate specific subtasks of vehicle guidance to these systems or to let the automation take over vehicle guidance completely in emergency situations. However, the scientifically proven advantages of ADAS are accompanied by an important disadvantage: increasing complexity. Today, most ADAS are developed separately, with the consequence that each of these systems has its own user interface and interaction concept. Indeed, the uncountable number of buttons placed all over the dashboard in a modern car almost recalls the cockpit of an aircraft. This complexity runs counter to the original goal of enhanced comfort and safety. The automation of driving might be a solution to some of the challenges of future mobility concepts. Work on fully autonomous cars has undeniably made extensive advances in the last ten years. But fundamental challenges, for example the question of possible approval processes for road traffic, are still unsolved. On the other hand, ADAS capabilities are increasing rapidly and many have already proven themselves in real road traffic and in usage by customers.

Automotive Microcontroller

In the Automobile industry an electronic control unit (ECU) is an embedded electronic device, basically a digital computer, that reads signals coming from sensors placed at various parts and in different components of the car and depending on this information controls various important units e.g. engine and other automated operations within the car.

Cranes Varsity

CHAPTER -26

MISRA - C

Embedded C and Coding standard according to Automotive

The C programming language is growing in importance and use for real-time embedded applications within the automotive industry. This is due largely to the inherent language

flexibility, the extent of support and its potential for portability across a wide range of hardware. Specific reasons for its use include:

a. For many of the microprocessors in use, if there is any other language available besides assembly language then it is usually C. In many cases other languages are simply not available for the hardware.
b. C gives good support for the high-speed, low-level, input/output operations, which are essential to many automotive embedded systems.
c. Increased complexity of applications makes the use of a high-level language more appropriate than assembly language
d. C can generate smaller and less RAM-intensive code than many other high-level languages.
e. A growth in portability requirements caused by competitive pressures to reduce hardware costs by porting software to new, and/or lower cost, processors at any stage in a project lifecycle.

Language insecurities and the C language

No programming language can guarantee that the final executable code will behave exactly as the programmer intended. There are a number of problems that can arise with any language, and these are broadly categorized below. Examples are given to illustrate insecurities in the C language.

- The programmer makes mistakes.
- The programmer misunderstands the language.
- The compiler doesn't do what the programmer expects.
- The compiler contains errors.
- Run-time errors.

It should be clear from the above section that great care needs to be exercised when using C within safety-related systems. Because of the kinds of issues identified above, various concerns have been expressed about the use of C on safety-related systems. Certainly it is clear that the full C language should not be used for programming safety-related systems. However in its favor as a language is the fact that C is very mature, and consequently well analyzed and tried in practice. Therefore its deficiencies are known and understood. Also there is a large amount of tool support available commercially which can be used to statically check the C source code and warn the developer of the presence of many of the problematic aspects of the language. If, for practical reasons, it is necessary to use C on a safety-related system then the use of the language must be constrained to avoid, as far as is practicable, those aspects of the language which do give rise to concerns.

Introduction to MISRA- C Automotive Standard

The MISRA consortium published its Development Guidelines for Vehicle Based Software in 1994. This document describes the full set of measures that should be used in software development. In particular, the choices of language, compiler and language features to be used, in relationship with integrity level, are recognized to be of major importance. One of the measures recommended is the use of a subset of a standardized

Cranes Varsity

language, which is already established practice in the aerospace, nuclear and defense industries.

Guidelines main targets

- MISRA has developed a set of coding guidelines for the programming language C while other languages (like C++) are under discussion.
- The C guidelines are intended to be applied during the development of software used in safety-critical applications.
- Even if these guidelines are produced for the automotive industry, they are often applied to other industries (like medical devices).
- Most of the guidelines can be enforced by performing static code analysis on application source code.

The guidelines specify that all of the rules apply equally to human and machine generated code. Some rules have their basis in psychological findings (i.e. how developers read the source). Such issues are not important in machine generated code (because such code is never read by humans). Those rules that are motivated by how humans process source code are flagged as such, so that they may be allowed in machine generated code.

Understanding the rules of MISRA – C

5 Use of characters are required to be in the source character set. This excludes the characters $ and @, among others.
22 Declarations of identifiers denoting objects should have the narrowest block scope unless a wider scope is necessary.
34 The operands of the && and || operators shall be enclosed in parenthesis unless they are single identifiers.
67 Identifiers modified within the increment expression of a loop header shall not be modified inside the block controlled by that loop header.
103 Relational operators shall not be applied to objects of pointer type except where both operands are of the same type and both point into the same object.

Please visit as reference
http://www.misra.org.uk/
http://www.misra-c2.com/
http://www.knosof.co.uk/misracom.html
http://en.wikipedia.org/wiki/MISRA_C

Static Analysis of selected programs for Compilance

Note: Using PC Lint Tool

Overview of Code Optimization Techniques

- Jumps/branches are expensive. Minimize their use whenever possible.
- Think about the order of array indices

Cranes Varsity

- Avoid/reduce the number of local variables
- Reduce the number of function parameters.
- For the same reason as reducing local variables – they are also stored on the stack.
- Pass structures by reference, not by value.
- If you do not need a return value from a function, do not define one.
- Use shift operations >> and << instead of integer multiplication and division, where possible.
- Avoid dynamic memory allocation during computation.

EXAMPLE PROGRAMS

1. **Program 1**

```
#include<stdio.h>
        int main()
        {
int x;
x = 5 + 4 << 2;
printf("the value of x is:%d\n", x);
return 0;
        }
```

Warning: Unusual shift operation

```
#include<stdio.h>
int main()
{
    int x;
    x = 5 + (4 << 2);
    printf("the value of x is:%d\n", x);
    return 0;
}
```

2. **Program 2**

```
#include <stdio.h>
int main()
{
            int i;
      int a[] = {1,2,3};
```

Cranes Varsity

```
    int n = sizeof(a)/sizeof(int);
    for(i=0;i<=n;i++)
    {
            printf("a[%d]=%d\n",i,a[i]);
    }
   return 0;
}
```

Warning: Possible access of out-of-bounds pointer (1 beyond end of data) by operator

```
#include <stdio.h>
int main()
{
    int i;
    int a[] = {1,2,3};
    int n = sizeof(a)/sizeof(int);
    for(i=0;i<=(n-1);i++)
    {
        printf("a[%d]=%d\n",i,a[i]);
    }
    return 0;
}
```

Cranes Varsity

3. Program 3

```
#include<stdio.h>
int a=0,b=0;
int main()
{
    int a=1,b=2,sum,sum1;
    printf("Enter two numbers\n");
scanf("%d%d",&a,&b);
    sum=a+b;
```

```
    sum1 = a+b;
  printf("sum=%d\n",sum);
}
```

Warning: Declaration of symbol 'a' hides symbol 'a' (line 2)
Warning: Declaration of symbol 'b' hides symbol 'b' (line 2)

```
#include<stdio.h>
int a1=0,b1=0;
int main()
{
      int a=1,b=2,sum,sum1;
      printf("Enter two numbers\n");
      scanf("%d%d",&a,&b);
      sum=a+b;
      sum1 = a1+b1;
      printf("sum=%d\n",sum);
      printf("sum1=%d\n",sum1);
}
```

4. **Program 4**

```
#include<stdio.h>
int main()
{
int a = 0x80000000;
a = a >>1;
return 0;
}
```

Warning: Loss of information (initialization) (32 bits to 7 bits)
Shift right of signed quantity (int)

```
#include<stdio.h>
int main()
{
Unsigned int a = 0x80000000;
a = a >>1;
return 0;
}
```

5. **Program 5**

Cranes Varsity

```
#include<stdio.h>
int main()
{
 int x =1,i;
 for(i=0;i<3;i++)
{
if(x)
{
    x--;
}
}
}
```

```
#include<stdio.h>
int main()
{
 int x =1,i;
 for(i=0;i<3;i++)
{
if(x!=0)
{
    x--;
}
}
}
```

6. **Program 6**

```
#include<stdio.h>
 int main()
 {
     int a;
for( ; ; )
     {
   a++;
    printf("%d", a);
}
 }
```

Cranes Varsity

```
#include<stdio.h>
  int main()
  {
      int a;
 while(1)
      {
    a++;
      printf("%d", a);
}
  }
```

7. Program 7

```
#include<stdio.h>
int main()
{
int a,b,c,big;
printf("enter three numbers : ");
scanf("%d%d%d", &a,&b,&c);

if(a)
{
    if(a>b)
    {
        if(a>c)
        {
            big=a;
        }
        else
            big = c;
    }
    else
    {
        if(b>c)
            {
                big=b;
            }
            else
                big = c;
    }
        }
else
printf("number is Zero");
```

Cranes Varsity

```
printf("biggest number is %d\n", big);
return 0;

  }
```

```
#include<stdio.h>
   int main()
   {
 int a,b,c,big;
 printf("enter three numbers : ");
scanf("%d%d%d", &a,&b,&c);

 if(a!=0)
{
      if(a>b)
      {
          if(a>c)
          {
                  big=a;
           }
          else
                  big = c;
     }
     else
     {
           if(b>c)
          {
                  big=b;
          }
          else
                  big = c;
     }
          }
else
printf("number is Zero");

printf("biggest number is %d\n", big);
return 0;

      }
```

8. **Program 8**

```
#include<stdio.h>
```

Cranes Varsity

```
int main()
{
int count;
while(count<99)
{
    count++;
    printf("number:%d", count);
    delay();
}
}

void delay(int c)
{
 int i,j;
for(i=0;i<c;i++);
for(j=0;j<c;j++);
}
```

Warning: Symbol 'count' (line 4) not initialized
Suspicious use of ;

```
#include<stdio.h>
int main()
{
int count=0;
while(count<99)
{
    count++;
    printf("number:%d", count);
    delay();
}
}

void delay(int c)
{
 int i,j;
for(i=0;i<c;i++)
{
    for(j=0;j<c;j++)
    {
    }
}

}
```

Cranes Varsity

9. **Program 9**

```
#include<stdio.h>
int main()
    {
     int i=3, j = 3;
          while(i<5 && j!=0)
          {
           i++;
           j--;
     }
        printf("%d %d\n", i,j);
         return 0;
 }
```

Warning: Negative Indentation

```
#include<stdio.h>
int main()
    {
     int i=3, j = 3;
          while(i<5)
          {
           i++;
          if(j==0)
          break;
          j--;
     }
        printf("%d %d\n", i,j);
          return 0;
 }
```

10. **Program 10**

```
#include<stdio.h>
 int main()
{
     int a= 21;
     int b= 09;
printf("%d %d\n", a,b);
 }
```

Warning: Octal constants (other than zero) and octal escape sequences

Cranes Varsity

shall not be used.

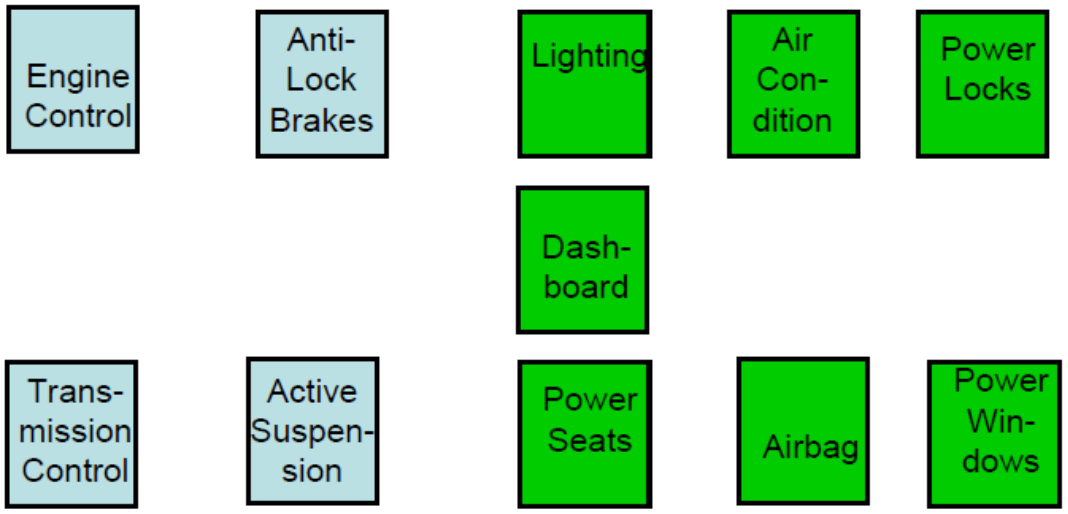
```
#include<stdio.h>
int main()
{
    int a= 21;
    int b= 9;
printf("%d %d\n", a,b);
}
```

Cranes Varsity

CHAPTER - 27
CONTROL AREA NETWORK

INTRODUCTION

The automobile industry witnessed the advent of various electronic control systems that have been developed for safety, comfort, pollution prevention, and low cost.



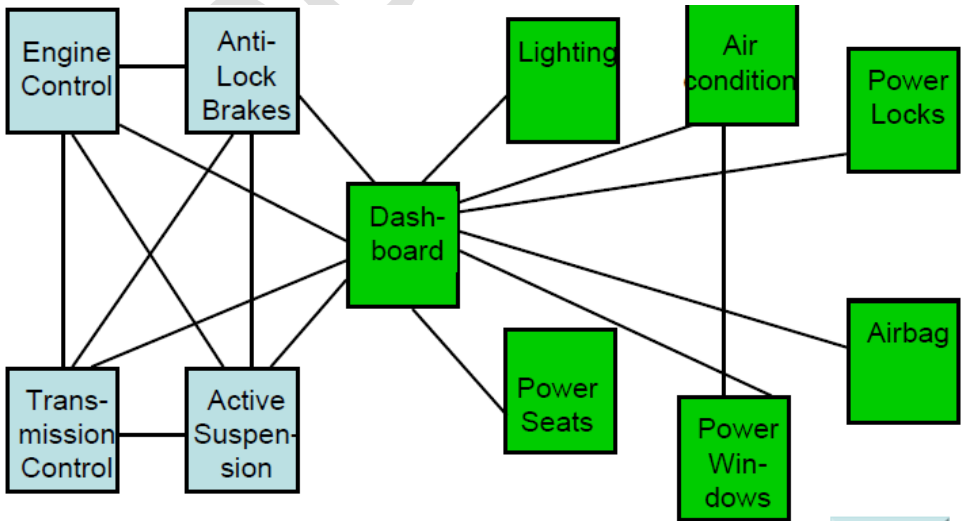
These ECU's presented a drawback in that since the communication data types, required reliability, etc. differed between each system, they were configured in multiple bus lines, resulting in increased wire harnesses.

Need to reduce size of wiring harness around the car

1948 average car wiring: 10lbs, 150 ft, 35 connectors, 75 terminals, 55 wire

1994 average car wiring:65lbs, 5280 ft, 300 connectors, 2000 terminals, 1500 wires

1993 corvette had three miles and 200 pounds if wires



The need arose for

a) Reducing the number of wire harnesses,

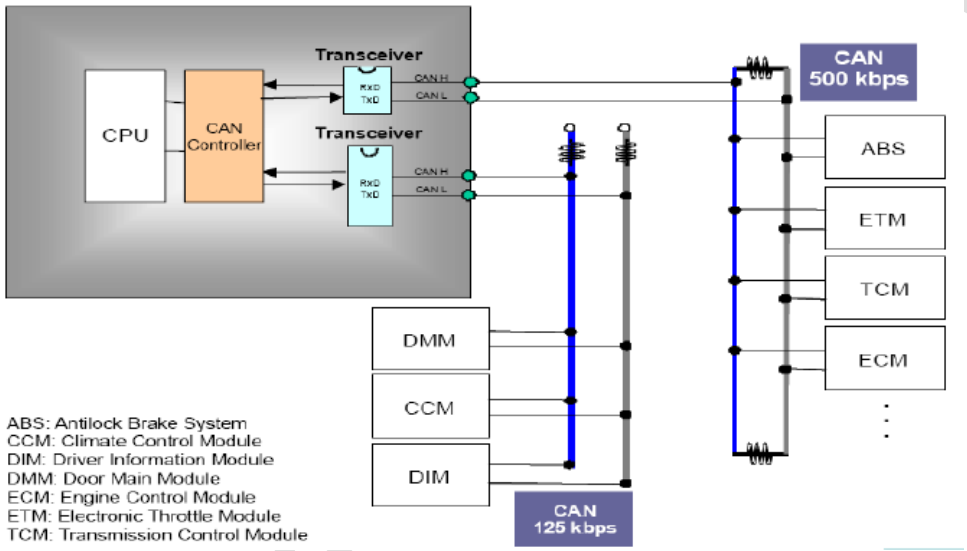
b) Transferring large amounts of data at high speed via multiple LANs, and so on.

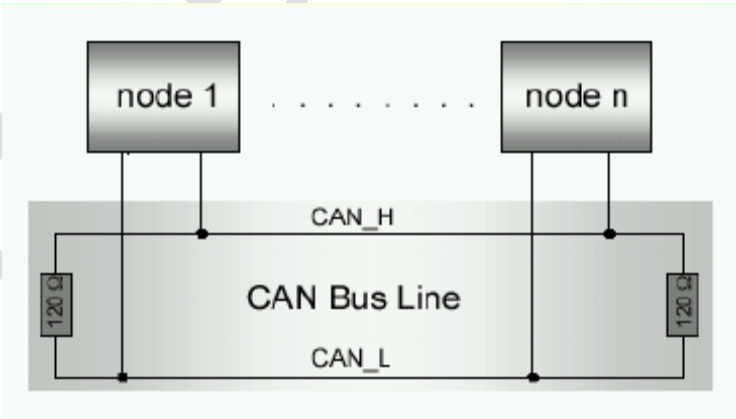
To meet the need, BOSCH developed CAN in 1986 as a communication protocol for automotives. The first CAN silicon was then fabricated in 1987 by Intel.

Thereafter, CAN was standardized in ISO 11898(for high speed applications) and ISO 11519(for lower speed applications), establishing itself as the standard protocol for in-vehicle networking now.

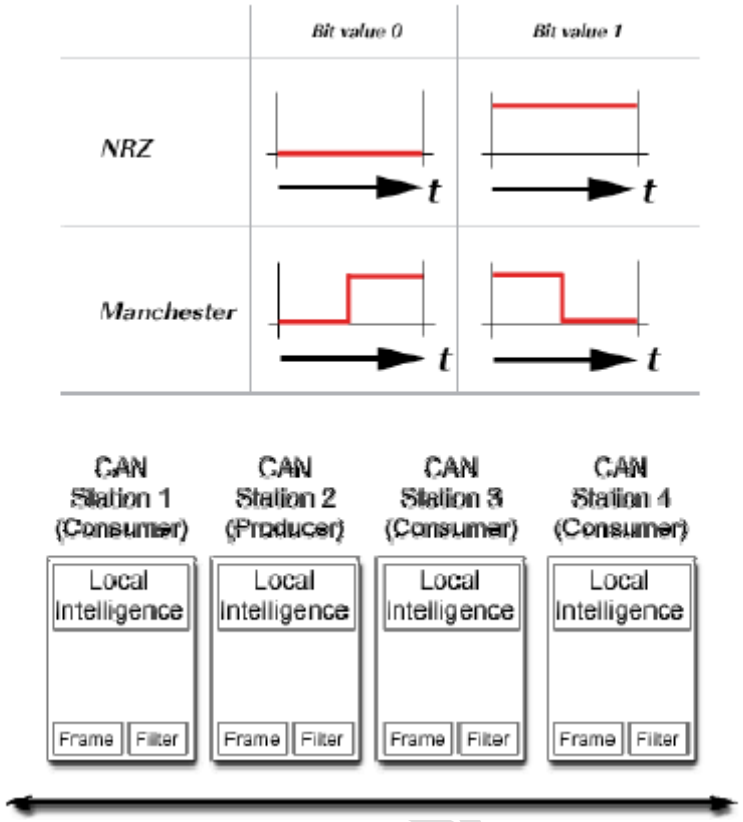
• Standard CAN (Version 2.0A). Uses 11 bit identifiers.

• Extended CAN (Version 2.0B). Uses 29 bit identifiers.





Bit Encoding : CAN is NRZ - Non Return to Zero



Features of CAN

CAN protocol has the following features

(1) Multimaster

When the bus is free, all of the units connected to it can start sending a message (multimasters).

The unit that first started sending a message to the bus is granted the right to send. If multiple units start sending a message at the same time, the unit that is sending a message who's ID has the highest priority is granted the right to send.

(2) Message transmission

Information is sent on the bus in a fixed format messages frames

Any node may transmit when the bus is idle

If two or more units start a message at the same time, contention for the bus is arbitrated according to the ID of each message by comparing the IDs bitwise.

Highest priority message wins the bus access. Transmitting nodes which lose arbitration becomes receivers. The unit that won the arbitration (i.e., the one that has the highest priority) can continue to send, while the units that lost in arbitration immediately stop sending and go to a receive operation. Uses CSMA/ CD+AMP (Carrier Sense Multiple Access/Collision Detection with Arbitration on Message Priority).

No data or time is wasted, someone always wins. Nodes which lose arbitration will automatically re-transmit.

Message identifier determines the message priority

Identifier is intended to label message content not physical address. The ID does not indicate the destination to which a message is sent, but rather indicates the priority of messages in which order the bus is accessed.

(3) System flexibility

The units connected to the bus have no identifying information like an address. Therefore, when a unit is added to or removed from the bus, there is no need to change the software, hardware, or application layer of any other unit connected to the bus. Data messages transmitted from any node on a CAN bus do not contain addresses of either the transmitting node, or of any intended receiving node. The content of the message is labeled by an identifier that is unique throughout the network.

(4) Communication speed

Any communication speed can be set that suits the size of a network. Within one network, all units must have the same communication speed. If any unit with a different communication speed is connected to the network, it will generate an error, hindering communication in the network. This does not apply to units in other networks.

(5) Remote data request

Data transmission from other units can be requested by sending a “remote frame” to those units.

(6) Error detection, error notification, and error recovery functions

All units can detect an error (error detection function).

The unit that has detected an error immediately notifies all other units of the error simultaneously (error notification function).

If a unit detects an error while sending a message, it forcibly terminates message transmission and notifies all other units of the error. It then repeats re-transmission until the message is transmitted normally (error recovery function).

All nodes will check the consistency of the messages and flag an inconsistent message to entire network

Cranes Varsity

All receiving nodes will acknowledge a valid message

A message is received correctly by all nodes are no nodes

All nodes apply message filtering to determine whether to accept or reject the message

Any number of nodes can simultaneously receive and accept messages

(7) Error confinement

There are two types of errors occurring in the CAN:

A temporary error where data on the bus temporarily becomes erratic due to noise from the outside or for other reasons, and

A continual error where data on the bus becomes continually erratic due to a unit's internal failure, driver failure, or disconnections.

The CAN has a function to discriminate between these types of errors. This function helps to lower the communication priority of an error-prone unit in order to prevent it from hindering communication of other normal units, and if a continual data error on the bus is occurring, separate the unit that is the cause of the error from the bus.

(8) Connection

There are no logical limits to the number of connectable units. However, the number of units that can actually be connected to a bus is limited by the delay time and electrical load in the bus. A greater number of units can be connected by reducing the communication speed. If the communication speed is increased, the number of connectable units decreases.

(9) Bus Lengths

Bus length (metres)	Maximum bit rate (bit/s)
40	1 Mbit/s
100	500 kbit/s
200	250 kpit/s
500	125 kbit/s
6 km	10 kbit/s

Cranes Varsity

CAN Basic Concepts

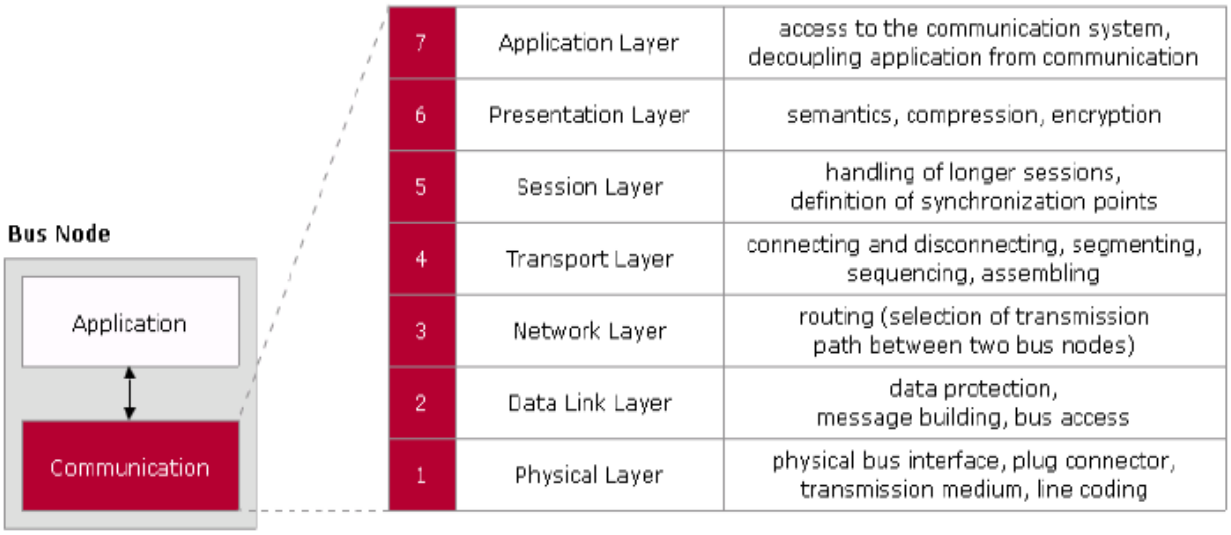
- Bit oriented serial communication Protocol
- Variable bit rate - 5 Kbps to 1 Mbps
- Peer - to - Peer: Any node can transmit at any frame
- Multicast without routing - All nodes receive all messages
- CSMA/CR - Non destructive bit wise arbitration
  - Pre set time for beginning of transmission
  - Transmitter “listen” before attempting to access
  - If channel is busy, the interface unit waits
  - Collision Resolution by bit wise arbitration
- Message Identifier
  - CAN has no node address
    - Every node receives every messages and decides itself whether to use it or not
- Extensive error checking
  - 5 different checks
  - Every connected node participates
- System wide data consistency
  - A message is accepted by all nodes are none

CAN Media Access

- Carrier Sense Multiple Access / Collision Detect with Collision Resolution
- All nodes must sense bus before transmission (CS)
- Equal access opportunity for all nodes (MA)
- Must detect collision during transmission (CD)
- Once collision detected, distributed arbitration applied (CR)

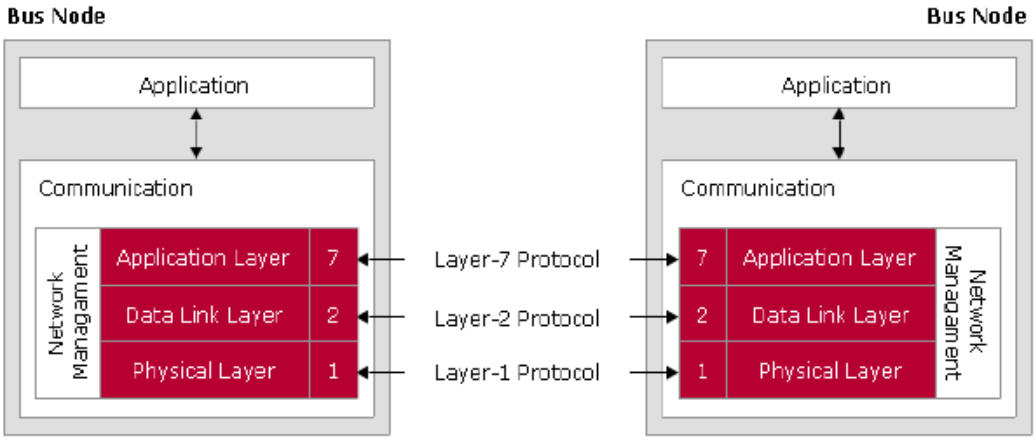
CAN from OSI perspective:

Cranes Varsity



7	Application Layer	access to the communication system, decoupling application from communication
6	Presentation Layer	semantics, compression, encryption
5	Session Layer	handling of longer sessions, definition of synchronization points
4	Transport Layer	connecting and disconnecting, segmenting, sequencing, assembling
3	Network Layer	routing (selection of transmission path between two bus nodes)
2	Data Link Layer	data protection, message building, bus access
1	Physical Layer	physical bus interface, plug connector, transmission medium, line coding

For serial data exchange between electronic control units in the motor vehicle it is primarily the two lowest layers that are relevant (bit transmission and data protection layers), and functions of the unconsidered layers can be added to the uppermost layer (application layer),this reduces the seven layer model to a **three layer model**.



Data Link Layer

Logical link Control

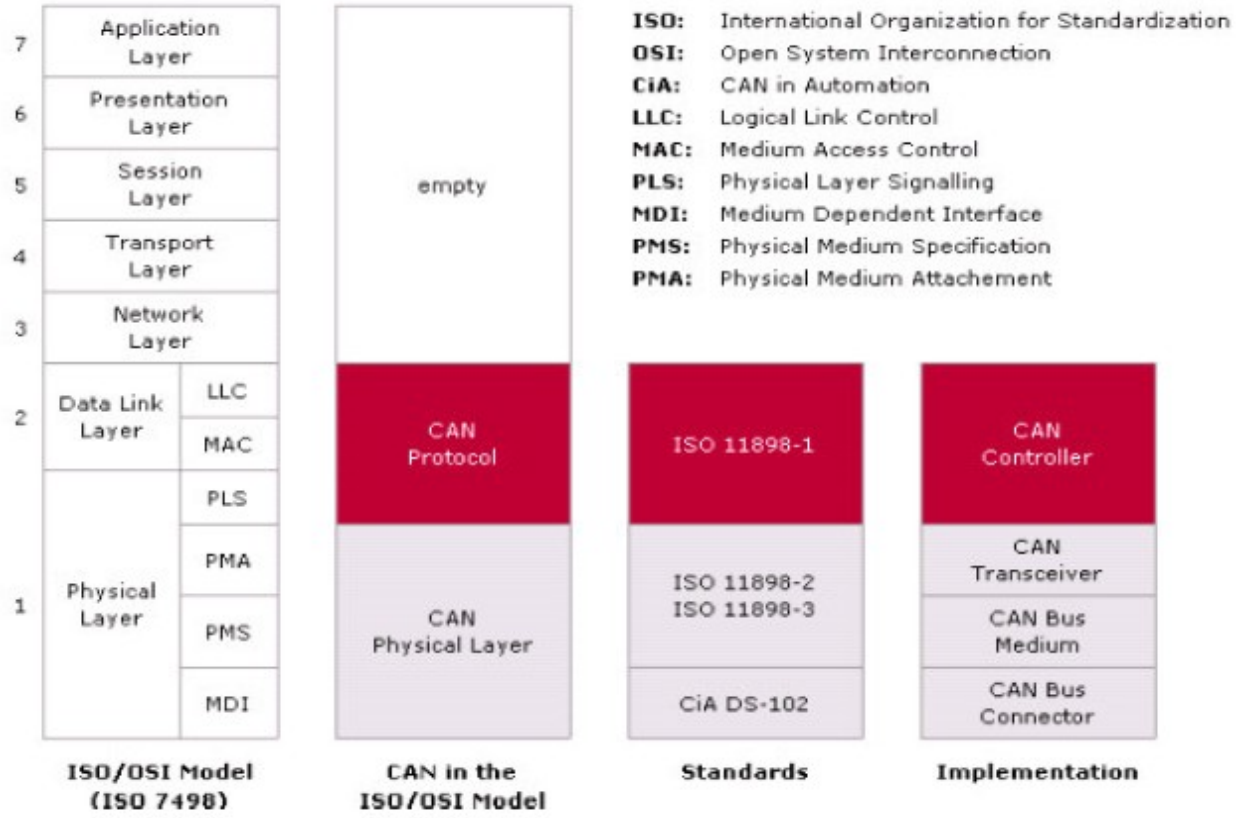
- Acceptance filtering
- Overload notification
- Recovery management

Medium Access control

- Data encapsulation
- Frame coding
- Error detection/signaling
- Acknowledgement
- Serialization/de serialization

Physical Layer

- Bit encoding/decoding
- Bit timing
- Synchronization



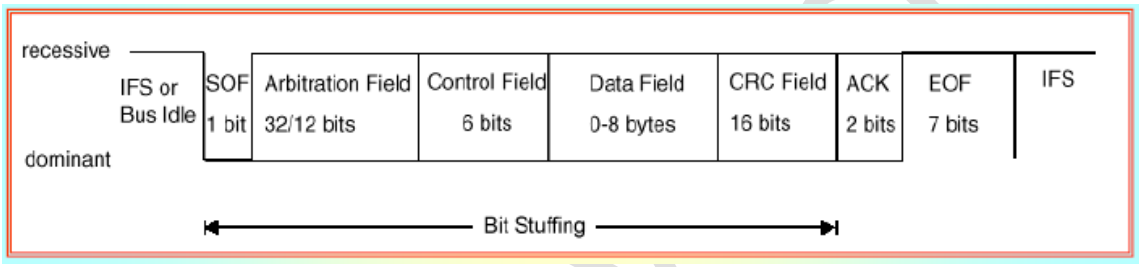
Types of Frames

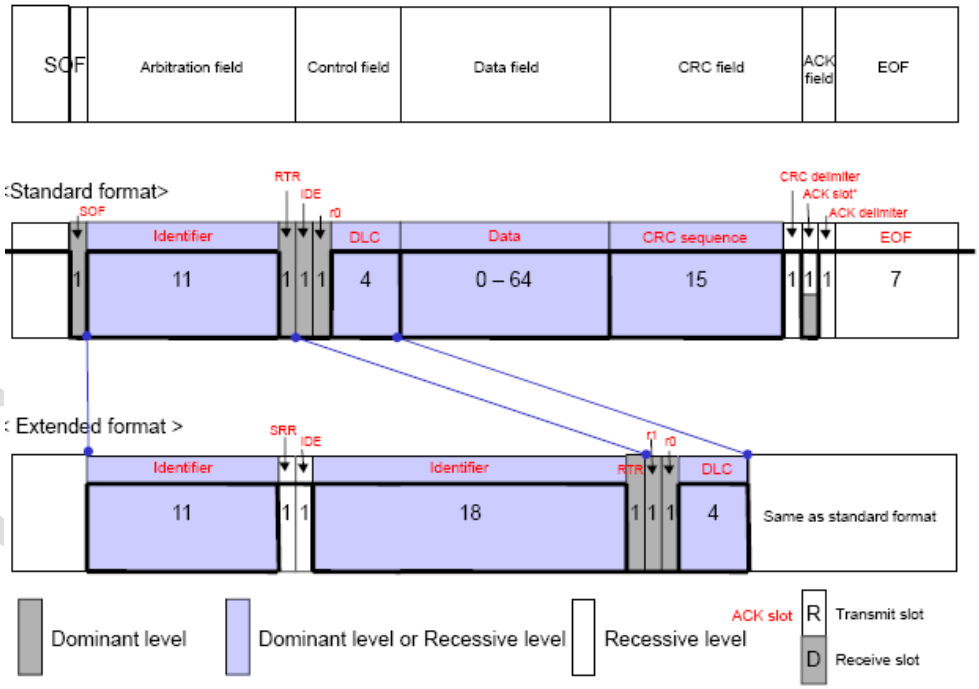
Communication is performed using the following five types of frames.

- Data frame: Used to transmit up to 8 bytes of data
- Remote frame: Used to request a data frame
- Error frame: Used to indicate error to the entire network (Not under CPU) control
- Overload frame: Used to create an extra delay between frames
- Interframe space:

Frame	Roles of frame	User settings
Data frame	This frame is used by the transmit unit to send a message to the receive unit.	Necessary
Remote frame	This frame is used by the receive unit to request transmission of a message that has the same ID from the transmit unit.	Necessary
Error frame	When an error is detected, this frame is used to notify other units of the detected error.	Unnecessary
Overload frame	This frame is used by the receive unit to notify that it has not been prepared to receive frames yet.	Unnecessary
Interframe space	This frame is used to separate a data or remote frame from a preceding frame.	Unnecessary

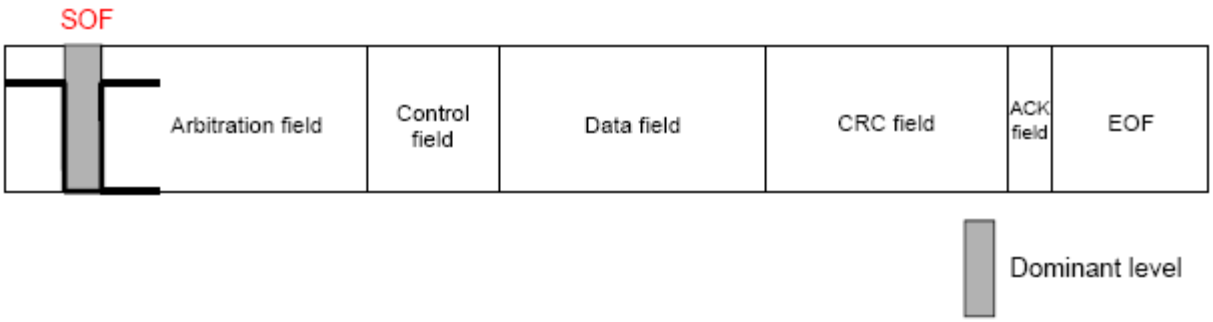
Data Frame





Start of Frame (SOF):

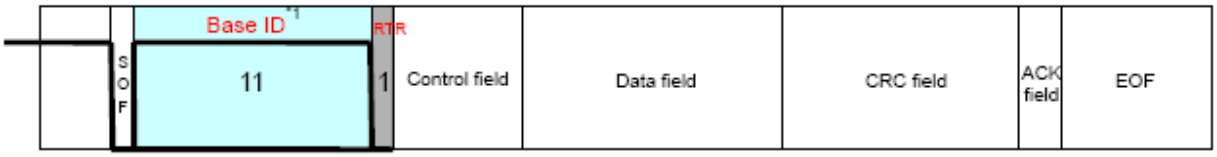
This field indicates the beginning of a frame. It consists of one dominant bit.



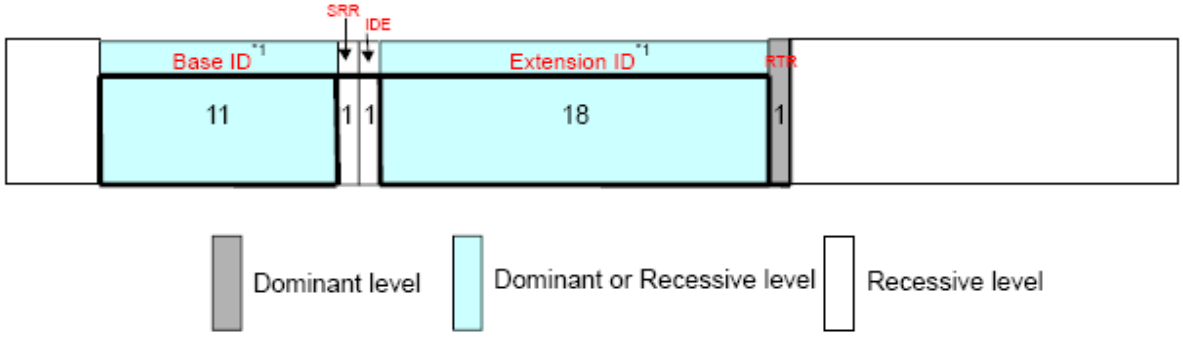
Arbitration field:

This field indicates the priority of data. The structure of this field differs between the standard and extended formats.

<Standard format>



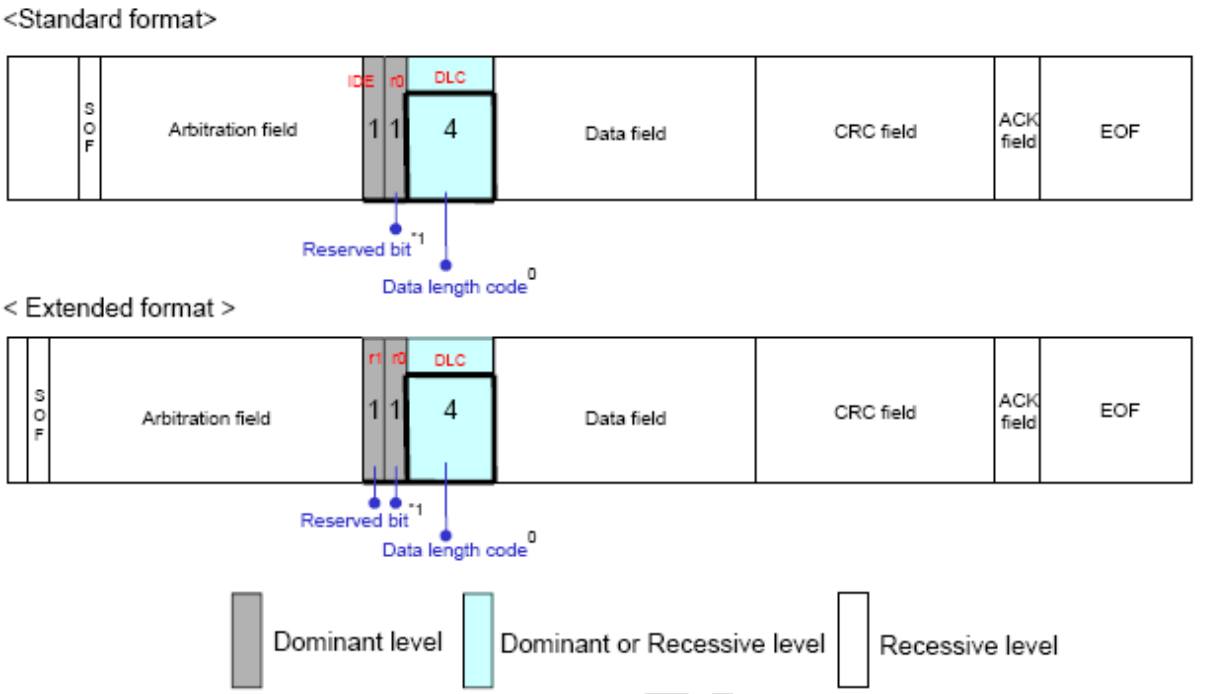
< Extended format >



- 11 or 29 bits
  - Controllers support CAN 2.0B
- Arbitration is done on this part
  - Lower the number, highest the priority
- RTR bit
  - 0 = Data frame
  - 1 = Remote Frame
- IDE bit
  - 0 = Standard ID (11 bit)
  - 1 = Extended ID (29 bit)
- SRR bit - Always 1, replacement for RTR

Control field:

This 6-bit field indicates the number of data bytes in a message to be transmitted. The structure of this field differs between the standard and extended formats.



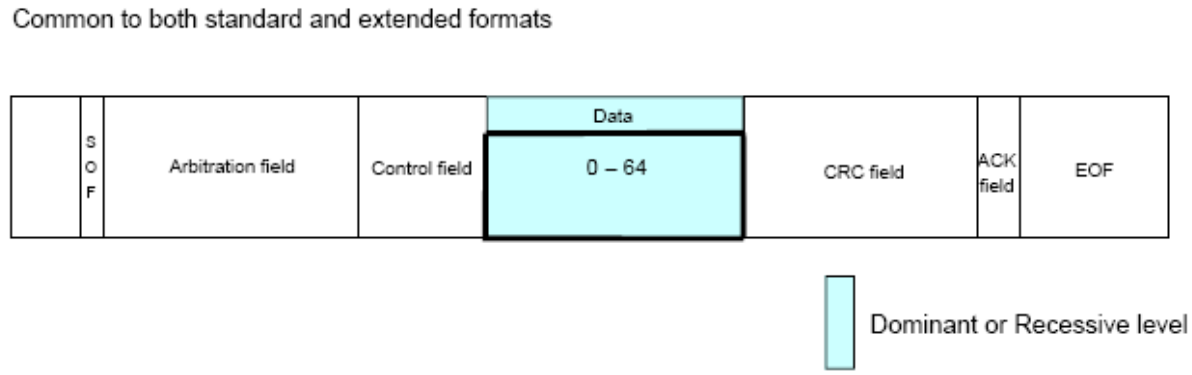
- DLC - Data Length Code
  - 0 to 8 bytes maximum (0000 to 1000)
  - Two bits are reserved and always 0.
  - In CAN 2.0B, the first reserved bit r1 is called as IDE to differentiate between 2.0A and2.0B

Control: Data Length Code and the Number of Data Bytes.

Number of data bytes	Data length code			
	DLC3	DLC2	DLC1	DLC0
0	D	D	D	D
1	D	D	D	R
2	D	D	R	D
3	D	D	R	R
4	D	R	D	D
5	D	R	D	R
6	D	R	R	D
7	D	R	R	R
8	R	D or R	D or R	D or R

D: Dominant level, R: Recessive level

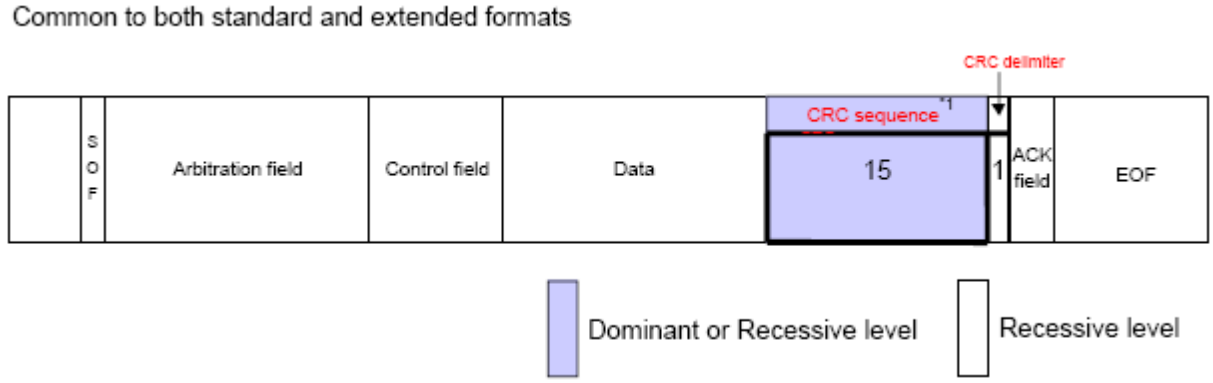
Data field: This field indicates the content of data. Zero to 8 bytes of data set in the control field can be transmitted. The data is output beginning with the MSB side. Contains information/set of information like Engine Speed, Torque and Vehicle Speed etc. Data More than 8 bytes must use techniques in software to handle data.



CRC field (Cyclic Redundancy Check): This field is used to check the frame for a transmission error. It consists of a 15-bit CRC sequence and a 1-bit CRC delimiter (separating bit).

A CRC sequence which is known to all connected modules is used for determining the 15 bit CRC reminder which is transmitted in this field

Receiver calculates CRC reminder using the received bits and matches with received CRC reminder. No match is considered as CRC error and error frame is generated.

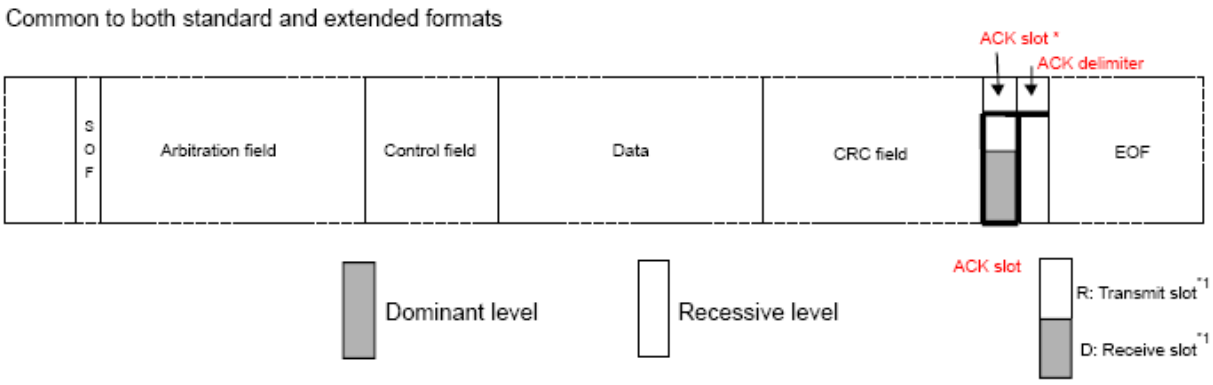


Acknowledge Field: This field indicates a signal for confirmation that the frame has been received normally. It consists of 2 bits, one for ACK slot and one for ACK delimiter.

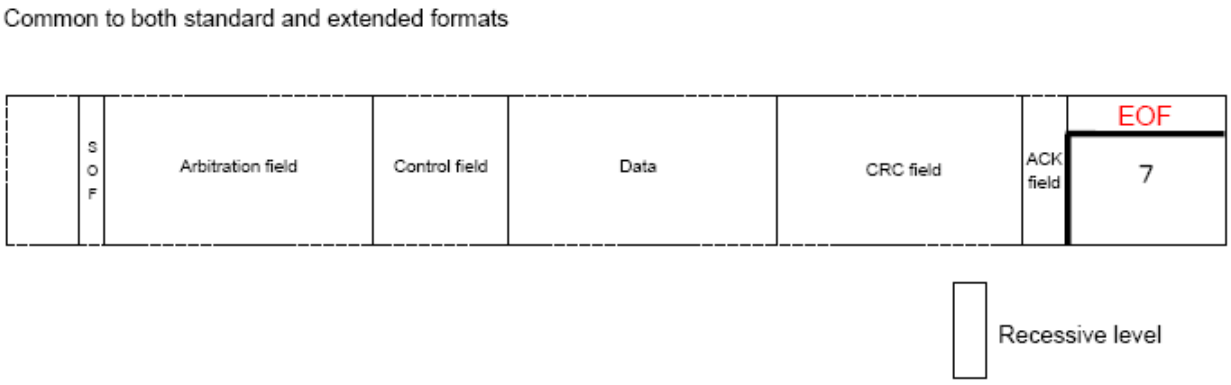
A Transmitter transmits with ACK Slot and ACK Del as 1.

If any one receiver receives the message properly, it pulls down the ACK slot field to 0.

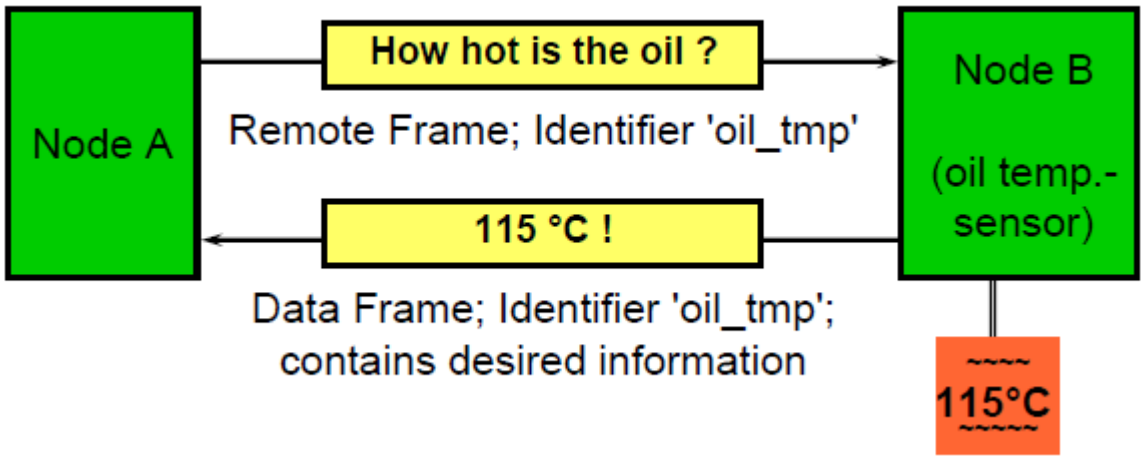
Transmitter monitors ACK Slot and if it doesn't find a 0, it is error and retransmits the message.

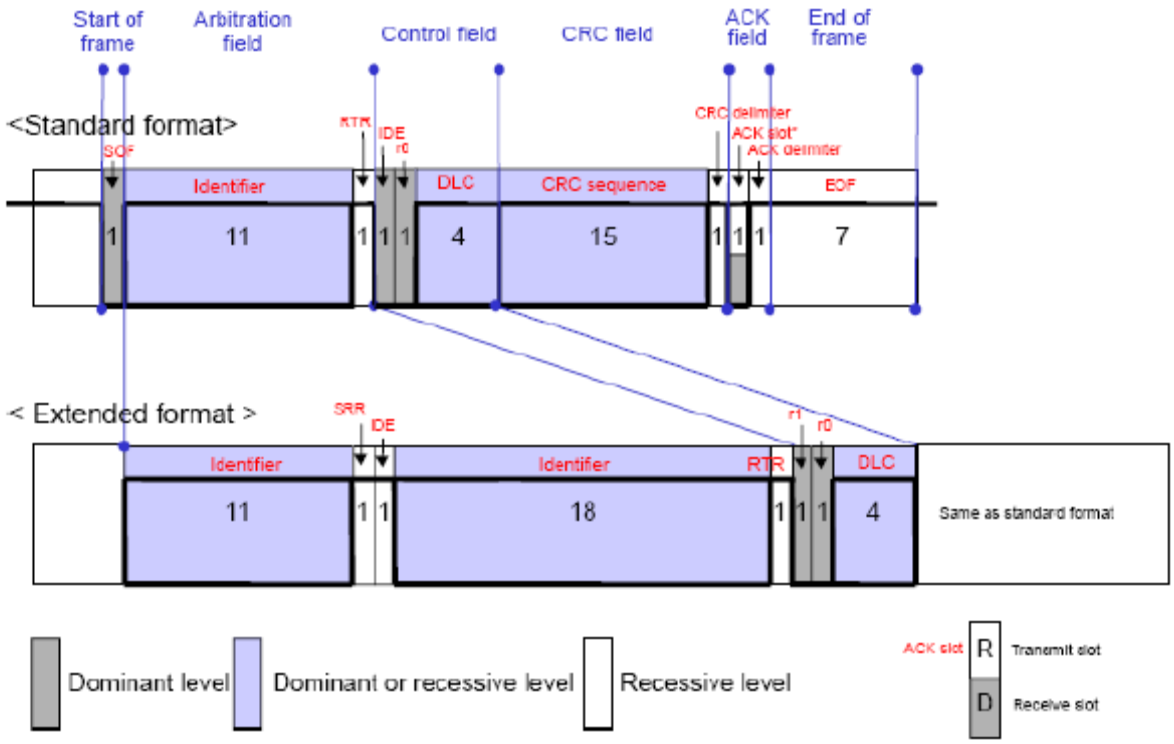


End of Frame (EOF): This field indicates the end of a frame. It consists of 7 recessive bits.



Remote Frame





Remote Frame and Data Frame

Differences between the data frame and remote frame

• The remote frame differs from a data frame in that it does not have a data field, and that the RTR bit in its arbitration field is of a recessive level.

• The data frame without a data field and the remote frame can be discriminated by the RTR bit.

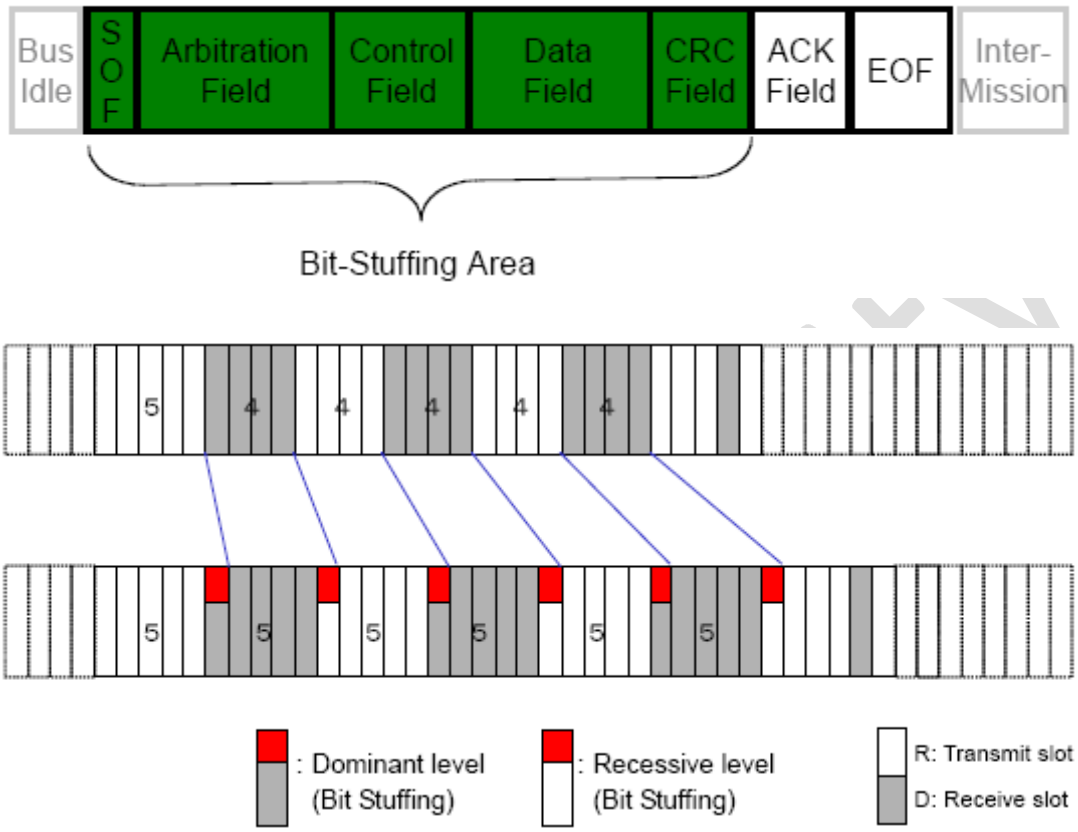
What does the data length code of the remote frame that has no data fields indicate?

• The value of the data length code of the remote frame indicates the data length code of the corresponding data frame.

For what is the data frame without a data field used?

• For example, this data frame may be used by each unit to confirm or respond for connection periodically, or to place real information in the arbitration field itself.

Bit Stuffing



- 5 consecutive bits of same polarity renders stuff bit of opposite polarity
- Transmitter inserts the stuff bit in the message sequence automatically when transmitting
- Receiver automatically de stuff the sequence

Bit stuffing Advantages:

- Stuff bit provides extra edges for synchronization - This is useful as CAN is NRZ
- Also used for stuff error - Discussed Later

Error Frame

- Types of Errors:-
  - Bit Error: Detected by transmitter transmits dominant and monitors recessive on the bus and vice versa
  - Stuff Error: Detected if the stuff bit rule is violated. If 6 consecutive bits of same polarity is received
  - Form Error: Detected if the fixed format in the frame is violated
  - CRC Error: Detected if the checksum mismatch between transmitter and receiver

Acknowledgement Error: Detected by transmitter if it finds no acknowledgement (Not finding dominant “Ack Slot” field).

Type of error	Content of error	Target frame (field) in which errors detected	Unit by which errors detected
Bit error	This error is detected when the output level and the data level on bus do not match when they are compared (Level comparison: Dominant output stuffing bits are compared, whereas arbitration fields and ACK bits during transmission are not compared).	• Data frame (SOF to EOF) • Remote frame (SOF to EOF) • Error frame • Overload frame	Transmit unit Receive unit
Stuffing error	This error is detected when the same level of data is detected for 6 consecutive bits in any field that should have been bit-stuffed.	• Data frame (SOF to CRC sequence) • Remote frame (SOF to CRC sequence)	Transmit unit Receive unit
CRC error	This error is detected if the CRC calculated from the received message and the value of the received CRC sequence do not match.	• Data frame (CRC sequence) • Remote frame (CRC sequence)	Receive unit
Form error	This error is detected when an illegal format is detected in any fixed-format bit field.	• Data frame (CRC delimiter, ACK delimiter, EOF) • Remote frame (CRC delimiter, ACK delimiter, EOF) • Error delimiter • Overload delimiter	Transmit unit Receive unit
ACK error	This error is detected if the ACK slot of the transmit unit is found recessive (i.e., the error that is detected when ACK is not returned from the receive unit).	• Data frame (ACK slot) • Remote frame (ACK slot)	Transmit unit

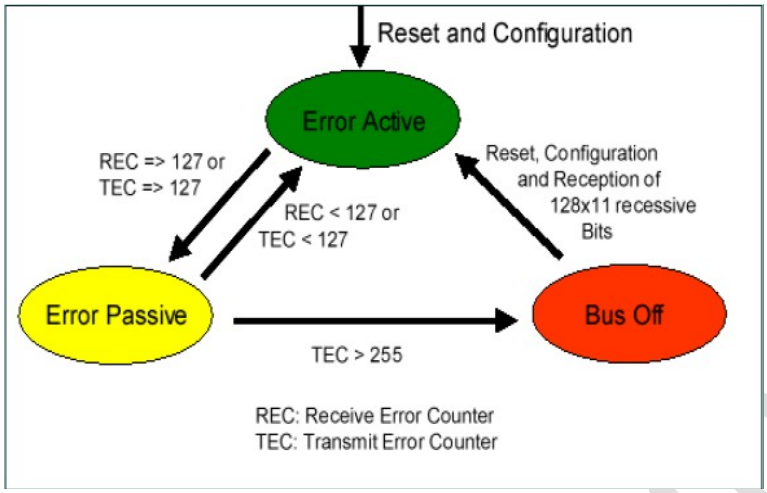
Output Timing of an Error Frame: Error Signaling and Confinement

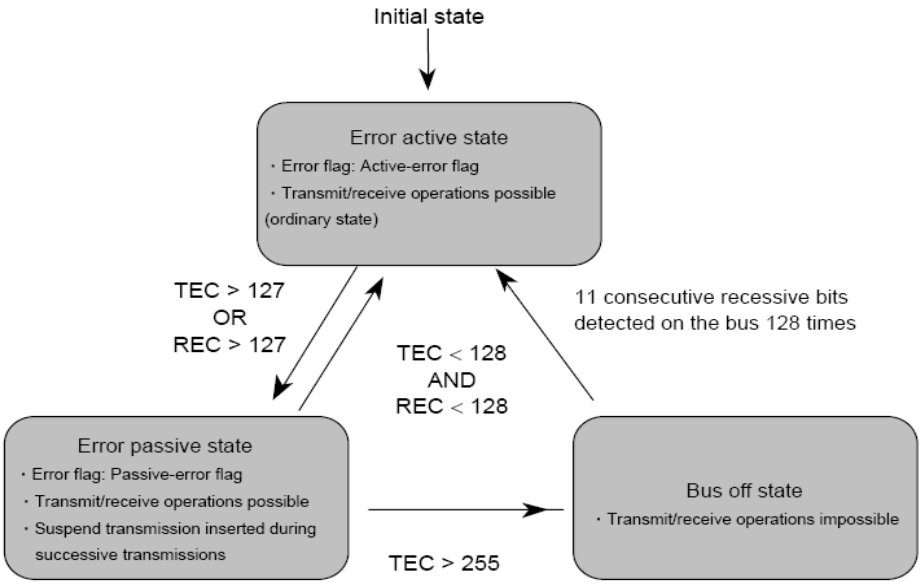
- When Bit, Stuff, Form or Acknowledgement error is detected error flag is started at the next bit
- When CRC error is detected, error flag is started at the bit following Ack Del

Type of error	Output timing
Bit error Stuffing error Form error ACK error	The error flag is output beginning with the bit that immediately follows the one in which an error was detected.
CRC error	The error flag is output beginning with the bit next to the ACK delimiter.

Two error counters are maintained, one for transmit and one for receive

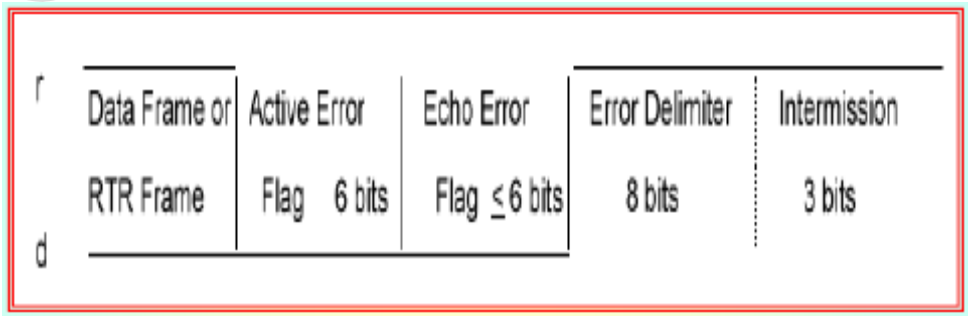
The error counters at each node are incremented in a specific format when an error frame is detected Based on the error counters value three states are determined (Refer Below Figure).





The error frame consists of an error flag and an error delimiter.

Error frames are transmitted by the hardware part of CAN.



(1) Error flag

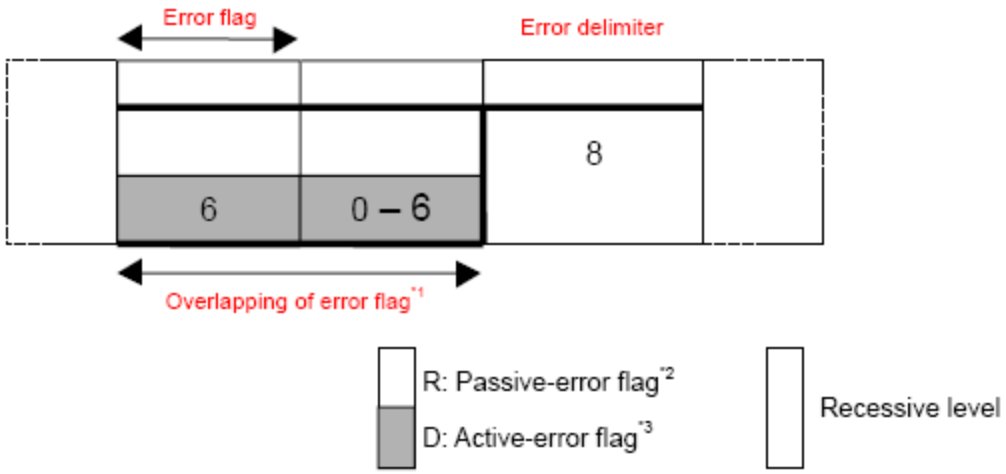
There are two types of error flags: active-error flag and passive-error flag.

• Active-error flag: Transmitted by Active Error Node, consists of 6 dominant bits

• Passive-error flag: Transmitted by Passive Error Node, consists 6 recessive bits

(2) Error delimiter

The error delimiter consists of 8 recessive bits.



*1 : Error flag overlapping

Depending on the timing at which an error is detected by each unit connected to the bus, error flags may overlap one on top of another, up to 12 bits in total length.

*2 : Passive-error flag

This error flag is output by a unit in an error-passive state when it detected an error.

*3 : Active-error flag

This error flag is output by a unit in an error-active state that it detected an error.

Overload Frame

The overload frame is used by the receive unit to notify that it has not been prepared to receive frames yet. Not used by many controllers

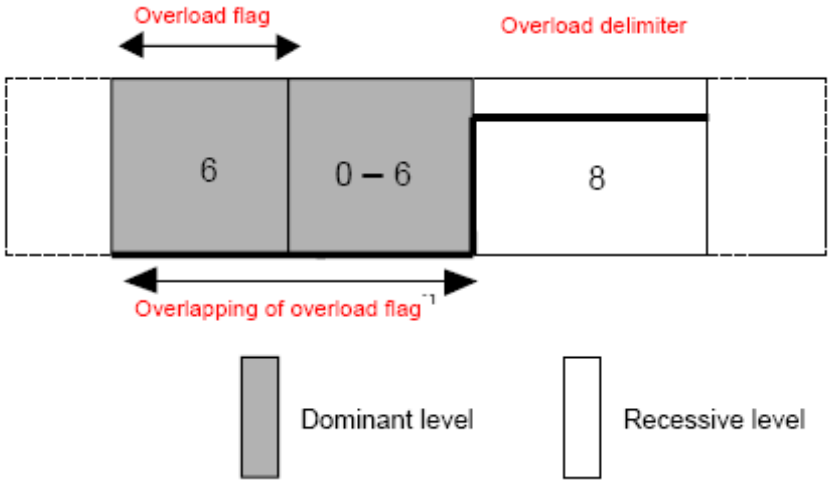
It consists of an overload flag and an overload delimiter.

(1) Overload flag

It consists of 6 dominant bits. The overload flag is structured the same way as the active-error flag of the error frame.

(2) Overload delimiter

It consists of 8 recessive bits. The overload delimiter is structured the same way as the error delimiter of the error frame.



*1 Error flag overlapping

Depending on timing as for the error flag, overload flags may overlap one on top of another, up to 12 bits in total length.

Interframe Space

No interframe spaces are inserted preceding overload and error frames.

(1) Intermission

Consists of 3 recessive bits. If a dominant level is detected during an intermission, an overload frame must be transmitted. However, if the third bit of an intermission is of a dominant level, the intermission is recognized as the SOF.

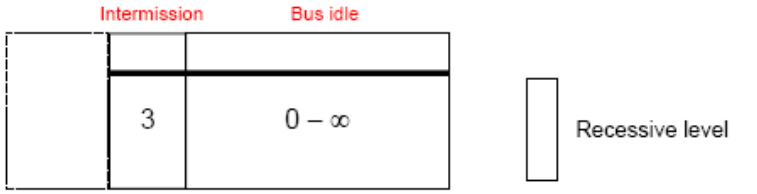
(2) Bus idle

Consists of a recessive level. There are no limitations on its length (it can be zero bits in length). While in this state, the bus is thought to be free, and any transmit unit can start sending a message.

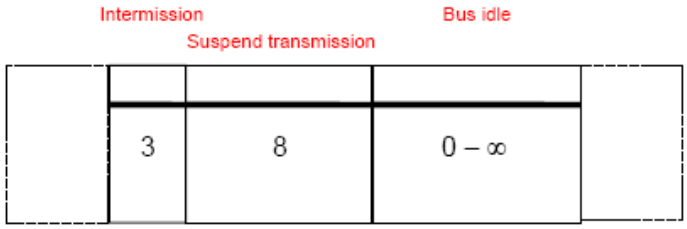
(3) Suspend transmission (transmission pause period)

Consists of 8 recessive bits. This field is included in only an interframe space if the immediately preceding message transmit unit was in an error-passive state.

Cranes Varsity



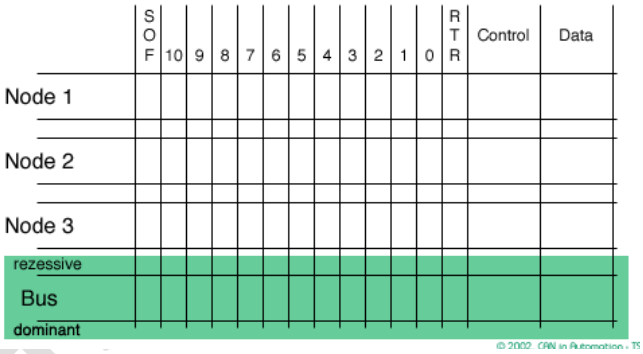
<When the immediately preceding message transmit unit is in an error-passive state>

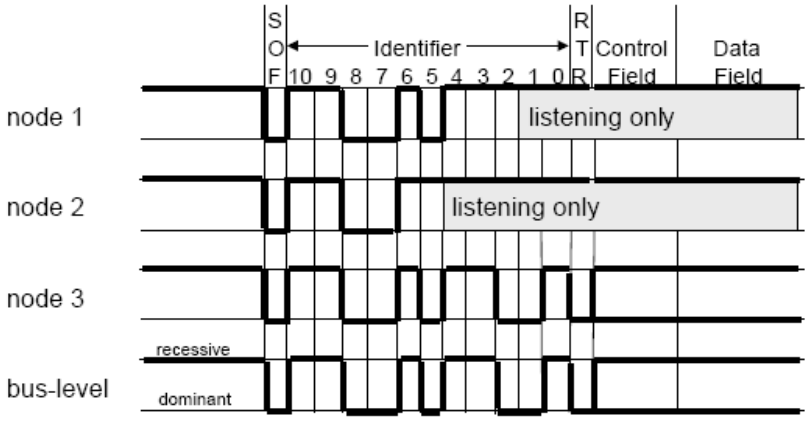


Priority Resolution by Arbitration

Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority (CSMA/CD + AMP)

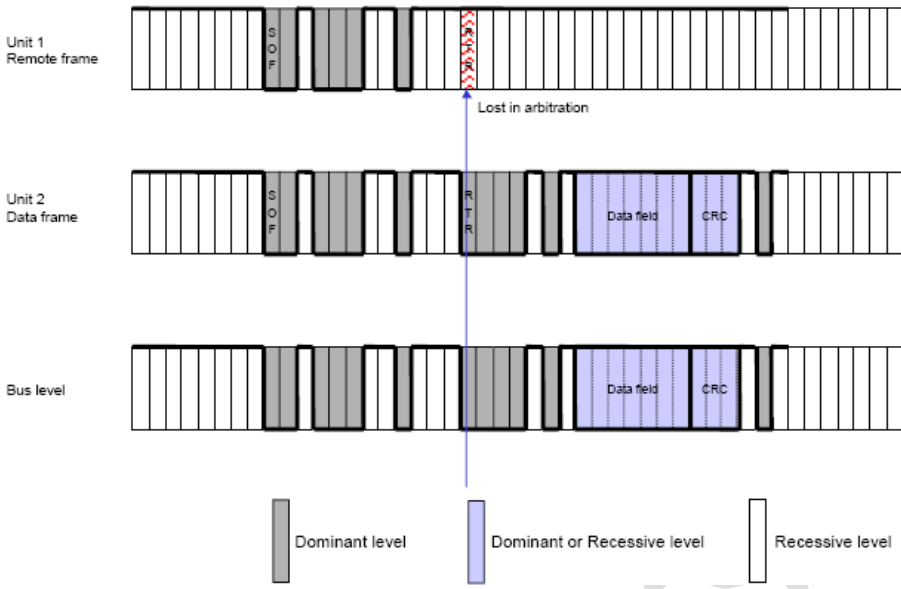
- Dominant/Recessive logic
  - Dominant Always wins
  - Wired And



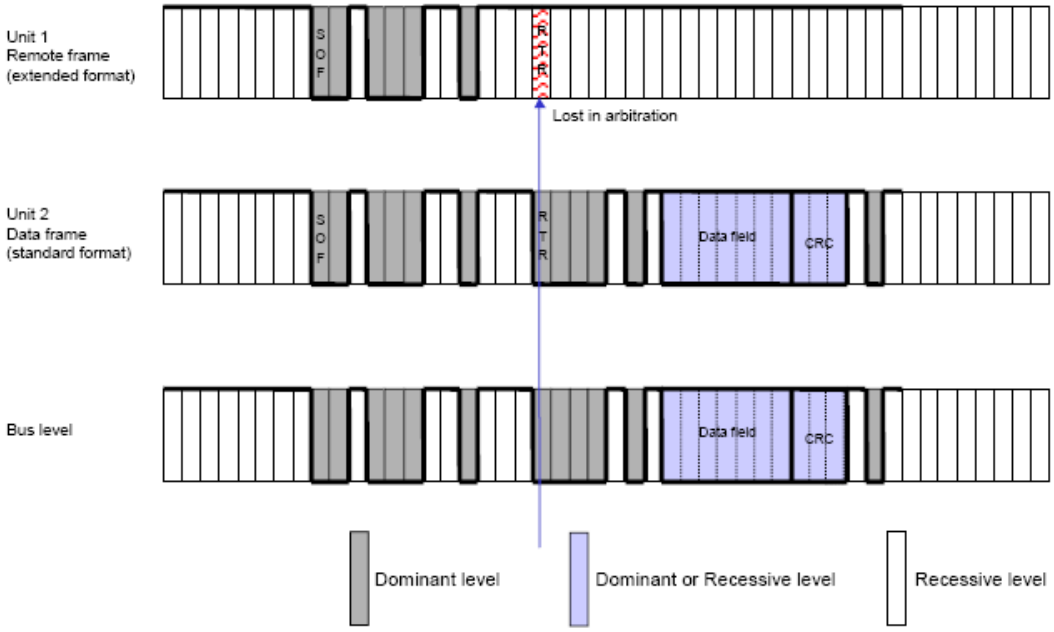


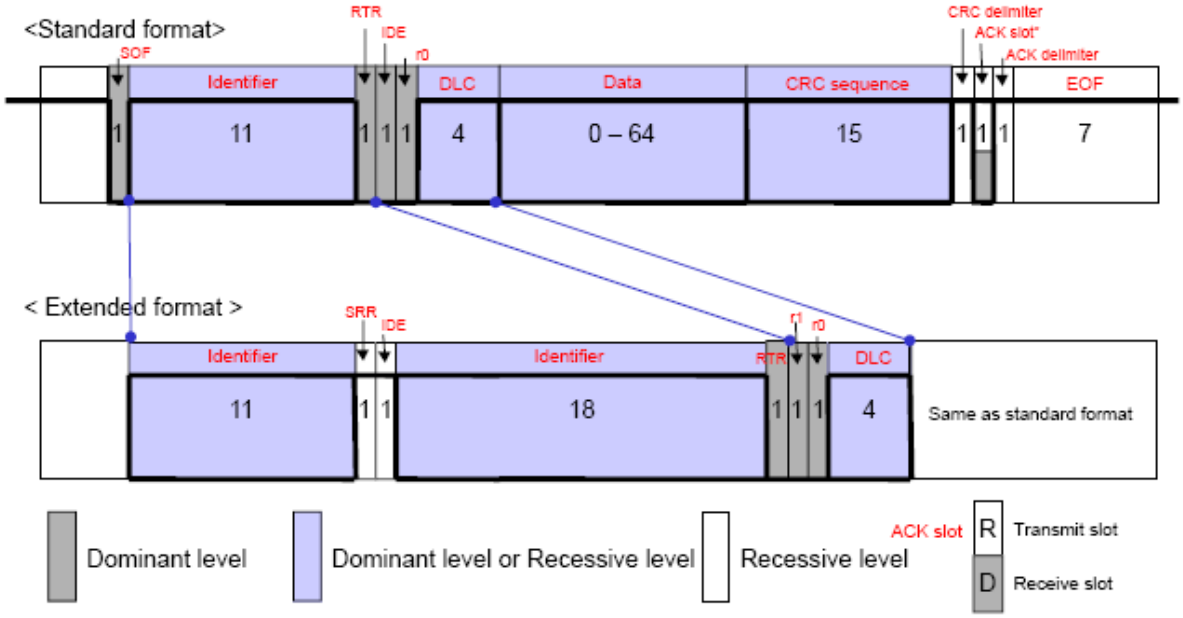
Node 3 wins arbitration and transmits his data.

Priority of data and remote frames

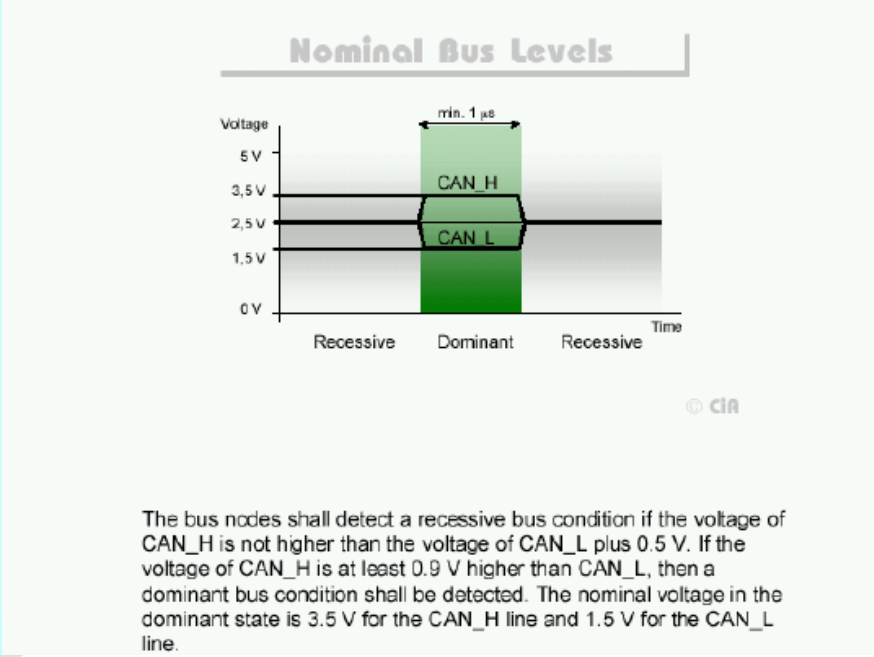


Priority of standard and extended formats





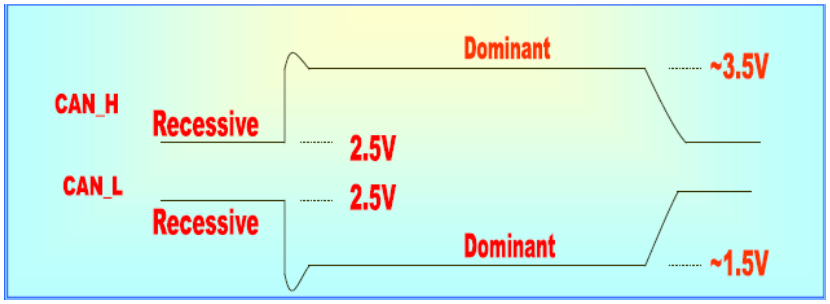
Bus Levels



The bus nodes shall detect a recessive bus condition if the voltage of CAN_H is not higher than the voltage of CAN_L plus 0.5 V. If the voltage of CAN_H is at least 0.9 V higher than CAN_L, then a dominant bus condition shall be detected. The nominal voltage in the dominant state is 3.5 V for the CAN_H line and 1.5 V for the CAN_L line.

Two wire high speed CAN - Differential

- Dominant - CAN_H at 3.5V and CAN_L at 1.5V
- Recessive - CAN_H and CAN_L at 2.5V



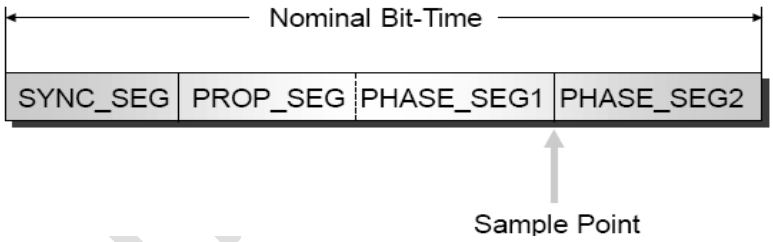
CAN Bit Timing

Basically the CAN bit period/time can be subdivided in to four time segments. These are measured in time quanta's (Tq). 1 Time Quanta = 1/CAN Freq. Each time segment consists of a number of Time Quanta.

Bit Time = Sync_Seg + Prop_Seg + Pgase_Seg1 + Phase_Seg2

Also Expressed as

- TBIT = TSYNC + TSEG1 + TSEG2
  - TSEG1 = Prop_Seg + Phase_Seg1
  - TSYNC = 1 Tq
  - TSEG1 = [2..16] Tq
  - TSEG2 = [1..8] Tq
  - TBIT = [8..25] Tq



• Synchronization segment (SS): 1 Time Quanta long. It is used to synchronize the various bus nodes.

• Propagation time segment (PTS): Programmable to be 1, 2 …8 Time Quanta long. It is used to compensate for signal delays across the network.

• Phase buffer segment 1 (PBS1): Programmable to be 1, 2 …8 Time Quanta long. It is used to compensate for edge phase errors and may be lengthened during resynchronization.

• Phase buffer segment 2 (PBS2): Maximum of Phase buffer segment 1 and the Information Processing Time long. It is also used to compensate edge phase errors and may be shortened during resynchronization.

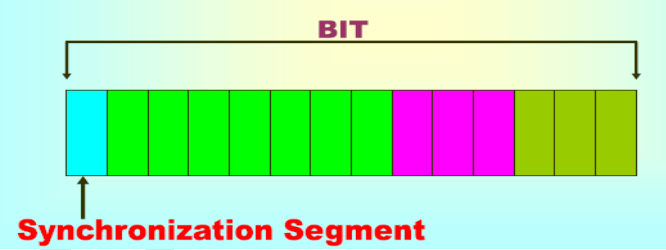
Information Processing Time is less than or equal to 2 Time Quanta long.

Total number of Time Quanta has to be from 8 to 25. Programming of the sample point allows optimizing the Bit Timing. A late sampling for example allows a maximum bus length. An early sampling allows slower rising and falling edges.

Segment name	Roles of segment	Number of Tq's	
Synchronization segment (SS)	Multiple units connected to the bus are timed to be coincident during the interval of this segment as they send or receive a frame. A recessive to dominant or a dominant to recessive transition edge is expected to exist in this segment.	1	8 – 25 Tq's
Propagation time segment (PTS)	This segment absorbs physical delays on network, which include an output delay of the transmit unit, a propagation delay of signal on bus, and an input delay of the receive unit. The duration of this segment is twice the sum of each delay time.	1 – 8	
Phase buffer segment 1 (PBS1)	This segment is used to correct for errors when signal edges do not occur within SS. Since each unit is operating with their own clock, even a minute clock error in each unit will accumulate. This segment absorbs such an error. To absorb a clock error, add or subtract within the SJW setup range for PBS1 and PBS2. Larger the values of PBS1 and PBS2, the greater the tolerance, but the communication speed slows down.	1 – 8	
Phase buffer segment 2 (PBS2)		PBS1 or IPT[-1] whichever is larger[-2]	
Resynchronization jump width (SJW)	Some units may get out of sync for reasons of a drift in clock frequency or a delay in transmission path. SJW is the maximum width in which this out-of-sync is corrected for.	1 – 4 and ≦PBS1	

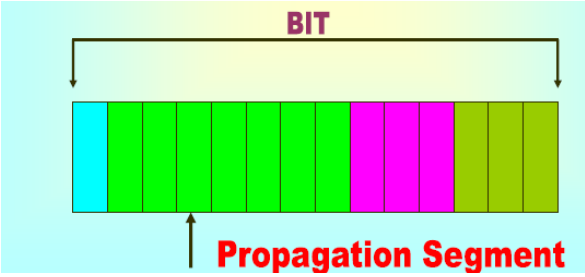
Synchronization Segment:

- Synchronize the various CAN nodes on the bus
- Edge is expected within this segment
- Always 1 Tq long



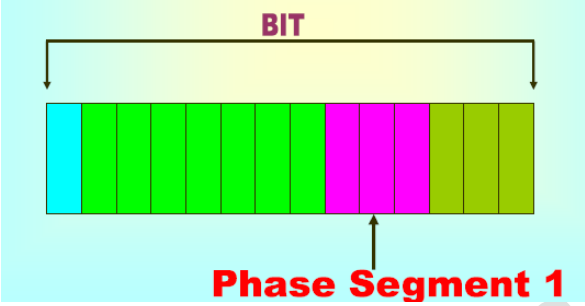
Propagation Segment:

- Needed to compensate physical delays in the bus line
- Is programmable to be 1..8 Tq long
- t propagation = 2 (t cable + t controller + t optocoupler + t transceiver)
  CAN controller (50 ns to 62 ns), Optocoupler (40 ns to140 ns), Transceiver (120 ns to 250 ns), and Cable (about 5 ns/m).



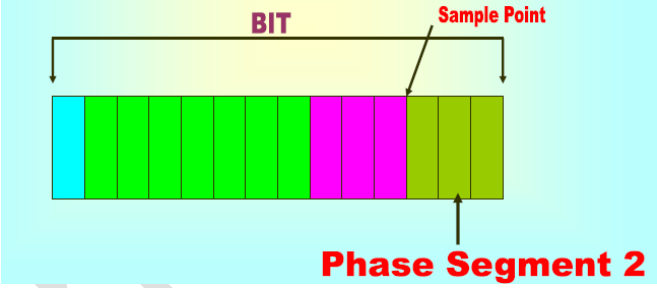
Phase Segment 1:

- Used to compensate for positive phase errors
- Can be lengthened during resynchronization
- Is programmable to be 1...8 Tq long

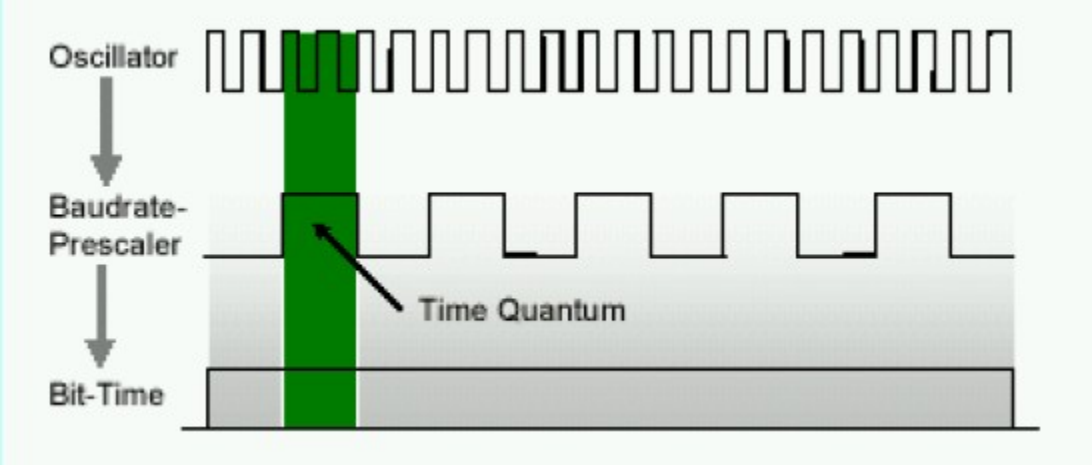


Phase Segment 2:

- Used to compensate for negative phase errors
- Can be shortened during resynchronization
- Maximum of Phase_Seg1
- Sample point lies before the start of TSEG2



Bit Timing (the transmission rate):



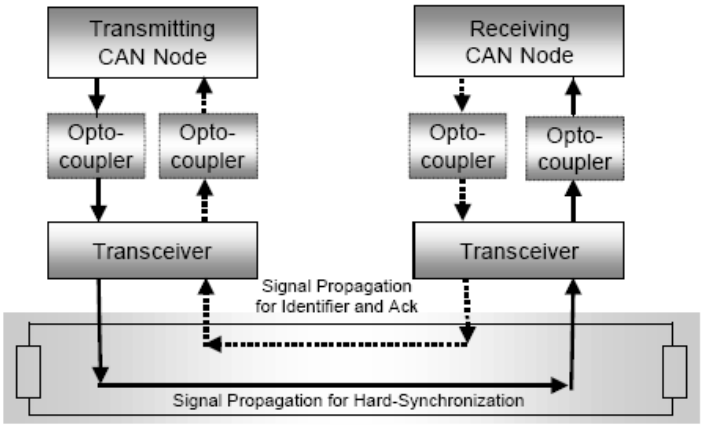
Number of Bits / Frame:

In an Extended CAN network the time is calculated as follows:

1 Start bit
+ 11 Identifier bits
+ 1 SRR bit
+ 1 IDE bit
+ 18 Identifier bits
+ 1 RTR bit
+ 6 Control bits
+ 64 Data bits
+ 15 CRC bits
+ 23 (maximum) Stuff bits
+ 1 CRC delimiter
+ 1 ACK slot
+ 1 ACK delimiter
+ 7 EOF bits
+ 3 IFS (Inter Frame Space) bits
= 154 bits

This results in a maximum delay time for the message with the highest priority of 154 bit times (e.g. 154 µs with maximum transmission rate of 1 MBit/s).
The data rate is the ratio of data bits to the total number of bits of the message (including all necessary framing bits) and depends on the type of message, data length and transmission rate.

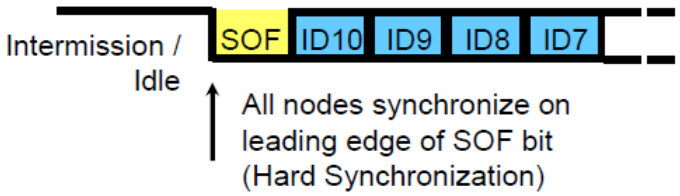
Signal Propagation delays:



tpropagation = 2 (tcable + tcontroller + toptocoupler + ttransceiver)
tpropagation = 2(50ns/m + 50 ns to 62 ns + 40 ns to 140 ns + 120 ns to 250 ns)

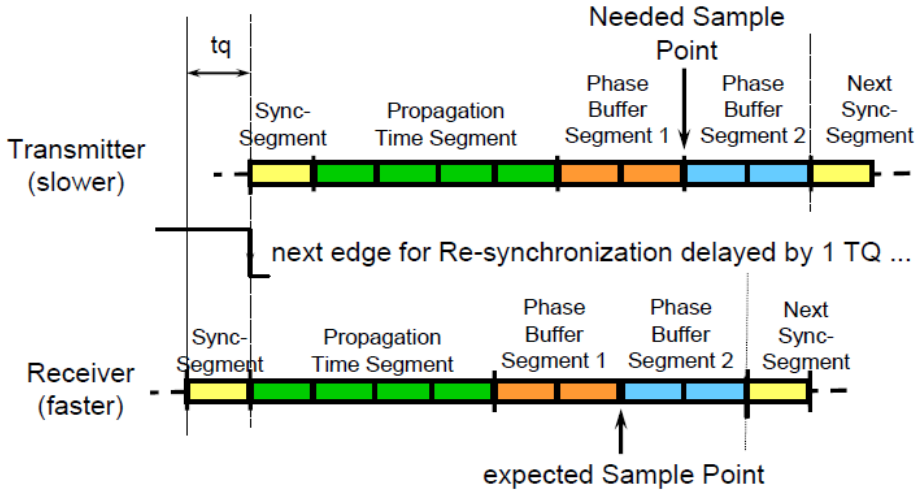
Synchronization:

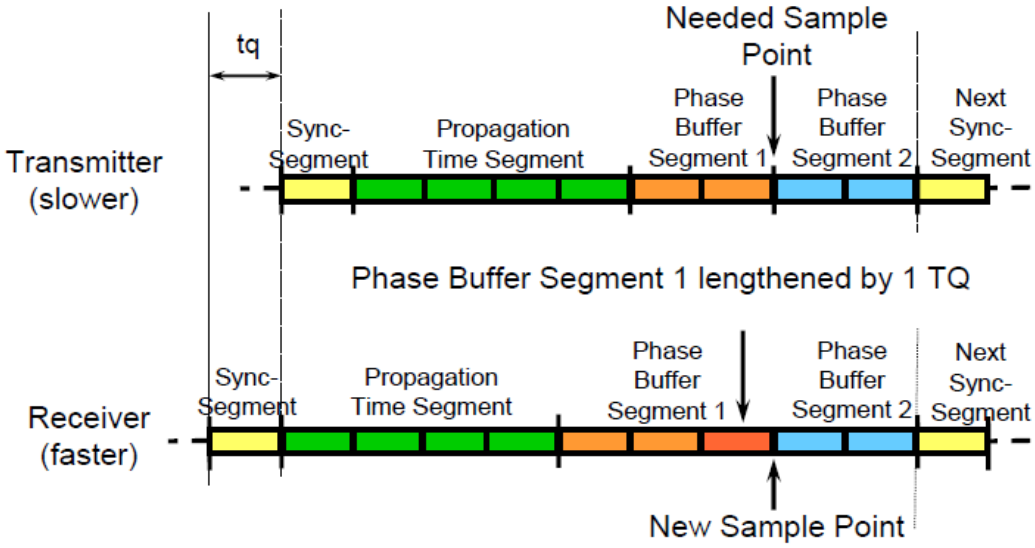
Hard Synchronization: Occurs on the recessive-to-dominant transition of the start bit. The bit time is restarted from that edge



Soft Synchronization: Occurs when a bit edge doesn't occur within the Synchronization Segment in a message. One of the Phase Segments are shortened or lengthened with an amount of Synchronization Jump Width which is programmable.

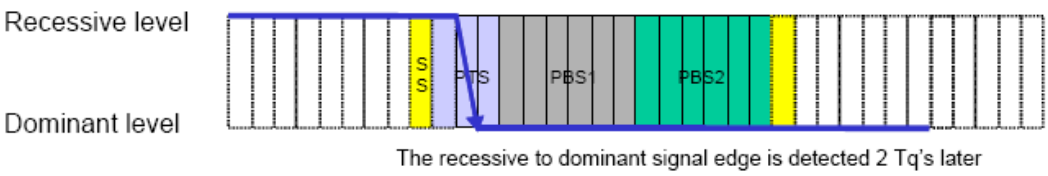
Resynchronization (Bit Lengthening) :



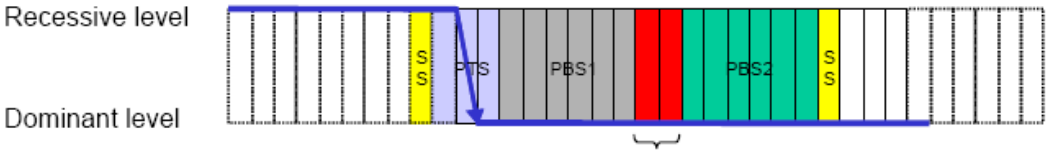


Slower transmitter:

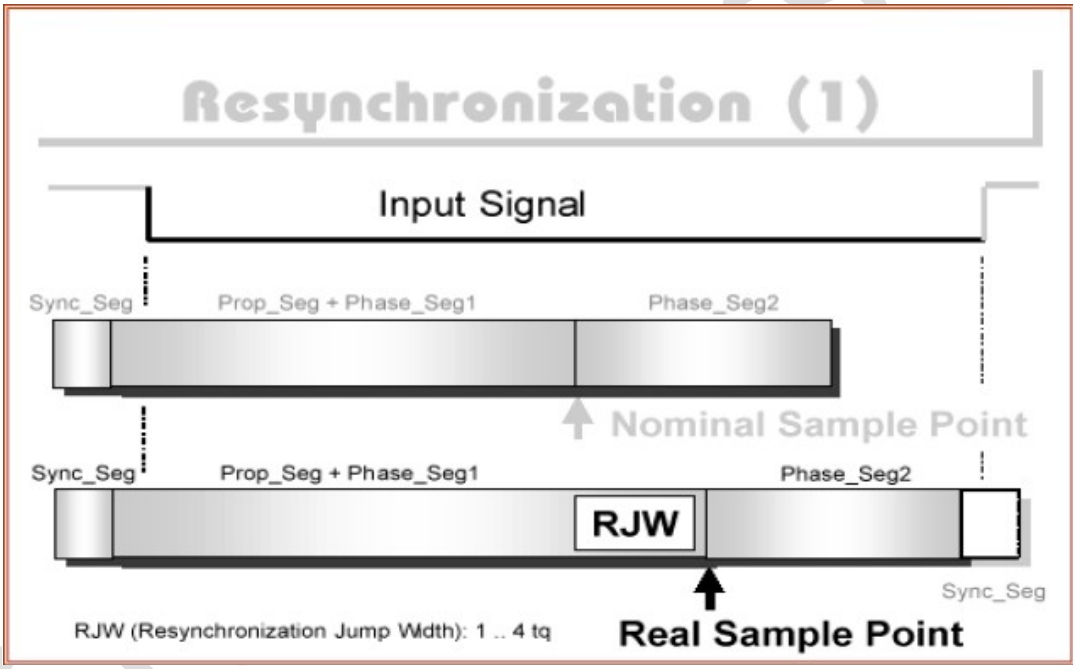
When a recessive to dominant signal edge occurs during PTS (example for SJW = 2)



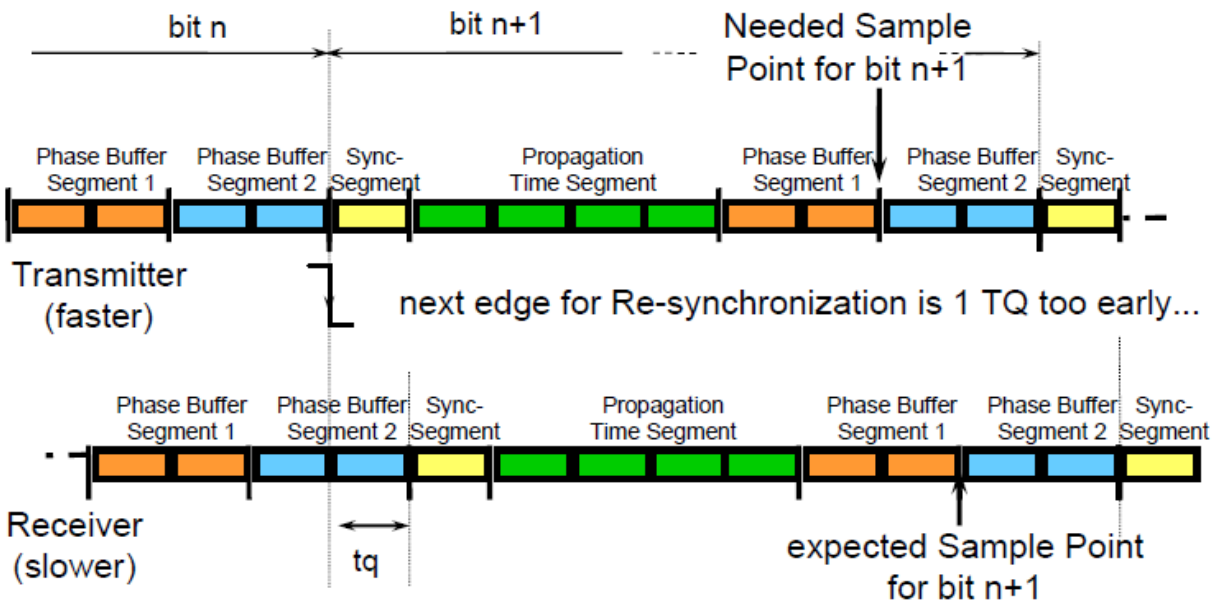
The recessive to dominant signal edge is detected 2 Tq's later

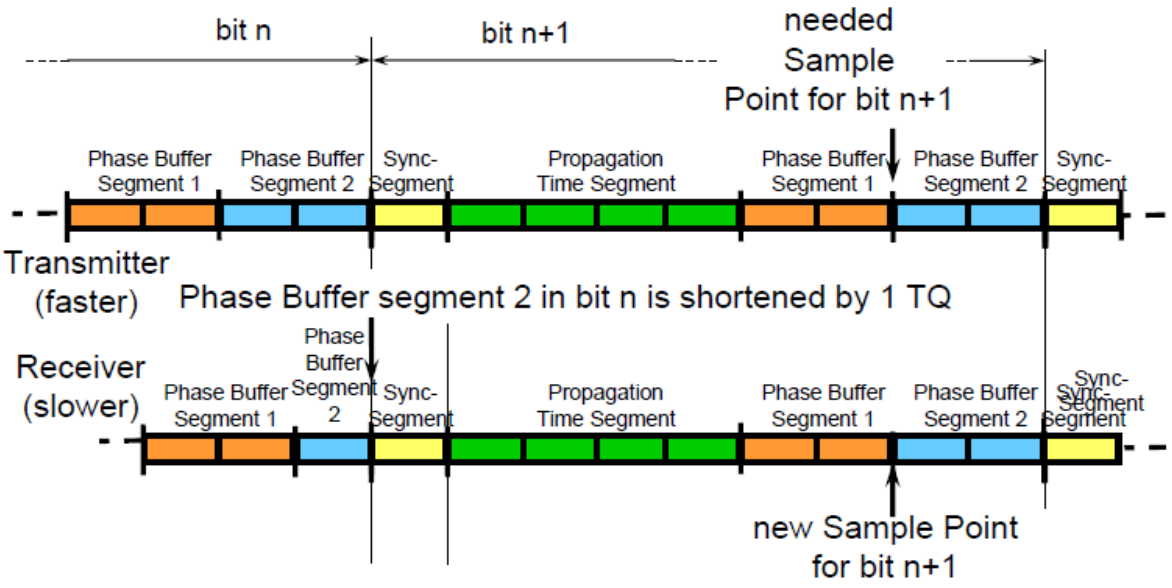


The SJW length for PSB1 is extended (by adding 2 Tq's) to adjust the timing error for resynchronization.



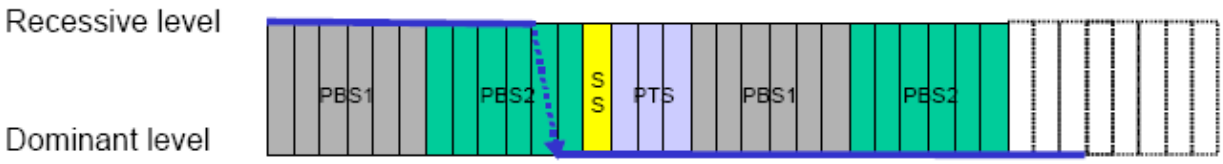
Resynchronization (Bit Shortening) :



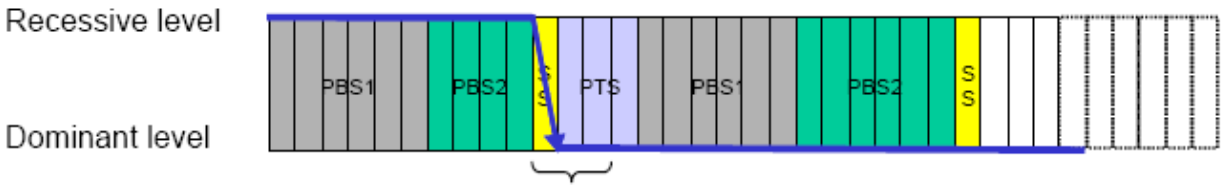


Faster transmitter:

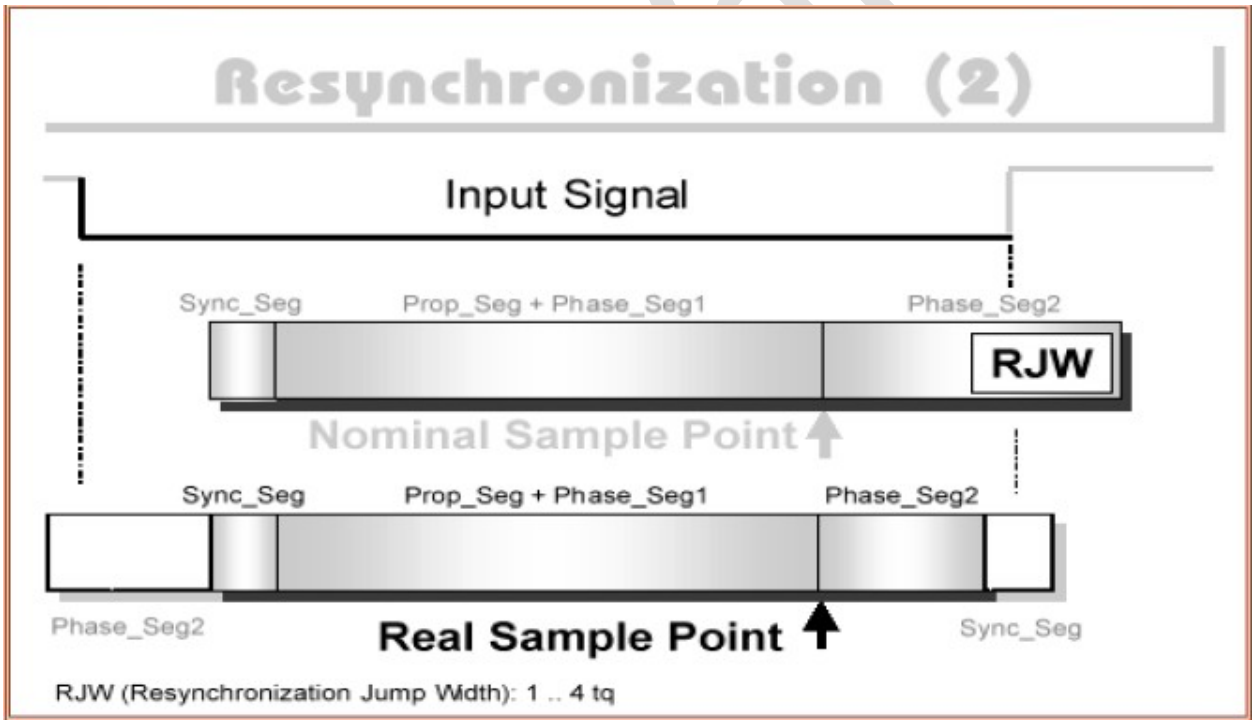
When a recessive to dominant signal edge occurs during PBS2 (example for SJW = 2)



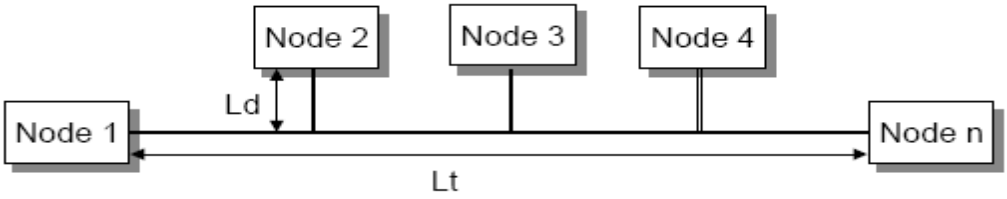
The recessive to dominant signal edge is detected 2 Tq's earlier



The SJW length for PSB2 is contracted (by subtracting 2 Tq's) to adjust the timing error for resynchronization.



Calculation of Cable Drop Length :



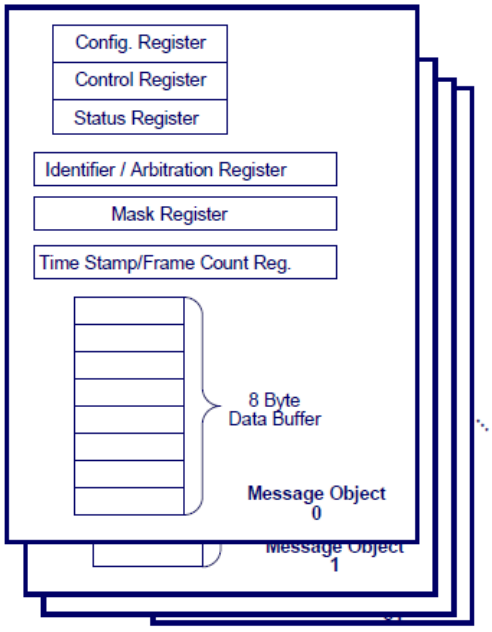
Ld = Drop Length *Lt = Trunk Length*

$t_{PROPSEG}$: length of the propagation segment of the bit period
t_P : specific line delay per length unit

Example: bit rate = 500 kbit/s: $t_{PROPSEG}$ = 12 * 125ns = 1500 ns; t_P = 5 ns/m

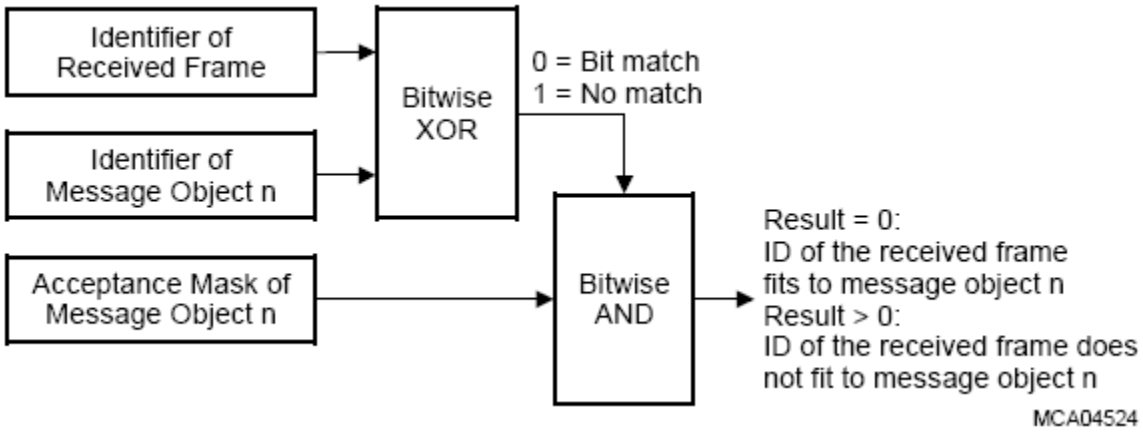
$Ld < 1500\ ns / (50 * 5\ ns/m) = 6\ m$; $\sum_{n} Ldi < 1500\ ns / (10 * 5\ ns/m) = 30\ m$

CAN Driver Implementation:

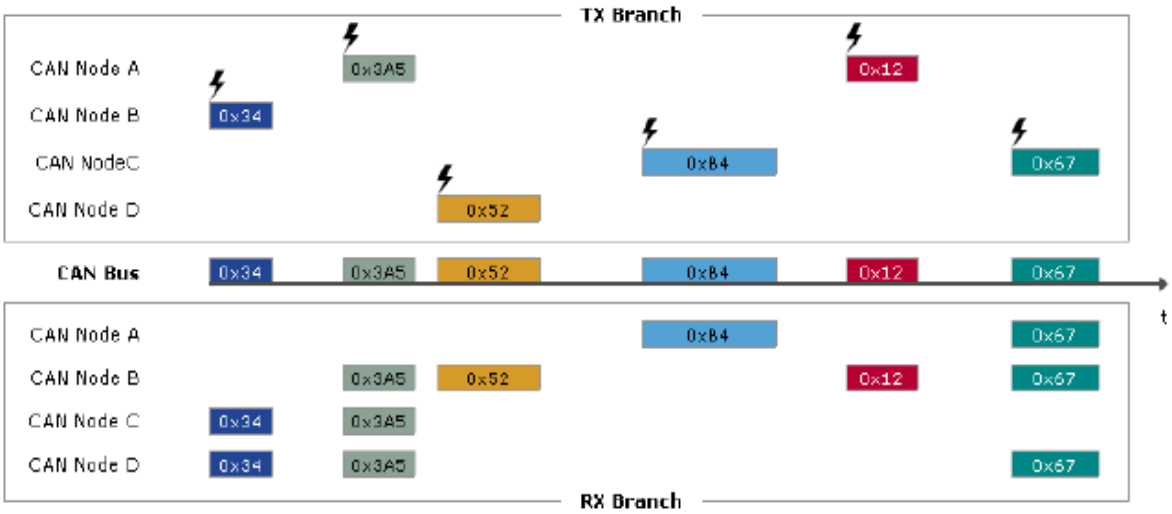


- A Message Object is more than a Buffer
  - Config Registers
    - Tx, Rx, 11 or 29-bit ID, etc.
  - Control Registers
    - Tx Request, Interrupt enable, etc.
  - Status Registers
    - Rx & Tx Status, Interrupt request, etc.
  - Identifier/Arbitration Registers
    - 11 or 29 bit ID
  - Mask Registers*
    - Specifies which bits of ID are "don't cares"
  - Time Stamp / Frame Count*
    - Indicates when the message arrived or was transmitted
  - 8-byte buffer for Rx or Tx data

Acceptance Mask Registers:



Data Frame	CAN Node A	CAN Node B	CAN Node C	CAN Node D
ID = 0x12	Sender	Receiver		
ID = 0x34		Sender	Receiver	Receiver
ID = 0x52		Receiver		Sender
ID = 0x67	Receiver	Receiver	Sender	Receiver
ID = 0xB4	Receiver		Sender	
ID = 0x3A5	Sender	Receiver	Receiver	Receiver



CAN Applications:

- Elevators

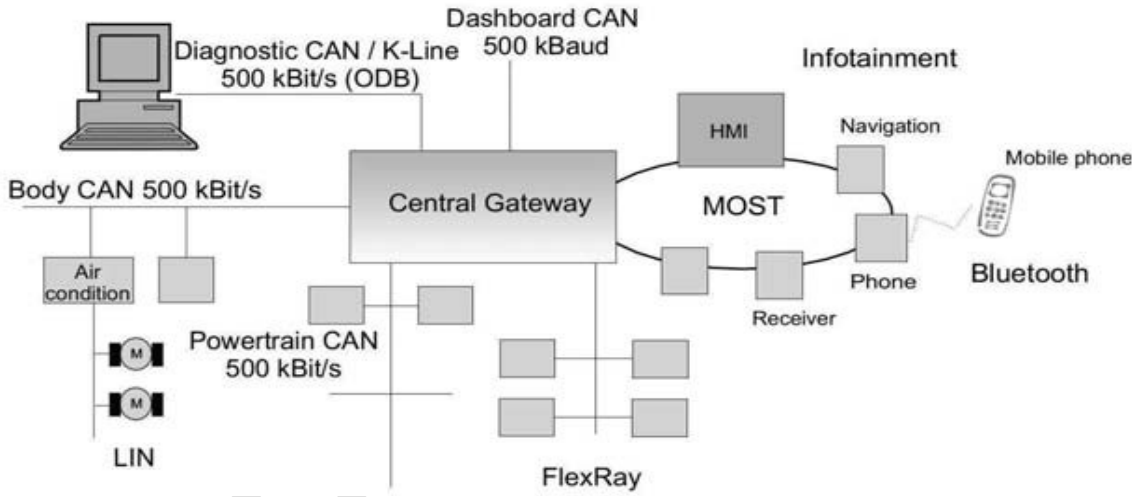
- Escalators
- Agriculture Machinery
- Medical systems
- Textile Production
- Home Appliances
- Vending Machines
- Robots, numeric machine tools
- Office control systems
- Trains, Ships, Planes etc.

Cranes Varsity

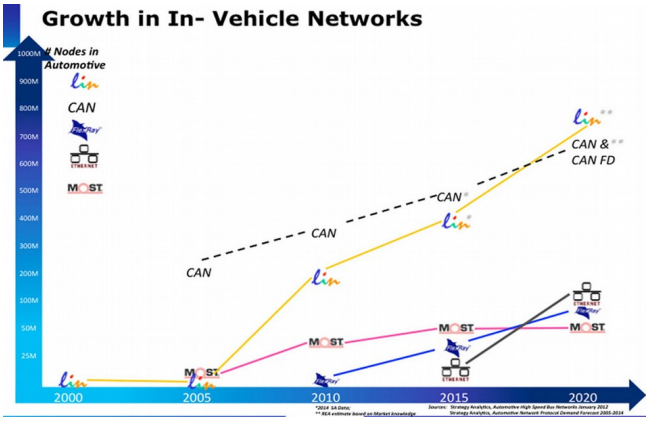
CHAPTER -28
Introduction to Vehicle Networks

Introduction to Bus Network in vehicles

Automotive communication protocols are changing rapidly as users and systems are demanding more information to be available in-vehicle. This dramatically increases bandwidth requirements to support new functionality, interaction between modules, and "hungry" signals such as multimedia. In designing with this trend in mind, automotive engineers face time limitations of 100 seconds during end of line programming, demand for increasing vehicle network bandwidth and cost challenges. Using an analysis of the existing automotive networks available, the opportunities for protocol improvements and vehicle network analysis, this article will outline how engineers can address these challenges.



Vehicle communications have been growing at a steady rate since the first Electronic Control Modules (ECMs) started to "think outside" of their own box and interact with other ECMs. For years, the radio was the only electronic device in vehicle, but starting with regulations implemented to reduce automobile emissions and followed by the growth of the semiconductor industry, electronics made their way into almost every facet of the vehicle. Within the last decade (Figure below) protocols such as Local Interconnect Network (LIN) and CAN (Controller Area Network), have grown in popularity even as higher bandwidth protocols such as MOST (Media Oriented Systems Transport), FlexRay, and Ethernet have become accepted standards in automotive communications design. Furthermore, it is expected that LIN and CAN networks will continue to dominate vehicle communication over the next decade. This article will discuss changes in these protocols and how engineers can continue to use these protocols given the increased bandwidth requirements.



LIN and CAN are amongst the most popular protocols in-vehicle and have existed as standardized communication protocols for some time. LIN, which emerged in the mid 1990s, is single master, single wire 12 Volt and transfers data up to 20 Kbit/s. CAN proliferated in the late 1990s as a multi- master, typically dual wire (at higher frequencies) 5 Volt and up to 500 Kbit/s. These two protocols' success has ensured that their use will continue to increase well into the 2020's.

Automotive Networking Protocols(CAN, FLEXRAY,LIN,MOST)

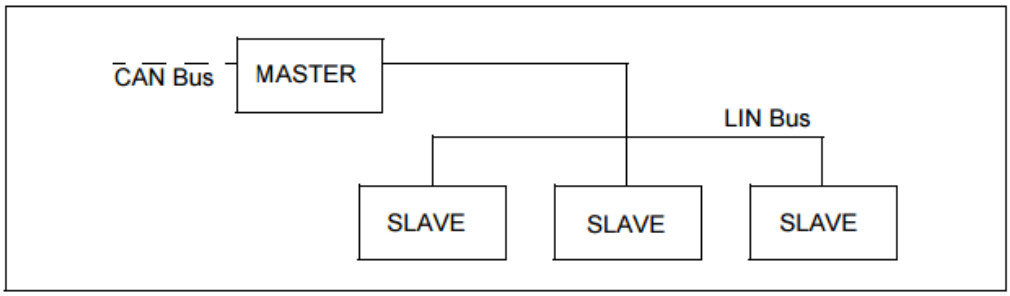
LIN

- **Introduction**

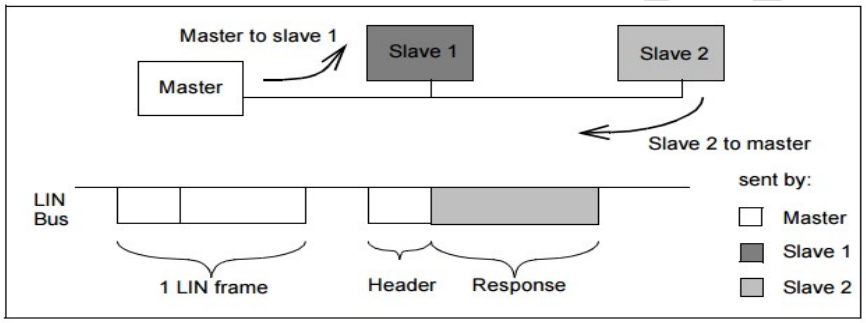
  LIN - developed by the LIN Consortium and a de-facto standard - was specially developed to achieve cost-effective communication for intelligent sensors and actuators in motor vehicles, and it is used wherever the bandwidth and versatility of CAN are not needed. The LIN specification includes the LIN protocol, a uniform format for the description of an entire LIN network and the interface between a LIN network and the application.

  The LIN bus is a sub-bus system based on a serial communications protocol. The bus is a single master / multiple slave bus that uses a single wire to transmit data. To reduce costs, components can be driven without crystal or ceramic resonators. Time synchronization permits the correct transmission and reception of data. The system is based on a UART / SCI hardware interface that is common to most microcontrollers. The bus detects defective nodes in the network. Data checksum and parity check guarantee safety and error detection.

Cranes Varsity



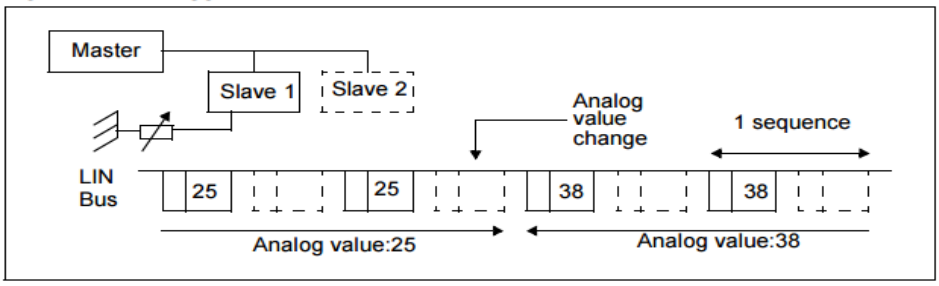
The LIN concept uses a single master / multiple slave model. Only the master is able to initiate a communication. A LIN frame consists of a header and a response part. To initiate a communication with a slave the master sends the header part. If the master wants to send data to the slave it goes on sending the response part. If the master requests data from the slave the slave sends the response part.



Direct communication between slaves is not possible.

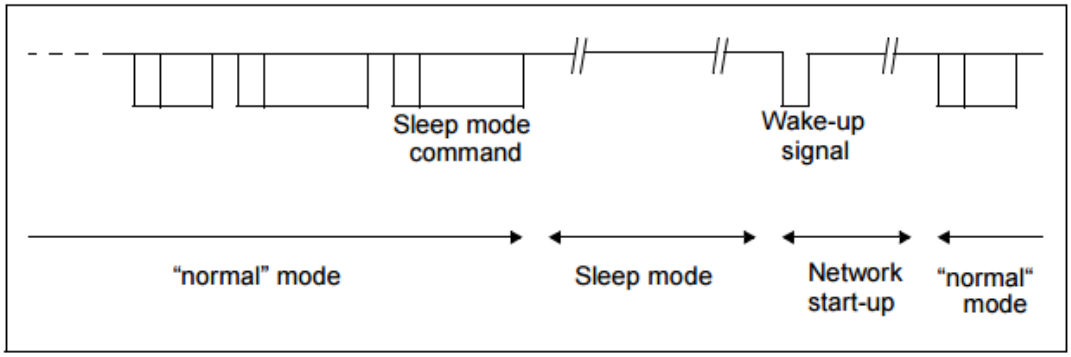
The LIN protocol is message-oriented and not address-oriented. The header contains the identifier which identifies the LIN frame and the data it contains. Different nodes may receive the same frame data. The response part consists mainly of data of selectable length (1 to 8 bytes). The data are secured by an 8 bit checksum. The LIN protocol is time-trigger oriented. The master periodically sends the same sequence of LIN frames. Each sequence, the master and the slaves update the data they send and receive. The sequence sent by the master may change depending on application events. Example: The slave is a sensor measuring a analog value which is communicated to the master via LIN. The slave continuously measures its analog input independently from the LIN communication. In response to a master request (periodical) the slave sends the up-to-date/ last measured value of the analog input.

Figure 3. Time-triggered protocol

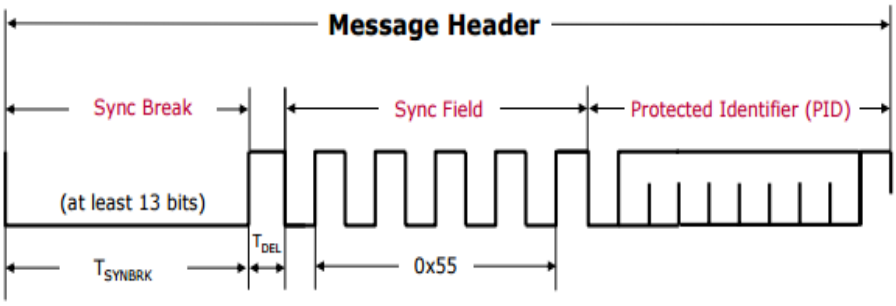


In order to achieve a good level of security, different mechanisms exist like parity bits on the identifier or checksum on data bytes. One important feature of the protocol is to enable the slave MCUs to run with low cost oscillators such as an integrated RC oscillator provided that the accuracy is better than +/-15%. For this purpose the header contains a sync field byte consisting of the constant 0x55. This byte enables each slave to measure the master bit time and to synchronize its clock accordingly. In order to obtain very low power consumption, the master is able to send a sleep frame. Any node can go into low power mode. To wake up the network, any node can send a so-called wake-up signal

Figure 4. Sleep mode - wake-up



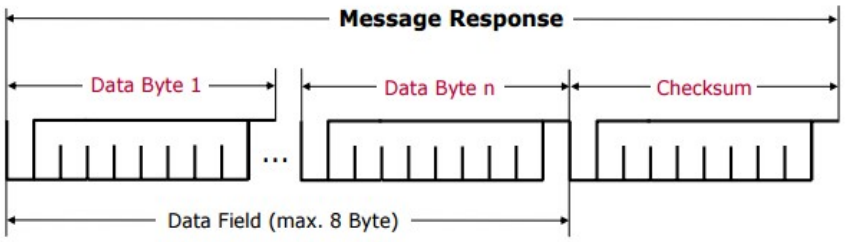
- **LIN Message Formats**
  - **Message Header**
    - The Token is referred to as the Message Header
    - The Message Header is sent by the Master Task
    - The Message Header is used for synchronization
    - The Message Header includes the identifier



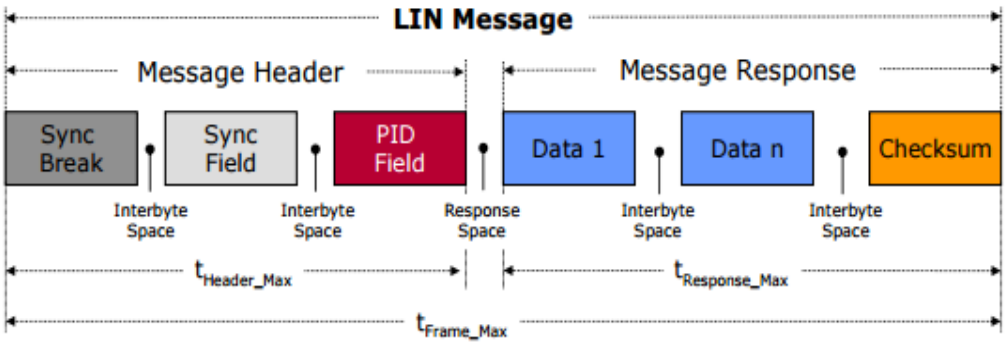
The token is referred as message header in LIN Network. The entire message header is transmitted by LIN master. It is made up of Sync Break, Sync Field and Protected Identifier (PID). Both Sync Break and Sync Field are used for initial synchronization. The PID is comprised of the message address (Identifier) and two parity bits. According to the message address the LIN nodes decide what they do immediately after the message header (send, receive or ignore the message response).

- **Message Response**

  - The Message Response is sent by a Slave Task
  - The Message Response is comprised of the data and checksum
    - Classic checksum over data field (LIN 1.x)
    - Enhanced checksum over data field and ID field (LIN 2.0)



`The message response is sent by a slave- task delegated for this purpose based on the message address. A maximum of eight data bytes may be transmitted with a message response. It should be noted that byte transmission begins with LSB. In principle a message response may be received and accepted by all slaves-tasks.



The data bytes are protected with the help of a checksum. Checksum formation is based on Modulo- 2 arithmetic and carry bit over all data bytes. The individual data bytes are added by Modulo-2 arithmetic. Overview bits are carried. Finally the result is inverted.

LIN Overview

1. LIN is a broadcast serial network comprising 16 nodes (one master and typically up to 15 slaves).

2. All messages are initiated by the master with at most one slave replying to a given message identifier. The master node can also act as a slave by replying to its own messages. Because all communications are initiated by the master it is not necessary to implement a collision detection.

3. The master and slaves are typically microcontrollers, but may be implemented in specialized hardware or ASICs in order to save cost, space, or power.

4. Single wire communications up to 19.2 kbit/s @ 40 meter bus length.In the LIN specification 2.2 the speed up to 20 kbit/s

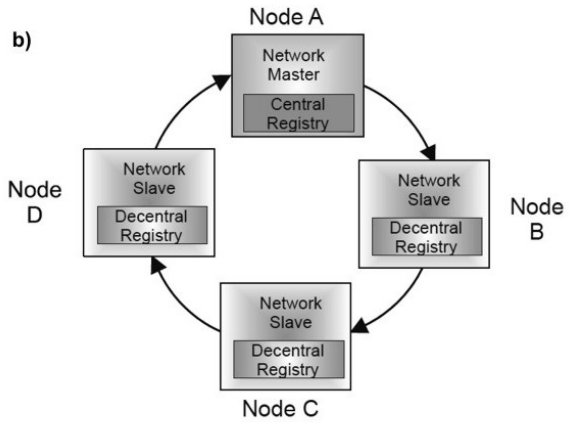
MOST

MOST (Media Oriented Systems Transport) is a high-speed multimedia network technology optimized by the automotive industry. It can be used for applications inside or outside the car. The serial MOST bus uses a daisy-chain topology or ring topology and synchronous data communication to transport audio, video, voice and data signals via plastic optical fiber (POF) (MOST25, MOST150) or electrical conductor (MOST50, MOST150) physical layers.

MOST technology is used in almost every car brand worldwide, including Audi, BMW, GeneralMotors, Hyundai, Jaguar,Lancia, LandRover, Mercedes-Benz, Porsche, Toyota, Volkswagen, SAAB, SKODA, SEAT and Volvo. SMSC and MOST are registered trademarks of Standard Microsystems Corporation ("SMSC"), now owned by Microchip Technology.

The first multimedia installation based on MOST bus and protocol was introduced in the year 2001. In the same year, MOST bus was applied in the next ten vehicle models. In the year 2013, MOST Cooperation consortium could report MOST introduction into 140 vehicle models including new models i.e. Audi A3 and Mercedes class S. MOST bus and protocol have been present in popular medium segment vehicles e.g. Volkswagen Golf and Opel Insignia as well as the models: Rolls Royce Ghost, Phantom and Wraith. The functioning of majority of wire communication buses in motor vehicles is based on linear bus topology. Therefore MOST bus is a unique solution because it is based on ring topology (Fig. 1). The application of fiber optic solution is another specific feature. The communication via cable connections is possible after transceivers replacement.

Cranes Varsity



MOST bus operation is typical for ring topology. The data block received from preceding node is used as information and commands source. The block received from preceding node is regenerated and forwarded. Turned off devices transmit optical signal without its analysis. The data transfer is finished when the block is received by its sender. The ring contains some special nodes responsible for the ring management i.e. commands generation on the basis of user activity and for the ring synchronization (Fig. 1b). MOST protocol and bus are dedicated to multimedia networks which are sometimes called Infotainment networks. High throughput levels are required for data stream in such networks. Despite MOST150 standard functioning since several years, this fact has been not mentioned in many publications. Most often the graphical presentations inform about the throughput of about 25 Mbps (Fig. 2)which is underestimated by three times. The throughput of 150 Mbps will be probably exceeded soon. The manufacturers of Plastic Optical Fibers (POF) indicate the throughputs of 500 Mbps along the section of 20 m or 170 Mbps along the section of 115 m. The transceiving equipment is prepared for operation with throughput of 5 Gbps. The current throughput is sufficient to use MOST as an element in the network supporting images received from security camera or from the games network.

Parameter	MOST25	MOST50	MOST150
Streaming data			
Minimum bandwidth [Mbps]	8.48	0.38	0
Maximum bandwidth [Mbps]	21.17	44.93	142.85
Packet data			
Minimum bandwidth [Mbps]	0	0	0
Maximum bandwidth [Mbps]	10.84	44.54	142.85
Control data			
Minimum bandwidth [kbps]	405.84	448.00	512.00
Maximum bandwidth [kbps]	405.84	810.62	1130.00

Bandwidth of MOST 25, 50 and 150 frame

Cranes Varsity

FLEXRAY

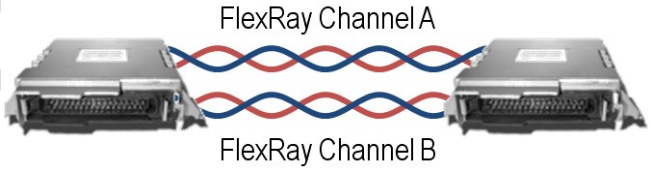
Many aspects of FlexRay are designed to keep costs down while delivering top performance in a rugged environment. FlexRay uses unshielded twisted pair cabling to connect nodes together. FlexRay supports single- and dual-channel configurations which consist of one or two pairs of wires respectively. Differential signaling on each pair of wires reduces the effects of external noise on the network without expensive shielding. Most FlexRay nodes typically also have power and ground wires available to power transceivers and microprocessors.

Dual-channel configurations offer enhanced fault-tolerance and/or increased bandwidth. Most first-generation FlexRay networks only utilize one channel to keep wiring costs down, but as applications increase in complexity and safety requirements; future networks will use both channels.

FlexRay buses require termination at the ends, in the form of a resistor connected between the pair of signal wires. Only the end nodes on a multi-drop bus need termination. Too much or too little termination can break a FlexRay network. While specific network implementations vary, typical FlexRay networks have a cabling impedance between 80 and 110 ohms, and the end nodes are terminated to match this impedance. Termination is one of the most frequent causes of frustration when connecting a FlexRay node to a test setup. Modern PC-based FlexRay interfaces may contain on-board termination resistors to simplify wiring.

FlexRay Topology and Layout

One of the things that distinguish FlexRay, CAN and LIN from more traditional networks such as ethernet is its topology, or network layout. FlexRay supports simple multi-drop passive connections as well as active star connections for more complex networks. Depending a vehicle's layout and level of FlexRay usage, selecting the right topology helps designers optimize cost, performance, and reliability for a given design.



Multi-drop Bus

FlexRay is commonly used in a simple multi-drop bus topology that features a single network cable run that connects multiple ECUs together. This is the same topology used by CAN and LIN and is familiar to OEMs, making it a popular topology in first-generation FlexRay vehicles. Each ECU can "branch" up to a small distance from the core "trunk" of the bus. The ends of the network have termination resistors installed that eliminate

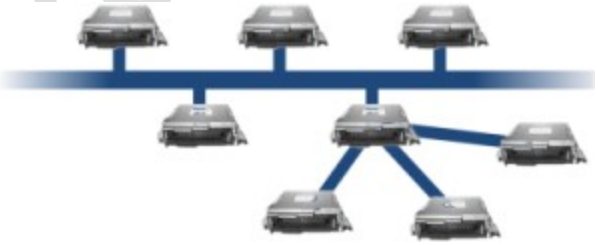
Cranes Varsity

problems with signal reflections. Because FlexRay operates at high frequencies, up to 10 Mbit/s compared to CAN's 1 Mbit, FlexRay designers much take care to correctly terminate and lay out networks to avoid signal integrity problems. The multi-drop format also fits nicely with vehicle harnesses that commonly share a similar type of layout, simplifying installation and reducing wiring throughout the vehicle.

Star Network

The FlexRay standard supports "Star" configurations which consist of individual links that connect to a central active node. This node is functionally similar to a hub found in PC ethernet networks. The active star configuration makes it possible to run FlexRay networks over longer distances or to segment the network in such a way that makes it more reliable should a portion of the network fail. If one of the branches of the star is cut or shorted, the other legs continuing functioning. Since long runs of wires tend to conduct more environmental noise such as electromagnetic emissions from large electric motors, using multiple legs reduces the amount of exposed wire for a segment and can help increase noise immunity.

Hybrid Network



The bus and star topologies can be combined to form a hybrid topology. Future FlexRay networks will likely consist of hybrid networks to take advantage of the ease-of-use and cost advantages of the bus topology while applying the performance and reliability of star networks where needed in a vehicle.

The FlexRay Protocol

The FlexRay protocol is a unique time-triggered protocol that provides options for deterministic data that arrives in a predictable time frame (down to the microsecond) as well as CAN-like dynamic event-driven data to handle a large variety of frames. FlexRay accomplishes this hybrid of core static frames and dynamic frames with a pre-set communication cycle that provides a pre-defined space for static and dynamic data. This space is configured with the network by the network designer. While CAN nodes only needed to know the correct baud rate to communicate, nodes on a FlexRay network must know how all the pieces of the network are configured in order to communicate.

As with any multi-drop bus, only one node can electrically write data to the bus at a time. If two nodes were to write at the same time, you end up with contention on the bus and data becomes corrupt. There are a variety of schemes used to prevent contention on a bus. CAN, for example, used an arbitration scheme where nodes will yield to other nodes if they see a message with higher priority being sent on a bus. While flexible and easy to expand, this technique does not allow for very high data rates and cannot guarantee timely delivery of data. FlexRay manages multiple nodes with a Time Division Multiple Access or TDMA scheme. Every FlexRay node is

Cranes Varsity

synchronized to the same clock, and each nodes waits for its turn to write on the bus. Because the timing is consistent in a TDMA scheme, FlexRay is able to guarantee determinism or the consistency of data deliver to nodes on the network. This provides many advantages for systems that depend on up-to-date data between nodes.

Embedded networks are different from PC-based networks in that they have a closed configuration and do not change once they are assembled in the production product. This eliminates the need for additional mechanisms to automatically discover and configure devices at run-time, much like a PC does when joining a new wired or wireless network. By designing network configurations ahead of time, network designers save significant cost and increase reliability of the network.

For a TDMA network such as FlexRay to work correctly, all nodes must be configured correctly. The FlexRay standard is adaptable to many different types of networks and allows network designers to make tradeoffs between network update speeds, deterministic data volume, and dynamic data volume among other parameters. Every FlexRay network may be different, so each node must be programmed with correct network parameters before it can participate on the bus.

To facilitate maintaining network configurations between nodes, FlexRay committee standardized a format for the storage and transfer of these parameters in the engineering process. The Field Bus Exchange Format, or FIBEX file is an ASAM-defined standard that allows network designers, prototypers, validaters, and testers to easily share network parameters and quickly configure ECUs, test tools, hardware-in-the-loop simulation systems, and so on for easy access to the bus.

Comparison of different Vehicle network protocols

BUS	LIN	CAN	FLEXRAY	MOST
Cost/Node [$]	1.50	3.00	6.00	4.00
Used in	Subnets	Soft real-time	Hard real-time	Multimedia
Applications	Body	Chassis, Powertrain	Chassis, Powertrain	Multimedia, Telematics
Message transmission	Synchronous	Asynchronous	Synchronous & Asynchronous	Asynchronous & Synchronous
Data rate	20 kbps	1 Mbps	10 Mbps	24 Mbps
Physical layer	Single Wire	Dual-Wire	Dual-Wire (Optical-Fiber)	Optical-Fiber (Dual-Wire)
Latency jitter	Constant	Load dependent	Constant	Data stream
Extensibility	High	High	Low	High

Cranes Varsity

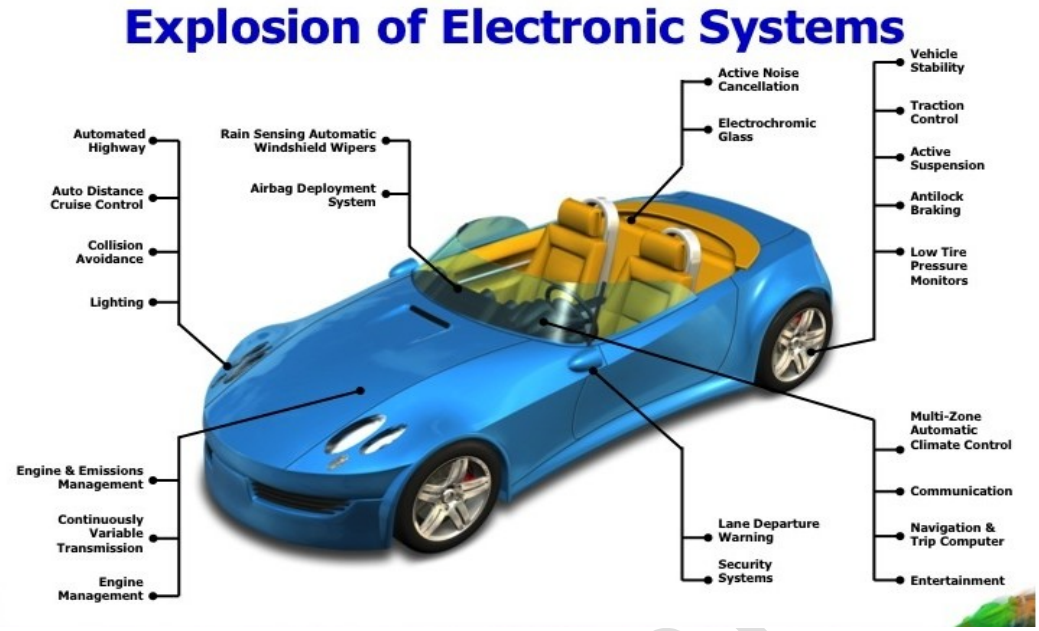
CHAPTER -30
Electronic Control Unit

ECU's inside a CAR

The automotive ECU can be subdivided into three major categories,

- Power train Control Module
- Body Control Unit
- Chassis System.

Let us consider Engine control unit which is a part of the Power train Control Module (PCM). In modern vehicles, the basic working principle of engine operation is still based on combustion, only difference is that the process is now controlled by the ECU. The engine ECU controls the opening and closing of the input/output valve, by taking input from the accelerator of pedal of the vehicle. The engine ECU is also responsible for the clockwork of the amount of fuel injection and spark ignition. In this way, the Engine ECU results in accurate synchronization, rendering more power, efficiency and highly functional engines, to the vehicles. In this way, ECU controlled vehicles are able to deliver higher efficiency as compared to mechanical automobiles.



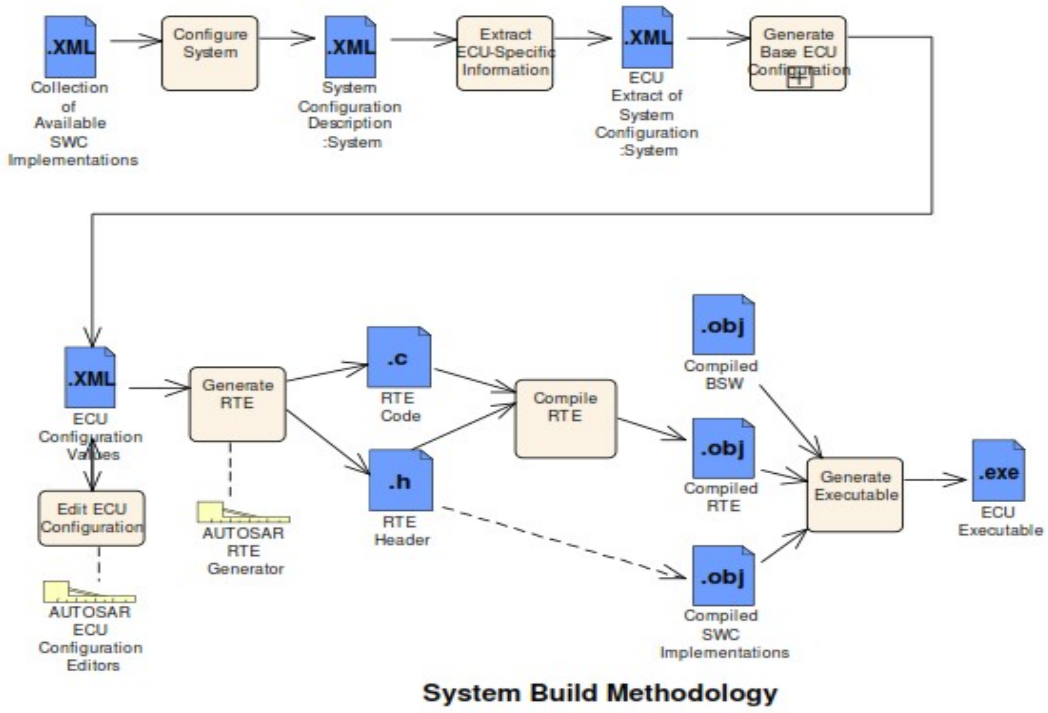
Source: chip estimate blog

An ECU consists of a number of functional blocks:

1. Power Supply – digital and analog (power for analog sensors)
2. MPU – microprocessor and memory (usually Flash and RAM)
3. Communications Link – (e.g. CAN bus)
4. Discrete Inputs – On/Off Switch type inputs
5. Frequency Inputs – encoder type signals (e.g. crank or vehicle speed)
6. Analog Inputs – feedback signals from sensors
7. Switch Outputs – On/Off Switch type outputs
8. PWM Outputs – variable frequency and duty cycle (e.g. injector or ignition)
9. Frequency Outputs – constant duty cycle (e.g. stepper motor – idle speed control).

The ECU uses closed-loop control, a control scheme that monitors outputs of a system to control the inputs to a system, managing the emissions and fuel economy of the engine (as well as a host of other parameters). Gathering data from dozens of different sensors, the ECU performs millions of calculations each second, including looking up values in tables, calculating the results of long equations to decide on the best spark timing or determining how long the fuel injector is open. A modern ECU might contain a 32-bit, 40-MHz processor, which may not sound fast compared to the processors we probably have in our PCs, but the processor in our car runs a much more efficient code. The code in an average ECU takes up less than 1 megabyte (MB) of memory. By comparison, we probably have at least 2 gigabytes (GB) of programs on our computers -- 2,000 times the amount in an ECU.

Overview about Design and Development of ECU



System Build Methodology

Overview about tools

Steps	Tools
Logical and Software architecture design	PREE Vision
Development of Application Software	vVIRTUAL target
SWC detailed Design	Da vinci developer
RTE & BSW (layers) Configuration	Da vinci configuaration Pro
Calibration	CANape
ECU System test	CANoe
ECU Monitoring and Debugging	CANoe.AMD

Cranes Va

CHAPTER -31
Introduction to AUTOSAR

31.1 AUTOSAR

Today's vehicles that are rolling out from the factories are basically computers with wheels. A modern vehicle consists of approximately 50-60 ECUs and the numbers are increasing for each year. Each unit is in charge of a specific functionality and they communicate with each other over some kind of bus, e.g. CAN, Flex-Ray, LIN etc. For each additional unit that is connected to the system, the system complexity increases. Applications and software must be configured for each system and with new hardware the applications must be rewritten each time to support any changes in the hardware.

In order to handle this increased complexity, which would eventually become unmanageable, a handful of leading Original Equipment Manufacturers (OEM) and Tier 1 suppliers from the automotive industry, decided to work together to form a new standard that would serve as a platform for future vehicle applications, and their result is AUTOSAR.

AUTOSAR (AUTomotive Open System ARchitecture) is a worldwide development partnership of automotive interested parties founded in 2003. It pursues the objective of creating and establishing an open and standardized software architecture for automotive electronic control units (ECUs). Goals include the scalability to different vehicle and platform variants, transferability of software, the consideration of availability and safety requirements, a

Cranes Varsity

collaboration between various partners, sustainable utilization of natural resources, and maintainability throughout the whole "Product Life Cycle".

The development of AUTOSAR is today a global cooperation between car manufacturers, suppliers and other companies from electronics, semiconductors and software industry. The cooperation started in 2003 with just a couple of the larger companies from the automotive industry and together they been developing a standardized software architecture for the automotive industry that is still being evolved.
One important thing to remember is that AUTOSAR does not specify a specific implementation; it is up to the tool vendors and other companies to follow the AUTOSAR standard specifications and implement according to them. One of the fundamental ideas behind AUTOSAR is reusable Software Components (SWCs) that can deal with the increasing complexity today and in the future. Software is tightly coupled with the ECU where it is going to be executed. If something is changed in the ECU, the software must be rewritten to suit with the hardware. It is problematic to buy software from one manufacturer and hardware from another, if they are not made to work with each other. Another example is for example one car manufacturer that has done all the work for one car in one of their production series, has to redo much work again to make the first system fit another car in a different production series. Depending on the manufacturer, models used and how variations are handled, this problem can be smaller or larger but the problem is still present in different scales. With a standardized interface it would be possible to buy software and hardware from different manufacturers, and they would all work together. This would not be as smooth with the conventional solution.

31.2 Why we need AUTOSAR

- Manage increasing E/E complexity – Associated with growth in functional scope.
- Improve flexibility – More room for updates, upgrades and modifications.
- Improve scalability – The system can in a more graceful manner be enlarged.
- Improve quality and reliability – Proven software applications can be reused.
- Detection of errors in early design phases.

Future engineering does not aim at optimizing single components but optimizing on system level which requires an open architecture as well as scalable and exchangeable software modules.Paves the way for innovative electronic systems that further improve performance, safety and environmental friendliness. Is a strong global partnership that creates one common standard: "Cooperate on standards, compete on implementation"

Is a key enabling technology to manage growing electrical/electronic complexity It aims to be prepared for the upcoming technologies and to improve cost-efficiency without making any compromise with respect to quality facilitates the exchange and update of software and hardware over the service life of the vehicle
To reduce development effort and improve quality are important reasons for introducing a uniform procedure independent of system platform. Hardware and software are decoupled from one another to assure such results. The AUTOSAR concept is based on modular components with defined interfaces.

31.3 Key Features

31.3.1 Modularity and configurability

- Definition of a layered basic software architecture for automotive electronic control units in order to encapsulate the HW dependencies
- Consideration of HW dependent and HW independent SW modules
- Enable the integration of basic SW modules provided by different suppliers to increase the functional reuse
- Enable the transferability of functional SW-components within a particular E/E-system at least at the final software linking process
- Resource optimized configuration of the SW infrastructure of each ECU depending on the function deployment
- Scalability of the E/E-system across the entire range of vehicle product lines

31.3.2 Standardized interfaces

- Standardization of different APIs to separate the AUTOSAR SW layers
- Facilitate encapsulation of functional SW-components
- Definition of the data types of SW-components
- Standardization of interfaces of basic SW modules of the SW infrastructure

31.3.3 Runtime Environment (RTE)

- Provision of inter- and intra-ECU communication across all nodes of a vehicle network
- Located between the functional SW-components and the basic SW-modules
- All entities connected to the AUTOSAR RTE must comply with the AUTOSAR specification
- Enables the easy integration of customer specific functional SW-modules

31.4 AUTOSAR Layer Model

Cranes Varsity

AUTOSAR uses a three-layered architecture:

- **Basic Software**: standardized software modules (mostly) without any functional job itself that offers services necessary to run the functional part of the upper software layer.
- **Runtime environment:** Middleware which abstracts from the network topology for the inter- and intra-ECU information exchange between the application software components and between the Basic Software and the applications. The RTE integrates the application layer with the BSW. It implements the data exchange and controls the integration between the application software component (SWCs) and BSW.
- **Application Layer:** application software components that interact with the runtime environment.

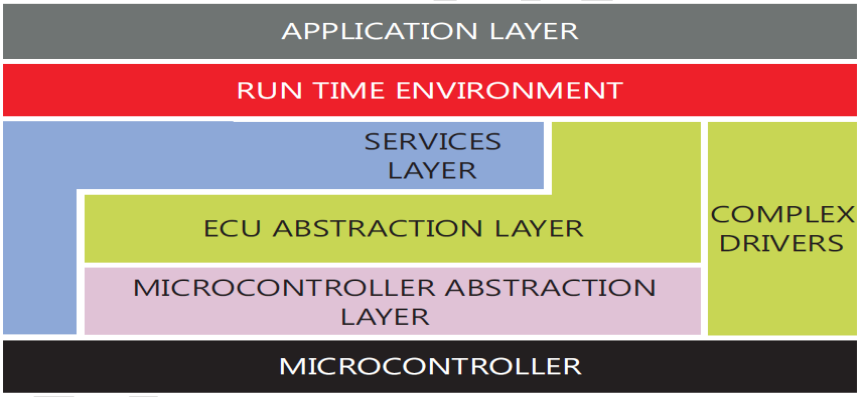
Basic Software

- The Basic Software (BSW) consists of Basic Software Modules (BSWM) as a collection of software files (code and description) that define certain basic software functionality present on an ECU.
- Standard software may be composed of several software modules that are developed independently. A software module may consist of Integration Code, and/ or standard software.
- the Basic Software can be classified as
  - **Services Layer:** The Service Layer provides various types of background services such as vehicle network communication and management services, diagnostic services, memory management, ECU state management, mode management and Logical and temporal program flow monitoring. The operating system is also part of this layer.
    - Communication Services
    - Memory Services
    - System Services
    - Diagnostic Services
  - **ECU Abstraction Layer**: The ECU Abstraction Layer offers uniform access to all features of an ECU like communication, memory or I/O, no matter if these features are part of the microcontroller or realized by peripheral components. The drivers for such external peripheral components reside in this layer.

Cranes Varsity

- Communication HW Abstraction
- Memory HW Abstraction
- Onboard Device Abstraction
- I/O HW Abstraction

- **Microcontroller Abstraction Layer:** The Microcontroller Abstraction Layer contains internal drivers, which are software modules with direct access to the micro controller and internal peripherals.
  - Communication Drivers
  - I/O Drivers
  - Memory Drivers
  - Microcontroller Drivers

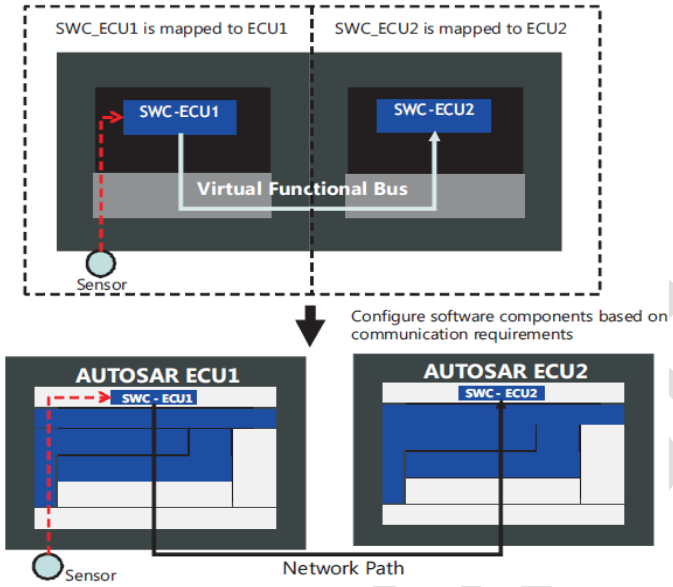


AUTOSAR Layered Architecture

31.5 Design of software Components

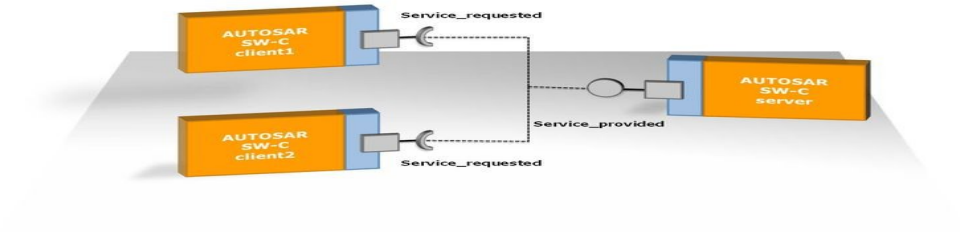
A fundamental design concept of AUTOSAR is the separation between Application and Infrastructure. An application in AUTOSAR consists of interconnected "AUTOSAR Software Components". The interfaces of each SWC are formally defined. Communication between SWCs takes place chiefly over two kinds of ports, Client/ Server ports where server is a provider of a service and the client is a user of a service and Sender/ Receiver ports where a sender distributes information to one or several receivers in synchronous as well as asynchronous environment. The implementation architecture of SWC is formally defined in terms of so-called runnable entities. They correspond to procedures and are executed on a specific event such as a periodic activation or reception of new input value. During system design phase the SWCs can be integrated with their environment (e.g. hardware, driver, OS, etc) based on Virtual Functional Bus (VFB). The virtual functional bus is the abstraction of the AUTOSAR Software Components

interconnections of the entire vehicle. Once the system of SWCs is deployed to the concrete vehicle network architecture, the RTE and BSW of involved ECUs realize the communication between the SWC either as ECU-local communication or as network based communication.



31.5 Communication:

31.5.1 Client-Server Communication

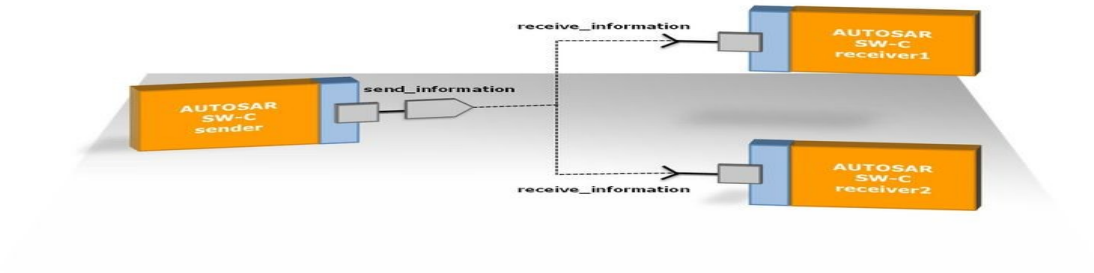


A widely used communication pattern in distributed systems is the client-server pattern, in which the server is a provider of a service and the client is a user of a service.

The client initiates the communication, requesting that the server performs a service, transferring a parameter set if necessary. The server waits for incoming communication requests from a client, performs the requested service, and dispatches a response to the client´s request. The direction of initiation is used to categorize whether an AUTOSAR Software Component is a client or a server. A single component can be both a client and a server, depending on the software realization.

The client can be blocked (synchronous communication) or non-blocked (asynchronous communication), respectively, after the service request is initiated until the response of the server is received. The image gives an example how client-server communication for a composition of three software components and two connections is modeled in the VFB view.

31.5.2 Sender-Receiver Communication



The sender-receiver pattern gives solution to the asynchronous distribution of information, where a sender distributes information to one or several receivers. The sender is not blocked (asynchronous communication) and neither expects nor gets a response from the receivers (data or control flow), i.e. the sender just provides the information and the receivers decides autonomously when and how to use this information. It is the responsibility of the communication infrastructure to distribute the information.

The sender component does not know the identity or the number of receivers to support transferability and exchange of AUTOSAR Software Components. The image illustrates an example how sender-receiver communication is modeled in the AUTOSAR VFB view.

Cranes Va

Cranes Varsity