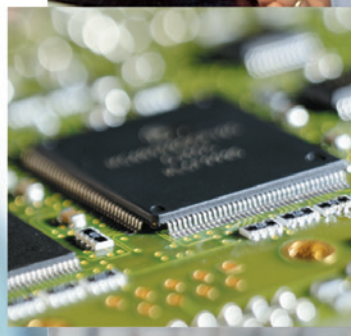
hitex
DEVELOPMENT TOOLS

The Insider's Guide To The STM32 ARM® Based Microcontroller



An Engineer's Introduction To The STM32 Series

www.hitex.com

Published by Hitex (UK) Ltd.

ISBN: 0-9549988 8

First Published February 2008

Hitex (UK) Ltd.
Sir William Lyons Road
University Of Warwick Science Park
Coventry, CV4 7EZ
United Kingdom

Credits

Author: Trevor Martin
Illustrator: Sarah Latchford

Editors: Michael Beach, Alison Wenlock
Cover: Wolfgang Fuller

Acknowledgements

The author would like to thank Matt Saunders and David Lamb of ST Microelectronics for their assistance in preparing this book.

Contents

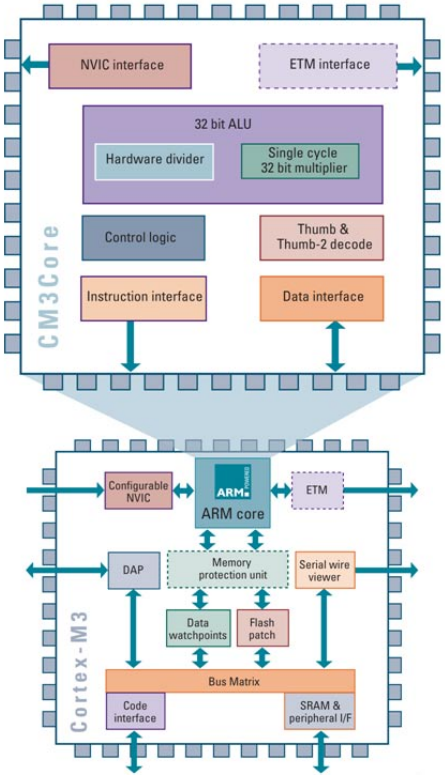
1. Introduction

Over the last six or seven years one of the major trends in microcontroller design is the adoption of the ARM7 and ARM9 as *the* CPU for general purpose microcontrollers. Today there are some 240 ARM-based microcontrollers available from a wide range of manufacturers. Now ST Microelectronics have launched the STM32, their first microcontroller based on the new ARM Cortex-M3 microcontroller core. This device sets new standards of performance and cost, as well as being capable of low power operation and hard real-time control.

1.1 So What Is Cortex?

The ARM Cortex family is a new generation of processor that provides a standard architecture for a wide range of technological demands. Unlike the other ARM CPUs, the Cortex family is a complete processor core that provides a standard CPU and system architecture. The Cortex family comes in three main profiles: the A profile for high end applications, R for real time and M for cost-sensitive and microcontroller applications. The STM32 is based on the Cortex-M3 profile, which is specifically designed for high system performance combined with low power consumption. It has a low enough cost to challenge traditional 8 and 16-bit microcontrollers.

While the ARM7 and ARM9 CPUs have been successfully integrated into standard microcontrollers, they do show their SoC heritage. This is particularly noticeable in the area of exception and interrupt handling, because each specific manufacturer has designed their own solution. The Cortex-M3 provides a standardised microcontroller core which goes beyond the CPU to provide the entire heart of a microcontroller (including the interrupt system, SysTick timer, debug system and memory map). The 4Gbyte address space of the Cortex-M3 is split into well-defined regions for code, SRAM, peripherals and system peripherals. Unlike the ARM7, the Cortex-M3 is a Harvard architecture and so has multiple busses that allow it to perform operations in parallel, boosting its overall performance. Unlike earlier ARM architectures, the Cortex family allows unaligned data accesses. This ensures the most efficient use of the internal SRAM. The Cortex family also supports setting and clearing of bits within two 1Mbyte regions of memory by a method called bit banding. This allows efficient access to peripheral registers and flags located in SRAM memory without the need for a full Boolean processor.



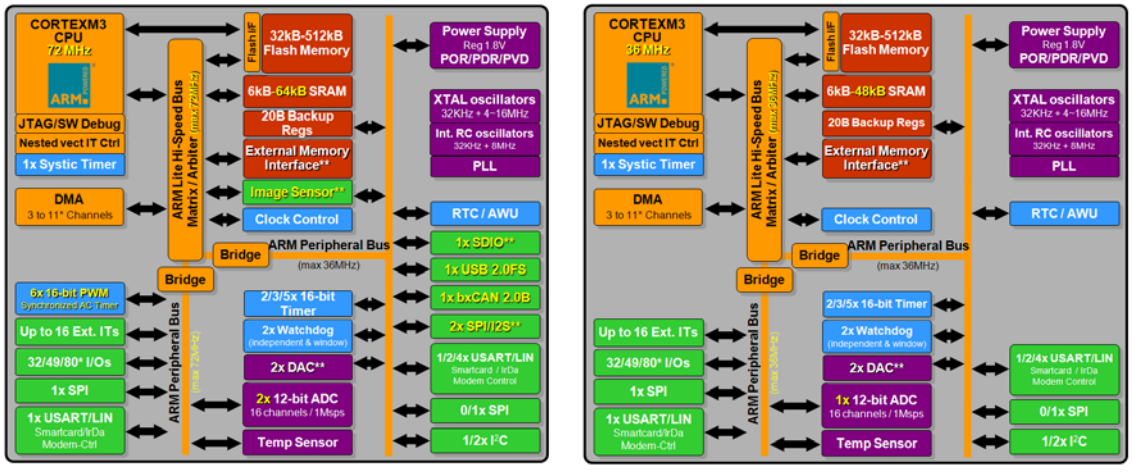
The heart of the STM32 is the Cortex-M3 processor. The Cortex M3 processor is a standardised microcontroller including 32 bit CPU, bus structure, nested interrupt unit, debug system and standard memory layout.

One of the key components of the Cortex-M3 core is the Nested Vector Interrupt Controller (NVIC). The NVIC provides a standard interrupt structure for all Cortex based microcontrollers and exceptional interrupt handling. The NVIC provides dedicated interrupt vectors for up to 240 peripheral sources where each interrupt source can be individually prioritised. The NVIC has been designed for extremely fast interrupt handling. The time taken from receiving an interrupt to reaching the first line of code in your service routine is just twelve cycles. This is achieved in part by automatic stack handling which is done by microcode within the CPU. In the case of back to back interrupts, the NVIC uses a “tail chaining” method that allows successive interrupts to be served with only a six cycle latency. During the interrupt stacking phase, a high priority interrupt can pre-empt a low priority interrupt without incurring any additional CPU cycles. The interrupt structure is also tightly coupled to the low power modes within the Cortex-M3 core. It is possible to configure the CPU to automatically enter a low power on exit from an interrupt. The core then stays asleep until another exception is raised.

Although the Cortex-M3 is designed as a low cost core, it is still a 32-bit CPU and as such has support for two operating modes: Thread mode and Handler mode, which can be configured with their own stacks. This allows more sophisticated software design and support for real-time operating systems. The Cortex core also includes a 24-bit auto reload timer that is intended to provide a periodic interrupt for an RTOS kernel. While the ARM7 and ARM9 CPUs have two instruction sets (the ARM 32-bit and Thumb 16-bit instruction sets) the Cortex family is designed to support the ARM Thumb-2 instruction set. This blends both 16 and 32-bit instructions, to deliver the performance of the ARM 32-bit instruction set with the code density of the Thumb 16-bit instruction set. The Thumb-2 instruction set is a rich instruction set that is designed as a target for C/C++ compilers. This means that a Cortex application can be entirely coded in C.

1.2 A Look At The STM32

ST already have four ARM7 and ARM9 based microcontroller families, but the STM32 is a significant step up the price/performance curve. With volume pricing at just over one Euro, the STM32 is a serious challenge to existing 8-bit microcontrollers.. . The STM32 was initially released with fourteen different variants.These are split into two groups: the Performance line which operates up to CPU clock speeds of 72MHz and the Access line which runs up to 36MHz. Both sets of variants are pin and software compatible and offer FLASH ROM sizes up to 128K and 20K SRAM. Since the initial release the STM32 road map has been extended to include devices with larger RAM and FLASH memories and more complex peripherals.



The STM32 family has two distinct branches. The Performance line which runs to 72MHz and has the full set of peripherals and the Access line which runs to 36MHz and has a reduced set of peripherals.

1.2.1 Sophistication

At first glance the peripheral set looks like a typical small microcontroller, featuring peripherals such as Dual ADC, general purpose timers, I2C,SPI,CAN,USB and a real-time clock. However, each of these peripherals is very

feature-rich. For example the 12-bit ADC has an integral temperature sensor and multiple conversion modes and devices with dual ADC can slave both ADCs together in a further nine conversion modes. Similarly, each of the four timers has four capture compare units and each timer block may be combined with the others to build sophisticated timer arrays. An advanced timer has additional support for motor control, with 6 complimentary PWM outputs with programmable dead time and a break input line that will force the PWM signal to a pre programmed safe state. The SPI peripheral has a hardware CRC generator for 8 and 16 words to support interfacing to SD and MMC cards.

Surprisingly for a small microcontroller, the STM32 also includes a seven channel DMA unit. Each channel can be used to transfer data to and from any peripheral register on memory location as 8/16 or 32-bit words. Each of the peripherals can be a DMA flow controller sending or demanding data as required. An internal bus arbiter and bus matrix minimise the arbitration between the CPU data accesses and the DMA channels. This means that the DMA unit is flexible, easy to use and really automates data flow within the microcontroller.

In an effort to square the circle the STM32 is a low power as well as high performance microcontroller. It can run from a 2V supply and at 72MHz with everything switched on it consumes just 36mA. In combination with the Cortex low power modes the STM32 has a standby power consumption of just 2µA. An internal 8MHz RC oscillator allows the chip to quickly come out of low power modes while the external oscillator is still starting up. This fast entry and exiting from low power modes further reduces overall power consumption.

1.2.2 Safety

As well as demanding more processing power and more sophisticated peripherals, many modern applications have to operate in safety-critical environments. With this in mind, the STM32 has a number of hardware features that help support high integrity applications. These include a low power voltage detector, a clock security system and two separate watchdogs. The first watchdog is a windowed watchdog. This watchdog must be refreshed in a defined time frame. If you hit it too soon, or too late, the watchdog will trigger. The second watchdog is an independent watchdog which has its own external oscillator separate from the main system clock. A further clock security system can detect failure of the main external oscillator and fail safely back onto an internal 8MHz RC oscillator.

1.2.3 Security

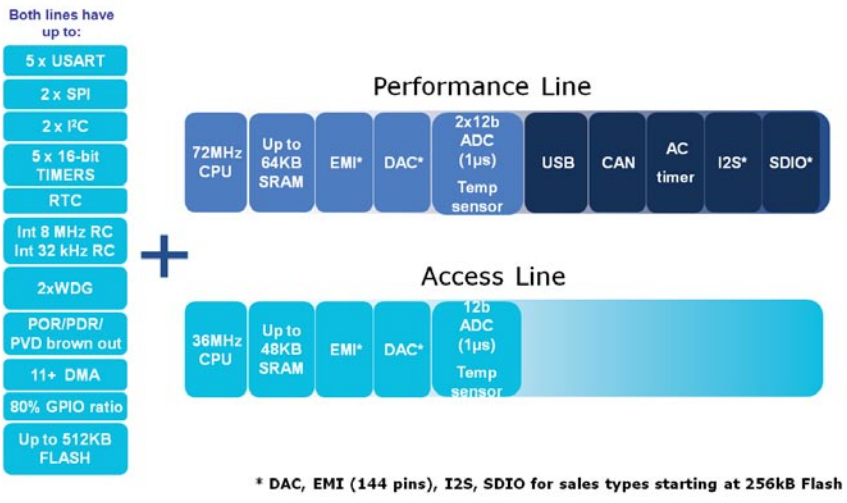
One of the other unfortunate requirements of modern design is the need for code security to prevent software piracy. Here the STM32 FLASH can be locked for FLASH READ accesses via the debug port. When READ protection is enabled, the FLASH memory is also WRITE protected to prevent untrusted code from being inserted on the interrupt vector table. Further WRITE protection can be enabled over the remainder of the FLASH memory. The STM32 also has a real-time clock and a small area of battery backed SRAM. This region has an anti-tamper input that can trigger an interrupt on a state change. In addition an anti-tamper event will automatically clear the contents of the battery backed SRAM.

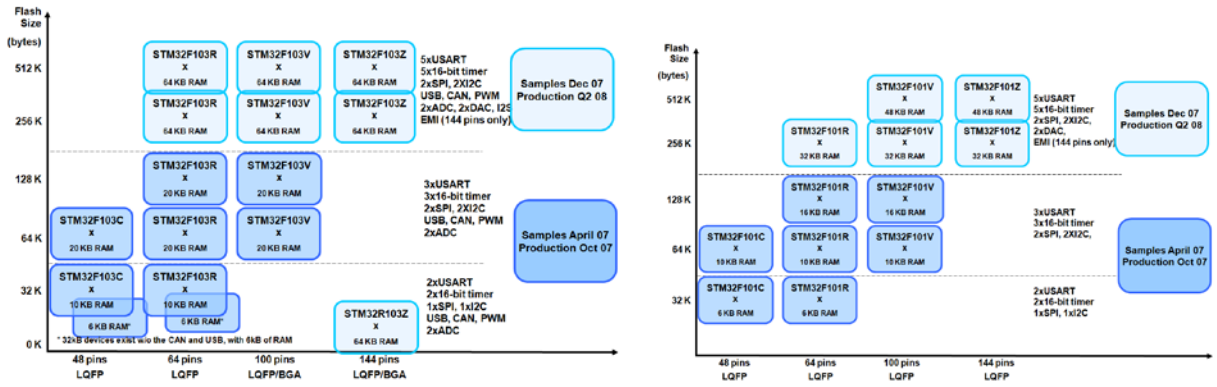
1.2.4 Software Development

If you are already using an ARM-based microcontroller, the good news is that the chances are that your development tools already support the Thumb-2 instruction set and the Cortex family. The worst case is a software upgrade to get the necessary support. ST also provide a peripheral driver library, a USB developer library as an ANSI C library and source code that is compatible with earlier libraries published for their STR7 and STR9 microcontrollers. Ports of these libraries are already available for popular compiler tools. Similarly, many open source and commercial RTOS and middleware (TCP/IP, file system etc) are available for the Cortex family.
The Cortex-M3 also comes with a whole new debug system called CoreSight. Access to the CoreSight system is through the Debug Access Port which supports either a standard JTAG connection or a serial wire (2 Pin) interface. As well as providing debug run control, the CoreSight system on the STM32 provides a data watchpoint and an instrumentation trace. The instrumentation trace can send selected application information up to the debug tool. This can provide extended debug information and can also be used during software testing.

1.2.5 STM32 Performance Line And Access Line

The STM32 family has two distinct branches: the Performance line and Access line. The Performance line has the full set of peripherals and runs to the maximum 72MHz. The Access line has a reduced set of peripherals and runs to a maximum 32MHz. Importantly the package types and pins layouts are the same between both the Access and Performance line variants. This allows different versions of the STM32 to be interchanged without having to re-spin the PCB.





2. Cortex Overview

As we saw in the introduction, the Cortex processor is the next generation embedded core from ARM. It is something of a departure from the earlier ARM CPUs in that it is a complete processor core, consisting of the Cortex CPU and a surrounding set of system peripherals, providing the heart of an embedded system. As a result of the wide variety of embedded systems, the Cortex processor is available in a number of application profiles. These are denoted by the letter following the Cortex name. The three profiles are as follows:

Cortex-A Series, applications processors for complex OS and user applications.
Supports the ARM, Thumb and Thumb-2 instruction sets.

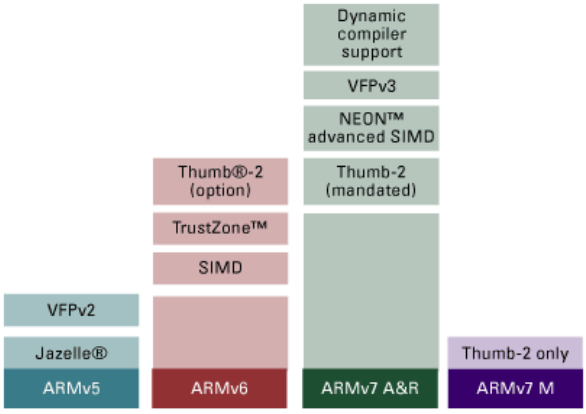
Cortex-R Series, real-time systems profile.
Supports the ARM, Thumb, and Thumb-2 instruction sets.

Cortex-M Series, microcontroller profile optimized for cost-sensitive applications.
Supports Thumb-2 instruction set only.

The number at the end of the Cortex name refers to the relative performance level, with 1 the lowest and 8 the highest. Currently performance level 3 is the highest performance level available in the microcontroller profile. The STM32 is based on the Cortex-M3 processor.

2.1 ARM Architectural Revision

ARM also somewhat confusingly denote each of their processors with an architectural revision. (This is written ARMV6, ARMV7 etc.) The Cortex M3 has the architectural revision ARMV7 M.



The Cortex-M3 processor is based on the ARMV7 architecture and is capable of executing the Thumb-2 instruction set.

Thus the documentation for the Cortex-M3 consists of the Cortex-M3 Technical Reference Manual and the ARMV7 M Architectural Reference Manual. Both of these documents can be downloaded from the ARM website at www.arm.com

2.2 Cortex Processor And Cortex CPU

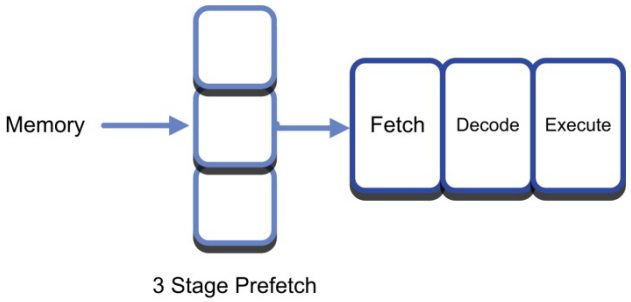
Throughout the remainder of this book, the terms Cortex processor and Cortex CPU will be used to distinguish between the complete Cortex embedded core and the internal RISC CPU. In the next section we will look at the key features of the Cortex CPU followed by the system peripherals in the Cortex processor.

2.3 Cortex CPU

At the heart of the Cortex processor is a 32-bit RISC CPU. This CPU has a simplified version of the ARM7/9 programmer's model, but a richer instruction set with good integer maths support, better bit manipulation and 'harder' real-time performance.

2.3.1 Pipeline

The Cortex CPU can execute most instructions in a single cycle. Like the ARM7 and ARM9 CPUs this is achieved with a three stage pipeline.

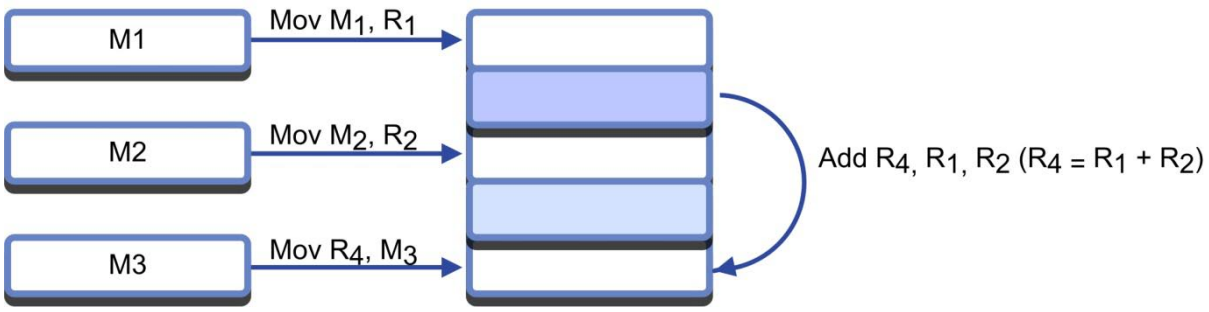


Like the ARM7 and ARM9 CPUs the Cortex-M3 has a three stage pipeline. However, the Cortex-M3 also has branch prediction to minimise the number of pipeline flushes.

Whilst one instruction is being executed, the next is being decoded and a third is being fetched from memory. This works very well for linear code, but when a branch is encountered the pipeline must be flushed and refilled before code can continue to execute. In the ARM7 and ARM9 CPUs branches are very expensive in terms of code performance. In the Cortex CPU the three stage pipeline is enhanced with branch prediction. This means that when a conditional branch instruction is reached, a speculative fetch is performed, so that both destinations of the conditional instruction are available for execution without incurring a performance hit. The worst case is an indirect branch where a speculative fetch cannot be made and the only course of action is to flush the pipeline. While the pipeline is key to the overall performance of the Cortex CPU, no special considerations need to be made in the application code.

2.3.2 Programmer's Model

The Cortex CPU is a RISC processor which has a load and store architecture. In order to perform data processing instructions, the operands must be loaded into a central register file, the data operation must be performed on these registers and the results then saved back to the memory store.



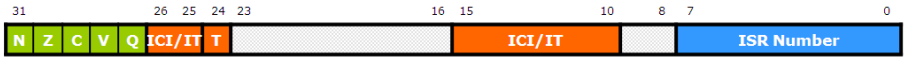
The Cortex-M3 is a load and store architecture. All data has to be moved into a central register file before a data processing instruction can act on it.

Consequently all the program activity focuses around the CPU register file. This register file consists of sixteen 32-bit wide registers. Registers R0-R12 are simple registers that can be used to hold program variables. The Registers R13-R15 have special functions within the Cortex CPU. Register R13 is used as the stack pointer. This register is banked, which allows the Cortex CPU to have two operating modes each with their own separate stack space. This is typically used by an RTOS which can run its 'system' code in a protected mode. In the Cortex CPU the two stacks are called the main stack and the process stack. The next register R14 is called the link register. This register is used to store the return address when a call is made to a procedure. This allows the Cortex CPU to make a fast entry and exit to a procedure. If your code calls several levels of subroutines, the compiler will automatically store R14 on the stack. The final register R15 is the program counter; since this is part of the central register file it can be read and manipulated like any other register.

The Cortex-M3 has a CPU register file of 16 32-bit wide registers. Like the earlier ARM7/9 CPUs R13 is the stack pointer. R14 is the link register and R15 is the PC. R13 is a banked register to allow the Cortex-M3 to operate with two stacks: a process stack and a main stack.

2.3.2.1 XPSR

In addition to the register file there is a separate register called the Program Status Register. This is not part of the main register file and is only accessible through two dedicated instructions. The xPSR contains a number of fields that influence the execution of the Cortex CPU.



The Program Status Register contains status fields for instruction execution. This register is aliased into the Application, Execution and Interrupt Status Registers

The xPSR register can also be accessed through three special alias names that allow access to sub-ranges of bits within the xPSR. The top five bits are the condition code flags and are aliased as the Application Program Status Register. The first four condition code flags N,Z,C,V (Negative, Zero, Carry and Overflow) will be set and cleared depending on the result of a data processing instruction. The Q bit is used by the DPS saturated maths instructions to indicate that a variable has reached its maximum or minimum value. Like the ARM 32-bit instruction set, certain Thumb-2 instructions are only executed if the instruction condition code matches the state of the Application Program Status Register flags. If the instruction condition codes do not match, the instruction passes through the pipeline as a NOP. This ensures that instructions flow smoothly through the pipeline and minimises pipeline flushes. In the Cortex CPU, this technique is extended with the Execution Program Status

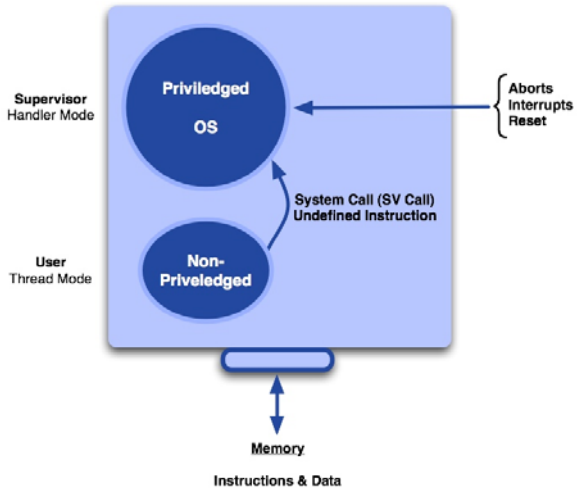
Register. This is an alias of bits 26 – 8 of the xPSR. This contains three fields: the “If then” field the “interrupt continuable instruction” and the Thumb instruction field. The Thumb-2 instruction set has an efficient method of executing small ‘if then’ blocks of instructions. When a conditional test is true, it can set a value in the IT field that tells the CPU to execute up to four following instructions. If the conditional test fails, these instructions will pass through the pipeline as a NOP. Thus a typical line of C would be coded as follows:

```
If (r0 ==0)
       CMP r0,#0     compare r0 to 0
       ITTEE EQ      if true execute the next two instructions
Then r0 = *r1 +2;
       LDR r0,[r1]   load contents of memory location into r0
       ADDr0,#2      add 2
```

While most Thumb-2 instructions execute in a single cycle, some (such as load and store instructions) take multiple cycles. So that the Cortex CPU can have a deterministic interrupt response time, these instructions must be interruptible. When an instruction is terminated early, the interrupt continuable instruction field stores the number of the next register to be operated on in the load or store multiple instruction. Thus once the interrupt has been serviced, the load/store multiple instruction can resume execution. The final Thumb field is inherited from the earlier ARM CPUs. This field indicates if the ARM or Thumb instruction set is currently being executed by the CPU. In the Cortex-M3 this bit is always set to one. Finally, the interrupt status field contains information on any interrupt request that was pre-empted.

2.3.3 CPU Operating Modes

While the Cortex processor is designed to be a low gate count, fast and easy to use microcontroller core, it has been designed to support the use of a real-time operating system. The Cortex processor has two operating modes: Thread mode and Handler mode. The CPU will run in Thread mode while it is executing in non-interrupt background mode and will switch to the Handler mode when it is executing exceptions. In addition, the Cortex CPU can execute code in a privileged or non-privileged mode. In privileged mode, the CPU has access to the full instruction set. In unprivileged mode certain instructions are disabled (such as the MRS and MSR instructions which allow access to the xPSR and its aliases). Additionally, access to most registers in the Cortex processor system control space is also disabled. Stack usage can also be configured. The main stack (R13) can be used by both Thread and Handler mode. Alternatively, Handler mode can be configured to use the process stack (R13 banked register).



The Cortex-M3 can be used in a 'flat' simple mode. It is also designed to support real-time operating systems. It has Handler and Thread modes that can be configured to use the main and process stacks and have privileged access to the Cortex system control registers.

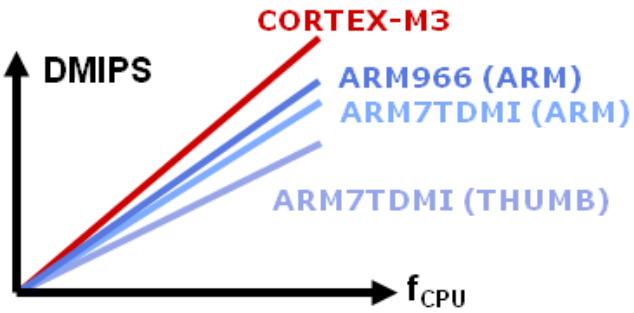
		Operations (privilege out of reset)	Stacks (Main out of reset)
Modes (Thread out of reset)	**Handler** - An exception is being processed	Privileged execution Full control	Main Stack Used by OS and Exceptions
	Thread - No exception is being processed - Normal code is executing	Privileged/Unprivileged	Main/Process

Out of reset the Cortex processor will run in a 'flat' configuration. Both Thread and Handler modes execute in privileged mode, so there are no restrictions on access to any processor resources. Both the Thread and Handler modes use the main stack. In order to start execution, the Cortex processor simply needs the reset vector and the start address of the stack to be configured before you can start to execute your application C code. However, if you are using an RTOS or are developing a safety-critical application, the chip can be used in a mode advanced configuration where Handler mode (exceptions and the RTOS) runs in privileged mode and uses the main stack while application code runs in Thread mode with unprivileged access and uses the process stack. This way the system code and the application code are partitioned and errors in the application code will not cause the RTOS to crash.

2.3.4 Thumb-2 Instruction Set

The ARM7 and ARM9 CPUs can execute two instruction sets: the ARM 32-bit instruction set and the Thumb 16-bit instruction set. This allows a developer to optimise his program by selecting the instruction set used for

different procedures: 32-bit instructions for speed and 16-bit instructions for code compression. The Cortex CPU is designed to execute the Thumb-2 instruction set which is a blend of 16 and 32 bit instructions. The thumb-2 instruction set gives a 26% code density improvement over the ARM 32-bit instruction set and a 25% improvement in performance over the Thumb 16-bit instruction set. The Thumb2 instruction set has some improved multiply instructions which can execute in a single cycle and a hardware divide that takes between 2 – 7 cycles.



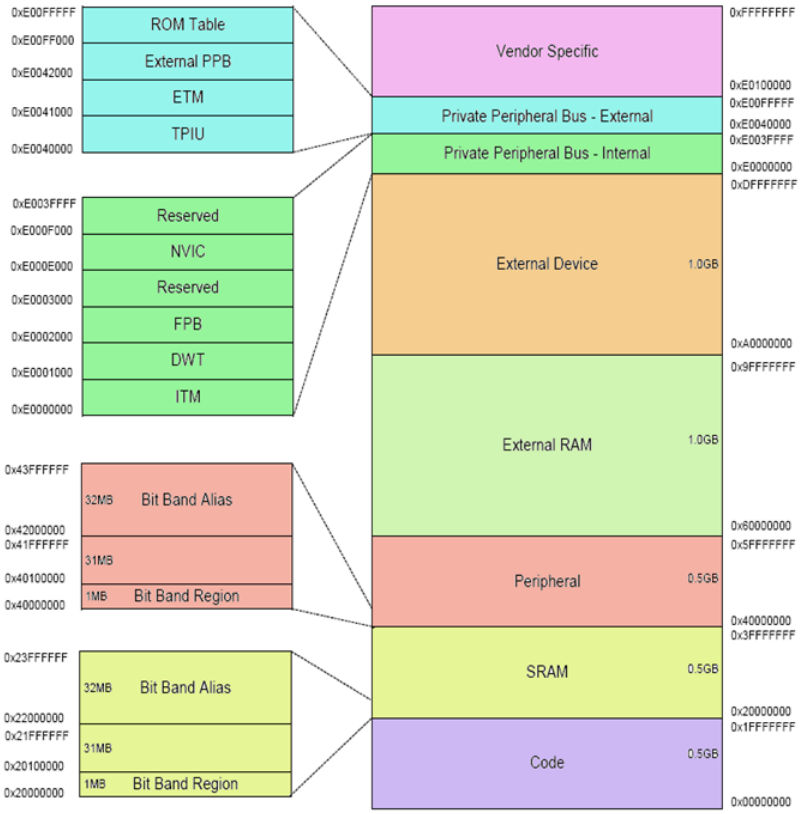
The Cortex processor benchmarks give a performance level of 1.2 DMIPS/MHz, which is 1.2 Clock cycles per instruction.

Source	Destination	Cycles
16b x 16b	32b	1
32b x 16b	32b	1
32b x 32b	32b	1
32b x 32b	64b	3-7*

The Thumb-2 instruction set also has: improved branching instructions including test and compare, if/then conditional execution blocks and for data manipulation byte ordering and byte and half word extraction instructions. While still a RISC processor, the Cortex CPU also has a rich instruction set that is specifically designed as a good target for a C compiler. A typical Cortex-M3 program will be written entirely in ANSI C, with minimal non-ANSI keywords and only the exception vector table written in Assembler.

2.3.5 Memory Map

The Cortex-M3 processor is a standardised microcontroller core and as such has a well-defined memory map. Despite the multiple internal busses this memory map is a linear 4 Gbyte address space.

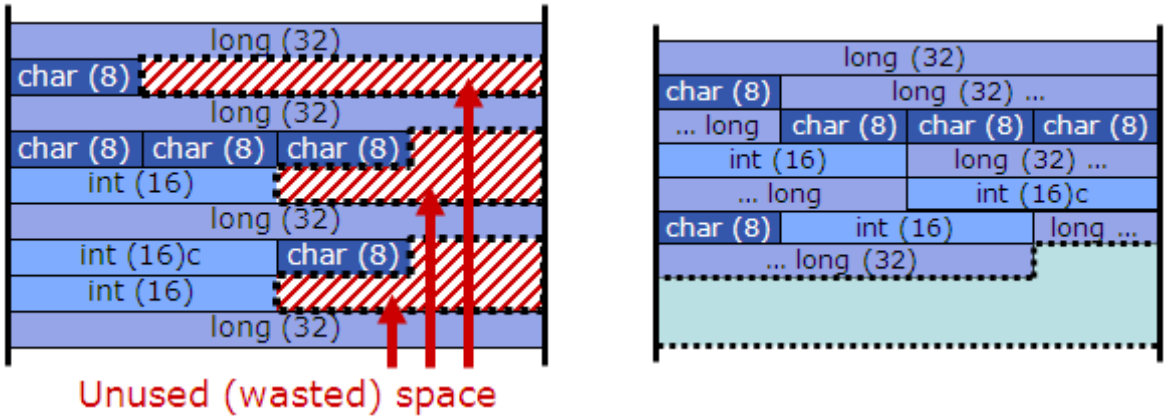


The Cortex-M3 defines a fixed 4 Gb memory map that specifies regions for code SRAM peripherals, external memory and devices and the Cortex system registers. This memory map is common to all Cortex-based devices.

The first 1Gbyte of memory is split evenly between a code region and a SRAM region. The code space is optimised to be executed from the I-Code bus. Similarly, the SRAM is reached with the D-code bus. Although code can be loaded and executed from the SRAM, the instructions would be fetched using the system bus, which incurs an extra wait state. It is likely that code would run slower from SRAM than from on-chip FLASH memory located in the code region. The next 0.5 Gbyte of memory is the on-chip peripheral region. All user peripherals provided by the microcontroller vendor will be located in this region. The first 1 Mbyte of both the SRAM and Peripheral regions is bit-addressable using a technique called bit banding. Since all the SRAM and all the user peripherals on the STM32 are located in these regions all the memory locations of the STM32 can be manipulated in a word-wide or bitwise fashion. The next 2 Gbyte address space is allocated to external memory-mapped SRAM and peripherals. The final 0.5 Gbyte is allocated to the internal Cortex processor peripherals and a region for future vendor specific enhancements to the Cortex processor. All of the Cortex processor registers are at fixed locations for all Cortex-based microcontrollers. This allows code to be more easily ported between different STM32 variants and indeed other vendors' Cortex-based microcontrollers. One processor to learn, one set of tools to invest in and large amounts of reusable code across a wide range of microcontrollers.

2.3.6 Unaligned Memory Accesses

The ARM7 and ARM9 instruction sets are capable of accessing byte, half word and word signed and unsigned variables. This allows the CPU to naturally support integer variables without the need for the sort of software library support typically required in 8 and 16-bit microcontrollers. However, the earlier ARM CPUs do suffer from a disadvantage in that they can only do word or half-word aligned accesses. This restricts the compiler linker in its ability to pack data into the SRAM and some valuable SRAM will be wasted. (This can be as much as 25% depending on the mix of variables used.)

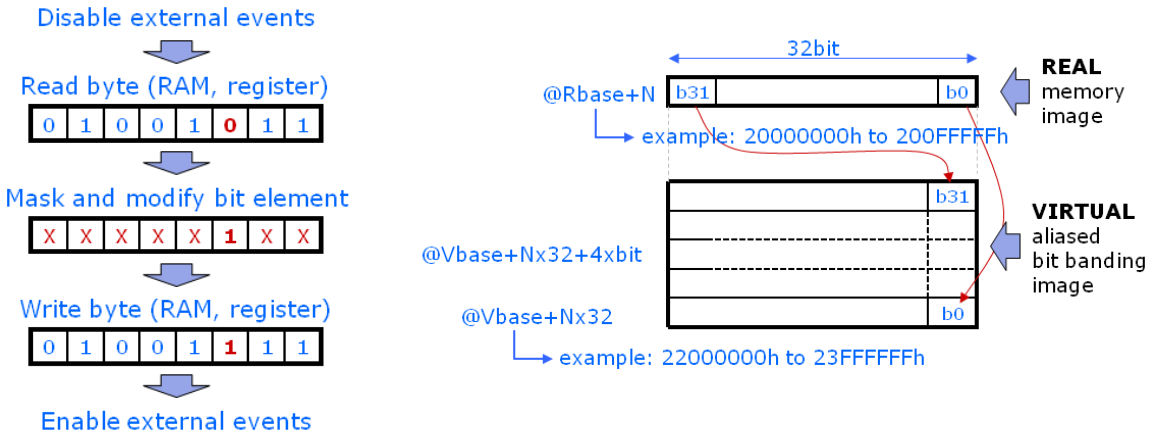


The Cortex-M3 can make unaligned memory accesses, which ensures that the SRAM is efficiently used.

The Cortex CPU has addressing modes for word, half-word and byte, but is able to make unaligned memory accesses. This gives the compiler linker complete freedom to order the program data in memory. The additional bit banding support on the Cortex CPU allows program flags to be packed into a word or half-word variable rather than using a byte for each flag.

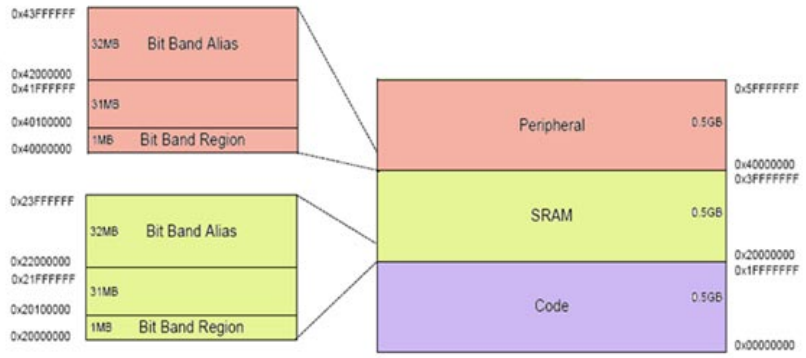
2.3.7 Bit Banding

The earlier ARM7 and ARM9 CPUs were only able to perform bit manipulations on SRAM and peripheral memory locations by using AND and OR operations. This requires a READ MODIFY WRITE operation which is expensive in terms of the number of cycles taken to set and clear individual bits and the overall code space required for each bit manipulation.



The bit banding technique allows atomic bit manipulation while keeping the Cortex-M3 CPU to a minimal gate count.

To overcome this limitation it would be possible to introduce a dedicated bit set and clear instructions, or a full Boolean processor, but this would increase the size and complexity of the Cortex CPU. Instead, a technique called bit banding allows direct bit manipulation on sections of the peripheral and SRAM memory spaces, without the need for any special instructions. The bit addressable regions of the Cortex memory map are composed of the bit band region (which is up to 1Mbyte of real memory or peripheral registers) and the bit band Alias region which takes up to 32Mbyte of the memory map. Bit banding works by mapping each bit in the bit band region to a word address in the Alias region. So by setting and clearing the aliased word address we can set and clear bits in the real memory.



Bit Banding is supported over the first 1Mb of the SRAM and Peripheral regions. This covers all the resources of the STM32.

This allows us to perform individual bit manipulation without the need for special instructions and keeps the overall size of the Cortex core as small as possible. In practice, we need to calculate the address of the bit band alias word for a given memory location in the peripheral or SRAM space. The formula to calculate the alias address is as follows:

Address in the bit band alias region	= Bit band alias base address + bit word offset
Where bit word offset	= Byte offset from bit band base X 0x20 + bit number x 4

This is much easier than it may look at first glance. For a practical example, the GPIO output data register is written to in order to set and clear individual IO lines. The physical address of the Port B output register is 0x40010C0C. In this example we want to be able to set and clear bit eight of this word using the above formula.

Word address	= 0x40010C0C
Peripheral bit band base	= 0x40000000
Peripheral bit band Alias base	= 0x42000000
Byte offset from bit band base	= 0x40010c0c – 0x40000000 = 10c0c
Bit word offset	= (0x10c0c x 0x20) +(8x4) = 0x2181A0
Bit Alias address	= 0x42000000 + 0x2181A0 = 0x422181A0

We can now create a pointer to this address using the following line of C:

```
#define PortBbit8   (*((volatile unsigned long *) 0x422181A0 ))
```

This pointer can then be used to set and clear the IO port bit:

```
PB8 = 1;        //led on
```

Which generates the following assembly instructions:

```
MOVS     r0,#0x01
LDR      r1,[pc,#104]
STR      r0,[r1,#0x00]
```

Switching the LED off:

```
PB8 = 0;  //led off
```

Generates the following assembly instructions:

```
MOVS     r0,#0x00
LDR      r1,[pc,#88]
STR      r0,[r1,#0x00]
```

Both the set and clear operations take three 16-bit instructions and on the STM32 running at 72 MHz these instructions are executed in 80nsec. Any word in the peripheral and SRAM bit band regions can also be directly addressed word-wide so we could perform the same set and clear using the more traditional AND and OR approach:

```
GPIOB->ODR |= 0x00000100;         //LED on
 LDR      r0,[pc,#68]
 ADDS     r0,r0,#0x08
 LDR      r0,[r0,#0x00]
 ORR      r0,r0,#0x100
 LDR      r1,[pc,#64]
 STR      r0,[r1,#0xC0C]

GPIOB->ODR &=!0x00000100;         //LED off
 LDR      r0,[pc,#40]
 ADDS     r0,r0,#0x08
 LDR      r0,[r0,#0x00]
 MOVS     r0,#0x00
 LDR      r1,[pc,#40]
 STR      r0,[r1,#0xC0C]
```

Now each set and clear operation takes a mixture of 16 and 32-bit operations, which take a minimum of 14 bytes for each operation and at the same clock frequency take a minimum of 180 nSec. If you consider the impact of bit banding on a typical embedded application that sets and clears lots of bits in the peripheral registers and uses semaphores and flags in the SRAM, you are very clearly going to make significant savings in both code size and execution time and it is all handled in the STM32 header file for you.

2.4 Cortex Processor

2.4.1 Busses

The Cortex-M3 processor has a Harvard architecture with separate code and data busses. These are called the Icode bus and the Dcode bus. Both of these busses can access code and data in the range 0x00000000 – 0x1FFFFFFF. An additional system bus is used to access the Cortex system control space in the range 0x20000000-0xDFFFFFFF and 0xE0100000-0xFFFFFFFF. The Cortex on-chip debug system has an additional bus structure called the Private Peripheral Bus.

2.4.2 Bus Matrix

The system and data busses are connected to the external microcontroller via a set of high speed busses arranged as a bus matrix. This allows a number of parallel paths between the Cortex busses and other external bus masters such as DMA to the on-chip resources such as SRAM and peripherals. If two bus masters (i.e. the Cortex CPU and a DMA channel) try to access the same peripheral, an internal arbiter will resolve the conflict and grant bus access to the highest priority peripheral. However, in the STM32 the DMA units are designed to work in concert with the Cortex CPU, as we will see when we examine the operation of the DMA unit.

2.4.3 System Timer

The Cortex core also includes a 24-bit down counter, with auto reload and end of count interrupt. This is intended to provide a standard timer for all Cortex-based microcontrollers. The SysTick timer is intended to be used to provide a system tick for an RTOS, or to generate a periodic interrupt for scheduled tasks. The SysTick Control and status register in the Cortex-M3 System control space unit allows you to select the SysTick clock source. By setting the CLKSOURCE bit the SysTick timer will run at the CPU frequency. When cleared the timer will run at 1/8 CPU frequency.

The SysTick Timer is a 24-bit auto-reload timer located within the Cortex-M3 processor. It is intended to provide a timer tick for a Real Time Operating System.

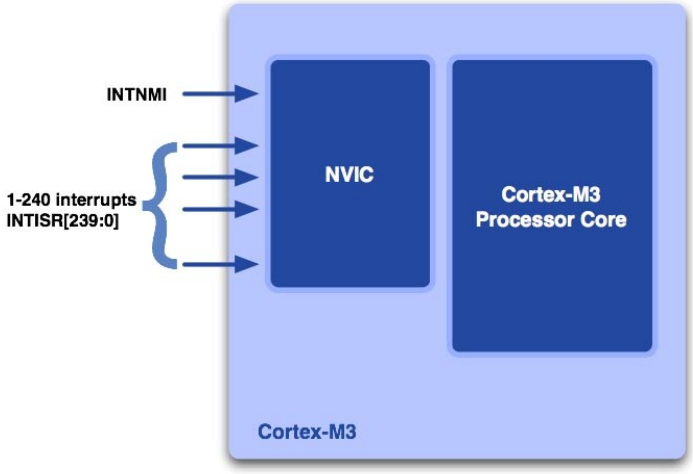
The SysTick timer has three registers. The current value and reload value should be initialised with the count period. The control and status register contains an ENABLE bit to start the timer running and a TICKINT bit to enable its interrupt line. In the next section we will look at the Cortex interrupt structure and use the SysTick timer to generate a first exception on the STM32.

2.4.4 Interrupt Handling

One of the key improvements of the Cortex core over the earlier ARM CPUs is its interrupt structure and exception handling. The ARM7 and ARM9 CPUs had two interrupt lines: the fast interrupt and the general purpose interrupt line. These two interrupt lines had to support all of the interrupt sources within a given manufacturer's microcontroller. How this was done varied according to the implementation, so while the techniques used were broadly the same, the implementation differed between manufacturers. The ARM7 and ARM9 interrupt structure suffers from two further problems. Firstly it is not deterministic; the time taken to terminate or abort an instruction under execution when the interrupt occurs is variable. This may not be a problem for many applications, but it is a big issue in real-time control. Secondly, the ARM7 and ARM9 interrupt structure does not naturally support nested interrupts; further software is required: either Assembler macros or an RTOS. One of the key criteria of the Cortex core is to overcome these limitations and provide a standard interrupt structure which is both extremely fast and deterministic.

2.4.5 Nested Vector Interrupt Controller

The Nested Vector Interrupt Controller is a standard unit within the Cortex core. This means that all Cortex-based microcontrollers will have the same interrupt structure, regardless of manufacturer. Thus application code and operating systems can be easily ported from one microcontroller to another and the programmer does not need to learn a whole new set of registers. The NVIC is also designed to have a very low interrupt latency. This is both a feature of the NVIC itself and of the Thumb-2 instruction set which allows multi-cycle instructions such as load and store multiple to be interruptible. This interrupt latency is also deterministic, with several advanced interrupt handling features that support real-time applications. As its name implies, the NVIC is designed to support nested interrupts and on the STM32 there are 16 levels of priority. The NVIC interrupt structure is designed to be programmed entirely in 'C' and does not need any Assembler macros or special non-ANSI directives.

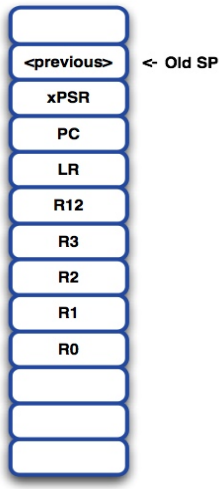


The STM32 processor includes a Nested Vector Interrupt Controller which can support a maximum of 240 external peripherals.

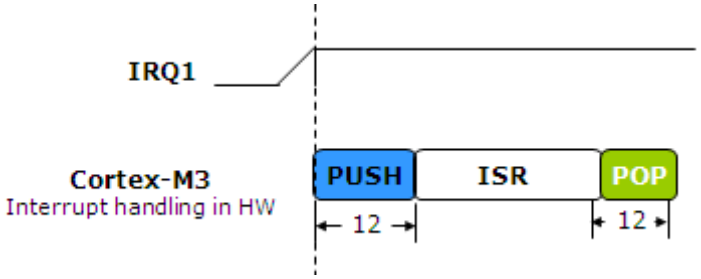
Although the NVIC is a standard unit within the Cortex core, in order to keep the gate count to a minimum the number of interrupt lines going into the NVIC is configurable when the microcontroller is designed. The NVIC has one non-maskable interrupt and up to a further 240 external interrupt lines which can be connected to the user peripherals. There are an additional 15 interrupt sources within the Cortex core, which are used to handle internal exceptions within the Cortex core itself. The STM32 NVIC has been synthesised with a maximum of 43 maskable interrupt lines.

2.4.5.1 NVIC Operation Exception Entry And Exit

When an interrupt is raised by a peripheral, the NVIC will start the Cortex CPU serving the interrupt. As the Cortex CPU enters its interrupt mode, it will push a set of registers onto the stack. Importantly this is done in microcode, so there is no instruction overhead in the application code. While the stack frame is being saved, the starting address of the interrupt service routine is fetched on the instruction bus. Thus the time taken from the interrupt being raised to reaching the first instruction in the interrupt routine is just 12 cycles.



The NVIC will respond to an interrupt with a latency of just six cycles. This includes a microcoded routine to automatically push a set of registers onto the stack.



The stack frame consists of the Program Status Register, the program counter and the link register. This saves the context that the Cortex CPU was running in. In addition registers R0 – R3 are also saved. In the ARM binary interface standard these registers are used for parameter passing, so saving these gives us a set of CPU registers that can be used by the ISR. Finally R12 is also saved; this register is the intracall scratch register. This register is used by any code that runs when function calls are made. For example, if you have enabled stack checking in the compiler, the additional code generated will use R12 if it need a CPU register. When the interrupt ends the process is reversed, the stack frame is restored automatically by microcode and in parallel the return address is fetched, so that the background code can resume execution in 12 cycles.

2.4.5.2 Advanced Interrupt Handling Modes

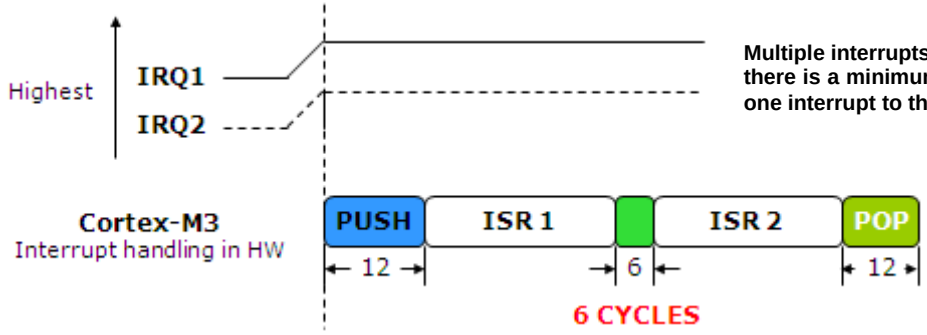
As well as being able to handle a single interrupt very quickly, the NVIC is designed to efficiently handle multiple interrupts in a very real time application. The NVIC has several clever methods of handling multiple interrupt sources the minimum delay between interrupts and to ensure that the highest priority interrupt is served first.

2.4.5.2.1 Interrupt Pre-emption

The NVIC is also designed to allow high priority interrupts to pre-empt a currently running low priority interrupt. In this case the running interrupt is halted and a new stack frame is saved in the standard 12 cycles after which the high priority interrupt runs. When the high priority interrupt is finished, the stack is automatically POPed and the low priority interrupt can resume execution.

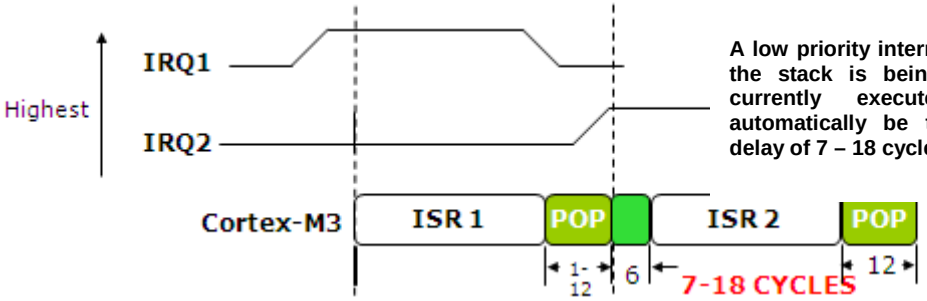
2.4.5.2.2 Tail Chaining

If a high priority interrupt is running and a low priority interrupt is raised, the Cortex NVIC uses a method called tail chaining to ensure that there is a minimum delay between servicing interrupts.



Multiple interrupts will be 'tail chained' so there is a minimum delay from the end of one interrupt to the start of the next.

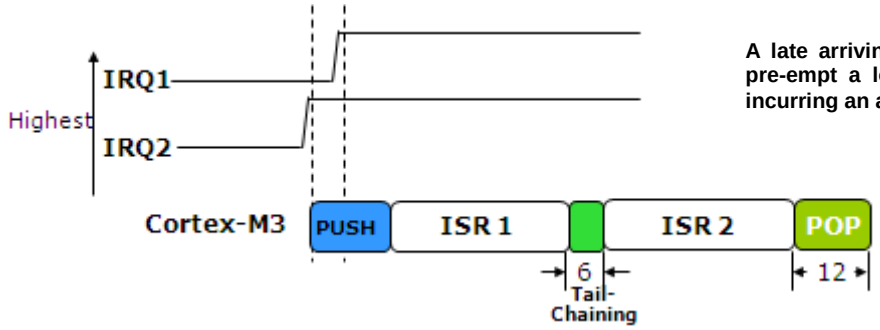
If two interrupts are raised, the highest priority interrupt will be served first and will begin execution in the standard 12 cycles. However, at the end of the interrupt routine the Cortex CPU does not return to the background code. The stack frame is not restored, only the entry address of the next highest priority ISR is fetched. This takes just six cycles and then the next interrupt service routine can begin execution. At the end of the pending interrupts the stack is restored and the return address is fetched, so the background code can begin execution in the standard 12 cycles. If the low priority interrupt arrives while the running interrupt is exiting, the POP will be abandoned and the stack pointer will be wound back to its original value. There is an additional 6 cycle delay while the new ISR address is fetched. This gives a latency of between 7 – 18 cycles before the new interrupt service routine can begin execution.



A low priority interrupt which occurs as the stack is being unwound from a currently executed interrupt will automatically be tail chained with a delay of 7 – 18 cycles

2.4.5.2.3 Late Arrival

In a real-time system there may often be a condition where we have started to serve a low priority interrupt, only for a high priority interrupt to be raised. If this condition occurs during the initial PUSH the NVIC will switch to serve the higher priority interrupt. The stacking continues and there will be a minimum of 6 cycles from the point at which the high priority interrupt is raised, while the new ISR address is fetched.



A late arriving high priority interrupt will pre-empt a low priority interrupt without incurring an additional stacking overhead.

Once the high priority interrupt has finished execution, the original low priority interrupt will be tail chained and begin execution six cycles later.

2.4.5.3 NVIC Configuration And Use

To use the NVIC we need to do three things. First configure the vector table for the interrupt sources we want to use. Next configure the NVIC registers to enable and set the priorities of the NVIC interrupts and lastly we must configure the peripheral and enable its interrupt support.

2.4.5.3.1 Exception Vector Table

The Cortex vector table starts at the bottom of the address range. However rather than start at zero the vector table starts at address 0x00000004 the first four bytes are used to store the starting address of the stack pointer.

No.	Exception Type	Priority	Type of Priority	Descriptions
1	Reset	-3 (Highest)	fixed	Reset
2	NMI	-2	fixed	Non-Maskable Interrupt
3	Hard Fault	-1	fixed	Default fault if other hander not implemented
4	MemManage Fault	0	settable	MPU violation or access to illegal locations
5	Bus Fault	1	settable	Fault if AHB interface receives error
6	Usage Fault	2	settable	Exceptions due to program errors
7-10	Reserved	N.A.	N.A.	
11	SVCall	3	settable	System Service call
12	Debug Monitor	4	settable	Break points, watch points, external debug
13	Reserved	N.A.	N.A.	
14	PendSV	5	settable	Pendable request for System Device
15	SYSTICK	6	settable	System Tick Timer
16	Interrupt #0	7	settable	External Interrupt #0
.....	.....	.....	settable	.....
256	Interrupt#240	247	settable	External Interrupt #240

The Cortex exception table contains the start address or an ISR which is loaded into the Program counter as the CPU enters the exception.

Each of the interrupt vector entries is four bytes wide and holds the start address of each service routine associated with the interrupt. The first 15 entries are for exceptions that occur within the Cortex core. These include the reset vector, non-maskable interrupt, fault and error management, debug exceptions and also the SysTick timer interrupt. The Thumb-2 instruction set also includes system service call instruction which when executed will generate an exception. The user peripheral interrupts start from entry 16 and will be linked to peripherals as defined by the manufacturer. In software, the vector table is usually maintained in the startup by locating the service routine addresses at the base of memory.

```
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors

__Vectors       DCD     __initial_sp               ; Top of Stack
                DCD     Reset_Handler              ; Reset Handler
                DCD     NMI_Handler                ; NMI Handler
                DCD     HardFault_Handler          ; Hard Fault Handler
                DCD     MemManage_Handler          ; MPU Fault Handler
                DCD     BusFault_Handler           ; Bus Fault Handler
                DCD     UsageFault_Handler         ; Usage Fault Handler
                DCD     0                          ; Reserved
                DCD     0                           ; Reserved
                DCD     0                          ; Reserved
                DCD     0                          ; Reserved
                DCD     SVC_Handler                ; SVCall Handler
                DCD     DebugMon_Handler           ; Debug Monitor Handler
                DCD     0                          ; Reserved
                DCD     PendSV_Handler             ; PendSV Handler
                DCD     SysTick_Handler            ; SysTick Handler
```

In the case of the SysTick timer we can create a service routine by declaring a ‘C’ routine with the matching symbolic name:

```
void SysTick_Handler (void)
{

   ….

}
```

Now with the vector table configured and the ISR prototype defined, we can configure the NVIC to handle the SysTick timer interrupt. Generally we need to do two things: set the priority of the interrupt and then enable the interrupt source. The NVIC registers are located in the system control space.

The NVIC registers are located in the Cortex-M3 System control space and may only be accessed when the CPU is running in privileged mode.

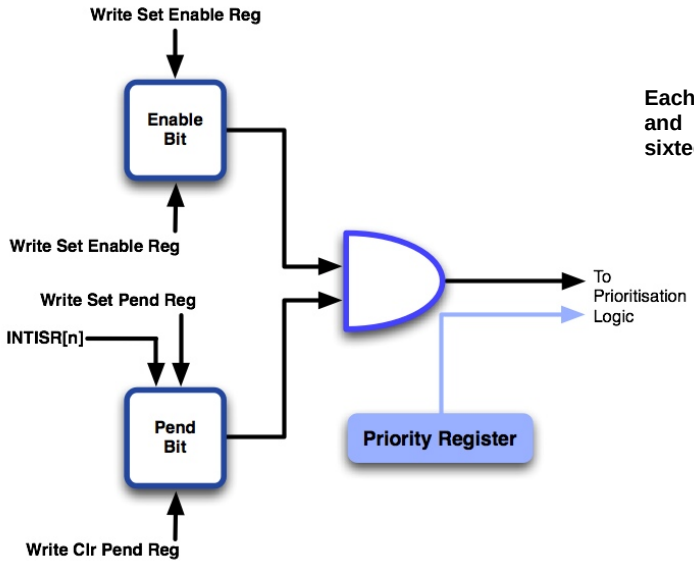
The Cortex internal exceptions are configured using the system control and system priority registers, while the user peripherals are configured using the IRQ registers. The SysTick interrupt is an internal Cortex exception and is handled in the system registers. Some of the internal exceptions are permanently enabled; these include the reset and NMI interrupts, but also the SysTick timer, so there is no explicit action required to enable the SysTick interrupt within the NVIC. To configure the SysTick interrupt we need to set the timer going and enable the interrupt within the peripheral itself:

```
SysTickCurrent = 0x9000;                    //Start value for the sys Tick counter
SysTickReload  = 0x9000;                    //Reload value
SysTickControl = 0x07;                      //Start and enable interrupt
```

The priority of each of the internal Cortex exceptions can be set in the system priority registers. The Reset, NMI and hard fault exceptions are fixed to ensure that the core will always fallback to a known exception. Each of the

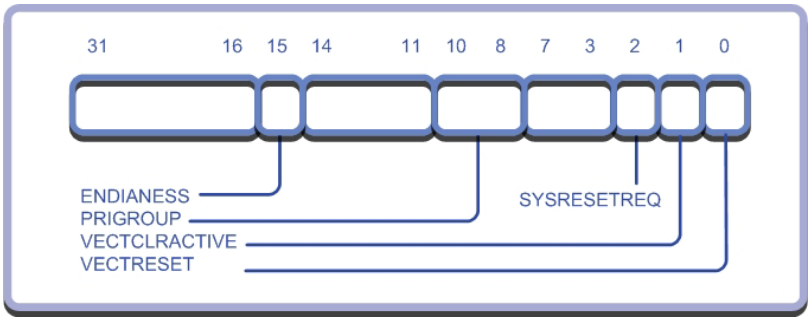
other exceptions has an eight bit field located in the three system priority registers. The STM32 only implements 16 levels of priority so only four bits of this field are active. However it is important to note that the priority is set by the four most significant bits.

Each of the USER peripherals is controlled by the IRQ register blocks. Each user peripheral has an Interrupt Enable bit. These bits are located across two 32-bit IRQ Set Enable registers. There are matching IRQ Clear Enable registers that are used to disable an interrupt source. The NVIC also includes pending and active registers that allow you to determine the current condition of an interrupt source.



Each interrupt source has an enable bit in the NVIC and in the peripheral. In the STM32 there are sixteen levels of priority.

There are sixteen priority registers. Each priority register is divided into four eight bit priority fields, each field being assigned to an individual interrupt vector. The STM32 only uses half of this field to implement 16 levels of priority. However, you should note that the active priority bits are in the upper nibble of each priority field. By default the priority field defines 16 levels of priority with level zero the highest and 15 the lowest. It is also possible to format the priority field into priority groups and subgroups. This does not provide extra levels of priority, but helps management of priority levels when you have a large number of interrupts by programming the PRIGROUP field in the Application Interrupt and Reset Control Register.



The PRIGROUP field splits the priority levels into groups and subgroups. This is useful for software abstraction when dealing with a large number of interrupts.

PRIGROUP (3 Bits)	Binary Point (group.sub)		Preempting Priority (Group Priority)		Sub-Priority	
			Bits	Levels	Bits	Levels
011	4.0	gggg	4	16	0	0
100	3.1	gggs	3	8	1	2
101	2.2	ggss	2	4	2	4
110	1.3	gsss	1	2	3	8
111	0.4	ssss	0	0	4	16

The three bit PRIGROUP field allows you to split the 4-bit priority fields into groups and subgroups. For example, PRIGROUP value 3 creates two groups, each with four levels of priority. In your application code you can now define a high priority group of interrupts and a low priority group. Within each group you can specify subgroup levels of low, medium, high and very high. As mentioned above this does not provide you anything “extra” but provides a more abstracted view of the interrupt structure which is useful to the programmer when managing a large number of interrupts. Configuring a peripheral interrupt is very similar to configuring an internal Cortex exception. In the case of the ADC interrupt we must first set the interrupt vector and provide the ISR routine:

```
DCD     ADC_IRQHandler  ;

void ADC_Handler void
{

}
```

Then the ADC must be initialised and the interrupt must be enabled within the peripheral and the NVIC:

```
ADC1->CR2       = ADC_CR2;   //Switch on the ADC and continuous conversion
ADC1->SQR1      = sequence1; //Select number of channels in sequence conversion
ADC1->SQR2      = sequence2; //and select channels to convert
ADC1->SQR3      = sequence3;
ADC1->CR2       |= ADC_CR2;  //Rewrite on bit

ADC1->CR1       = ADC_CR1;//Start regular channel group, enable ADC interrupt

GPIOB->CRH      = 0x33333333; //Set LED pins to output

NVIC->Enable[0] = 0x00040000; //Enable ADC interrupt
NVIC->Enable[1] = 0x00000000;
```

2.5 Power Modes

We will have a look at the full power management options within the STM32 later. In this section we will look at the power management modes within the Cortex core. The Cortex CPU has a sleep mode that places the Core into its low power mode and halts execution of instructions within the Cortex CPU. A small part of the NVIC is kept awake, so that interrupts generated from the STM32 peripherals can wake the Cortex core up.

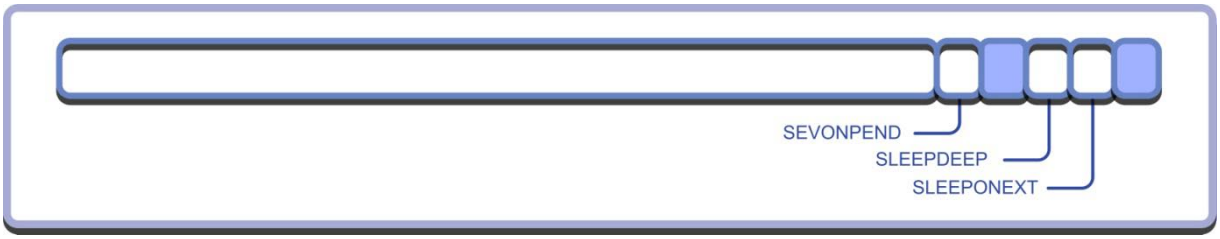
2.5.1 Entering Low Power Mode

The Cortex core can be placed into its sleep mode by execution of either a Wait For Interrupt (WFI) or Wait For Event (WFE) instruction. In the case of the WFI instruction, the Cortex core will resume execution and serve the pending interrupt. Once the ISR routine has completed, there are two possibilities. Firstly, the Cortex CPU can return from the ISR and continue execution of the background code as normal. By setting the SLEEPON EXIT bit in the System Control Register, the Cortex core will automatically enter the sleep mode once the ISR has completed. This allows a low power application to be entirely interrupt-driven, so that the Cortex core will wake up, run the appropriate code and then re-enter the sleep mode with minimal code being used for power management.

The WFE interrupt allows the Cortex core to resume execution from the point it was placed into the sleep mode. It does not jump to a service routine. A wake-up event is simply a peripheral interrupt line that is not enabled as an interrupt within the NVIC. This allows a peripheral to signal the Cortex core to wake up and continue processing without the need for an interrupt service routine. The WFI and WFE instructions are not reachable from the C language, but compilers for the Thumb-2 instruction set provide intrinsic macros that can be used inline with standard C commands:

```
__WFI

__WFE
```

In addition to the SLEEPNOW and SLEEPONEXIT low power modes the Cortex core can issue a SLEEPDEEP signal to the rest of the microcontroller system.



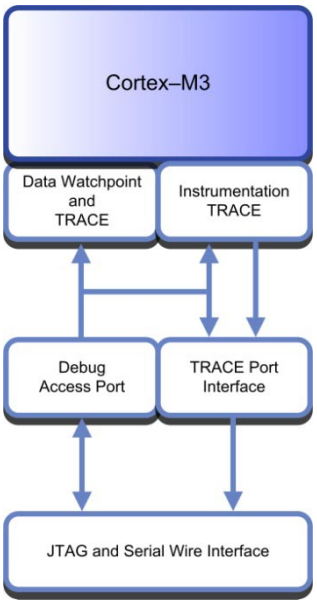
The System control register configures the Cortex processor sleep modes. The STM32 has additional low power modes that use the DeepSleep signal which is exported from the Cortex processor.

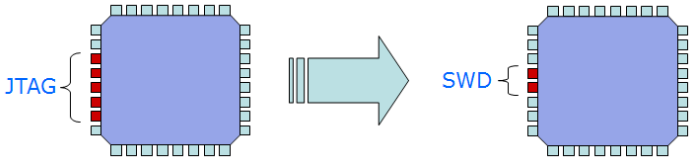
This allows additional functions such as the PLL and user peripherals to be halted, so that the STM32 can enter its lowest power modes.

2.5.2 CoreSight Debug Support

All of the ARM CPUs have their own on-chip debug system. The ARM7 and ARM9 CPUs have as a minimum a JTAG port which allows a standard debug tool to connect to the CPU and download code into the internal RAM or FLASH memory. The JTAG port also supports basic run control (single step and setting breakpoints etc) as well as being able to view the contents of memory locations. The ARM7 and ARM9 CPUs can also provide a real-time trace through an additional debug peripheral called the embedded trace macro cell (ETM). While this system works fine, it does have some limitations. The JTAG debug peripheral can only provide debug information to the development tools when the ARM CPU is halted, so there is no possibility of real-time updates. Also, the number of hardware breakpoints is limited to two, though the ARM7 and ARM9 instructions sets include a breakpoint

instruction which can be patched into the code by the development tool (typically called soft breakpoints.) Similarly for the real-time trace to work, the ETM must be fitted by the manufacturer at additional cost. Consequently this is not always supported. With the new Cortex core a whole new debug system called CoreSight has been introduced.





The Cortex CoreSight debug system uses a JTAG or serial wire interface. CoreSight provides run control and trace functions. It has the additional advantage that it can be kept running while the STM32 is in a low power mode. This is a big step on from standard JTAG debugging.

The full core sight debug system has a debug access port which allows connection to the microcontroller by a JTAG tool. The debug tool can connect using the standard 5 pin JTAG interface or a serial 2 wire interface.

In addition to the JTAG debug features, the full CoreSight debug system contains a Data Watch trace and an embedded trace macro cell. For software testing there is instrumentation trace and FLASH patch block. The STM32 implements the CoreSight debug system with the omission of the embedded trace macro cell.

In practice, the CoreSight debug structure on the STM32 provides an enhanced real-time version of the standard JTAG debug features. The STM32 CoreSight debug system provides 8 hardware breakpoints which can be non-intrusively set and cleared while the Cortex CPU is running. In addition the Data Watch trace allows you to view the contents of memory locations non intrusively while the Cortex CPU is running. The CoreSight debug system can stay active when the Cortex core enters a low power or sleep mode. This makes a world of difference when debugging a low power application. Additionally the STM32 timers can be halted when the CPU is halted by the CoreSight system. This allows you to single-step your code and keep the timers in sync. with the instructions executing on the Cortex CPU. The CoreSight debug infrastructure significantly improves the real-time debug capabilities of the STM32 over earlier ARM7 and ARM9 CPUs whilst still using the same low cost hardware.

3. Getting It Working

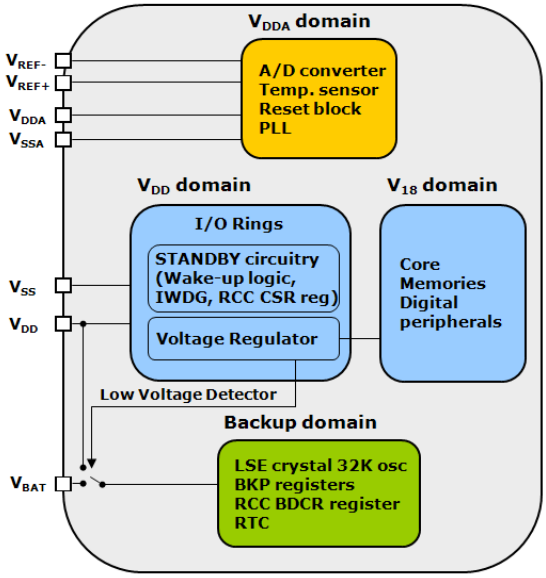
A minimal STM32 design really can be very minimal. To make the STM32 work you really just need to add a power supply. The microcontroller contains its own internal RC oscillators and an internal reset circuit. This section will look at the main hardware considerations you will need to address to build a practical design.

3.1 Package Types and Footprints

The STM32 Access line and Performance line variants are designed with matching package types, to allow an easy hardware upgrade without any need to redesign the PCB. All the STM32 microcontrollers are available in LQFP packages ranging from 48 pins up to 144 pins.

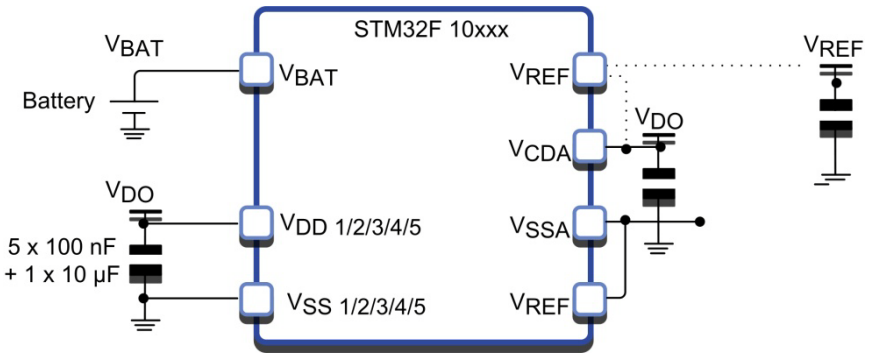
3.2 Power Supply

The STM32 requires a single power supply which must be in the range 2.0V to 3.6V. An internal regulator is used to generate a 1.8V supply for the Cortex core. The STM32 has two other optional power supplies. The real time clock and a small number of registers are located on a separate power domain, which can be battery-backed to preserve data when the rest of the STM32 is placed in a deep power down state. If the design is not using battery back up, then V_{BAT} must be connected to V_{DD}.



The STM32 runs from a single 2.0V-3.6V supply. There is an additional backup power domain and a separate supply for the ADC converter (144 pin package only).

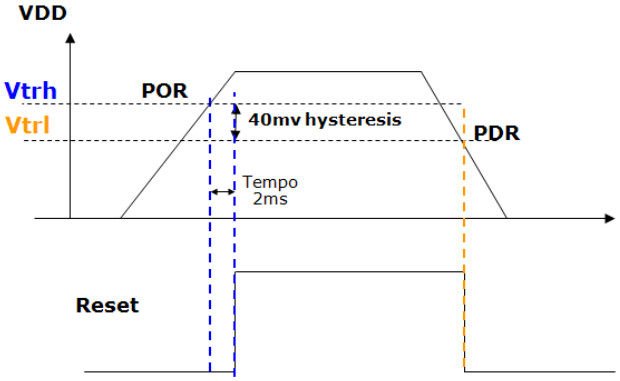
The second optional power supply is used to power the ADC. If the ADC is used, the main V_{DD} power supply range is limited to 2.4V to 3.6V. On the 100 pin package, the ADC has additional voltage reference pins V_{REF+} and V_{REF-}. The V_{REF-} pin must be connected to V_{DDA} and V_{REF+} can vary from 2.4 to V_{DDA}. On all other packages the voltage reference is internally connected to the ADC voltage supply pins. Each of the power supplies requires stabilisation capacitors as shown below.



With an internal reset and an internal voltage regulator, the STM32 only needs seven external capacitors.

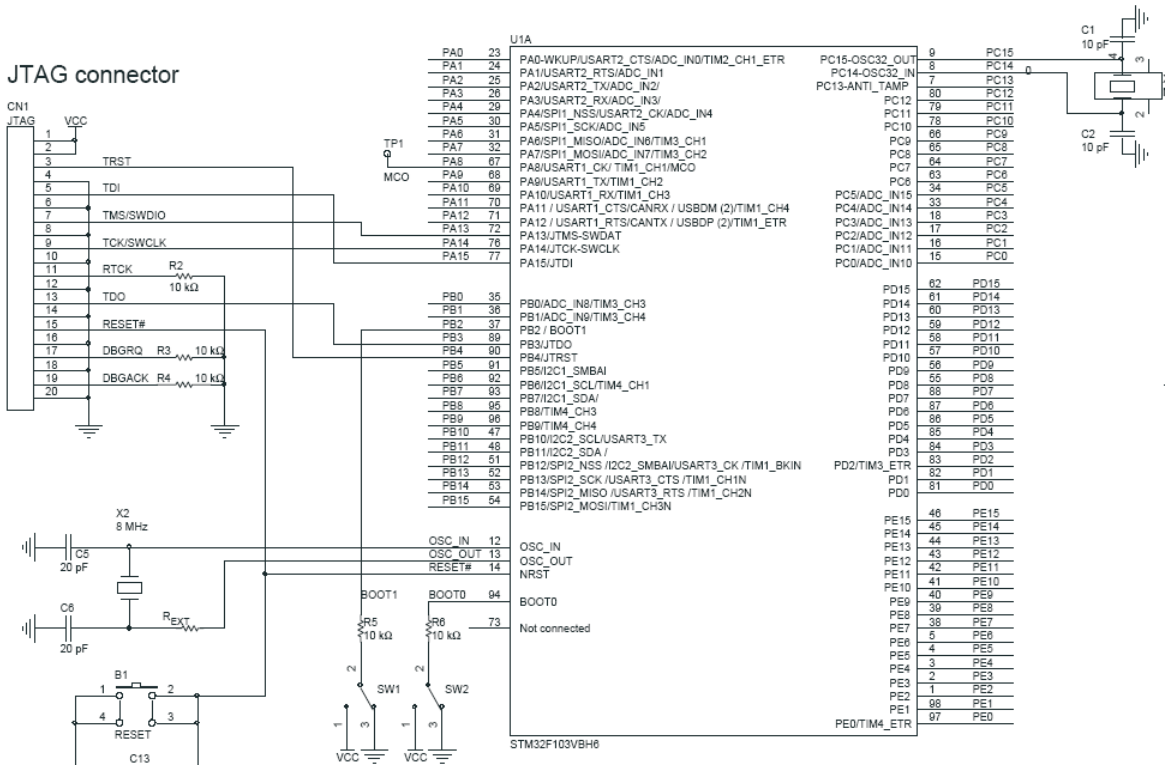
3.3 Reset Circuit

The STM32 contains an internal reset circuit that holds the device in reset as long as V_{DD} is below 2.0V with a hysteresis of 40mV.



The internal power on reset and power down reset ensure the processor only runs with a stable power supply. No external reset circuit is required.

3.3.1.1.1 Basic Hardware Schematic



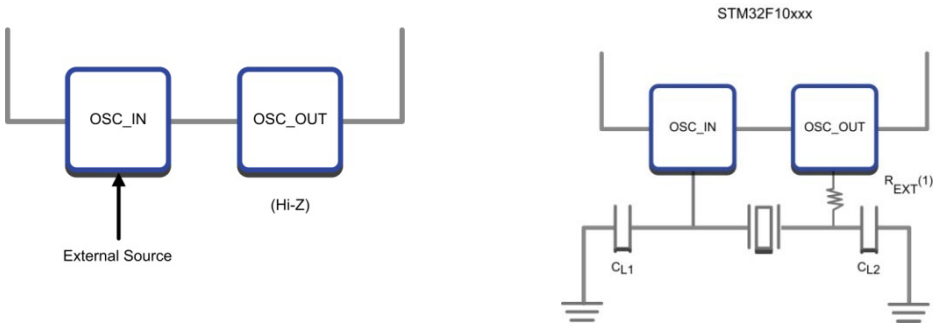
Strictly speaking, an external reset circuit is not a necessary part of an STM32 design. However, during development the nRST pin can be connected to a simple reset switch. nRST is also routed to the JTAG debug port, so that a development tool can force a reset on the microcontroller. The STM32 has a number of internal reset sources that can detect faulty operating conditions and we will have a look at these in the safety section later on.

3.4 Oscillators

The STM32 has internal RC oscillators which are capable of supplying a clock to the internal PLL. This will allow the microcontroller to run at its maximum 72 MHz clock frequency. The internal oscillators are not as accurate or stable as an external crystal; consequently for most designs you will need at least one clock source.

3.4.1 High Speed External Oscillator

The main external clock source is used to derive the Cortex processor and the STM32 peripheral clocks. This clock source is called the High Speed External (HSE Osc) Oscillator and can be a crystal/ceramic resonator or a user provided clock source. If a user clock is selected, it can be a square, sine or triangular waveform, but it must have a duty cycle of 50% and a maximum frequency of 25 MHz.



The External Oscillator can be run from a crystal or external clock source.

If an external crystal/ceramic resonator is used, it must be in the range 4 MHz – 16 MHz. Since the internal PLL multiplies the HSE Osc frequency up by integer values, the external clock should be a factor of 72 MHz so that you can easily derive the full operating frequency.

3.4.2 Low Speed External Oscillator

The STM32 can have a second external oscillator called the Low Speed External (LSE Osc) Oscillator. This is used to clock source to the real-time clock and the windowed watchdog.
Like the HSE Osc, the LSE Osc can be an external crystal or a user-provided clock which can again have a square, sine or triangular waveform, as long as the duty cycle is 50%. In both cases the LSE Osc should have a frequency of 32.768 KHz, as this will provide an accurate working frequency for the real-time clock. The internal low speed oscillator can be used to supply the real-time clock, but it is not highly accurate so if you plan to use the RTC to any extent in your design you should fit the LSE oscillator.

3.4.3 Clock Output

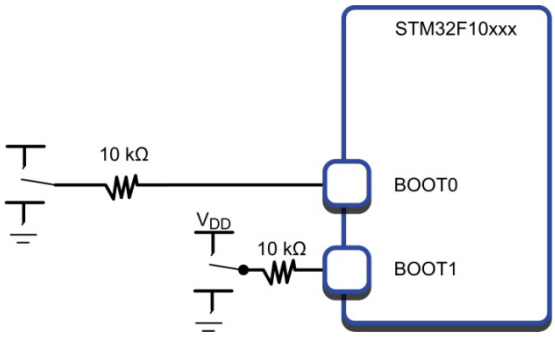
One of the GPIO pins can be configured to be a Microcontroller Clock out pin. In this mode the MCO pin can output one of four internal clock sources. We will look at this in more detail when we examine the internal clock tree configuration.

3.4.4 Boot Pins And Field Programming

The can be started in one of three different boot modes. These modes are selected by two external boot pins BOOT0 and BOOT1. By changing the boot mode the microcontroller will alias different areas of the memory map to the bottom of memory. This allows us to execute code from user FLASH, internal SRAM or system memory. If system memory is selected, the STM32 will start to execute a factory- programmed bootloader which allows the user flash to be reprogrammed.

3.4.5 Boot Modes

For normal operation BOOT0 must be tied to ground. If you wish to use the other modes you must provide jumpers to allow different settings on the two boot pins.



The external boot pins are used to select which region of memory aliased to the first 2k of memory. This can be user Flash, the internal bootloader or the first 2k of SRAM

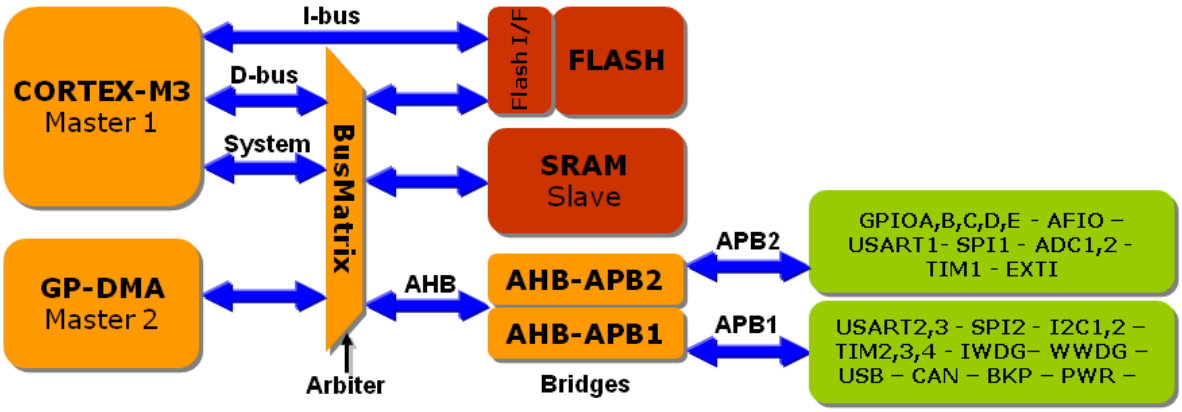
This is typically necessary if you want to do a field upgrade using the internal bootloader. If you do plan to use the bootloader, USART1 is the default serial interface used to download code from a PC, so you will need to add an RS232 driver chip.

3.4.6 Debug Port

Finally, we need to add the hardware debug port to allow a debugger to connect to the STM32. The Cortex CoreSight debug system supports two connection standards: the five pin JTAG port and the 2 pin Cortex serial wire port. Both of these configurations sacrifice GPIO pins for use by the debugger. After reset, the Cortex CPU places these pins in their alternate function setting so that the debug port is available. If you wish to use them you must program the alternate function registers to convert them back to GPIO pins. The five pin JTAG interface is brought out to a 20 pin IDC type connector which has a standard pinning for all JTAG tools. The serial wire interface uses Port A 13 for the serial data and Port A 14 for the serial clock.

4. STM32 System Architecture

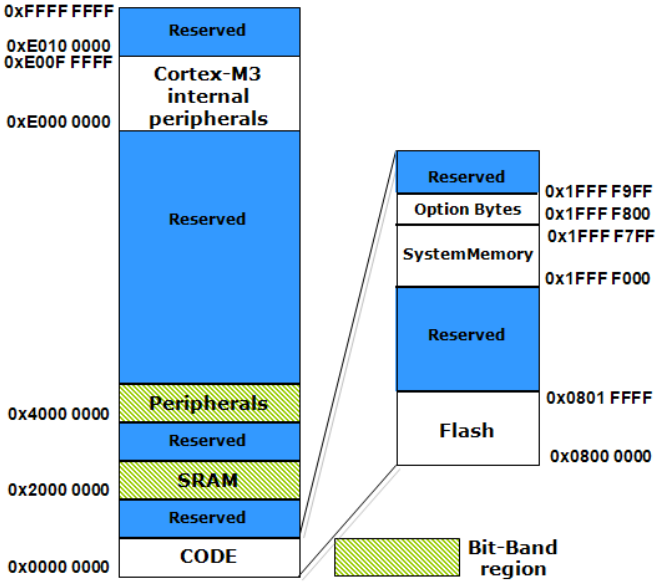
The STM32 is composed of the Cortex core which is connected to the FLASH memory by the dedicated Instruction bus. The Cortex Data and System busses are connected to a matrix of ARM Advanced High Speed Busses (AHB). The internal SRAM is connected directly to the AHB bus matrix as is the DMA unit. The peripherals are located on two ARM Advanced Peripheral Busses (APB). Each of the APB busses is bridged onto the AHB bus matrix. The AHB bus matrix is clocked at the same speed as the Cortex core. However, the AHB busses have separate prescalers and may be clocked at slower speeds to conserve power. It is important to note that APB2 can run at the full 72MHz while APB1 is limited to 36MHz. Both the Cortex and the DMA unit can be bus masters. Because of the inherent parallelism of the bus matrix, they will only arbitrate if they are both attempting to access the SRAM, APB1 or APB2 at the same time. However, as we will see in the DMA section, the bus arbiter will guarantee 2/3 access time for the DMA and 1/3 for the Cortex CPU.



The internal bus structure provides a dedicated bus for program instructions and a bus matrix which provides several data paths for the Cortex and DMA units to access the on-chip resources.

4.1 Memory Layout

Although the STM32 has numerous internal busses, to the programmer it offers a linear 4Gbyte address space. As the STM32 is a Cortex-based microcontroller, the memory map must conform to the standard Cortex layout. Therefore the program memory starts from 0x00000000. The on-chip SRAM starts from 0x20000000 and all the internal SRAM is located in the initial bit band region. The user peripherals are memory mapped starting from 0x40000000 and are also located in the peripheral bit band region. Finally all the Cortex registers are at their standard locations starting from 0xE0000000.



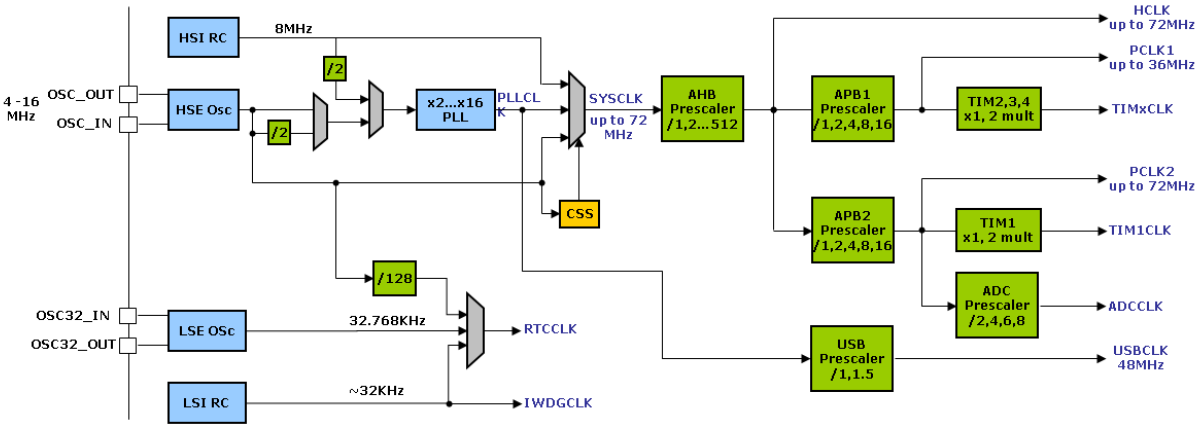
BOOT Mode Selection Pins		Boot Mode	Aliasing
BOOT1	BOOT0		
x	0	User Flash	User Flash is selected as boot space
0	1	SystemMemory	SystemMemory is selected as boot space
1	1	Embedded SRAM	Embedded SRAM is selected as boot space

The STM32 memory map follows the Cortex standard. The first 2K of memory is mapped from FLASH, System Memory or SRAM depending on the condition of the boot pins.

The FLASH memory region is composed of three sections. First is the User FLASH memory starting at 0x0000000. Next is the System Memory also called the big information block. This is 4K of FLASH memory that is factory programmed with a bootloader. The final section from 0x1FFFF800 is called the little information block and contains a group of option bytes that allow you to configure some system settings for the STM32. The bootloader is designed to download code over USART1 and program it into the User FLASH memory. To place the STM32 in bootloader mode the external BOOT0 and BOOT1 pins must be held low and high respectively. With the boot pins held in this pattern, the system memory block will appear at 0x00000000. After a reset the STM32 will begin to execute the bootloader rather than the application code held in the User FLASH. A bootloader application for the PC is available for download from the ST website. This program will communicate with the bootloader code and can be used to erase and reprogram the User FLASH memory. The PC download software is also delivered as a DLL which allows you to write custom bootloader software for field upgrading or production programming. The bootpins also allow the internal SRAM to be mirrored at 0x00000000 in place of the User FLASH. This allows programs under development to be downloaded and executed from the internal SRAM. This speeds the download process and saves continual burning of the FLASH memory.

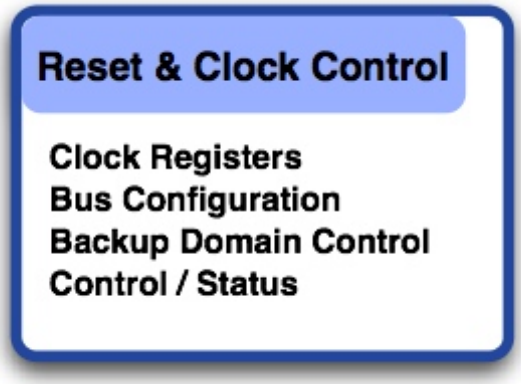
4.2 Maximising Performance

In addition to the two external oscillators the STM32 has two internal RC oscillators. After reset the initial clock source for the Cortex core is the High Speed Internal Oscillator which runs at a nominal 8 MHz. The second internal oscillator is a Low Speed Oscillator running at a nominal 32.768 KHz. This oscillator is intended for the real time clock and watchdogs.



The STM32 has a sophisticated clock tree with two external and two internal oscillators, plus a Phase Locked Loop. The High Speed External Oscillator may also be monitored by a clock security system.

The Cortex processor can be clocked by either the Internal or External High Speed Oscillators or from an internal Phase Locked Loop. The Phase Locked Loop can be driven from either the Internal or External High Speed Oscillator. So it is possible to run the STM32 at 72 MHz without an external oscillator. The downside is that the internal oscillator is not an accurate and stable 8 MHz clock source. In order to use the serial communications peripherals or do any accurate timing functions the external oscillator should be used. Regardless of whichever oscillator is selected, the Phase Locked Loop must be used to derive the full72MHz clock frequency for the Cortex core. All of the oscillator PLL and bus configuration registers are located in the Reset and Clock Control (RCC) group.

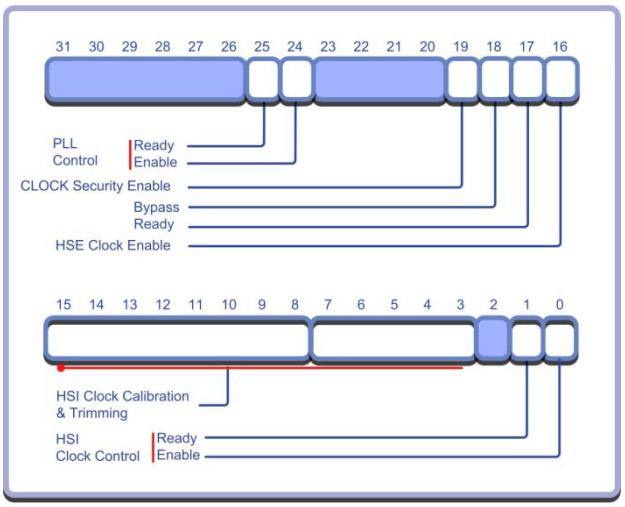


The Reset And Clock Control unit controls the clock tree bus bridges and backup domain.

4.2.1 Phase Locked Loop

After reset the STM32 will derive its CPU clock from the HSI oscillator. At this point the external oscillator is switched off. The first step in running the STM32 up to full speed is to switch on the HSE oscillator and wait for it to stabilise

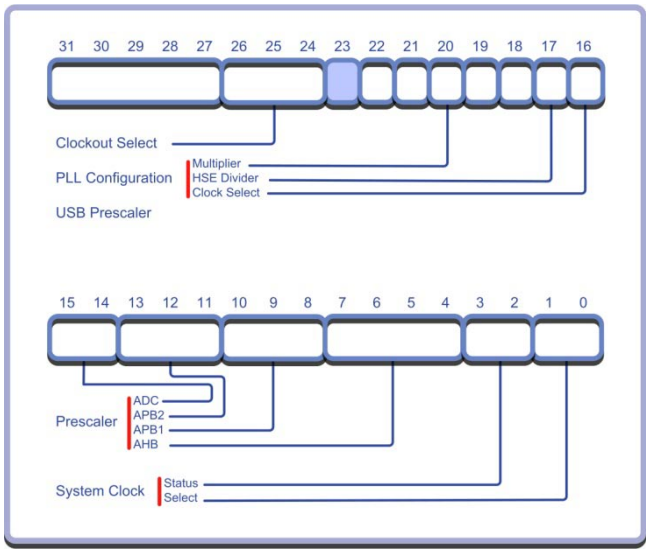
After reset the STM32 runs from the Internal High Speed Oscillator. The External Oscillator must be switched on.




```
RCC->CR |= 0x10000; //HSE on

// Wait until HSE stable
while(!(RCC->CR &0x00020000))
{
   ;
}
```

The external oscillator can be switched on in the RCC_Control register. A ready bit indicates when the external oscillator is stable. Once the external oscillator is stable, it can be selected as the input of the PLL. The output frequency of the PLL is defined by selecting an integer multiply value which is stored in the RCC_PLL_configuration register. In the case of an 8 MHz oscillator, the PLL must multiply the input frequency by 9 to generate the maximum 72 MHz clock frequency. Once the PLL multiplier has been selected, the PLL can be enabled in the control register. Once it is stable, the PLL ready bit will be set and the PLL output can be selected as the Cortex CPU clock source.

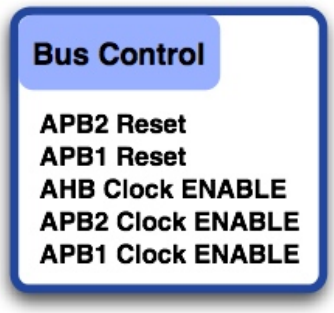
Once the HSE oscillator is on it can be used to supply the PLL. Once the PLL is stable it can become the system clock




```
//HSE clock,PLLx9
RCC->CFGR = 0x001D0000; //Enable PLL
RCC->CR |= 0x01000000;

while(!(RCC->CR & 0x02000000))
{
   ;
}

//Set the remaining control fields
RCC->CR |= 0x00000001;

//Set the remaining configuration
//fields
RCC->CFGR |= 0x005D0402;
```

4.2.1.1.1 Bus Configuration

Once the PLL has been selected as the system clock source the Cortex CPU will be running at 72 MHz. To get the remainder of the chip running at its optimum speed you will need to configure the AHB and APB busses through their bridge registers




```
//Enable clocks to the AHB,APB1 and APB2 busses
AHBENR       = 0x00000014;
RCC->APB2ENR = 0x00005E7D;
RCC->APB1ENR = 0x1AE64807;

//Release peripheral reset line on APB1 and APB2 buses
RCC->APB2RSTR= 0x00000000
RCC->APB1RSTR= 0x00000000;
```

After rest many of the peripherals have their clock disabled and are held in reset. Before using a peripheral enable its clock and release it from reset.

4.2.2 FLASH Buffer

When we looked at the system architecture of the STM32 we saw that the Cortex-M3 core is connected to the internal FLASH by a dedicated I-Bus. This bus is running at the same frequency as the CPU, so with the PLL enabled the core will be trying to run at the full 72 MHz. Since the Cortex CPU is essentially a single cycle machine, it will be trying to access the FLASH every 1.3ns. When the STM32 starts up it is running from the internal oscillator at 8 MHz, so the access time of the internal FLASH is not an issue. However, once the PLL is enabled and becomes the clock source, the FLASH access time is simply too long (35 ns.) to allow the Cortex CPU to run at maximum performance. In order to allow the Cortex CPU to run at 72 MHz with zero waitstates, the FLASH memory has a prefetch buffer which is made up of two 64-bit buffers. Both of these buffers can do a 64-bit-wide read of the FLASH memory and then pass the individual 16 or 32-bit instructions to the Cortex CPU for execution. This technique works well with the conditional execution features of the Thumb-2 instruction set and the branch prediction of the Cortex pipeline. During normal operation the programmer does not need to take any special precautions because of the FLASH buffer. However you must make sure that it is enabled before switching to the PLL as the main clock source. The FLASH buffer is controlled by the FLASH access control register. As well as enabling the prefetch buffer, you must tune the number of waitstates required for the FLASH prefetch buffer to read the 8 bytes of instructions from the FLASH memory. The latency settings are as follows:

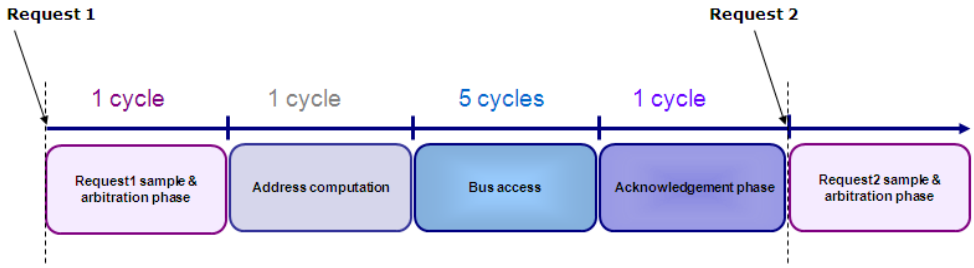
0< SYSCLK <24MHz	0 waitstate
24< SYSCLK <48MHz	1 waitstate
48<SYSCLK <72MHz	2 waitstate

These waitstates are between the prefetch buffer and the FLASH memory and do not impact on the Cortex CPU. As the CPU is executing instructions held in the first half of the buffer, the second half is loading so that the CPU can seamlessly continue execution at its optimum rate.

4.2.3 Direct Memory Access

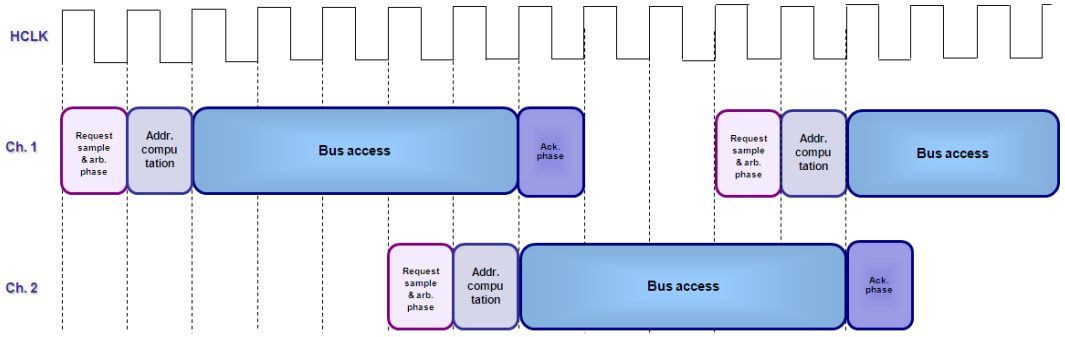
While the Cortex CPU can be used to move data between peripherals and the internal SRAM, many of these operations can be automated with the internal DMA unit. The STM32 DMA unit has seven independently configurable channels that can perform autonomous transfers from memory to memory, peripheral to memory, memory to peripheral and peripheral to peripheral. The memory to memory transfers will be performed as fast as the DMA channel can move the data. In the case of the peripheral transfers, the DMA unit is placed under the control of a selected peripheral and the data is transferred on demand to or from the controlling peripheral. As well as transferring blocks of data, each DMA unit can continually transfer data to a circular buffer. Since most communications peripherals do not contain any FIFO buffers, the DMA units are used to stream data to and from

buffers in SRAM. The DMA unit has been specifically designed for the STM32 and as such it is optimised for the short but frequent data transfers that you typically find in microcontroller applications.



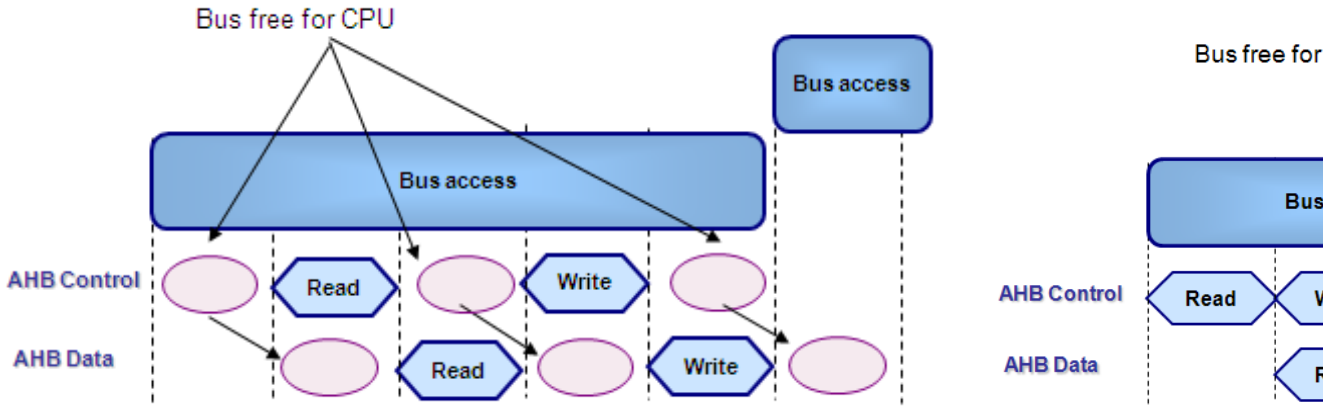
Each DMA memory to memory transfer is made up of four phases each 1 cycle long except for the bus access phase that takes five cycles for each word transferred

Each DMA transfer is made up of four phases: a sample and arbitration phase, an address computation phase, bus access phase and a final acknowledgement phase. Each phase takes a single cycle, with the exception of the bus access phase. The bus access phase (which is where the actual data transfer takes place), takes three cycles for each word transferred. The DMA unit and the Cortex CPU are designed to work together in an interleaved fashion so the DMA will not block the CPU and vice versa. We will see how this works in a moment, but it is only necessary to prioritise DMA transfers between different DMA channels. Each DMA channel is assigned one of four priority levels by the application software. During the arbitration phase the highest priority level will be granted the bus. If two DMA units have pending transfers and both have the same priority level, the unit with the lowest channel number will be granted the bus.



The DMA unit is designed for fast but short data transfers of the kind found in typical small embedded systems. The DMA unit only occupies the bus during the bus access phase.

The DMA unit can perform both the arbitration and address computation phase while another DMA Channel is in its bus access phase. As soon as the active channel finishes its data transfer on the internal bus, the next DMA channel transfer is ready to begin immediately while the original transfer finishes off its transfer by performing its acknowledgement phase. So the DMA channels not only transfer data faster than the CPU, but are also tightly interleaved and only occupy the bus for actual data transfer.

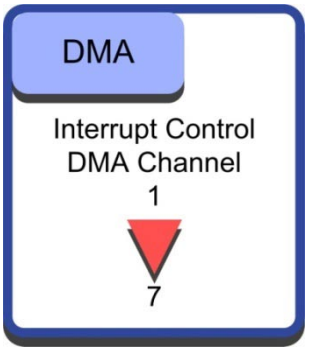


During the bus access phase three cycles per transfer are free for the CPU. In a memory to memory transfer this guarantees the Cortex-M3 60% of the bus even when the DMA is running continuously (remember this is for data transfer only). The Cortex has a separate I-code bus to fetch instructions.

In the case of a memory to memory transfer each DMA channel only occupies the data bus during its bus access phase and takes five cycles to transfer each word of data. That is, one cycle to read the data and one cycle to write, interleaved with idle cycles where the bus is left free for the Cortex CPU. This means that the DMA units will consume a maximum of 40% for the data bus bandwidth even during continuous maximum data transfer. In the case of peripheral to peripheral and peripheral to memory transfers the situation is a little more complicated. Transfers over the AHB bus take two cycles at the AHB clock frequency and transfers involving the APB take two cycles at the APB bus frequency and a further 2 cycles at the AHB clock frequency. Each DMA transfer consists of two bus transfer periods and a free cycle. So for example, a transfer from the SPI peripheral to the SRAM will consist of a transfer from the SPI, plus a transfer to the SRAM plus one free cycle thus;

SPI to SRAM DMA transfer = SPI transfer (APB) + SRAM transfer (AHB) + free cycle(AHB)
= (2 APB cycles + 2 AHB cycles) + 2 AHB cycles + 1 AHB cycle
= 2APB Cycles + 5 AHB cycles

Remember that this only refers to data transfers as all of the Cortex program instructions are fetched on the separate I-Bus.

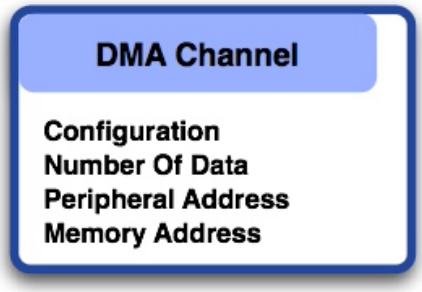


The DMA unit has seven channels. Each channel is fully orthogonal

The next good news about the DMA unit is that is very easy to use. The first thing you must remember to do is to switch in its clock and release it from reset. This is done in the AHB clock enable register within the reset and clock control unit.

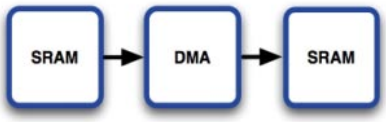
```
RCC->AHBENR |= 0x00000001;        // enable the DMA clock
```

Once the DMA unit is powered up, each DMA channel is controlled by four registers. Two registers hold the source and destination addresses for the peripheral register and the memory location. The size of the transfer is held in the “number of data” register and the configuration register defines the overall characteristics of the DMA transfer.



Each DMA channel is controlled by four registers and has three interrupt sources, finished, half-finished and error.

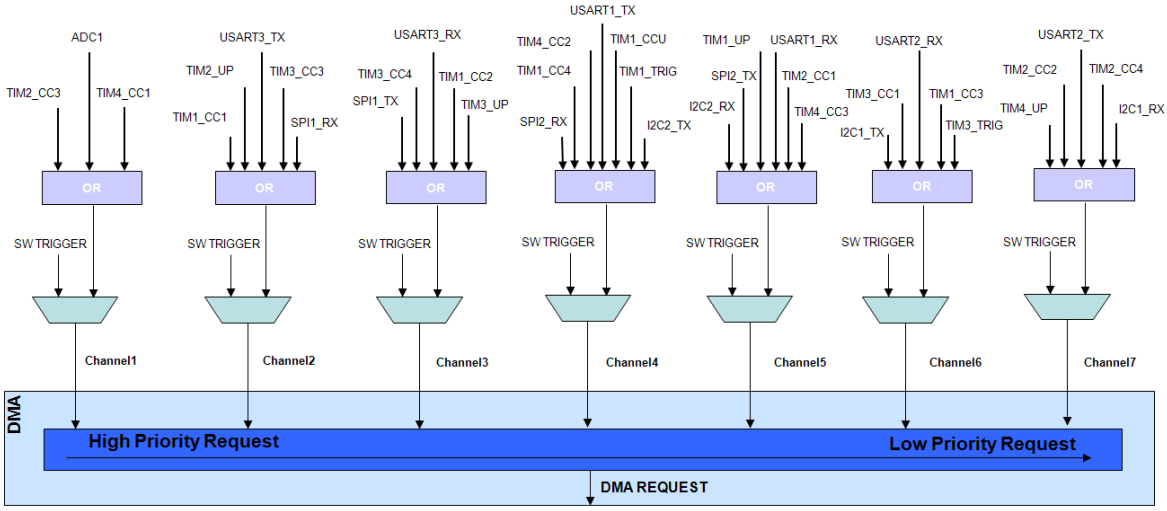
Each DMA channel can be assigned a priority of “very high”, “high”, “medium” and “low”. The transfer word size can also be defined separately for the memory and the peripheral. For example, we can flow a 32-bit word into the DMA channel (3 cycles) from memory and then flow four 8-bit words to the UART data register (a total of 35 cycles rather than 64 cycles if everything was moved as 8-bit quantities.) It is also possible to increment the memory and peripheral addresses, so for example you can transfer data repeatedly from the ADC results register but increment the memory address, so that the results are written to an array in memory for processing. The Transfer Direction Bit allows us to specify if the flow of data is memory to peripheral or peripheral to memory. For a memory to memory transfer we must set bit 14 to enable a “fast as you can” data transfer between two SRAM buffers. Though you can use the DMA channels in a polled mode, each DMA channel has three interrupt sources: transfer finished, half-finished and transfer error. Finally, once the DMA transfer is fully configured, the Channel Enable Bit must be set and the transfer will begin. A memory to memory transfer can be performed with the following code:



This code shows a simple DMA memory to memory transfer and uses an internal timer to count the number of cycles taken.

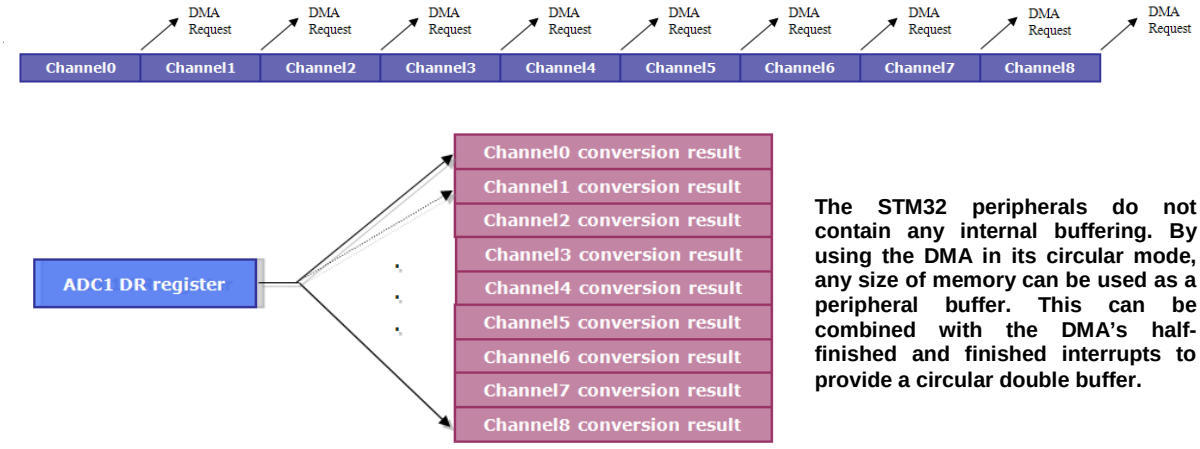
```
DMA_Channel1->CCR   = 0x00007AC0;               //configure for mem2mem transfer
DMA_Channel1->CPAR  = (unsigned int)src_arry;   //set source and destination
DMA_Channel1->CMAR  = (unsigned int)arry_dest;
DMA_Channel1->CNDTR = 0x000A;                   //set size of transfer

TIM2->CR1           = 0x00000001;               //start a timer
DMA_Channel1->CCR    |= 0x00000001;             //start the DMA transfer
while(!(DMA->ISR & 0x0000001))                  //wait till the transfer ends
{
;
}
TIM2->CR1 = 0;                                  //halt the Timer
TIM2->CNT = 0;                                  //Clear the count
TIM2->CR1 = 1;                                  //restart timer
for(index = 0;index <0xA;index++)               //repeat the operation using the CPU
{
arry_dest[index] = arry_src[index];
}
TIM2->CR1 = 0;                                  //halt the timer

}
```

The above code transfers ten words of data between two arrays in SRAM first with the DMA and then using the Cortex CPU. In both cases a timer is started at the beginning of the transfer and when the transfer ends. In this example the DMA unit takes 220 cycles and the CPU takes 536.



Each peripheral with DMA support is assigned to a specific channel. When enabled, the peripheral becomes the flow controller for the DMA transfer. This allows it to sink or source data as it requires without taking any CPU cycles.

While memory to memory transfers can be useful for initialising regions of memory and performing block transfers, most of the time the DMA channels will be used to move data between memory and the various user peripherals. In this case, each of the DMA channels is mapped to a selection of peripherals. First we must initialise the peripheral and enable its DMA support, then we must configure the matched DMA channel to transfer data on request of the supported peripheral. We will have a more detailed look at the ADC later, but in a simple conversion mode it can perform continuous 10-bit conversions to a single results register. Without the DMA the Cortex CPU would be continually responding to ADC conversion complete interrupts and these would start to steal useful amounts of CPU runtime. However, by using the DMA the ADC can request a DMA transfer at the end of each conversion. The DMA will transfer the ADC results data to an incrementing address in the SRAM. The ADC data can then be processed once a suitable sample of data has been transferred.



The STM32 peripherals do not contain any internal buffering. By using the DMA in its circular mode, any size of memory can be used as a peripheral buffer. This can be combined with the DMA's half-finished and finished interrupts to provide a circular double buffer.

To make this process more efficient, we can enable the circular buffer support so that the ADC data will continuously write to our buffer. Then, by using the half complete and transfer complete interrupts, we can create a double buffer. So when the first half of the buffer is full, an interrupt will be generated and we can process this data while the DMA continues to fill the second half. Once the second half is full, we can process this data while the DMA starts to refill the buffer from the top. All the other peripherals with DMA support are handled in a similar way. It should be noted that the communication peripherals have separate transmit and receive DMA channels. For example, the SPI can simultaneously flow data in both directions.

5. Peripherals

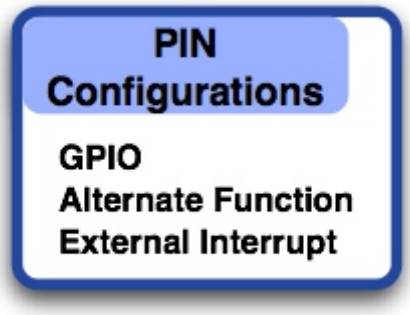
This chapter will present an introduction to the user peripherals on the existing STM32 variants. For convenience, these are split into two groups: general purpose microcontroller peripherals and communications peripherals. All of the peripherals on the STM32 have a high level of sophistication and are tightly integrated with the DMA unit. Each peripheral has some form of extended hardware functionality, which can be useful in minimising the amount of CPU time required to drive a given peripheral. In other words, there are lots of clever features that can help you to automate the hardware, reducing the CPU effort in driving the peripherals.

5.1 General Purpose Peripherals

The general purpose peripherals on the STM32 consist of: general purpose IO, external interrupt controller, analogue to digital converters, general purpose and advanced timer units, and real-time clock with backup registers and tamper pin.

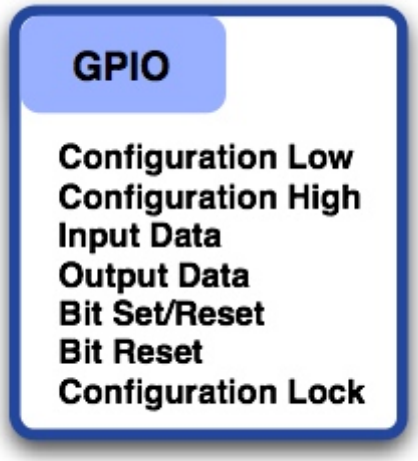
5.1.1 General Purpose IO

The STM32 is well served with general purpose IO pins, having up to 80 bidirectional IO pins. The IO pins are arranged as five ports each having 16 IO lines.



Each digital pin may be configured as GPIO or as an alternate function. Each pin can simultaneously be configured as one of 16 external interrupt lines.

These ports are names A-E and are all 5v tolerant. Many of the external pins may be switched from general purpose IO to serve as the Input/Output of a user peripheral, for example a USART or I2C peripheral. Additionally there is an external interrupt unit which allows 16 external interrupt lines to be mapped onto any combination of GPIO lines.



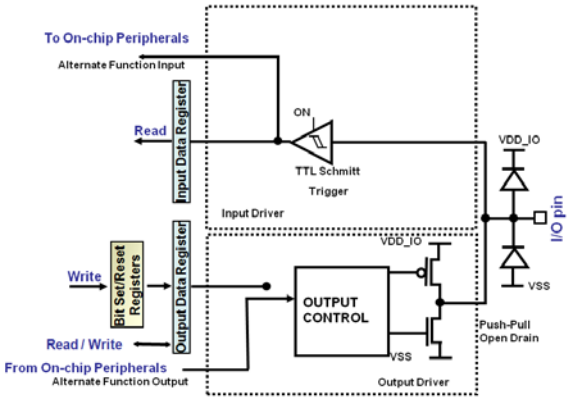
Each GPIO port can configure individual pins as input or output with different driver configurations. It has registers to write word-wide or for atomic bit manipulation. Once a configuration is defined it can be locked.

Each GPIO port has two 32-bit wide configuration registers; these two registers combine to give a 64-bit wide configuration register. Within these 64 bits each pin has a four bit field that allows its characteristics to be defined. The four bit configuration field is made up of a two bit wide mode field and a two bit wide configuration field. The

mode field allows the user to define the pin as an input or an output, while the configuration field defines the drive characteristics:

As well as being able to define a port pin as an input or output, its drive characteristics may also be selected. In the case of an input, an internal resistor can be connected as a pull up or pull down resistor. For an output, each port pin may be configured with a push pull or an open drain driver. Each output pin can also be configured with a maximum output speed of 2MHz,10MHz or 50MHz.

Configuration Mode	CNF1	CNF0	MOD1	MOD0
Analog Input	0	0	00	
Input Floating (Reset State)	0	1		
Input Pull-Up	1	0		
Input Pull-Down	1	0		
Output Push-Pull	0	0	00: Reserved 01: 10 MHz 10: 2 MHz 11: 50 MHz	
Output Open-Drain	0	1		
AF Push-Pull	1	0		
AF Open-Drain	1	1		



Once the port configuration has been set, these parameters can be protected by writing to the configuration lock register. In this register each pin has a lock bit which when set will prevent any writes to the matching configuration and mode fields. When all of the required lock bits have been set, the lock can be activated by writing a sequence of 1,0,1 to bit 16 in the lock register, followed by two reads of the same bit which will return 0,1 if the lock has been successfully activated. The input and output data registers allow port wide access to the IO pins. Atomic bit manipulation is supported by either using the Cortex bit banding technique on the input and output data registers, or through two dedicated bit manipulation registers. The bit set/reset register is a 32-bit wide register. The upper 16 bits are mapped to each port pin. Writing a logic 1 to these locations will reset the matching port pin. Similarly, writing a logic 1 to any of the lower 16 bits will set the matching port pin. The second bit manipulation register is a bit reset register. This is a 16-bit wide register where writing logic 1 in the lower 16 bits will reset the matching port pin. The combination of port registers, bit banding and atomic bit manipulation registers allows you very fine control of all the STM32 port pins and can be used very efficiently for IO intensive applications.

5.1.1.1 Alternate Functions

The alternate function registers allow you to remap the port pins from GPIO to alternate peripheral functions. To allow flexibility in hardware design, a given peripheral function can be mapped to one of several pins.

A late-arriving high priority interrupt will pre-empt a low priority interrupt without incurring an additional stacking overhead.

The STM32 alternate functions are controlled in the remap and debug IO register. Each of the digital user peripherals (USART,CAN, timers, I2C and SPI) has a 1 or two bit field which allows mapping to several different pin combinations. Once the alternate function pins have been selected, the GPIO configuration registers must be used to switch from IO to alternate function. The remap register also controls the configuration of the JTAG debug

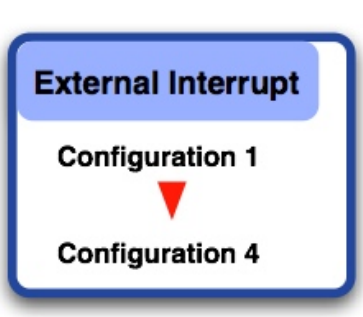
pins. After reset, the JTAG port is enabled with the data trace disabled. The JTAG can be switched to serial wire (two pin debug) or disabled, the unused pins in each case can be used as GPIO.

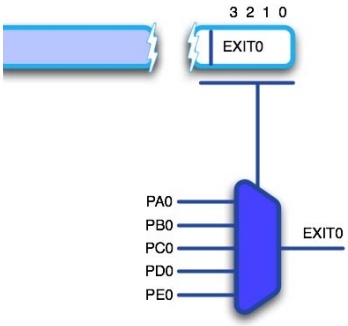
5.1.1.2 Event Out

The Cortex processor can generate an event out pulse that is intended to wake up a separate microcontroller from a low power mode. Typically, the event out pulse would be connected to the wake up pin of a second STM32. The event out pulse is generated by executing the SEV Thumb-2 instruction. In the STM32 the event control register is used to route the event out pulse to a selected GPI pin. The event control register contains fields to select the port and pin within the port. Once the port pin is selected, the event out enable bit is set to finish the configuration.

5.1.2 External Interrupts

The external interrupt unit has 19 interrupt lines that are connected to interrupt vectors via the NVIC. Sixteen of these interrupt lines are connected to GPIO pins and can generate an interrupt on a rising or falling edge, or both. The remaining three exit lines are connected to the RTC alarm interrupt, the USB wake up and the Power voltage detect unit. The NVIC provides individual interrupt vectors for EXTI lines 0-4, the RTC alarm, Power voltage detect and the USB wake up. The remaining EXTI lines are connected in groups of lines 5-9 and lines 10 – 15 to two additional interrupt vectors. The EXTI is important for power control on the STM32. As it is not a clocked peripheral, it can be used to wake up the microcontroller from its STOP mode where both the main oscillators are halted. The EXTI can generate both interrupts to exit Wait for interrupt mode and events to exit Wait for event mode.





The STM32 has 16 external interrupt lines which may be connected to each of the port pins.

The 16 EXTI lines dedicated to the GPIO pins can be mapped to any combination of port pins. This is done through four configuration registers. In these registers each EXTI line is mapped to a four bit field. This field allows each EXTI line to be mapped onto any of the five IO ports, so for example EXTI line zero can be mapped onto pin 0 of port A, B, C, D or E. This scheme allows any external pin to be mapped to an interrupt line. The EXTI can also be used in conjunction with an alternate function that has been remapped to an external pin.

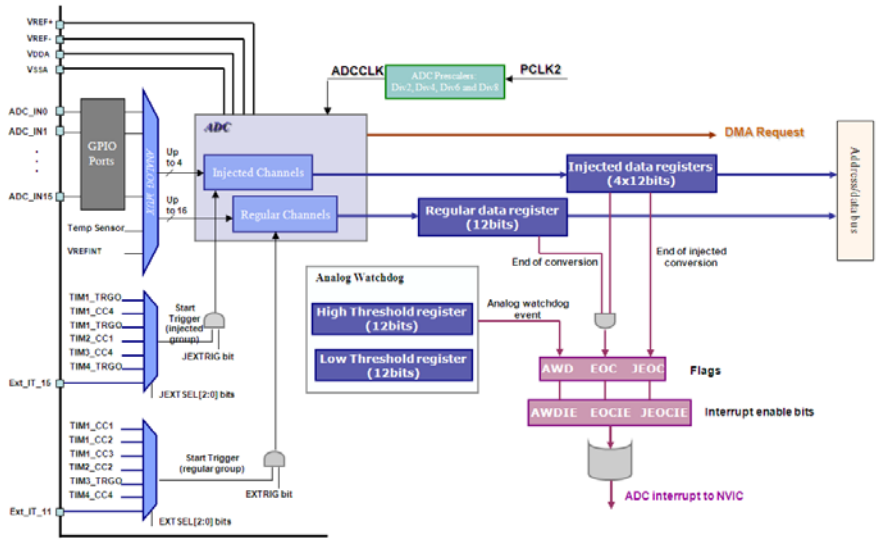

```
//Map the external interrupts to port pins
AFIO->EXTICR[0] = 0x00000000;
//Enable External interrupt sources
EXTI->IMR       = 0x00000001;
//Enable wake up events
EXTI->EMR       = 0x00000000;
//Select falling edge trigger sources
EXTI->FTSR      = 0x00000001;
//Select Rising edge trigger sources
EXTI->RTSR      = 0x00000000;
//Enable interrupt sources in the NVIC
NVIC->Enable[0] = 0x00000040;
NVIC->Enable[1] = 0x00000000;
```

Once mapped the external interrupt pins can generate an interrupt on a rising and/or falling edge.

Once the EXTI configuration registers have been set, each external interrupt can be configured to generate an interrupt or event on rising or falling edges. It is also possible to force an EXTI interrupt by writing to the matching bit in the software interrupt register.

5.1.3 ADC

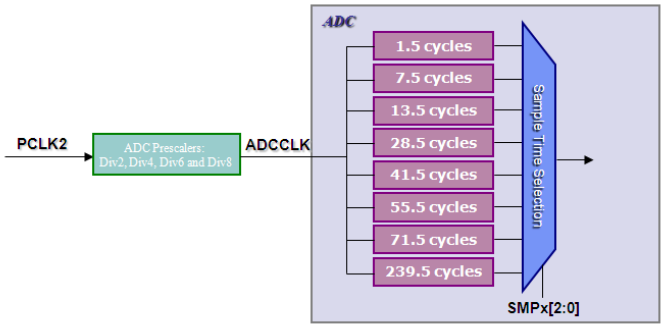
The STM32 features up to two independent analogue to digital converters, depending on variant. The ADC has an independent supply which can be between 2.4V to 3.6V, depending on the package type. The ADC reference is connected internally to the ADC supply, or brought out to a dedicated pin. The ADC converters offer a 12 bit resolution with a 1 MHz conversion rate. With up to 18 multiplexed channels, 16 can be available to measure external signals. Of the remaining two one is connected to an internal temperature sensor and the second is connected to an internal reference voltage.



The STM32 is a highly featured 12 bit 1MHz sample rate converter with internal band gap and temperature sensor.

5.1.3.1 Conversion Time And Conversion Groups

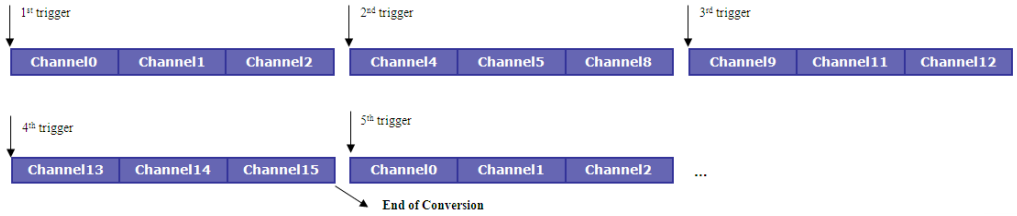
When the ADC is configured, it is also possible to individually program the conversion time for each channel. There are eight discrete conversion times ranging from 1.5 cycles to 239.5 cycles.



Each ADC channel may be configured with an individual sample rate.

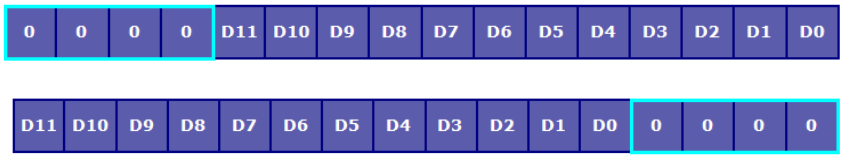
Each ADC has two basic conversion modes: regular and injected. In regular mode conversion allows you to specify a channel or group of channels to be converted on a round robin basis. The regular conversion group can be configured to have up to 16 channels. Additionally, the order in which the channels are converted can also be

programmed and in one conversion cycle a channel can be converted several times. The regular group conversion can be started by software, or by a hardware event from a range of timer signals or EXTI line 1. Once triggered, the regular group can perform continuous conversions. Alternatively it can operate in a discontinuous mode, whereby a selected number of channels are converted and then conversion halts until the next regular group trigger occurs.



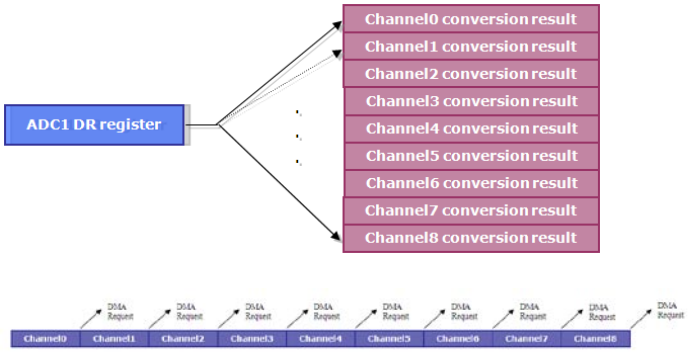
The regular group conversion sequence can run in a continuous round robin cycle. Or, in discontinuous mode it can convert a selected number of channels after each trigger event.

Each time a regular group conversion is made, the result is stored in a single results register and an interrupt can be generated. The 12-bit result is stored in a 16-bit wide register and the converted value can be aligned left or right.



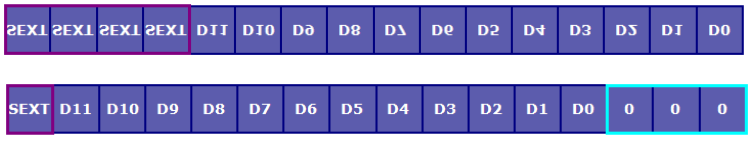
The 12-bit result can be left or right aligned in the 16-bit results register.

ADC1 has a dedicated DMA channel that can be used to transfer each conversion value from the results register to a buffer in memory. Using this method a regular group conversion cycle can be copied to memory, with a single DMA interrupt being generated at the end of the group conversion cycle. If you want to be clever, you can make the memory buffer double the size of the regular group conversion cycle and use the DMA half-finished and finished interrupts to create a double buffer. This can be combined with the DMA circular buffer mode to place a lot of ADC results-handling under hardware control.



ADC1 has DMA support which can automatically transfer results to a user-defined buffer in SRAM.

The second conversion group is called the injected group. The injected group is a conversion sequence of up to four channels that can be triggered by a software or hardware event. Once triggered, it will halt the regular conversion group, perform its sequence of conversions and then allow the regular group to continue. Like the regular group, any sequence of channels can be configured and a channel can be converted more than once in a conversion sequence. However, unlike the regular group, each injected conversion has its own results register and an offset register.

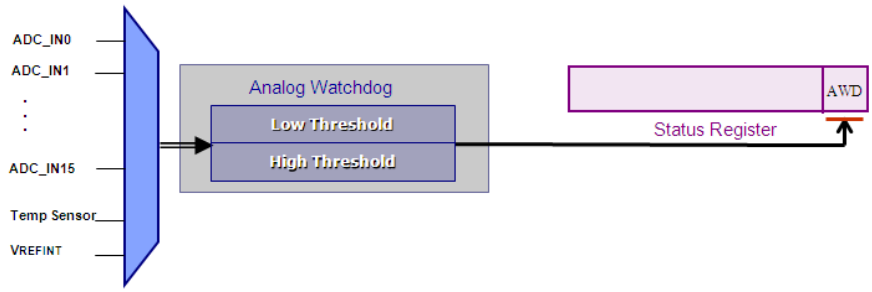


The injected group results registers are sign-extended and can be left or right justified.

The offset register can be programmed with a 16-bit value that will automatically be deducted from the ADC result. If the result is a negative value, the injected group results register is sign-extended. Like the regular group the result can be left or right justified.

5.1.3.2 Analogue Watchdog

In addition to the two conversion modes, the ADC has an analogue watchdog. This watchdog can be programmed with high and low threshold values to detect over or under voltage conditions. Once triggered, the analogue watchdog can generate an interrupt. The analogue watchdog can be used to monitor a selected regular and injected channel, or all injected and regular channels. In addition to voltage monitoring, the analogue watchdog could be used as a zero voltage crossing detector.

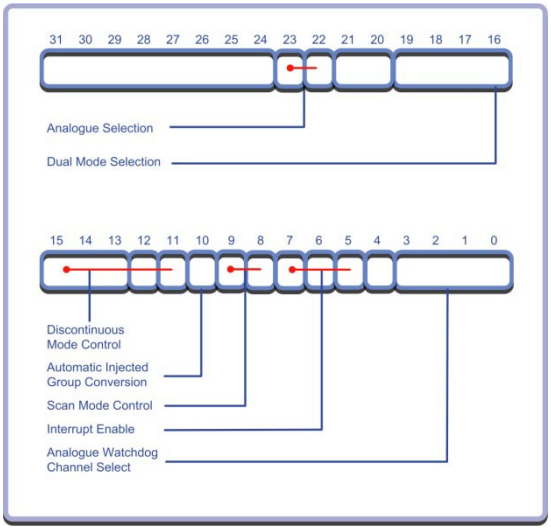


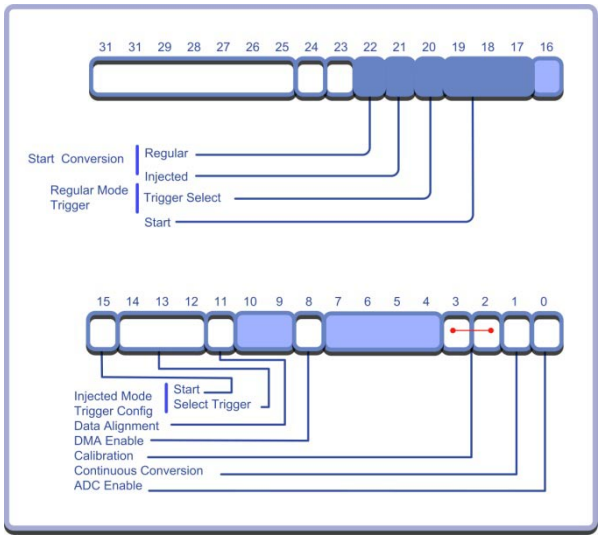
The analogue watchdog can monitor a channel or all channels for a user defined high and low threshold.

5.1.3.3 Basic ADC Configuration

The ADC registers breakdown into six groups, with the status and control registers defining the operating configuration of the ADC.

The ADC has register blocks to configure: the individual sample time, regular and injected conversion sequences along with the injected group offset values and watchdog threshold values. The overall ADC configuration is through the status and control registers.





The two control registers define the ADC operating mode. A single channel interrupt driven conversion is shown below.

```
ADC1->CR2   = 0x005E7003; //Switch on the ADC and enable continuous conversion
ADC1->SQR1  = 0x0000;     //set sequence length to one
ADC1->SQR2  = 0x0000;     //select conversion on channel zero
ADC1->SQR3  = 0x0001;
ADC1->CR2  |= 0x005E7003; //rewrite on bit

ADC1->CR1  =0x000100;     //Start  conversion  of  regular  channels,  enable  ADC
                          //interrupt

NVIC->Enable[0] = 0x00040000;    //enable ADC interrupt
NVIC->Enable[1] = 0x00000000;
```

In the ADC interrupt read the result register and copy the conversion result to a bank of port pins.

```
void ADC_IRQHandler  (void)
{
GPIOB->ODR = ADC1->DR<<5; // Copy ADC result to port pins
}
```

If the interrupt is not used a DMA channel can be used to transfer the ADC result directly to the port pins.

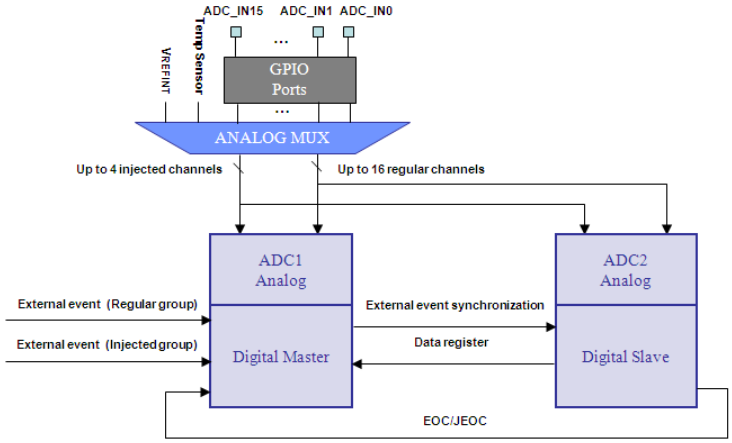
```
DMA_Channel1->CCR   = 0x00003A28;//Circular mode,
                                 //peripheral and memory increment disabled

//Load destination address into peripheral register,GPIO port data register
DMA_Channel1->CPAR  = (unsigned int) 0x4001244C;

//Load source address into memory register
DMA_Channel1->CMAR  = (unsigned int) 0x40010C0C;

DMA_Channel1->CNDTR = 0x1;       //Load number of words to transfer
DMA_Channel1->CCR  |= 0x00000001;//Enable the DMA transfer
```

In the ADC the DMA support must be enabled.

```
ADC1->CR2                |= 0x0100;
```

5.1.3.4 Dual Conversion Modes

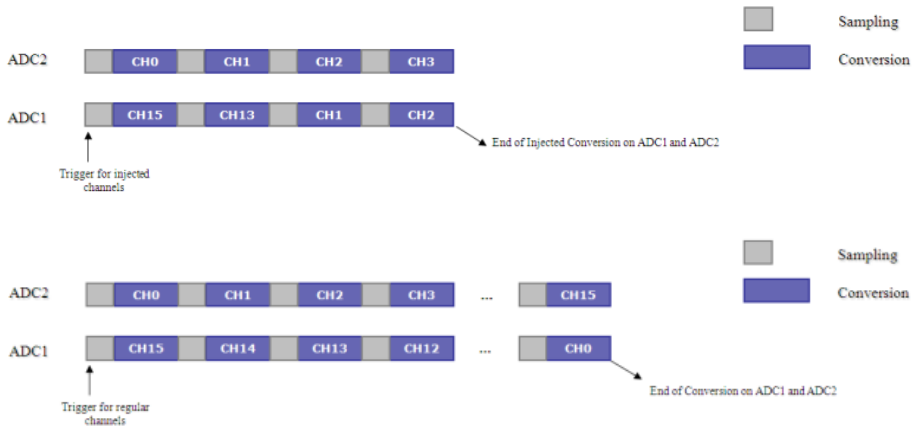
For a low cost general purpose microcontroller, the ADC on the STM32 is very sophisticated. You should take time to understand all of its features, as the ADC hardware can be configured to perform some operations that a more basic ADC would require additional software intervention to achieve. If all this isn't enough, on the STM32 variants with two ADC converters there are additional dual conversion modes.



The ADC dual conversion modes synchronise operation of the two converters providing eight additional modes.

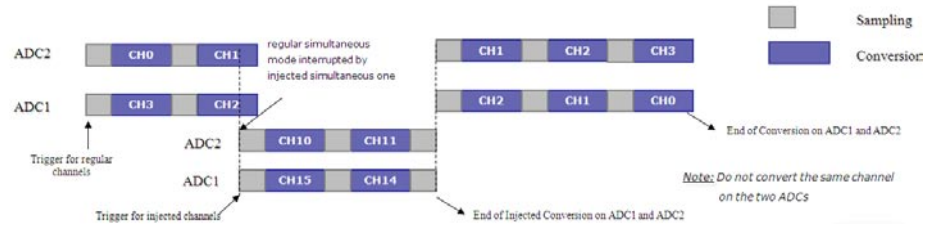
In the dual conversion modes ADC2 is slaved to ADC1 allowing eight additional conversion modes.

5.1.3.4.1 Injected Simultaneous Mode And Regular Simultaneous Modes



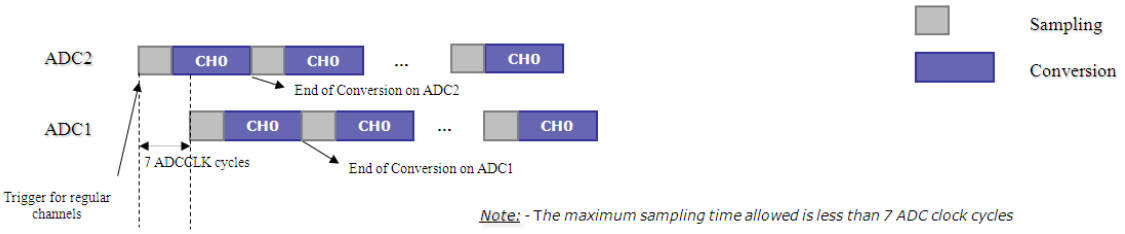
The first two dual conversion modes synchronise conversion of the regular and injected conversion groups on the two ADC converters. This is very useful if you need to measure two quantities such as voltage and current simultaneously.

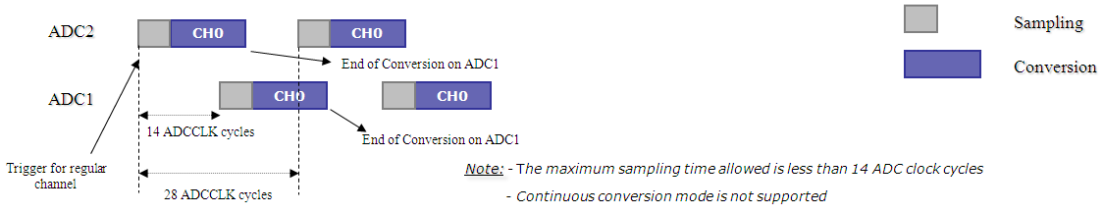
5.1.3.5 Combined Regular/Injected Simultaneous Mode



A further combined mode allows both the regular and injected groups on both ADCs to have their conversion sequences synchronised.

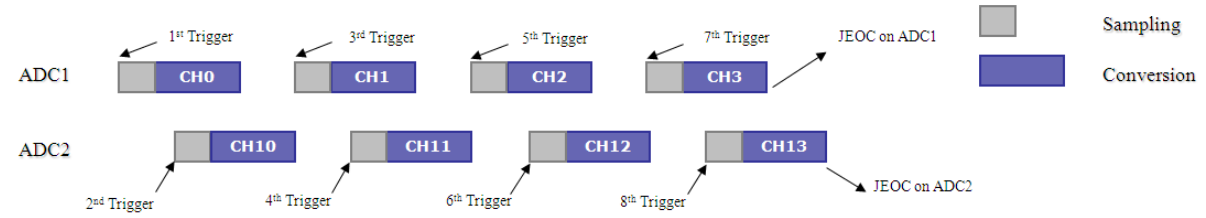
5.1.3.6 Fast Interleave And Slow Interleave Modes





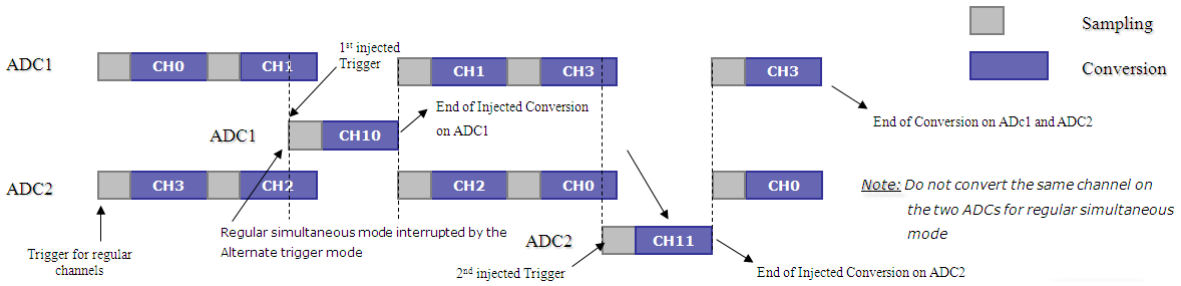
The fast and slow interleave modes synchronise the conversion of both ADC regular conversion groups, but unlike the simultaneous mode there is a delay between the start of conversion on ADC 1. In fast interleave mode this is seven ADC clock cycles after the start of ADC2 conversion. In slow interleave mode the delay is 14 ADC clock cycles. Both these modes can be used to increase the overall sampling rate by combining the two converters.

5.1.3.7 Alternate Trigger Mode



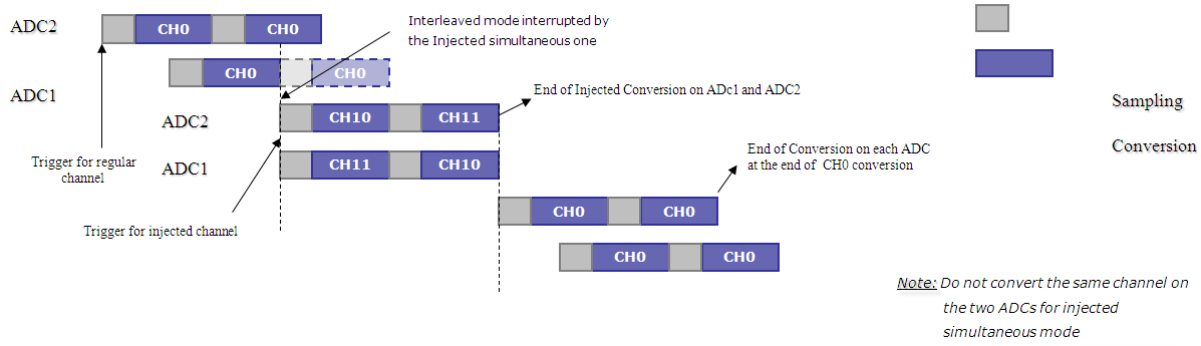
In alternate trigger mode a hardware trigger on ADC1 will first trigger an injected group conversion on ADC1. The next trigger will start a conversion on the injected group of ADC2.

5.1.3.8 Combined Regular Simultaneous And Alternate Trigger Mode



The alternate trigger mode can also be combined with the regular group simultaneous mode. This synchronises the regular conversions on both ADCs and makes alternate conversions on the two injected groups.

5.1.3.9 Combined Injected Simultaneous And Interleaved Mode



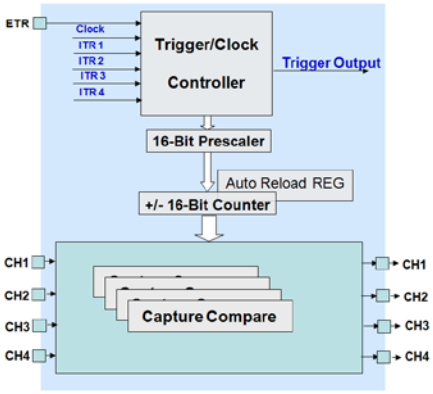
The final conversion mode performs an interleaved conversion on the two ADC regular groups but performs synchronised simultaneous conversion for the two injected groups.

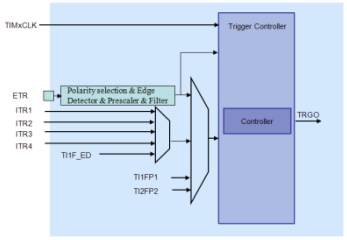
5.1.4 General Purpose and Advanced Timers

The STM32 has four timer units. Timer 1 is an advanced timer intended for motor control. The remaining timers are general purpose timer units. All of the timers have a common architecture; the advanced timer simply has additional hardware features. In this section we will look at the general purpose timers first and then move on to the advanced timer.

5.1.4.1 General Purpose Timers

All of the timer units are based on a 16-bit counter with a 16-bit prescaler and auto-reload register. The timer counter can be configured as count up, count down or centred counting (count up then count down). The clock input to the timer counter can be selected from eight different sources. These include: a dedicated clock generated from the main system clock, a trigger out clock from one of the other timers or an external clock through the capture compare pins. The timer trigger inputs and the external clock sources have a gated input to the timer counter which is controlled by an external trigger pin ETR.



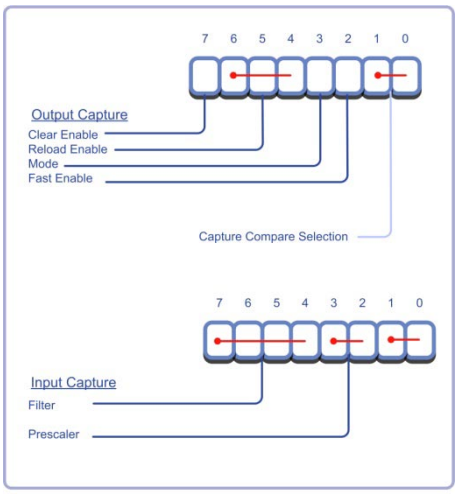


The four timers within the STM32 have a 16-bit counter with 16-bit prescaler and a four channel capture compare unit. They may be clocked from the system clock, external events or other timers.

In addition to the basic timer counter, each timer unit has a four channel capture compare unit. This unit can perform simple capture and compare functions but also has a number of special modes that allow common operations to be performed in hardware. Each of the timers has both interrupt and DMA support.

5.1.4.1.1 Capture Compare Unit

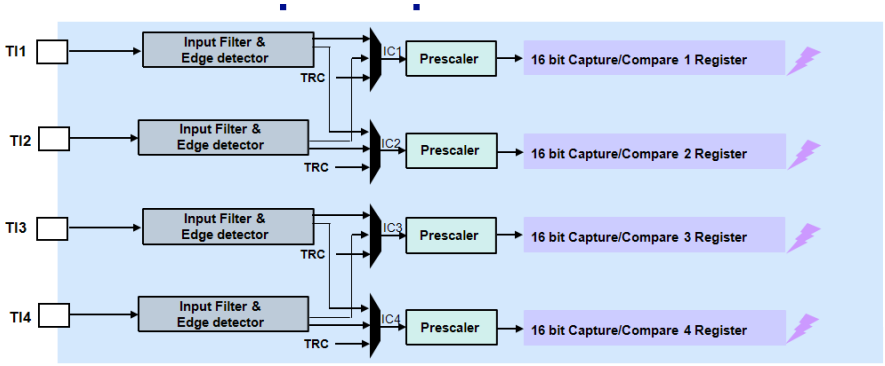
Each capture compare channel is controlled by a single register. This register has different functions, depending on the setting of the selection bits. In capture mode it has input filters and a special PWM measurement mode, plus support for encoder inputs. In compare mode it has standard compare functions and a PWM generation option plus a one pulse mode.



Each Capture/Compare channel has a single mode register. The Capture compare selection bits define the operating mode of the Cap Com channel

5.1.4.1.2 Capture Unit

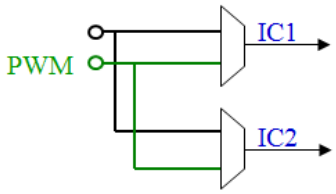
The basic capture unit has four input channels connected to configurable edge detectors. When a rising or falling edge is detected, the current timer count is captured into the channel's 16-bit capture/compare register. When a capture event occurs, the timer counter can be reset or halted. In addition, an interrupt or DMA transfer can be triggered.



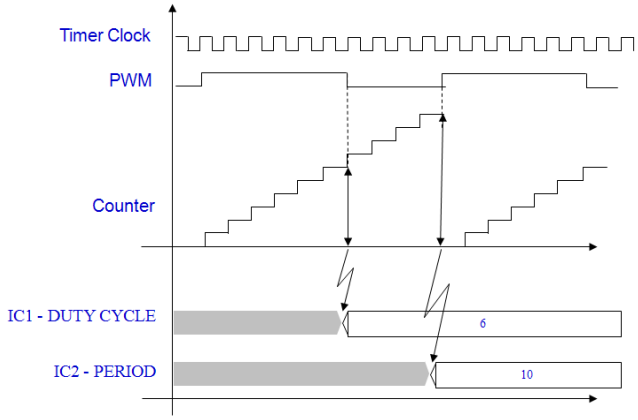
The four capture units have input filter and edge detection. A capture event can trigger an interrupt or DMA transfer.

5.1.4.1.3 PWM Input Mode

The capture unit can also be configured to use two capture channels to automatically measure an external PWM signal; both duty cycle and period can be measured.



In PWM measurement mode two channels are used to automatically capture the period and duty cycle of the PWM waveform.

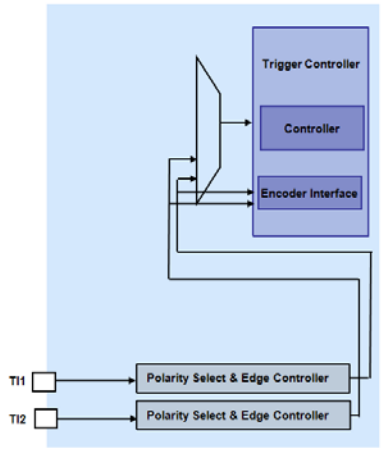



```
M3->CR1             = 0x00000000;       //default
TIM3->PSC           = 0x000000FF;       //set max prescaler
TIM3->ARR           = 0x00000FFF;       //set max reload count
TIM3->CCMR1         = 0x00000001;       //Input IC1 mapped to TI1
TIM3->CCER          |=0x00000000;       //IC1 triggers on rising edge
TIM3->CCMR1         |=0x00000200;       //Input IC2 mapped to TI1
TIM3->CCER          |=0x00000020;       //IC2 triggers on falling edge
TIM3->SMCR          = 0x00000054;       //Select TI1FP1 as input,rising edge trigger
                                        //resets the counter
TIM3->CCER          |=0x00000011;       //enable capture channels
TIM3->CR1           = 0x00000001;       //enable the timer
```

In PWM mode the input signal is routed to two capture channels. At the beginning of a PWM cycle the main counter is reset by capture channel 2 (using the rising edge of the PWM signal) and it will start counting up. On the falling edge of the PWM signal capture channel one is triggered, capturing the duty cycle value. On the next rising edge at the beginning of the next cycle capture 2 is again triggered, resetting the timer and capturing the PWM period value.

5.1.4.1.4 Encoder Interface

The capture unit on all of the timers are also designed to interface directly to an external encoder. A typical application of such an encoder is used to detect speed and angular position of a motor.

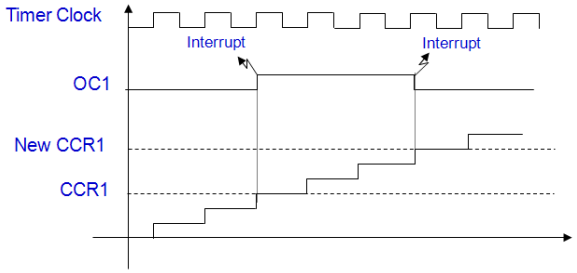


Each timer may be interfaced to a linear or rotary encoder to capture position, speed and direction information.

In this configuration the capture pins are providing the clock count to the timer counter. The count is then used to determine position. To obtain speed information a second timer must be used. This will provide a time base so that we can detect the number of encoder ticks that have occurred in a given period.

5.1.4.1.5 Output Compare

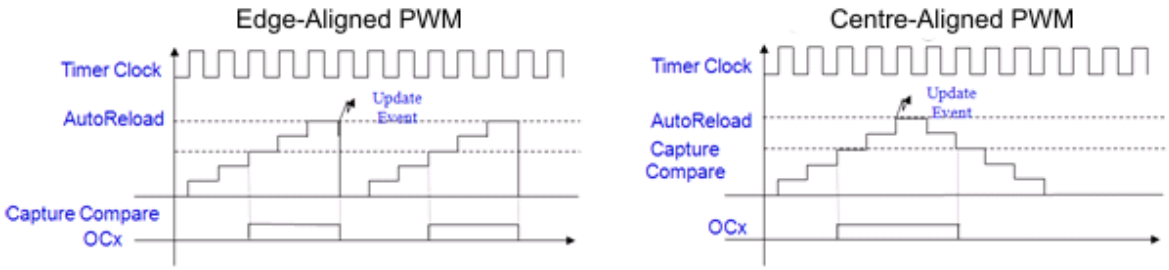
Each of the STM32 timer units also provides four channels of output compare. Using the basic compare mode, when the timer count matches the 16-bit value stored in the channel capture/compare register, a capture event is generated. This capture event can be used to: modify the state of the associated capture/compare channel pin, generate a timer reset, an interrupt or a DMA transfer.



In compare mode, each channel can be used to generate an interrupt, or change the state of the cap/com pin, when contents of the compare register match the timer count.

5.1.4.1.6 PWM Mode

In addition to the basic compare mode, each timer has a dedicated PWM generation mode. In this mode the PWM period is defined by the value stored in the timer auto reload register. The duty cycle is defined by the value stored in the channel capture/compare register. This allows each timer to generate up to four independent PWM signals. As we will see later the timers can be synchronised together so it is possible to generate up to 16 synchronised PWM signals.



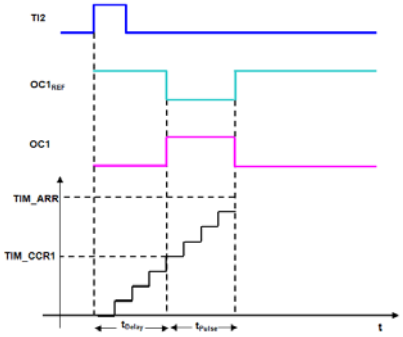
Each timer has a dedicated PWM mode that can generate edge or centre-aligned PWM waveforms.

Each channel can generate either an edge-aligned or centre-aligned PWM signal. In edge-aligned mode, the falling edge always coincides with the timer reload update event. Changing the capture/compare value simply modulates the rising edge of the PWM signal. In centre-aligned mode, the timer is configured as a centre counter (count up then count down). When a match is made with the capture/compare channel, the channel output pin is inverted.

```
TIM2->CR1           = 0x00000000;       //default
TIM2->PSC           = 0x000000FF;       //set max prescaler
TIM2->ARR           = 0x00000FFF;       //set max reload count
TIM2->CCMR1         = 0x00000068;       //Set PWM mode
TIM2->CCR1          = 0x000000FF;       //Set PWM start value
TIM2->CCER          = 0x00000101;       //Enable CH1 output
TIM2->DIER          = 0x00000000;       //enable update interrupt
TIM2->EGR           = 0x00000001;       //enable update
TIM2->CR1           = 0x00000001;       //enable timer
```

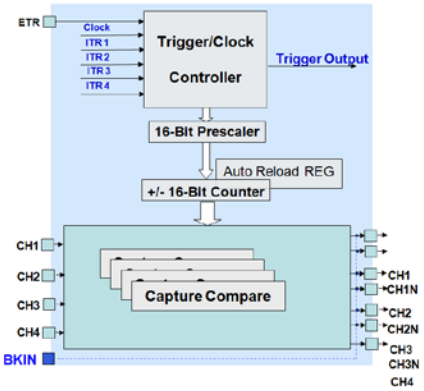
5.1.4.1.7 One Pulse Mode

In the basic compare and PWM mode the timer units will generate a continuous output waveform. Each of the timers also has a one pulse mode option. This is really a special case of the PWM mode, where an external trigger (an external pin or another timer trigger output) can start the PWM mode running for one cycle. This generates a single pulse with a programmable start delay and pulse width.



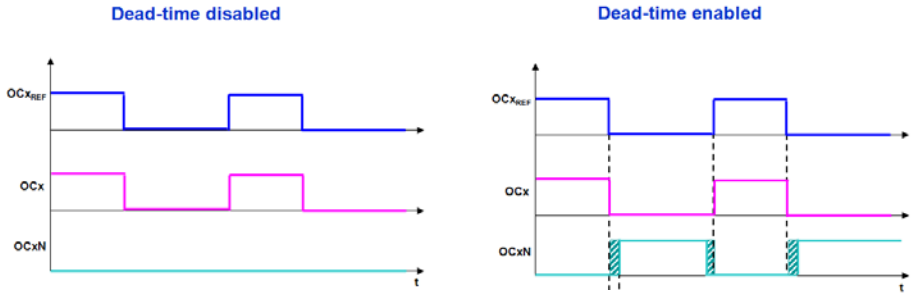
The one pulse mode allows you to define a single shot pulse with configurable delay and duration.

The advanced timer is timer unit 1. This timer contains additional hardware specifically intended for motor control. Three of the advanced timer channel output pins have complementary outputs. This provides a six channel PWM unit. As this unit is intended for three phase motor control, each channel has programmable dead time and there is a global break input line. There is also a Hall sensor interface in addition to the encoder interface.



The advanced timer has the same basic structure as the general purpose timers. Three of the compare channels have complementary outputs. There is an additional break input and Hall sensor interface.

Each of the three complementary PWM channels has a programmable dead time which places a delay between the switch-off of a PWM output and the switching on of its complementary channel.



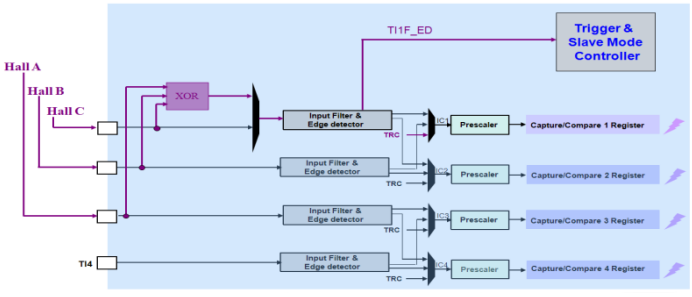
The advanced timer complementary PWM outputs have programmable dead time for motor control.

5.1.4.2.1 Break Function

The advanced timer can place its PWM outputs and their complementary outputs into a predefined configuration in response to a break input. This input can be from a dedicated external break pin or from the clock security system which monitors the external high speed oscillator. Once enabled, the break function operates entirely in hardware and guarantees to place the PWM outputs into a safe state if the STM32 system clock fails, or if there is a fault on the external hardware.

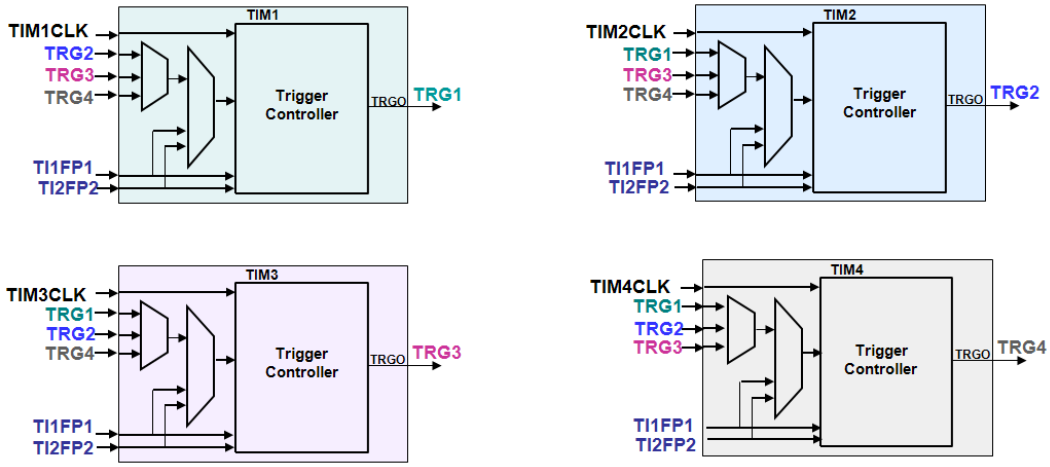
5.1.4.2.2 Hall Sensor Interface

Each of the timers including the advanced timer is designed to easily interface with a Hall Sensor to allow easy measurement of angular motor speed. The first three capture pins of each timer can be connected to channel 1 through an XOR gate. As the motor rotates and passes each sensor, a capture event will be generated on channel one. This captures the current timer count into the channel one capture register and also causes a reset on the timer. Thus the count value in the capture register can be related back to the motor speed.



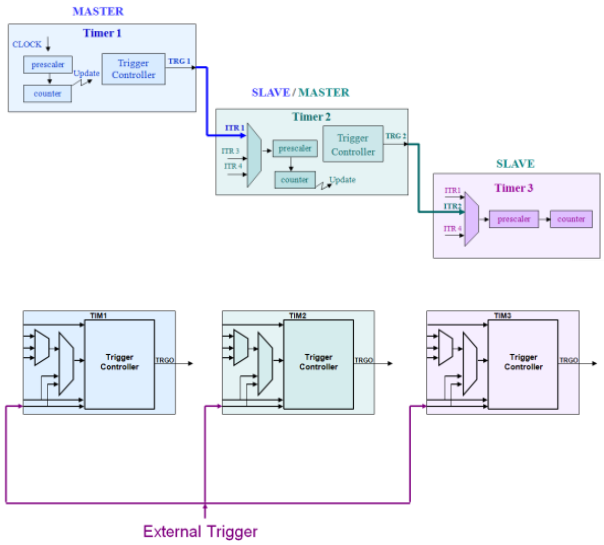
5.1.4.3 Timer Synchronisation

Although each of the timer units is a completely independent timer, any or all of the timer units can be synchronised together. This allows complex timer arrays to be designed in hardware, reducing the amount of software overhead required to perform a complex time-based function.



Each of the timers has trigger inputs from the other three timers as well as external inputs from the cap/com pins.

Each of the timer units has a trigger output which is routed as an input to the other three timers. Additionally, a capture input pin from timer 1 and timer 2 (TI1FP1 and TI2FP2) is routed to the trigger controller of each timer block. The timers may be synchronised in several different modes. The examples below show a couple of typical configurations.



All of the timers may be cascaded in a highly configurable fashion to build complex timer arrays.

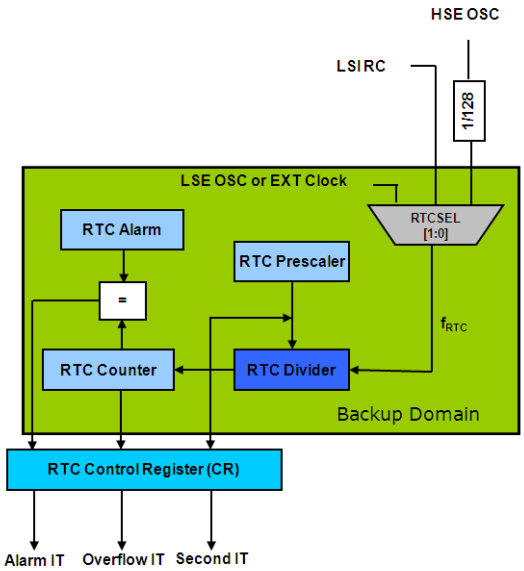
One timer acts as a master with two slaves. The master can provide a clock to the two slave timers, creating one large timer. Alternatively, it can provide a time delay that is used to trigger or gate the two slaves. Similarly, an external trigger can be used to gate the activity of each timer.

5.1.5 RTC And Backup Registers

The STM32 contains two power domains: the main STM32 system and peripheral power domain and the backup domain. Located within the backup domain are ten 16-bit wide registers, the RTC and the independent watchdog. The backup registers are simply ten memory locations that can be used to hold critical data values during the STM32 standby mode when the main power domain is switched off. In the low power modes both the RTC and

the independent watchdog can be kept running and may be used to wake up the main STM32 system, or to perform a chip reset.

The STM32 contains a basic real-time clock. This is a 32- bit counter that is optimised to increment each second if clocked from a 32.768 KHz clock source. When configuring the clock tree, the RTC oscillator can be selected from: the low speed internal oscillator, the low speed external oscillator, or the high speed external oscillator via a fixed divide by 128 prescaler. A further RTC prescaler allows you to get an accurate seconds count. The RTC counter itself can generate three interrupts: a seconds increment, a counter overflow and an alarm interrupt. The alarm interrupt occurs when the RTC counter reaches the value stored in a matching alarm register.



The real-time clock may use the internal or external low speed oscillator. It provides a seconds counter with an alarm register.

RTC is located in the backup power domain which is powered by the V_{BAT} voltage supply and the alarm interrupt is also connected to EXTI line 17. This means that when the main power domain of the STM32 is placed in a low power mode, the RTC will keep running. Through the EXTI it can generate an interrupt of events on the Cortex NVIC to wake up the main STM32 power domain. This configuration of the RTC is crucial for low power designs that need to spend most of their time in stop mode, but need some method of auto wakeup.

5.1.6 Backup Registers And Tamper Pin

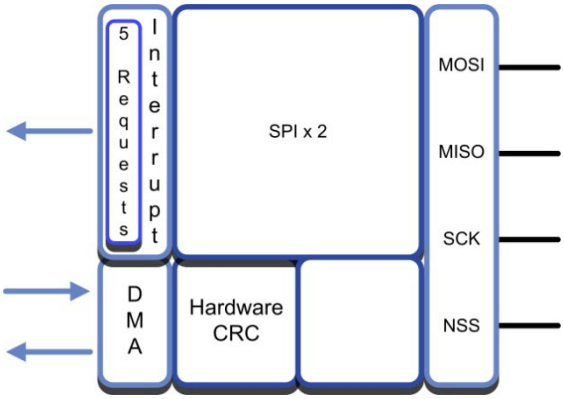
The backup power domain also contains ten 16-bit registers which act as battery-backed SRAM. The data held in these registers can be cleared by writing to the RCC backup control register. An external tamper pin can also be enabled in the same register. This pin can be configured to be high or low at startup. During normal operation a change in logic level will trigger a tamper detect event, which will clear the backup registers. A tamper interrupt can also be enabled, which allows the application software to take defensive action if a tamper condition is detected.

5.2 Connectivity

As well as having an excellent set of general purpose peripherals, the STM32 has five different types of communication peripheral. For communication between integrated circuits on the same PCB, the STM32 has SPI and I2C interfaces. For communication between different modules, there is a CAN bus interface and for communication to a PC, there is a USB device interface. Finally, there is the ever popular USART.

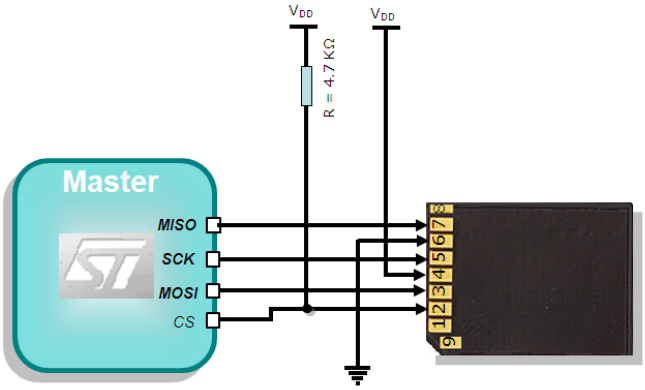
5.2.1 SPI

For fast communication between integrated circuits, the STM32 provides two SPI peripherals which can provide full duplex communication at rates up to 18 MHz. It is important to note that one SPI peripheral is located on the APB2 high speed peripheral bus, which can run at speeds up to 72 MHz. The second is on the low speed APB1, which can run at speeds up to 37 MHz. Each SPI peripheral has programmable clock polarity and phase and the data can be transmitted as 8 or 16-bit words, MSB first or LSB first. This allows each SPI peripheral to be configured as a master or slave, which can communicate with any other SPI device available.



Each SPI peripheral can operate as master or slave up to 18 MHz. Two DMA channels are provided for efficient data transfer

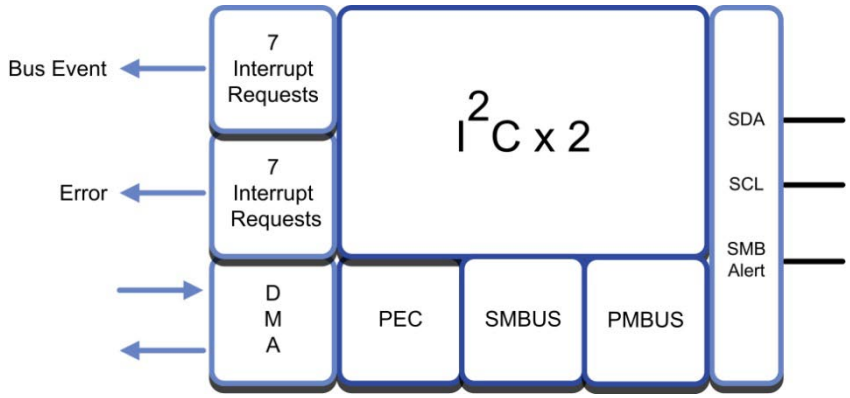
To support high data communication rates, each SPI peripheral has two DMA channels: one for transmitting data and one to read received data to memory. Using the DMA support allows high speed streaming of bi-directional data under hardware control. In addition to the standard SPI peripheral features, the STM32 SPI peripheral contains two hardware CRC units. One CRC unit is used for transmitted data and one for reception. Both units can generate and check CRC8 and CRC16 codes. This feature is particularly useful if you want to use either SPI peripheral as an interface to an MMC/SD card.



The SPI peripheral contains a hardware CRC unit which is designed to support interfacing with multimedia and SD memory cards.

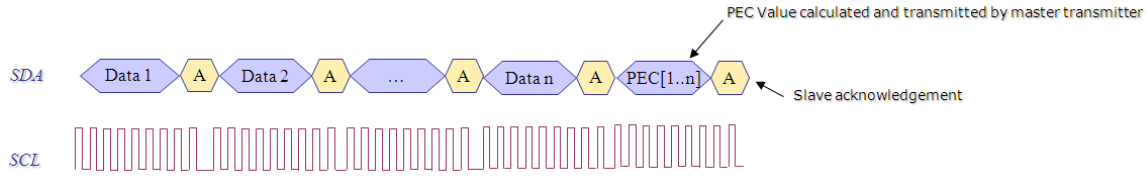
5.2.2 I2C

The STM32 can also communicate to other integrated circuits through a dedicated I2C interface. The I2C interface is capable of operating as a slave or bus master and is also capable of bus arbitration in a multi-master system. The I2C interface will support standard bus speeds of up to 100 kHz and fast speeds of up to 400 Khz. The peripheral also supports the I2C seven and ten bit addressing modes. The I2C peripheral is designed to simply read and write I2C data to and from the bus. Your software must control the I2C 'engine' to provide the protocol necessary for communication with various different bus devices. The I2C peripheral provides two interrupts to the Cortex processor: one for error containment and the other to control the communication address and data transmission. In addition, the DMA unit provides two DMA channels which can read and write data to the I2C transmit buffer. Thus, once the initial address and data transfer negotiation has been done, data can be streamed to and from the STM32 under hardware control.



The two I2C peripherals have enhanced support for the system management bus and the power management bus. They include hardware-packed error correction.

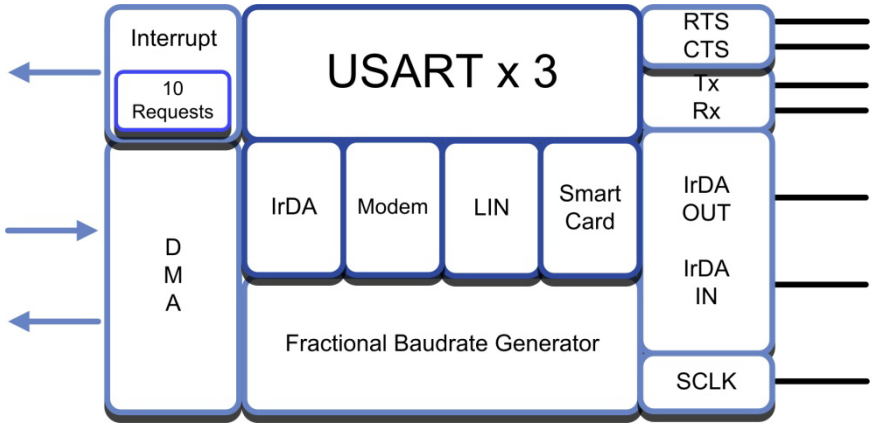
All of the above features make the STM32 I2C peripheral a fast and efficient bus interface. However there are some additional enhanced features that extend the basic I2C functionality. The STM32 I2 C peripheral contains hardware packet error checking (PEC). When enabled, the PEC will generate an 8-bit CRC error detection byte. This byte is automatically placed at the end of the transmitted data stream. The PEC will also error check received data against the PEC error protection byte.



The STM32 I2C peripheral is also designed to support two further communication protocols. These two protocols are System Management Bus (SMBus) and Power Management Bus (PMB). System Management Bus is a protocol defined by Intel in 1995 for use within PCs and servers. System Management Bus defines a data link layer which includes the use of the PEC and an additional networking layer standardising configuration communication between the PC BIOS and different manufacturers' devices. When operating in SMBus mode, the I2C peripheral has additional support for some SMBus features in addition to the PEC. These include support for the SMN address resolution protocol, host notify protocol and the SMBALERT signal. The Power Management Bus protocol is a version of System Management Bus designed for use within power conversion systems. PMBus is intended to allow configuration, programming and real-time monitoring of power systems.

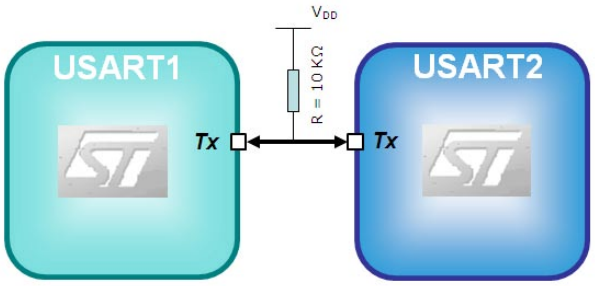
5.2.3 USART

Although serial communication ports have largely disappeared from PCs, they are still widely used in many embedded applications as a simple serial communication interface. Because of their utility and ease of use they will be with us for many years to come. The STM32 has up to three USARTs, each with several enhanced operating modes which support the latest serial communications applications. Each of the three USARTs are capable of up to 4.5 Mbps communication. Each USART has a fully programmable serial interface with programmable data size (8 or 9 bits), parity stop bit and baud rate. One USART is located on the APB2 bus which runs at up to 72 MHz, while the others are located on APB1 which runs at 36MHz.



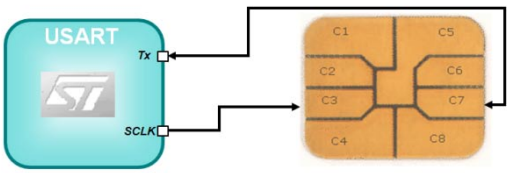
The USARTs are capable of supporting asynchronous communication with UARTS and modems as well as LIN IrDA and smart cards.

The baud rate generator on each USART is a fractional BAUD rate generator. This is more sophisticated than a simple clock divider and allows standard BAUD rates to be derived from any bus frequency. Like the other serial communications peripherals, each USART has two DMA channels which are used to transfer Tx and Rx data to and from memory. When used as a UART, the USART supports a number of special communication modes. The USART is capable of single wire half-duplex mode communication, using just the Tx pin. For modem communication and hardware flow control each USART has additional CTS and RTS control lines.

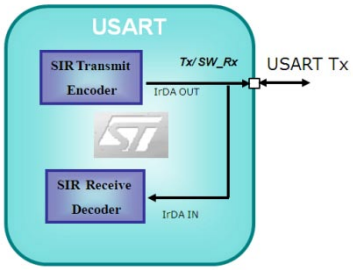


The USART's support single wire half-duplex communication.

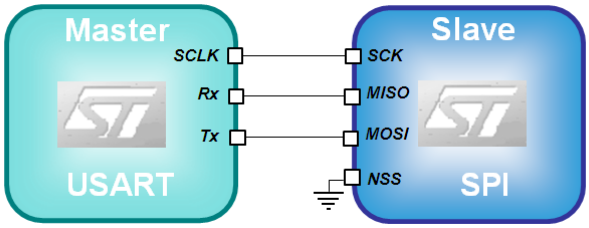
Each USART can also be used for local interconnect bus (LIN). This is an automotive standard for low cost networking to a cluster of microcontrollers. Each USART can be used as a serial infra red (SIR) encoder/decoder. This conforms to the IrDA standard for infra red communication for bit rate up to 115200bps, using half-duplex NRZ modulation with low power operation when the USART is clocked between 1.4MHz and 2.12MHz. Each USART has an additional smart card mode which conforms to the ISO 7618-3 standard.



The USARTS can support smartcard and IrDA communication.



In addition to high speed UART type operation, the USART can be configured for synchronous communication which allows a three wire connection to SPI peripherals. When in this mode, the USART acts as an SPI master and has programmable clock polarity and phase so it can communicate with any SPI slave.

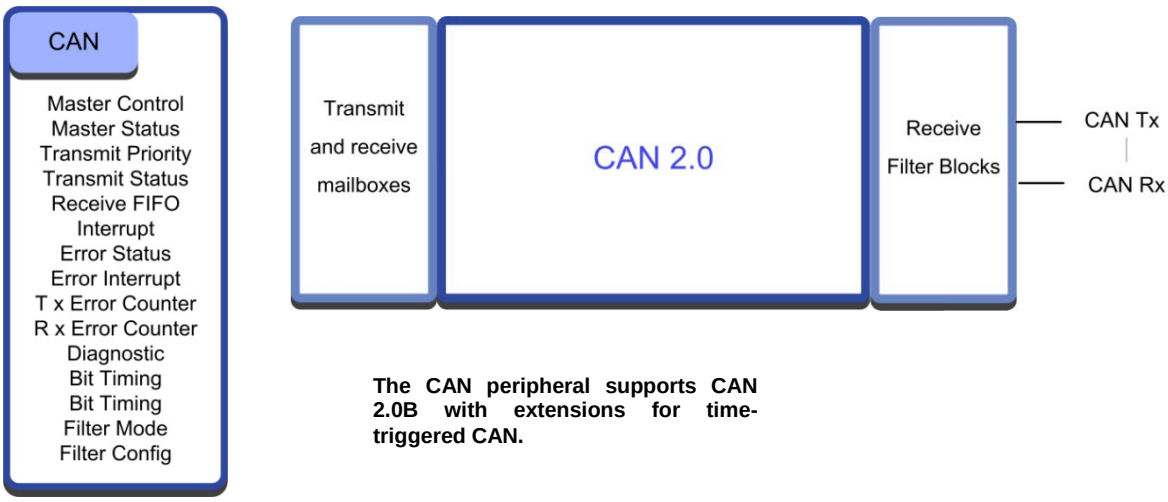


In synchronous operation the USARTS can be used as additional SPI masters.

The two remaining communication peripherals on the STM32 are the CAN controller and the USB full speed device interface. Both of these communication protocols are quite complicated. If you are new to either of these protocols you should read the CAN tutorial and/or the USB tutorial that comes with this book. Both of the USB and CAN peripherals require a relatively large amount of SRAM for message-filtering message buffers. The STM32 has a dedicated 512 byte region of SRAM that is shared between the CAN and USB peripherals. This memory can only be accessed by these peripherals. It is also assigned exclusively to the CAN peripheral or the USB peripheral. This means that you cannot use the CAN peripheral and the USB peripheral simultaneously, although it is possible to switch from one to the other in the same application.

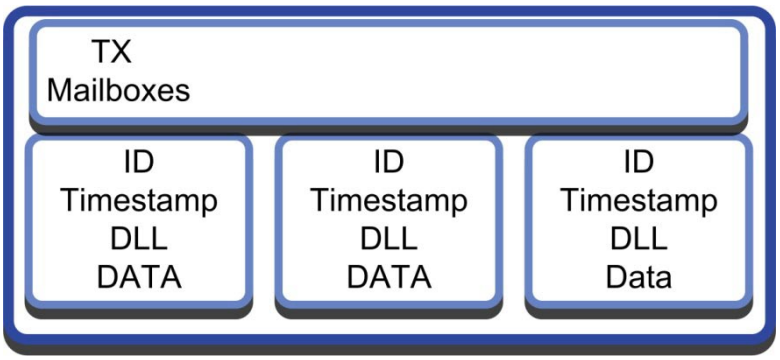
5.3.1 CAN Controller

The STM32 CAN controller is a fully-featured CAN node that supports CAB 2.0A and 2.0B active and passive with data rates up to the maximum 1 Mbit/s. The CAN controller also has extensions to support fully deterministic communication defined under the time-triggered CAN protocol TTCAN. When enabled, the TTCAN extensions support automatic message retransmission and will place a message timestamp in the last two data bytes of the CAN message packet. When enabled, these extensions allow the application software to use the CAN peripheral for hard real-time control.



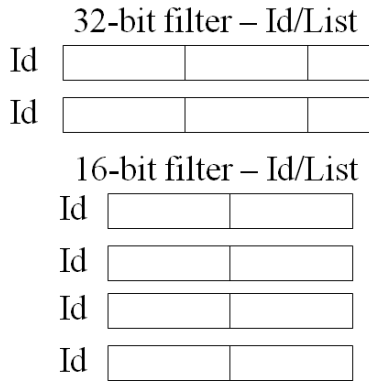
The CAN peripheral supports CAN 2.0B with extensions for time-triggered CAN.

The full name of the CAN controller is the bxCAN peripheral, where the bx stands for basic extended. A basic CAN peripheral is defined as having a single transmit and receive buffer, whereas an extended CAN peripheral has multiple transmit and receive buffers. The bx CAN peripheral is a hybrid of the two CAN peripheral architectures. The bxCAN peripheral has three transmit mailboxes and two receive mailboxes. Each of the receive mailboxes has a FIFO queue three messages deep. This design is a trade-off between having a low performance CAN module with a small silicon footprint and a high performance module that takes a large amount of the die area.



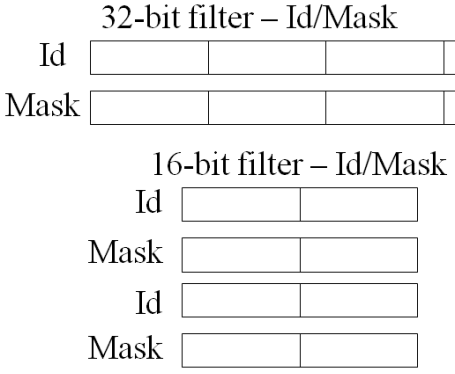
The CAN peripheral has three transmit mailboxes with automatic time stamping for TTCAN.

The next most important feature of a CAN controller is its receive message filtering. Because CAN is a broadcast network, every message transmitted is received by every node on the network. In a CAN network of any reasonable complexity there will be a large number of messages sent over the CAN bus. In such a network the CPU of a CAN node will spend all its runtime responding to CAN messages. To avoid this problem all CAN controllers have some form of message filtering that blocks unwanted messages from reaching the receive buffers. The CAN controller on the STM32 has 14 filter banks which can be used to block all CAN messages except selected message identifiers or groups of message identifiers.



The 14 message filters have two configurations that can be used to filter individual messages.

Each filter bank consists of two 32-bit registers. Each filter bank can be configured in one of four modes. The basic method programs each register of the filter bank with a message ID. When a message arrives, it must match this ID or be rejected. This mode has two configurations. In the first, the filter bank registers are used 3 bits wide and are able to filter the 11-bit and 29-bit message ID fields as well as the RTR and IDE bits in 16-bit mode.



The same filter banks can be used to filter groups of messages.

In the second configuration, the first 32-bit register is written with the message ID and the second register is used as a message mask. The mask register marks bits in the ID register as ‘care’ or ‘don’t care’. This allows a group of messages to be received through a single filter bank. When a message is received through the receive filters, a filter match index is stored with the message in the receive FIFO. This provides the application software with a shorthand method of determining the message data without having to read and decode the message packet ID.

All CAN controllers have two operating modes: a normal mode for receiving and transmitting message packets and an initialising mode for setting the communication parameters. The STM32 has a low power sleep mode. In sleep mode the clock to the bxCAN module is halted, but the mailbox registers may still be accessed. The bxCAN module will wake up when it detects activity on the CAN bus; it may also be reactivated by the application software. There are two additional sub modes when it is operating in normal mode. The first is silent mode, where the CAN controller may receive messages, but cannot transmit and does not generate error frames or message acknowledge bits. This mode is intended for passive monitoring of a CAN network Secondly, there is a loopback mode where all transmitted messages are looped back into the receive buffer. This mode is intended for self-testing and is also useful during code development. Both modes may be combined and this is ideal for self-testing when connected to a running network.

5.3.2 USB

The USB interface on the STM32 is a full speed (12Mb/sec) device interface, which can be controlled by a USB host such as a PC. The USB peripheral is a complete USB layer 1 and 2 protocol interface, which implements the USB physical layer interface and the data transfer layer with all packet error checking and retransmission. The USB device interface also supports the USB suspend and resume operations for low power operation. Developing a USB-based application does require a good knowledge of the USB specification and its application classes. A full USB developer's kit is available from the ST website. This provides a software stack to initialise the USB interface and has support for commonly used USB classes such as Human Interface Device (HID), mass storage, audio and legacy communications port. Using this stack, or a similar third party software stack, greatly speeds up development rather than reinventing the wheel.

The USB peripheral supports USB 2.0 device communications.

The USB interface supports up to eight endpoints, which are user configurable as endpoints for control, interrupt, bulk or isochronous pipes. The endpoint packet buffers are stored in the 512bytes of SRAM which is shared with the CAN controller. When the device is initialised, the application software divides this SRAM into a series of buffers.

The SRAM is configured into the endpoint buffers by a buffer description table held at the base of the SRAM. Here each endpoint is provided with a start address in the SRAM and a count to indicate its size. Each active control, interrupt and bulk endpoint is allocated an endpoint packet buffer, while isochronous endpoints are allocated a double buffer. This allows data to be received into one buffer while data in the second buffer is processed. When the next packet is received, the new data goes into the second buffer while the first is processed. This double buffer approach supports the streaming of real-time data such as audio.

The 512bytes of SRAM shared with the Can controller is used to store the USB packet data. During initialisation this region of memory is divided into individual buffers for each of the active endpoints. The endpoints used by isochronous pipes have a special double buffer so that data can be received into one buffer while an earlier packet is being read from a second

6. Low Power Operation

As well as being a high performance microcontroller, the STM32 has several low power modes in addition to its normal RUN mode. When used judiciously, the SLEEP, STOP and STANDBY low power modes make powering applications from batteries a practical prospect. The STM32 squares the circle by being a low power microcontroller with a high performance processor. In the Cortex overview we saw how the Cortex processor can enter a low power mode in which the CPU and Cortex peripherals are halted and consume minimal power. When the Cortex Processor enters a low power mode, it can export a SLEEPDEEP signal to the surrounding microcontroller, signalling it to enter a low power mode. All of the low power modes are entered by the Cortex CPU executing a WFI or WFE instruction. The low power mode that the STM32 then enters depends on the setting in the power control registers. In the next section we will have a look at each of the power modes in turn and look at comparison of their power consumption and wake up times.

6.1 RUN Mode

RUN mode is when the STM32 is executing program instructions and is at its highest level of power consumption. This section looks at various ways to reduce overall power consumption during program execution. It is important to remember that all of these features can be used dynamically as the code runs. This means that it is possible to run code in a low power, low performance configuration and then switch to high power, high performance configuration in response to an interrupt or program event.

During normal operation the Cortex processor and most of the STM32 can run at 72 MHz. When it is running at full speed, the STM32 consumes in excess of 30mA. The power consumption of the STM32 can be reduced by first gating the clocks of any unused peripherals. This stops any unused areas of the chip from consuming power. The peripheral clocks can be switched on and off dynamically through the Reset clock control module. Additionally, big power savings can be made by slowing down the system clock. If high speed operation is not a necessity, the PLL can be switched off and the STM32 can be clocked directly from the HSE oscillator. Further power reduction can be achieved by switching off the HSE oscillator and using the HSI oscillator. This has the disadvantage that the HSI oscillator is not as accurate a clock source as the HSE oscillator. Similarly, if the windowed watchdog and the real-time clock are not being used, the LSI oscillator can be switched off in order to shave off a bit more power consumption.

6.1.1 Prefetch Buffer And Half-Cycle Mode

If you are running directly from the HSE oscillator at a maximum of 8MHz, you can also disable the FLASH pre fetch buffer and enable the half-cycle operation. This incurs extra wait states, but reduces the RUN mode power consumption.

APB1	APB2	Peripheral	Frequency	Prefetch	Half Cycle	WFI	Oscillator	Typical consumption at 25 °C in mA
DIV4	DIV2	ALL_ON	72 MHz	ON	OFF	OFF	HSE	33.15
DIV 8	DIV 8	ALL_ON	72 MHz	ON	OFF	OFF	HSE	27.75
DIV 8	DIV 8	USART	72 MHz	ON	OFF	OFF	HSE	23.65
DIV4	DIV2	USART	8 MHz	ON	OFF	OFF	HSE	8.65
DIV4	DIV2	USART	8 MHz	OFF	OFF	OFF	HSE	8.48
DIV4	DIV2	USART	8 MHz	OFF	OFF	ON	HSE	1.68
DIV4	DIV2	USART	8 MHz	OFF	OFF	ON	HSI	0.9

Full speed power consumption is around 34mA, but at 8 MHz (9.6 DMIPS) the power consumption is below 1mA.

6.2 Low Power Modes

Careful configuration of the STM32 RUN mode can reduce power consumption to around 8.5mA. In order to get a true low power application we have to make use of the STM32 low power modes.

6.2.1 SLEEP

The first level of low power operation is the SLEEP mode. By default, when an WFE or WFI instruction is executed the Cortex processor will halt its internal clocks and stop executing the application code. In SLEEP mode the remainder of the STM32 will continue to operate. The STM32 will leave SLEEP mode when a peripheral generates an interrupt. When the STM32 enters SLEEP mode with all peripherals enabled and it is running at 72MHz from the HSE through the PLL, its SLEEP mode power consumption will be around 14.4mA. However, if the STM32 is prepared for low power operation by: firstly disabling all peripheral clocks (except for the peripheral used to wake up the Cortex processor) and secondly switching to the HIS oscillator (which can be further divided down to 1MHz or below) we can get power consumption figures of around 0.5mA.

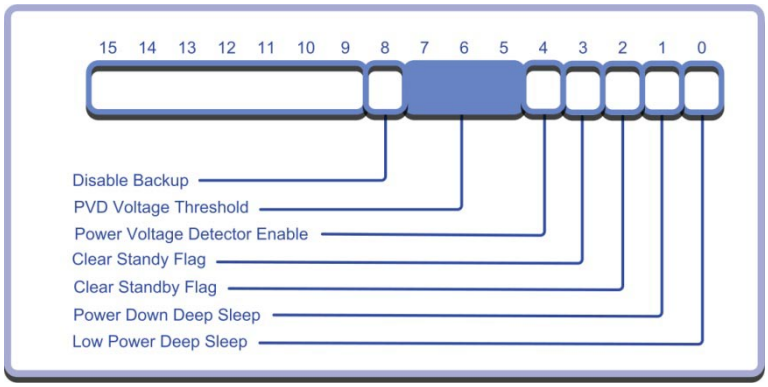
Conditions	f_{HCLK}	All APB peripherals enabled	All peripherals disabled	Unit
Running on HSE, AHB prescaler used to reduce the frequency	72 MHz	14.4	5.5	mA
	48 MHz	9.9	3.9	
	36 MHz	7.6	3.1	
	24 MHz	5.3	2.3	
	16 MHz	3.8	1.8	
	8 MHz	2.1	1.2	
	4 MHz	1.6	1.1	
	2 MHz	1.3	1	
	1 MHz	1.11	0.98	
	500 kHz	1.04	0.96	
	125 kHz	0.98	0.95	
Running on high speed internal RC (HSI), AHB prescaler used to reduce the frequency	64 MHz	12.3	4.4	
	48 MHZ	9.3	3.3	
	36 MHz	7	2.5	
	24 MHz	4.8	1.8	
	16 MHz	3.2	1.2	
	8 MHz	1.6	0.6	
	4 MHz	1	0.5	
	2 MHz	0.72	0.47	
	1 MHz	0.56	0.44	
	500 kHz	0.49	0.42	
	125 kHz	0.43	0.41	

SLEEP mode power consumption can be as low as 0.14mA.

In low power applications you should try to enter SLEEP mode as often as possible, in order to consume minimum power. The next issue is how long it takes for the STM32 to exit its low power mode and resume processing. The figures below show the wake up time for the Cortex CPU to resume processing using the HIS RC clock.

6.2.2 STOP Mode

The STM32 can be configured to enter the low power STOP Mode by setting the SLEEPDEEP bit in the Cortex power control register and clearing the Power Down Deep Sleep (PDDS) bit in the STM32 power control register.



When configured for STOP mode, execution of a WFI or WFE instruction will halt the Cortex processor and switch off the HIS and HSE oscillators. The FLASH, SRAM and peripherals are still powered, so the state of the STM32 is preserved. Like SLEEP mode, STOP mode can be left via an STM32 peripheral generating an interrupt. However, in STOP mode all the peripheral clocks are halted, with the exception of the External Interrupt peripheral. The use of the EXTI peripheral allows the STM32 to exit STOP mode when there is a state change on any GPIO pin. In addition, the EXTI has a line which can both request an interrupt and generate an interrupt from a real-time clock Alarm event. As the real-time clock has its own dedicated oscillator (either the LSI or LSE oscillator) it can provide a periodic interrupt to wake up the STM32 from STOP mode.

Once the STM32 has entered STOP mode, its power consumption drops mA in RUN mode to around 24 uA. Further power savings can be made by placing the internal regulator in a special low power mode when it enters STOP mode. The low power mode for the voltage regulator is selected by setting the LPDS bit in the STM32 power control register. With this bit set when the STM32 enters STOP mode, its power consumption will drop to 14uA. If the RTC is being used, a further 1.4 uA will be consumed.

Conditions	V_{DD}/V_{BAT} = 2.4 V	V_{DD}/V_{BAT} = 3.3 V	Unit
Regulator in Run mode, low-speed and high speed internal RC oscillators and high-speed oscillator OFF (no independent watchdog)	NA	24	µA
Regulator in Low Power mode, low speed and high speed internal RC oscillators and high speed oscillator OFF (no independant watchdog)	NA	14	

Symbol	Parameter	Conditions	Type	Unit
t_{WUSTOP}	Wakeup from Stop mode (regulator in run mode)	Wakeup on HSI RC clock	3.52	µs
	Wakeup from Stop mode (regulator in run mode + WFI)		5.42	
	Wakeup from Stop mode (regulator in Low power mode + WFE)		5.32	
	Wakeup from Stop mode (regulator in Low power mode + WFI)		7.21	

The wake up times you can expect in STOP mode are a worst case of 5.5 usec with the voltage regulator fully on and 7.3 usec with the regulator in its low power mode.

6.3 Standby

The STM32 can be configured to enter its standby mode by setting the SLEEPDEEP bit in the Cortex power control register and setting the Power Down Deep Sleep bit in the STM32. Now, when the WFI or WFE instructions are executed, the STM32 will drop into its lowest power mode. In Standby mode the STM32 is really switched off. The internal voltage regulator is switched off and the HSE and HIS oscillators are off. In this mode the STM32 consumes a mere 2uA.

Conditions	$V_{DD/}V_{BAT}$ = 2.4 V	$V_{DD/}V_{BAT}$ = 3.3 V	Unit
Low-speed internal oscillator and independent watchdog OFF, low speed oscillator and RTC OFF	NA	2	µA
Low-speed oscillator and RTC ON	1.08	1.4	

Symbol	Parameter	Conditions	Type	Unit
$t_{WUSTDBY}$	Wakeup from Standby mode	Wakeup on HSI RC clock	50	µs

In Standby mode power consumption is 2uA with a wakeup time of 50uS.

You can exit Standby mode by using an RTC alarm event in the same way as STOP mode. Additionally, you can use an external STM32 reset or a reset from the independent watchdog. Standby mode can also be exited by a rising edge on pin 0 of PortA. This pin must be configured as wakeup pin WKUP, by setting the EWUP bit in the power control and status register. As the lowest power mode, Standby mode takes the longest to leave and it will take around 50usec before the Cortex CPU will restart processing instructions. Once in Standby mode, all data in the SRAM, Cortex and STM32 registers is lost. An exit from Standby mode is effectively the same as a program reset.

6.4 Backup Region Power Consumption

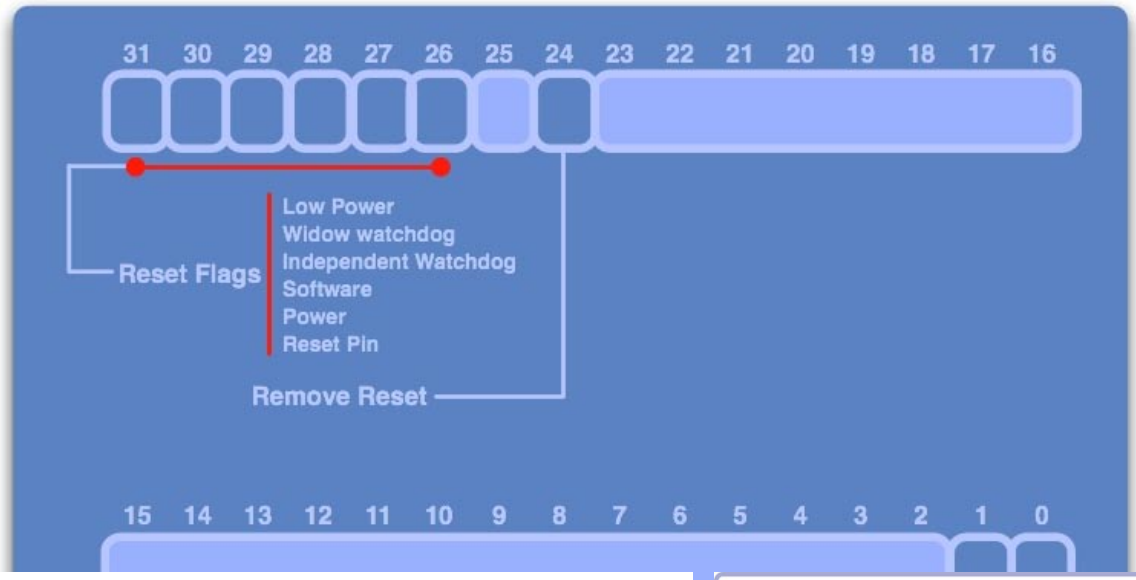
The backup region containing the battery backed RAM and the RTC will be kept alive during all power down modes. This power domain will consume around 1.4 uA at 3.3V.

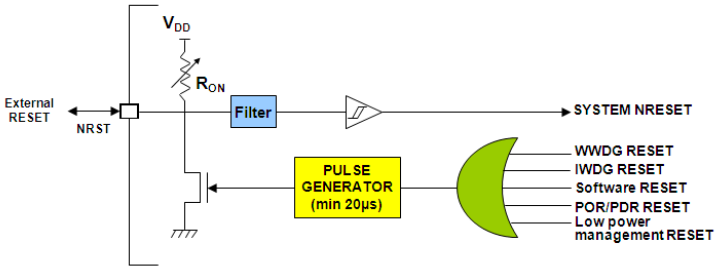
6.5 Debug Support

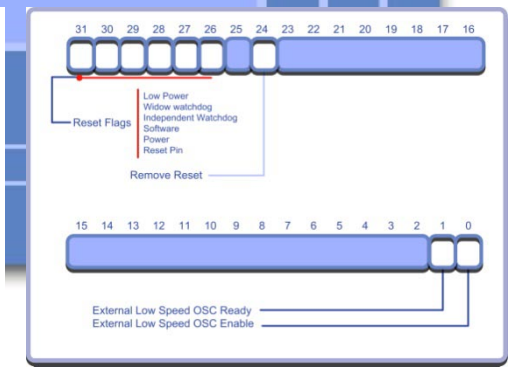
On traditional microcontroller systems, debugging an application which uses low power modes can be extremely painful. As soon as the microcontroller enters low power mode it stops responding to the debugger, which then throws an error or ceases to work. Within the STM32 it is possible to configure the low power modes to keep the HSI oscillator running in each of the low power modes, providing a dedicated clock path to the CoreSight debug architecture. This means that you can fully debug low power applications without having to remove entries into low power mode. This eliminates debug timeout problems. The STM32 enhanced debug features are configured with the DBG_MCU register.

7. Safety Features

The STM32 has also been designed with a number of inherent features that will detect incorrect operation of the application code or the STM32 itself. To ensure that there is a reliable power supply, the STM32 has its own internal reset that will place the chip in reset if the supply voltage is below minimum V_{DD}. Additionally there is a programmable power voltage detect circuit that can be used to detect power failure early. It will then generate an interrupt to place the chip into a safe state. The clock tree also includes a clock security system which monitors



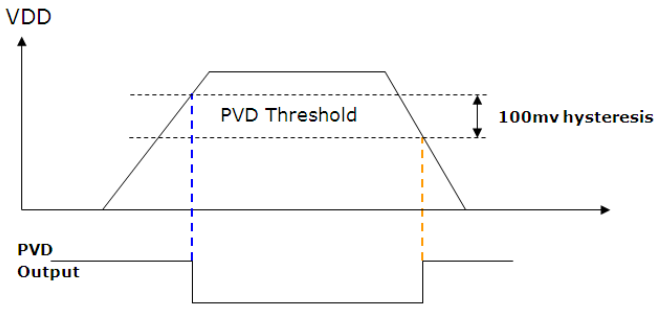




The STM32 may be reset from a number of sources. After reset the RCC control and status register reports the last reset source.

7.2 Power Voltage Detect

As part of the internal power supply supervisor the STM32 contains a power monitoring unit called the Power Voltage Detect (PVD). The PVD has a programmable threshold that can be set in the in steps of 0.1V from 2.2V to 2.9V. This threshold is configured in the power control register.

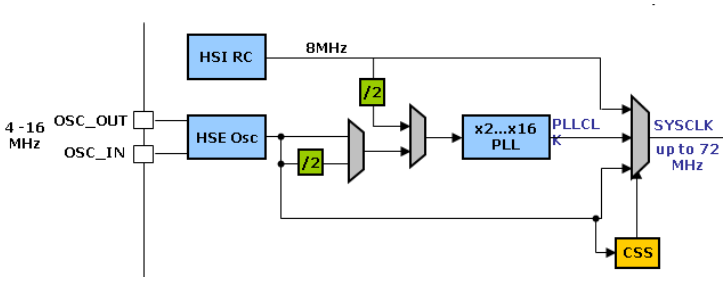


The power supply is monitored by the power voltage detect unit, which can generate an interrupt if the supply dips below a configured threshold.

The output of the PVD is connected to line 16 of the external interrupt unit. Since the EXTI lines can be configured to generate an interrupt on a falling or rising edge, or both, the PVD unit can be used to generate an interrupt for both under and over voltage conditions.

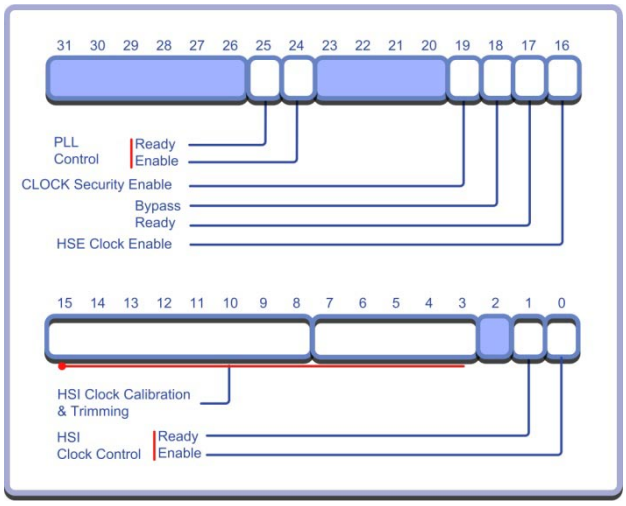
7.3 Clock Security System

In most STM32 applications the main system clock used for the Cortex processor and the STM32 peripherals will be derived from an external crystal connected to the HSE pins. This clock tree contains a Clock Security System that monitors the external crystal. If this crystal fails, it will cause the STM32 clock system to fail back to the internal 8 MHz oscillator.



The Clock Security System generates an interrupt if the external oscillator fails and switches to the internal RC oscillator.

The Clock Security System is enabled by setting the Clock Security Enable bit in the RCC control register.

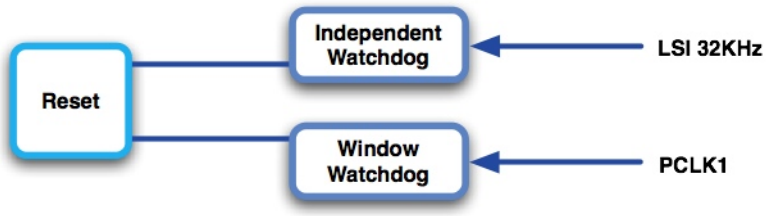


The Clock Security System is enabled by setting the CSS enable bit in the RCC control register.

The CSS has an interrupt line which is connected to the break interrupt of advanced timer 1, which in turn is connected to the Cortex NVIC non-maskable interrupt line. This ensures that if the main oscillator fails, the PWM outputs of the advanced timer will immediately be placed in a pre-programmed safe state by hardware control. This ensures that any hardware driven by the advanced timer PWM outputs will not be allowed to run while not under control of the Cortex processor. It is particularly important for motor control applications.

7.4 Watchdogs

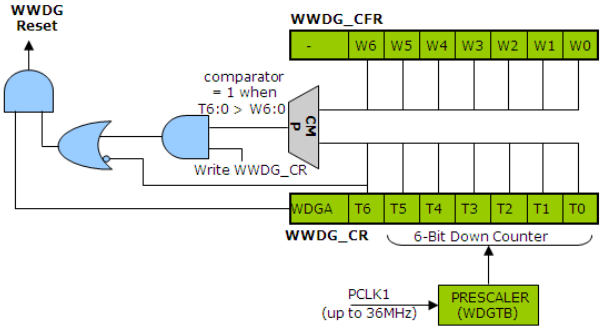
The STM32 contains two completely separate watchdogs. The independent watchdog is completely separate from the main STM32 system. It is located within the backup power domain and derives its clock from the internal Low Speed Oscillator (LSI). The windowed watchdog is part of the main STM32 system and is clocked via the peripheral bus 1 clock. Both watchdogs must be individually enabled and can be used simultaneously.

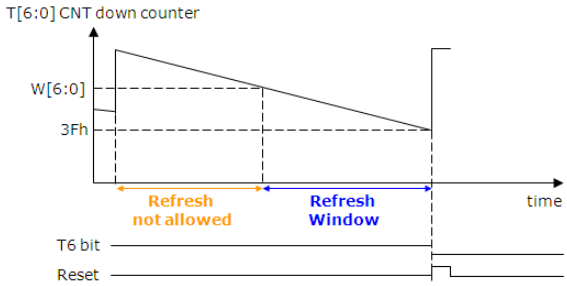


The STM32 has two internal watchdogs one of which has its own separate oscillator.

7.4.1 Windowed Watchdog

ersion of a traditional on-chip watchdog. Once enabled, the
t on a transition from 0x40 to 0x3F i.e. when bit T6 is cleared.
watchdog configuration register. This provides an upper count
chdog count while the actual watchdog counter value is greater
enerated. The windowed watchdog provides a programmable
ne watchdog can be written to. This allows you to build extra
within its expected parameters.





The windowed watchdog is a six-bit down counter, which is clocked from PCLK1 via a 12 bit prescaler that divides PCLK1 down by 4096. The prescaler has a further 4 bits that are user programmable allowing a further divide by 1,2,4 or 8. The prescaler bits are contained in bits 6,7 of the control register.

Hence the timeout period of the windowed watchdog is given by:

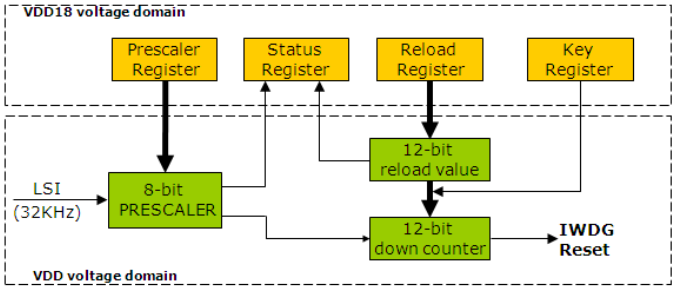
Twwdg = Tpclk1x4096x2POW (WDGTB) x (reload value+1)

With Pclk1 running at its maximum 36 MHz, the windowed watchdog minimum timeout period is 910uSec and the maximum period is 58.25mSec.

Once the windowed watchdog has been configured, it can be enabled by setting the watchdog activation bit in the control register. Once the windowed watchdog has been enabled by software it cannot be disabled, except by a reset.

7.4.2 Independent Watchdog

Although the independent watchdog is fabricated on the same silicon as the main STM32 system, it has its own oscillator separate from the main STM32 clock. The independent watchdog is also located within the V_{DD} voltage domain, which is kept alive in the STOP and STANDBY modes.



The independent watchdog is a 12-bit count down timer, which will force a reset on the STM32 when it underflows. It is clocked from the low speed internal oscillator via an 8 bit prescaler. The LSI oscillator has a nominal frequency of 32.768, but in practice this can vary between 30 KHz to 60 KHz. The independent watchdog is initialised by first setting the prescaler register, which divides down the LSI oscillator in powers of two between 4 to 256. The minimum timeout period for the independent watchdog is 0.mSec and the maximum period is just over 26 seconds. The timeout period value is programmed directly into the reload register.

The independent watchdog is a countdown watchdog with its own oscillator. It is also located in the backup domain so that it can remain active during Stop and Standby modes.

The option bytes in the FLASH memory small information block can be used to configure the independent watchdog to start after a reset, or by software command. If under software control, the independent watchdog can be started by writing 0xCCCC to the key register. The Independent watchdog will count down from an initial value of 0xFFF. The value 0xAAAA must be written to the key register to refresh the watchdog. This causes the reload value to be loaded into the down counter register, refreshing the count value.

Traditionally it is very difficult to debug small microcontrollers if the watchdog is enabled. As soon as the CPU is halted, the watchdog cannot be updated. It will timeout and force a reset, which destroys the debug session. Normally a watchdog has to be disabled so that it does not upset the debugger. Consequently it is very difficult to test and prove that watchdog refreshes are occurring at an optimum rate. Within the STM32 MCUDBG register it is possible to configure both the independent watchdog and the window watchdog to halt when the Cortex-M3 CPU is under control of the CoreSight debug system. This allows you to step through your code with both watchdogs enabled and they will be incremented in sync. with the number of cycles executed on the CPU.

7.5 Peripheral Features

The user peripherals have also been designed with a number of features that help to ensure the safe operation of the STM32. These are fully described in the relevant user peripheral section but will be reviewed here:

7.5.1 GPIO Port Locking

When the GPIO ports are initialised, each IO line will be configured as an input or output. Once configured, the STM32 GPIO port configuration can be locked. This prevents any further accidental changes to the port configuration. Each port may be locked on a bitwise basis.

7.5.2 Analog Watchdog

Each of the Analog to digital converters has two analog watchdogs. These watchdogs can be set to generate an interrupt on over-range or under-range voltages.

7.5.3 Break Input

For motor-based applications, the break line within the advanced timer can be used to place the three complementary PWM outputs into a predefined state, in response to an input on the break pin, or a failure in the main STM32 oscillator.

8. The FLASH Module

The on-chip FLASH memory of the STM32 is arranged in three main regions. First, there is the main FLASH memory designed to hold program instructions. This memory is 64 bits wide to provide efficient memory access with the prefetch buffer. For flash program and erase operations this memory is divided into 4K pages. This memory has a WRITE endurance of 10000 cycles and data retention of 30 years at 85 degrees C. Most microcontroller FLASH memory data retention is rated at 25 degrees C, so the STM32 has an exceptional FLASH memory. Aside from the main program memory, there are two smaller memory regions: the big information block and the small information block. The big information block is a further 2k of FLASH memory holding a factory programmed bootloader, which is designed to download code over USART 1. The small information block contains six configuration bytes, which are used to define the reset properties of the STM32 and its memory protection.

8.1 Internal FLASH Security And Programming

The internal FLASH memory can be updated by the internal bootloader, by a JTAG tool, or by in-application programming through a dedicated set of registers called the FLASH program and erase controller FPEC. The FPEC is also used to program the option bytes in the small information block.

The FPEC module is used to allow in-application programming of the FLASH memory. The FLASH memory can also be read-protected from debug tools and write-protected.

8.2 Erase And Write Operations

After reset, the FPEC registers are protected and must be unlocked by writing a special sequence to the key register. To unlock the FPEC you must write 0x45670123, followed by 0xCDEF89AB. If there is a mistake in this sequence, the FPEC will stay locked until the next reset. Once the FPEC has been unlocked, it is possible to erase and WRITE operations on the main FLASH memory. Within the main FLASH memory block it is possible to perform a mass erase or an erase of a selected 4k page. A mass erase is done by simply setting the mass erase and start bits in the control register. When the busy bit in the same register is reset, each location in the main FLASH memory will be reset to 0xFFFF. A page erase is equally easy to perform. First, you must program the start address of a FLASH page into the Address register, then set the page erase and start bits in the control register. Again, when the busy bit is clear, the page will be erased. New data can be written to a FLASH memory cell only after it has been erased. A WRITE operation is performed by setting the program bit in the control register and then performing a half-word write to the desired location. If the FLASH location is erased and not write protected, the FPEC will program the new value into the FLASH memory cell.

8.3 Option Bytes

The small information block contains eight user-configurable option bytes. Four of these bytes are used to define write protection on the main FLASH memory. The fifth is used to set read protection which prevents access to regions of memory when the chip is in debug mode. A sixth byte is used to configure low power and reset operation. The final two bytes are simple FLASH memory cells that are available for user-defined options. Before the Option bytes can be written to, the FPEC must be unlocked as described above. Then the Option bytes must be unlocked by writing the same two keys to the option key register. The Option bytes have a separate program

and erase procedure to the main FLASH memory. The small information block is erased by setting the OPTER bit in the control register and then the STRT bit. Once the BSY bit is reset, the small information block is erased. To program an Option byte, set the OPTPG bit in the FLASH control register and perform a half-word write to the Option byte. Each Option byte is stored as a half-word. The Option byte is stored in the lower byte of the half-word and its complemented value is stored in the upper half-word. You must write a correct value in the lower half-word and the FPEC will automatically calculate the complemented value.

8.3.1 Write Protection

When it is set, each bit in the write protection option bytes enables protection over a given FLASH page. Write protection can be disabled by an erase of the small information block.

8.3.2 Read Protection

When the read protection is set, all read accesses to the FLASH memory are disabled when the device enters debug mode. Access to the SRAM is still possible and code may be downloaded and executed in this region. So it is possible to disable the read protection by running a program out of SRAM. However, when read protection is disabled a mass erase of the internal FLASH is also performed, to ensure protection from software piracy. When read protection is enabled, the FLASH memory is also write protected to prevent a malicious program from being inserted into the memory region containing the vector table. The STM FLASH memory is protected if the read protection byte and its complement are set to 0xFF. The memory can be unprotected by writing 0xFA and its complement as a half-word to the read protection Option byte.

8.3.3 Configuration Byte

The configuration Option byte contains three active bits. Two of these bits govern how the STM32 enters Standby and Stop modes. Either mode can be configured to generate a reset on entry. This will configure the digital IO pins as inputs, reducing the overall power consumption of the STM32. The PLL and external oscillator will also be disabled and the chip will revert to using the internal high speed RC oscillator as the main system clock. The final bit in the configuration Option byte configures the activation of the independent watchdog. This watchdog has a hardware watchdog mode where it will start immediately after a processor reset, or software watchdog where it must be started under software control.

9. Development Tools

The adoption of ARM7 and ARM9 into standard microcontrollers has lead to an explosion in development tools' support for these CPUs. All of the major compiler developers such as GCC, Greenhills, Keil, IAR and Tasking provide ARM development tools. With the introduction of the Cortex processor, all of these development tools have been extended to support the Thumb-2 instruction set. If you are already using an ARM-based microcontroller, the chances are that it will already generate code for the STM32. The worst case scenario is that you will have to get an upgrade from your supplier.

If this is your first project with an ARM-based microcontroller, you will be able to select a toolset from your preferred manufacturer. While it is hard to find a bad toolset these days, two compilers are worth discussing further. Firstly the "GCC" or "GNU" compiler is an open source tool which can be downloaded and used for 'free'. The GCC compiler has been integrated into a number of commercial IDEs and debuggers to make low-cost development tools and evaluation kits. While the GCC compiler is a reliable and stable compiler, our experience has been that its code generation is not as efficient as the commercial compilers. There is also generally not a direct support route if you run into problems, which can slow down development. Of the commercial compilers, the ARM RealView compiler is the original and most refined C compiler and was developed by ARM for use with their CPUs. The RealView compiler is available as part of the ARM RealView toolset. This toolset is aimed at system-on-chip developers and is not really suitable for microcontroller projects. However since January 2006 the RealView compiler has been integrated into the Keil Microcontroller Development Kit (MDK-ARM). As its name implies, MDK-ARM is a complete tool chain designed exclusively for ARM-based microcontrollers. The MDK is easy to use (selecting about 4 options configures a whole project) and provides a tightly-integrated tool chain, which is controlled by one manufacturer.

If you are making a decision between using the GCC compiler and a commercial compiler, you will partly be driven by the project budget. A one off 'simple' project is unlikely to have the budget to justify a commercial toolset. However, if you plan to standardise on ARM-based microcontrollers, then an 'expensive' toolset will soon pay for itself both in reduced development time and a more compact final image. It is also important to bear in mind your relative level of experience. If you are a hard-core embedded developer, then you are likely to be able to develop a whole project with the GCC compiler. If, however, you are less experienced, or do some C coding, then it is possible to get into a huge mess.

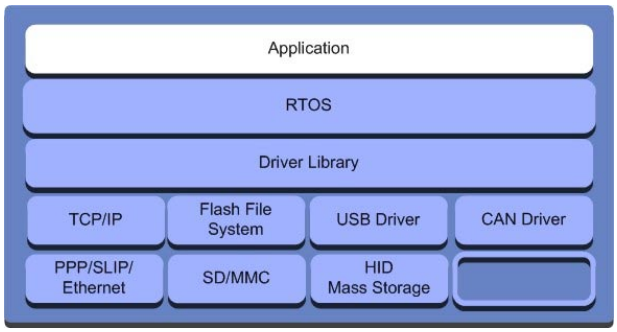
9.1.1 Evaluation Tools

Most compiler vendors will also provide an evaluation kit or starter kit. This is traditionally a hardware board and a cut down or time-limited version of their toolset. An up-to-date list of evaluation kits is available on the ST website. One of the best evaluation tools is the Hitex STM32 Performance Stick. Costing about 50 Eur, the Performance Stick is a complete evaluation tool for the STM32. It is designed as a USB dongle that allows you to develop and debug an unrestricted amount of code with the GCC or Tasking compilers, via the HiTOP IDE. In addition to the STM32, the Performance Stick hardware has a second microcontroller in the shape of the STR750. This microcontroller uses its ADC and timers to measure the STM32's power consumption and interrupt latency. This information is sent to a 'dashboard' application on the PC. The dashboard allows you to manually experiment with the different features of the STM32 and get some verification of the data sheet values for power consumption, wake up time etc.

The Hitex Performance Stick is a very low-cost evaluation tool for the STM32. It provides an unlimited development environment based on the HiTOP debugger and the GCC compiler. For full product development, the same IDE and compiler are available for the Hitex Tantino JTAG debugger.

9.1.2 Libraries And Protocol Stacks

To support rapid code development, ST have provided a STM32 firmware library as a free download from their website. The firmware library provides low- level driver functions for all of the on-chip peripherals. This gives you some basic building blocks on which to start building your application. The most complex peripheral on the current STM32 variants is the USB device controller. In order to help you build common USB class devices, ST also provide a free USB developer's kit. Like the firmware library, the USB developer's kit can be downloaded from the ST website. The USB developer's kit provides a USB library and demonstration applications for HID, Mass Storage, Audio and Device Field Upgrade.



With the increasing complexity of microcontroller peripherals, it is important to select a tool chain which is well supported with protocol and application software stacks.

As new variants of the STM32 are released, they will have more and more complex peripherals (Ethernet MAC, TFT interface etc). As this complexity increases, it becomes just about impossible to develop all the application code yourself. So when selecting development tools it is also important to consider the availability of protocol stacks, such as a TCP/IP stack and other application software, such as a GUI, which may be required on future projects. Ideally these should be from the same vendor and well integrated into your chosen toolset.

9.1.3 RTOS

If you are moving from an eight or sixteen bit microcontroller, the chances are that you are not currently using an RTOS. As we have seen, the Cortex-M3 provides you with significantly more processing power than comparably-priced microcontrollers and is designed to support a small footprint RTOS. Thus, if you have not been using an RTOS it is worth considering when you start work with the STM32. The use of an RTOS gives you the advantage of more abstract code development, enhanced code re-use, easier project management and enhanced debugging. The use of an RTOS also provides a structure to your code, which forces you to plan the application before you dive in and start writing. There are more RTOSes available for ARM and Cortex than for most embedded CPUs. Many compiler vendors will provide their own and have ports for third party RTOSes, but one of the most popular open source operating systems is "FreeRTOS", which is available from www.freertos.org. A commercial version of FreeRTOS is called "SafeRTOS", which has been tested to meet the IEC 61508 safety standard and is also available from the same site.

10. End Note

If you have read through this far, I think you will agree that the Cortex STM32 is a new generation of very low cost general purpose microcontrollers. Centred around a high performance processor with a deterministic interrupt system with sophisticated peripherals, the STM32 is suitable for many industrial and consumer applications. Additionally the low power modes make it suitable for battery-powered and hand held products.

11. Bibliography

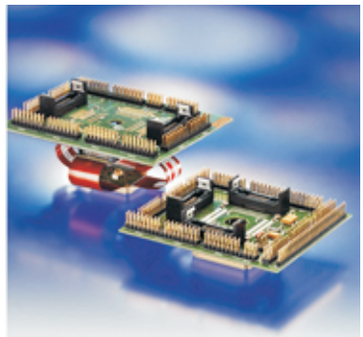
Cortex-M3 Technical reference manual	ARM Ltd
ARMv7-M architectural reference manual	ARM Ltd
ARM Architectural reference manual Thumb2 supplement	ARM Ltd
STM32F103xx User Manual	ST Microelectronics
STM32F10xxx FLASH Programming manual	ST Microelectronics

This book is intended as a hands-on guide for anyone intending to use the ST Microelectronics STM32 family of Cortex-M3 microcontrollers.

Over the last six or seven years one of the major trends in microcontroller design is the adoption of the ARM7 and ARM9 as the CPU for general purpose microcontrollers. Today there are some 240 ARM-based microcontrollers available from a wide range of manufacturers. Now ST Microelectronics have launched the STM32, their first microcontroller based on the new ARM Cortex-M3 microcontroller core. This device sets new standards of performance and cost, as well as being capable of low power operation and hard real-time control.

www.hitex.com

hitex
DEVELOPMENT TOOLS







ISBN 0-9549988-8-X
9 780954 998882

Hitex (UK) Ltd., Sir William Lyons Road, Science Park, Coventry, UK, CV4 7EZ.
Tel +44 (0) 2476 692066