The Cortex-M3/M4 Embedded Systems: The Cortex-M3/M4 Processor Basics

Refer to Chapter 1, 2, and 3 in the reference book
"The Definitive Guide the ARM Cortex-M3"
Refer to Chapter 1, 3, and 4 in the reference book
"The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

Processers in Embedded Systems

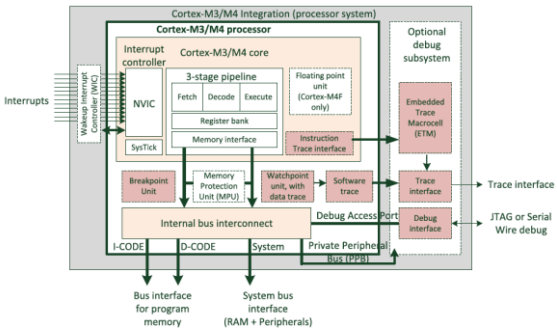
Embedded processors can be broken into two broad categories: ordinary **microprocessors** (μP) and **microcontrollers** (μC).

A fairly large number of basic CPU architectures are used.
- Von Neumann as well as Harvard architectures
- RISC as well as non-RISC

Word lengths vary from 4 bits to 64 bits and beyond (mainly in DSP processors).

Most architectures come in a large number of different variants and shapes.

Overview of Cortex-M3/M4 processor



Overview of Cortex-M3/M4 processor

32-bit microprocessor: 32-bit data path, 32-bit registers, 32-bit memory interfaces.

Harvard architecture: separate instruction bus and data bus, which allows instructions and data accesses to take place at the same time.

4GB Memory space

Registers: Registers (R0 to R15) and special registers.

Two operation modes: thread mode and handler mode

Two access levels: privileged level and user level.

Interrupts and Exceptions: a built-in *Nested Vectored Interrupt Controller*, supporting 11 system exceptions plus 240 external IRQs.

Overview of Cortex-M3/M4 processor

MPU: an optional Memory Protection Unit allows access rules to be set up for privileged access and user program access.

The Instruction Set: Thumb-2 instruction set allows 32-bit instructions and 16-bit instructions to be used together; no ARM instructions allowed

Fixed internal debugging components: provide debugging operation supports and features such as breakpoints, single step.

M4 has extra features: the DSP extensions and the optional single precision floating point unit

Features of Cortex-M3/M4 Processors

Greater performance efficiency, allowing more work to be done without increasing the frequency or power requirements

Low power consumption, enabling longer battery life

Enhanced determinism, guaranteeing that critical tasks and interrupts are serviced in a known number of cycles

Improved code density, ensuring that code fits in even the smallest memory

Ease of use, providing easier programmability and debugging

Lower-cost solutions, reducing 32-bit-based system costs at less than US$1 for the first time

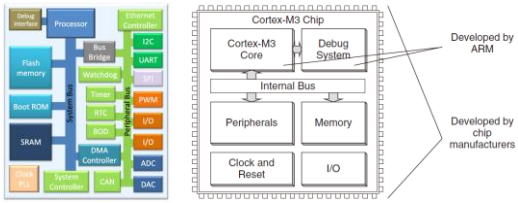
Wide choice of development tools, from low-cost or free compilers to full-featured development suites

Cortex-M3/M4 Processors vs MCUs

The Cortex-M3/M4 processor is the central processing unit (CPU) of a microcontroller chip

After chip manufacturers license the Cortex-M3/M4 processor, they can put the Cortex-M3/M4 processor in their silicon designs, adding memory, peripherals, input/output (I/O), and other features.

Cortex-M3/M4 processor based chips from different manufacturers will have different memory sizes, types, peripherals, and features

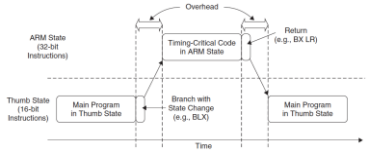


ARM Processors: Instruction State Switches

In the ARM state, the instructions are 32-bit and can execute all supported instructions with very high performance

In the Thumb state, the instructions are 16-bit, so there is a much higher instruction code density

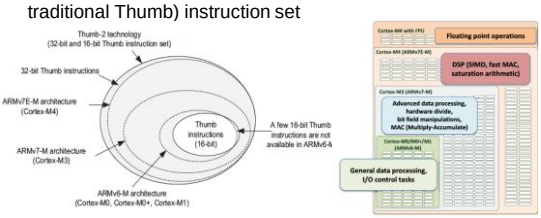
Can we combine them to achieve the best of both worlds?



Any cost?
1) overhead of state switches
2) complex software development

Instruction Set of Cortex-M3/M4

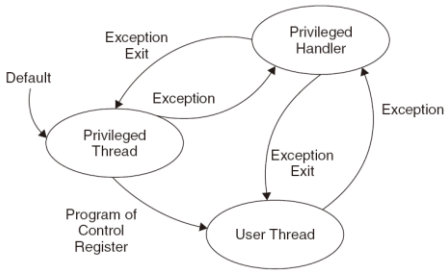
Cortex-M3/M4 supports only the **Thumb-2** (including the traditional Thumb) instruction set



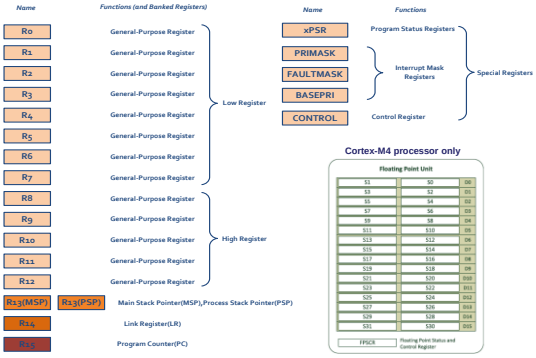
Advantages:
1) No state switching overhead, saving both execution time and instruction space
2) No need to separate ARM code and Thumb code source files, making software development and maintenance easier

Operation Modes in Cortex-M3/M4

Two modes and two privilege levels



Cortex-M3/M4 Basics: Registers



Cortex-M3 Basics: Registers

General-Purpose Registers

R0~R7 (low registers)

Can be accessed by all 16-bit Thumb instructions and all 32-bit Thumb-2 instructions

reset value is unpredictable

R8~R12 (high registers)

Accessible by all Thumb-2 instructions but not by all 16-bit Thumb instructions

reset value is unpredictable

Stack Pointer R13

Two stack pointers are banked so that only one is visible at a time.

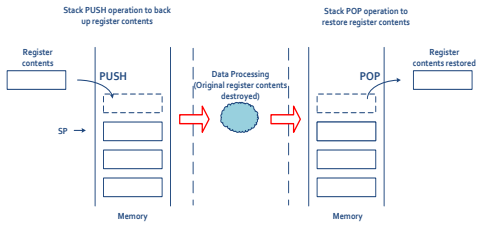
Main Stack Pointer (**MSP**) : This is the default stack pointer, used by the OS kernel, exception handlers, and privileged-mode programs

Process Stack Pointer (**PSP**) : Used by the user application code

Stack and Its Pointer

Stack: a first-in last-out buffer

- The stack pointers are used for accessing stack memory with **PUSH** and **POP**
- For example, to store register contents to stack memory at the start of a subroutine and then restore the registers from stack at the end of the subroutine.



Stack in Cortex-M3

Cortex-M3 uses a **full-descending** stack arrangement

The stack pointer decrements when new data is pushed in the stack

The assembly language syntax is as follows:

```
PUSH {R0}       ; R13=R13-4, then Memory[R13]=R0
POP  {R0}       ; R0=Memory[R13], then R13=R13+4
```

Either **R13** or **SP** can be used in your program codes (referring to the current stack pointer you are using, either MSP or PSP)

A particular stack pointer (**MSP**/**PSP**) can be accessed using special register access instructions (**MRS**/**MSR**)

Since PUSH and POP operations are always word aligned, the stack pointer R13 bit 0 and bit 1 are hardwired to zero and always read as zero (RAZ)

Cortex-M3: Stack Memory Operations

Multiple registers can be pushed and popped in one instruction, using comma to separate

```
PUSH {reglist}    ;push the largest numbered register first
POP {reglist}     ; pop the lowest numbered registers first
```

If a POP instruction includes PC in its reglist, a branch to this location is performed when the POP instruction has completed.

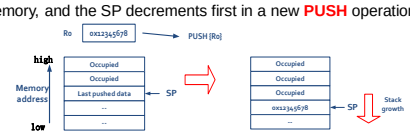
Note that Bit[0] of the value read for the PC must be 1 (used to update the APSR T-bit)

Example:

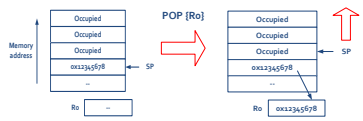
```
PUSH   {R0-R7, LR}   ; Save registers
…
POP    {R0-R7, PC}   ; Restore registers
```

Cortex-M3: Stack Implementation

The stack pointer (SP) points to the last data pushed to the stack memory, and the SP decrements first in a new **PUSH** operation.



For **POP** operations, the data is read from the memory location pointed by SP, then the stack pointer is incremented. The contents in the memory location are unchanged.



Cortex-M3 Basics: Registers

Link Register R14

R14 is the link register (LR), used to store the return program counter when a subroutine or function is called.

e.g., when using the **BL** (Branch with Link) instruction:

```
main    ; Main program
        …
        BL  function1 ; Call function1 using Branch with Link
                      ; instruction.
                      ; PC = function1 and
                      ; LR = the next instruction in main
        …
function1
        …             ; Program code for function 1
        BX  LR        ; Return
```

Cortex-M3 Basics: Registers

Program Counter R15

When you read this register you will find that the value is different than the location of the executing instruction by 4, due to the pipelining of the processor

Example:

```
0x1000 : MOV  R0, PC     ; R0 = 0x1004
```

When reading the PC, the LSB (bit 0) is always 0.

Why?

When writing to the PC, it will cause a branch. The LSB must be set to 1 to indicate the Thumb state operations (setting to 0 implies to switch to the ARM state, which will result in a fault exception in Cortex-M3)

Can you write to the PC in 8086? Why do we need to set LSB of R15 (true for R14) since all instructions are half-word or word aligned?

Cortex-M3 Basics: Registers

Special Registers

The special registers in the Cortex-M3 processor include:

- Program Status Registers (PSRs)
- Interrupt Mask Registers (PRIMASK, FAULTMASK, and BASEPRI)
- Control Register (CONTROL)

Can only be accessed via MSR and MRS instructions

```
MRS <reg>, <special_reg> ; Read special register
MSR <special_reg>, <reg> ; write to special register
```

Note: MSR and MRS cannot have memory addresses, only registers are allowed

Special Registers

Program Status Registers (PSRs)

The program status registers are subdivided into three status registers:

1. Application PSR (**APSR**), 2. Interrupt PSR (**IPSR**), 3. Execution PSR (**EPSR**)

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q											
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

When they are accessed as a collective item, the name xPSR is used (**PSR** used in program codes).

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Exception Number				

EPSR and **IPSR** are read-only:

```
MRS   R0, APSR      ; Read Flag state into R0
MRS   R0, IPSR      ; Read Exception/Interrupt state
MRS   R0, EPSR      ; Read Execution state
MSR   APSR, R0      ; Write Flag state
```

Accessing xPSR:

```
MRS R0, PSR     ; Read the combined program status word
MSR PSR, R0     ; Write combined program state word
```

Bit Fields in Cortex-M3 Program Status Registers

Bit	Description
N	Negative
Z	Zero
C	Carry/borrow
V	Overflow
Q	Sticky saturation flag
ICI/IT	Interrupt-Continuable Instruction (ICI) bits, IF-THEN instruction status bit
T	Thumb state, always 1; trying to clear this bit will cause a fault exception
Exception Number	Indicates which exception the processor is handling

Cortex-M3 Interrupt Mask Registers

PRIMASK, FAULTMASK and BASEPRI Registers

The PRIMASK, FAULTMASK, and BASEPRI registers are used to disable exceptions.

Register Name	Description
PRIMASK	A 1-bit register. When this is set, it allows NMI and the hard fault exception; all other interrupts and exceptions are masked; default is 0 (no masking)
FAULTMASK	A 1-bit register. When this is set, it allows only the NMI, and all interrupts and fault handling exceptions are disabled; default is 0
BASEPRI	A register of up to 9 bits. It defines the masking priority level. When this is set, it disables all interrupts of the same or lower level (larger priority value); default is 0

➢To access the PRIMASK, FAULTMASK, and BASEPRI registers, the MRS and MSR instructions are used.

Example:

```
MRS   R0, BASEPRI      ; Read BASEPRI register into R0
MRS   R0, PRIMASK      ; Read PRIMASK register into R0
MRS   R0, FAULTMASK    ; Read FAULTMASK register into R0
MSR   BASEPRI, R0      ; Write R0 into BASEPRI register
MSR   PRIMASK, R0      ; Write R0 into PRIMASK register
MSR   FAULTMASK, R0    ; Write R0 into FAULTMASK register
```

➢PRIMASK and BASEPRI are useful for temporarily disabling interrupts in timing-critical tasks; FAULTMASK is used by the OS kernel which cleans up a crashed task

➢The PRIMASK, FAULTMASK, and BASEPRI registers cannot be set in the user access level.

Special Registers

The Control Register

The Control register is used to define the privilege level and the stack pointer selection. This register has two bits.

Bit	Function
CONTROL[1]	Stack status: 1 = Alternate stack is used 0 = Default stack (MSP) is used If it is in the Thread or base level, the alternate stack is the PSP. There is no alternate stack for handler mode, so this bit must be zero when the processor is in handler mode.
CONTROL[0]	0 = Privileged in Thread mode 1 = User state in Thread mode If in handler mode (not Thread mode), the processor operates in privileged mode.

CONTROL[1]

In Cortex-M3, the CONTROL[1] bit is always 0 (MSP) in handler mode. However, in the Thread mode, it can be either 0 or 1.

This bit is writable only when the core is in Thread mode and privileged.

CONTROL[0]

The CONTRL[0] bit is writable only in privileged level.

To access the Control register, the MRS and MSR instructions are used:

```
MRS   R0, CONTROL     ; Read CONTROL register into R0
MSR   CONTROL, R0     ; Write R0 into CONTROL register
```

Cortex-M3 Basics: Operation Mode

Two modes and two privilege levels.

Operation Modes and Privilege Levels in Cortex-M3

	Privileged Level	*User Level*
When running an exception	*privileged Handler Mode*	
When running main program	*privileged Thread Mode*	*User Thread Mode*

Cortex-M3 Basics: Operation Mode

When the processor exits reset, it is in Thread mode with privileged access level

In the user access level (Thread mode), access to the System Control Space (SCS, a memory block for configuration registers and debugging components) is blocked.

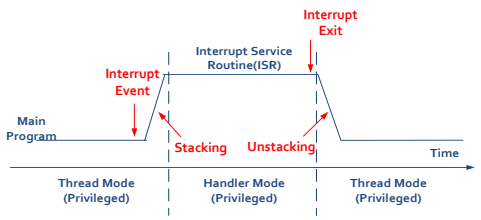
In the user access level, access to special registers (except the APSR) is also blocked. Fault exception will occur when trying to access SCS or special registers

Software in a privileged access level can switch the program into the user access level by setting CONTROL[0]=1

When an exception occurs, the processor will automatically switch to privileged state and return to the previous state when exiting the exception handler

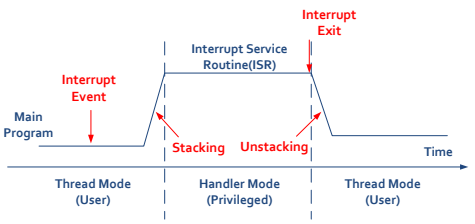
Therefore, in order to change to privileged level, a user program has to go through an exception/interrupt handler which can set the CONTROL register before returns.

When the CONTROL[0]=0, only the processor mode changes when an exception takes place.

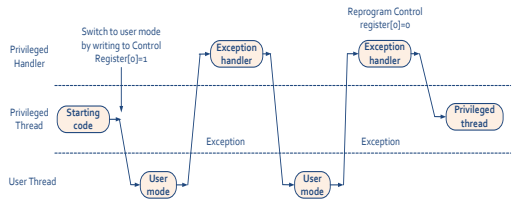


Switching Processor Mode at Interrupt

When CONTROL[0] = 1, both processor mode and access level change when an exception takes place.



Switching Processor Mode and Privilege Level at Interrupt



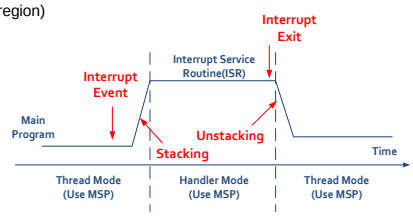
Switching of Operation Mode by Programming the Control Register or by Exceptions

Cortex-M3: The Two-Stack Model

The Cortex-M3 has two stack pointers: the Main Stack Pointer (MSP) and the Process Stack Pointer (PSP).

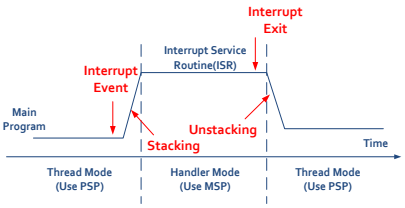
The SP register to be used is controlled by the bit 1 of the CONTROL register.

Control [1] = 0, both Thread mode and Handler mode use MSP (main program and handlers share the same stack memory region)



Cortex-M3: The Two-Stack Model

Control [1] = 1, the PSP is used in Thread mode and the MSP is used in handler mode (main program and handlers have separate stack memory regions)



Which stack will the automatic stacking and unstacking mechanism use?

Cortex-M3: Exceptions and Interrupts

The Cortex-M3 supports a fixed number of system exceptions and a number of interrupts (called *IRQ*), and a NMI input signal (e.g., connected to a watchdog timer or a voltage-monitoring block)

Exception Number	Exception Type	Priority	Function
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt
3	Hard fault	−1	All classes of fault, when the corresponding fault handler cannot be activated because it is currently disabled or masked by exception masking
4	MemManage	Settable	Memory management fault; caused by MPU violation or invalid accesses (such as an instruction fetch from a nonexecutable region)
5	BusFault	Settable	Error response received from the bus system; caused by an instruction prefetch abort or data access error
6	Usage fault	Settable	Usage fault; typical causes are invalid instructions or invalid state transition attempts (such as trying to switch to ARM state in the Cortex-M3)
7–10	-	-	Reserved
11	SVC	Settable	System service call via SVC instruction
12	Debug monitor	Settable	Debug monitor
13	-	-	Reserved
14	PendSV	Settable	Pendable request for System Service
15	SYSTICK	Settable	System Tick Timer
16–255	IRQ	Settable	IRQ input #0–239

Cortex-M3 Basics: Vector Table

The **vector table** is an array of word data, with each representing the starting address of the ISR for one exception/interrupt type.

The base address of the vector table is re-locatable (set the relocation register in the NVIC); initially, the base address is 0x0.

The word stored at address 0x00000000 is used as the starting value for the MSP.

Note that the LSB of all vectors in the table must be set to 1 (indicating the exception will be executed in Thumb state)

Example:

The reset is exception type 1. The address of the reset vector is 1 times 4, which equals 0x00000004; and NMI vector (type 2) is located at 2 * 4 = 0x00000008

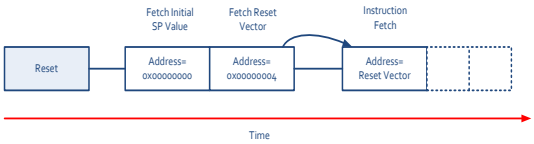
Vector Table Definition After Reset

Exception Type	Address Offset	Exception Vector
18–255	0x48–0x3FF	IRQ #2–239
17	0x44	IRQ #1
16	0x40	IRQ #0
15	0x3C	SYSTICK
14	0x38	PendSV
13	0x34	Reserved
12	0x30	Debug Monitor
11	0x2C	SVC
7–10	0x1C–0x28	Reserved
6	0x18	Usage fault
5	0x14	Bus fault
4	0x10	MemManage fault
3	0x0C	Hard fault
2	0x08	NMI
1	0x04	Reset
0	0x00	Starting value of the MSP

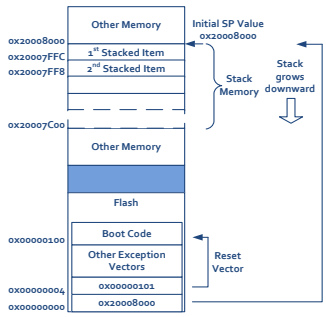
Cortex-M3: Reset Sequence

After the processor exits reset, it will read two words from memory:

- Address 0x00000000: default value of R13 (MSP)
- Address 0x00000004: Reset vector (the starting address of startup program)



Initial Stack Pointer Value and Initial Program Counter (PC) Value Example



Assume that the stack memory range from 0x20007C00 to 0x20007FFF

- Notice that in the Cortex-M3, vector addresses in the vector table should have their LSB set to 1 to indicate that they are Thumb code. For that reason, the previous example has 0x101 in the reset vector, whereas the boot code starts at address 0x100.
- It is necessary to have the stack pointer initialized, because some of the exceptions (such as NMI) can happen right after reset, and the stack memory could be required for the handler of those exceptions.

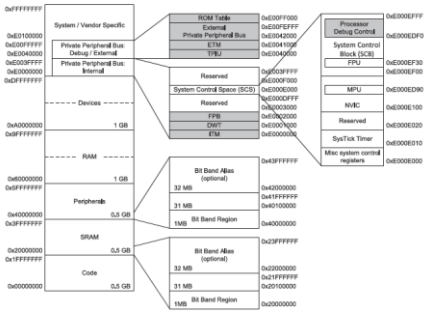
The Cortex-M3/M4 Embedded Systems: Cortex-M3/M4 Memory Systems

Refer to Chapter 5 in the reference book
"The Definitive Guide the ARM Cortex-M3"
Refer to Chapter 6 in the reference book
"The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

A Cortex-M3/M4 Predefined Memory Map

The Cortex-M3/M4 processor has a fixed memory map

The Cortex-M3/M4 processor has a total of 4GB of address space



A Cortex-M3 Predefined Memory Map

CODE: 0.5 GB
primarily for program code, including the default vector table. This region also allow data accesses

SRAM: 0.5 GB
primarily for connecting SRAM, mostly on-chip SRAM, but there is no limitation of exact memory type; program execution is allowed; accessed via the system bus interface; bit-band operation

On-chip peripherals: 0.5 GB
for on-chip peripherals; bit-band operation

External RAM: 1 GB
for external RAM; program execution is allowed

External devices: 1 GB
for and external devices

System: 0.5 GB
Internal Private Peripheral Bus (PPB): system components, e.g., NVIC, SysTick, MPU, debug components
External Private Peripheral Bus (PPB): additional optional debug components and vendor specific components.
vendor-specific area: vendor-specific components

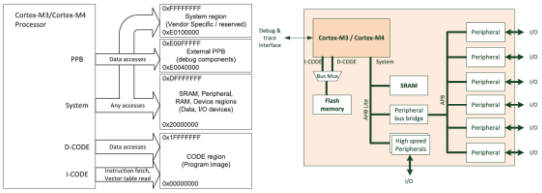
Connecting the Processor to Memory and Peripherals

the AHB (AMBA High-performance Bus) Lite protocol:

- used for the main bus interfaces

the APB protocol:

- used for the Private Peripheral Bus (PPB), mainly used for debug components.
- Additional bus segments based on APB can be added onto the system bus by using additional bus bridge components.



Default Memory Access Permissions

The Cortex-M3 memory map has a default configuration for memory access permissions, used when there is no MPU or MPU is disabled

Prevents user program (i.e., in user Thread mode) from accessing system control memory spaces

Memory Region	Address	Access in User Program
Vendor specific	0xE0100000-0xFFFFFFFF	Full access
ROM Table	0xE00FF000-0xE00FFFFF	Blocked; user access results in bus fault
External PPB	0xE0042000-0xE00FEFFF	Blocked; user access results in bus fault
ETM	0xE0041000-0xE0041FFF	Blocked; user access results in bus fault
TPIU	0xE0040000-0xE0040FFF	Blocked; user access results in bus fault
Internal PPB	0xE000F000-0xE003FFFF	Blocked; user access results in bus fault
NVIC	0xE000E000-0xE000EFFF	Blocked; user access results in bus fault, except Software Trigger Interrupt Register that can be programmed to allow user accesses
FPB	0xE0002000-0xE0003FFF	Blocked; user access results in bus fault
DWT	0xE0001000-0xE0001FFF	Blocked; user access results in bus fault
ITM	0xE0000000-0xE0000FFF	Read allowed; write ignored except for stimulus ports with user access enabled
External Device	0xA0000000-0xDFFFFFFF	Full access
External RAM	0x60000000-0x9FFFFFFF	Full access
Peripheral	0x40000000-0x5FFFFFFF	Full access
SRAM	0x20000000-0x3FFFFFFF	Full access
Code	0x00000000-0x1FFFFFFF	Full access

Bit-Band Operations

Bit-band operation allows a single load/store (read/write) operation to access a single data bit.

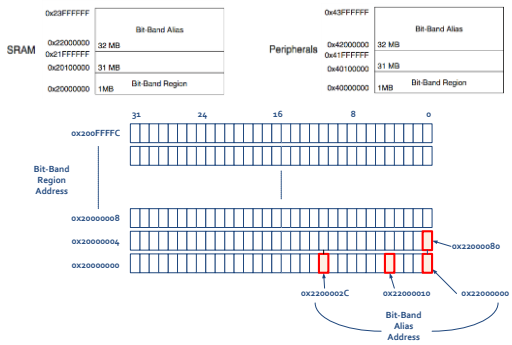
Two predefined memory regions, called ***bit-band regions***:

- the first 1 MB of the SRAM region
- the first 1 MB of the peripheral region

Two associated predefined memory regions, called ***bit-band alias***:

Bit band regions can be accessed like normal memory, but each bit within a bit band region can also be accessed using a word-aligned address in the corresponding bit-band alias

Bit-Band Region and Bit-Band Alias



Bit-Band Region and Bit-Band Alias

Remapping of Bit-Band Addresses in SRAM Region

Bit-Band Region	Aliased Equivalent
0x20000000 bit[0]	0x22000000 bit[0]
0x20000000 bit[1]	0x22000004 bit[0]
0x20000000 bit[2]	0x22000008 bit[0]
…	…
0x20000000 bit[31]	0x2200007C bit[0]
0x20000004 bit[0]	0x22000080 bit[0]
…	…
0x20000004 bit[31]	0x220000FC bit[0]
…	…
0x200FFFFC bit[31]	0x23FFFFFC bit[0]

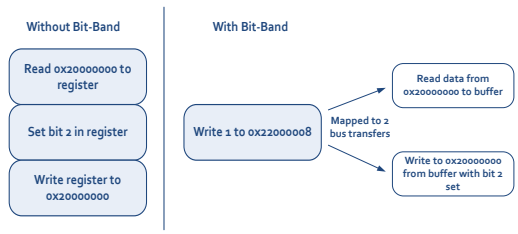
Remapping of Bit-Band Addresses in Peripheral Memory Region

Bit-Band Region	Aliased Equivalent
0x40000000 bit[0]	0x42000000 bit[0]
0x40000000 bit[1]	0x42000004 bit[0]
0x40000000 bit[2]	0x42000008 bit[0]
…	…
0x40000000 bit[31]	0x4200007C bit[0]
0x40000004 bit[0]	0x42000080 bit[0]
…	…
0x40000004 bit[31]	0x420000FC bit[0]
…	…
0x400FFFFC bit[31]	0x43FFFFFC bit[0]

Write to Bit-Band Alias

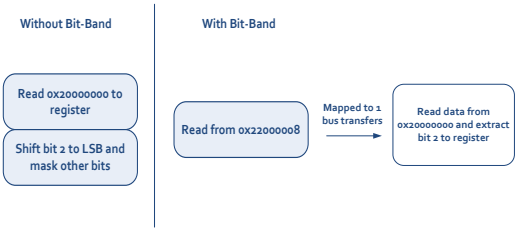
A **hardware** READ-MODIFY-WRITE is performed

For example, to set bit 2 in word data at address 0x20000000:



Read from Bit-Band Alias

To read bit 2 in word data in address 0x20000000:



Bit-Band Operation Example

Example:

1. Set address 0x20000000 to a value of 0x3355AACC.
2. Read address 0x22000008. This read access is remapped into read access to 0x20000000. The return value is 1 (bit[2] of 0x3355AACC).
3. Write 0x0 to 0x22000008.
4. Now read 0x20000000. That gives you a return value of 0x3355AAC8 (bit[2] =0).

Advantages of Bit-band Operations

Faster bit operations with fewer instructions

Exclusive read/write operations (by hardware)

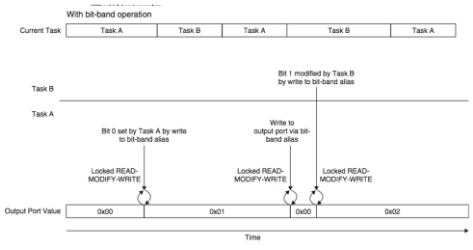


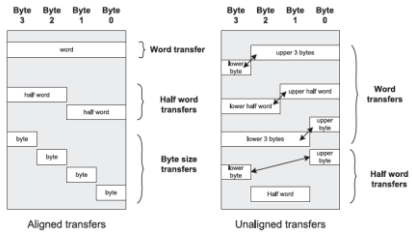
Figure 5.9 Data Loss Prevention with Locked Transfers Using the Bit-Band Feature

Data Alignment and Unaligned Transfers

Aligned transfers means the address value is a multiple of the size (in bytes).

The Cortex-M3 supports unaligned transfers on single accesses.

When unaligned transfers are issued by the processor, they are actually converted into multiple aligned transfers by the processor's bus interface unit.



Unaligned Transfers

There are a number of limitations:

1. Not supported in Load/Store multiple instructions.
2. Stack operations (PUSH/POP) must be aligned.
3. Exclusive accesses must be aligned.
4. Unaligned transfers are not supported in bit-band operations.

When unaligned transfers are used, they are actually converted into multiple aligned transfers by the processor's bus interface unit

1. Transparent to application programmers.
2. It takes more clock cycles for a single data access

Exclusive Accesses

Exclusive access instructions: LDREX,STREX

SWP instruction (swap) was used to check semaphore status in early ARM processors. But Cortex-M3 cannot support SWP any more.

1. What is semaphore?

Semaphores are commonly used for allocating shared resources to applications. When a resource is being used by one process, it is locked to that process and exclusive to others. A semaphore is the lock flag.

2. Why is exclusive access used instead of SWP instruction?

When a process or application want to use a resource, it needs to check whether the resource has been locked first.

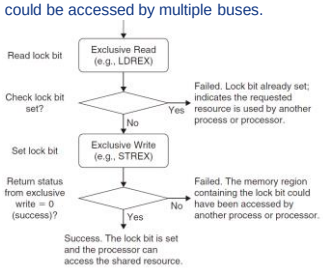
In ARM V7 architecture, the read/write access can be carried out on separated buses. Therefore, the SWP instructions (requiring the read and write in a locked transfer sequence must be on the same bus) can no longer be used to make the memory access atomic.

Therefore, the locked transfers are replaced by exclusive accesses.

Exclusive Accesses

3. The difference between SWP instructions and exclusive access.

The concept of exclusive access operation is quite simple but different from SWP, which allows the possibility that the memory location for a semaphore could be accessed by multiple buses.



Exclusive Accesses

Exclusive access instructions include:

1. LDREX (word)
2. LDREXB (byte)
3. LDREXH (half word)
4. STREX (word)
5. STREXB (byte)
6. STREXH (half word)

Exclusive Access Example

```
LockDeviceA
          ; A simple function to try to lock Device A
          ; Output R0 : 0 = Success, 1 = failed
          ; If successful, value of 1 will be written to variable
          ; DeviceALocked
          PUSH  {R1, R2, LR}
TryToLockDeviceA
          LDR       R1,=DeviceALocked     ; Get the lock status
          LDREX     R2,[R1]
          CMP       R2,#0                 ; Check if it is locked
          BNE       LockDeviceAFailed
DeviceAIsNotLocked
          MOV       R0,#1                 ; Try to write 1 to
                                          ; DeviceALocked
          STREX     R2,R0,[R1]            ; Exclusive write
          CMP       R2, #0
          BNE       LockDeviceAFailed     ; STREX Failed
LockDeviceASucceed
          MOV       R0,#0                 ; Return success status
          POP       {R1, R2, PC}          ; Return
LockDeviceAFailed
          MOV       R0,#1                 ; Return fail status
          POP       {R1, R2, PC}          ; Return
```

Endian Mode

The Cortex-M3/M4 supports both little endian (recommended) and big endian modes.

In the Cortex-M3/M4, the big endian scheme uses *byte-invariant big endian* (BE-8) byte lane usage.

Memory view

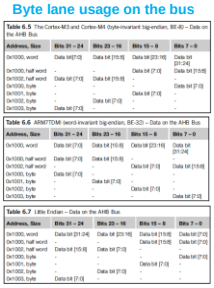
Table 6.3 Little Endian Memory View

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x0003 – 0x0000	Byte – 0x3	Byte – 0x2	Byte – 0x1	Byte – 0x0
...				
0x1003 – 0x1000	Byte – 0x1003	Byte – 0x1002	Byte – 0x1001	Byte – 0x1000
0x1007 – 0x1004	Byte – 0x1007	Byte – 0x1006	Byte – 0x1005	Byte – 0x1004
...				
	Byte – 4xN+3	Byte – 4xN+2	Byte – 4xN+1	Byte – 4xN

Table 6.4 Big Endian Memory View

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x0003 – 0x0000	Byte – 0x0	Byte – 0x1	Byte – 0x2	Byte – 0x3
...				
0x1003 – 0x1000	Byte – 0x1000	Byte – 0x1001	Byte – 0x1002	Byte – 0x1003
0x1007 – 0x1004	Byte – 0x1004	Byte – 0x1005	Byte – 0x1006	Byte – 0x1007
...				
	Byte – 4xN	Byte – 4xN+1	Byte – 4xN+2	Byte – 4xN+3

Byte lane usage on the bus



Endian Mode

The endian mode is set when the processor exits reset and it cannot be changed afterward.

Instructions are always in little endian, as are data accesses in the configuration control memory space and the external PPB memory range.

The data can be easily converted between little endian and big endian using instructions REV/REVH.

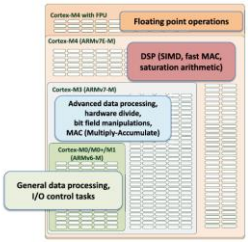
The Cortex-M3/m4 Embedded Systems: Cortex-M3/M4 Instruction Sets

Refer to Chapter 4 in the reference book
"The Definitive Guide the ARM Cortex-M3"
Refer to Chapter 5 in the reference book
"The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

ARM Instruction Set Architecture

There are quite a lot of instructions in the Cortex-M processors, but there is no need to learn them all in detail

- C compilers are good enough to generate efficient code
- the free CMSIS-DSP library and various middleware (e.g., software libraries)



Assembly Basics

The syntax depends on the assembler that you're using

With the ARM assembler, the common instruction format is as follows

```
label
 mnemonic operand1, operand2, ... ; Comments
```

label is optional, used as a reference to an address location

mnemonic is the name of the instruction

The number of operands depends on the type of instruction

Normally, the first operand is the destination of the operation

The text after each semicolon (;) is a comment

Assembly Basics

Immediate data are usually in the form #number

```
 MOV R0, #0x12 ; Set R0 = 0x12 (hexadecimal)
 MOV R1, #'A'  ; Set R1 = ASCII character A
```

Define constants using EQU

```
NVIC_IRQ_SETEN0 EQU 0xE000E100
NVIC_IRQ0_ENABLE EQU 0x1
```

Use DCB and DCD to define byte-size and word-size variables, respectively

```
MY_NUMBER
     DCD 0x12345678
HELLO_TXT
     DCB "Hello\n",0    ; null terminated string
```

Use of suffixes

Suffix	Description
S	Update APSR (flags); for example: `ADDS R0, R1 ; this will update APSR`
EQ , NE, LT, GT, and so on	Conditional execution; EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, etc. For example: `BEQ <Label> ; Branch if equal`

Using Unified Assembler Language

To allow better portability between architectures, UAL was developed

Using the same syntax for both make it easier to port applications

```
ADD  R0, R1     ; R0 = R0 + R1, using Traditional Thumb syntax
ADD  R0, R0, R1 ; Equivalent instruction using UAL syntax
```

The traditional Thumb syntax can still be used. Note that some of them change APSR without the ***S*** suffix

```
AND  R0, R1      ; Traditional Thumb syntax
ANDS  R0, R0, R1 ; Equivalent UAL syntax (S suffix is added)
```

You can specify whether to use Thumb or Thumb-2 instructions by adding suffixes of ***.N*** or ***.W***

```
ADDS   R0, #1 ; Use 16-bit Thumb instruction by default
              ; for smaller size
ADDS.N R0, #1 ; Use 16-bit Thumb instruction (N=Narrow)
ADDS.W R0, #1 ; Use 32-bit Thumb-2 instruction (W=wide)
```

Note: In most cases, applications will be coded in C, and the C compilers will use 16-bit instructions if possible due to smaller code size

Assembly Basics

32-bit Thumb-2 instructions can be half word aligned

```
0x1000 : LDR r0,[r1]   ; a 16-bit instructions (occupy 0x1000-0x1001)
0x1002 : RBIT.W r0     ; a 32-bit Thumb-2 instruction (occupy
                       ; 0x1002-0x1005)
```

Most of the 16-bit instructions can only access registers R0 to R7; 32-bit Thumb-2 instructions do not have this limitation

16-bit Instruction List

Table 4.2 16-Bit Data Processing Instructions

Instruction	Function
ADC	Add with carry
ADD	Add
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear (Logical AND one value with the logic inversion of another value)
CMN	Compare negative (compare one data with two's complement of another data and update flags)
CMP	Compare (compare two data and update flags)
CPY	Copy (available from architecture v6; move a value from one high or low register to another high or low register)
EOR	Exclusive OR
LSL	Logical shift left
LSR	Logical shift right
MOV	Move (can be used for register-to-register transfers or loading immediate data)
MUL	Multiply
MVN	Move NOT (obtain logical inverted value)
NEG	Negate (obtain two's complement value)
ORR	Logical OR
ROR	Rotate right
SBC	Subtract with carry
SUB	Subtract
TST	Test (use as logical AND; Z flag is updated but AND result is not stored)
REV	Reverse the byte order in a 32-bit register (available from architecture v6)
REVH	Reverse the byte order in each 16-bit half word of a 32-bit register (available from architecture v6)
REVSH	Reverse the byte order in the lower 16-bit half word of a 32-bit register and sign extends the result to 32 bits. (available from architecture v6)
SXTB	Signed extend byte (available from architecture v6)
SXTH	Signed extend half word (available from architecture v6)
UXTB	Unsigned extend byte (available from architecture v6)
UXTH	Unsigned extend half word (available from architecture v6)

16-bit Instruction List

Table 4.3 16-Bit Branch Instructions

Instruction	Function
B	Branch
B<cond>	Conditional branch
BL	Branch with link; call a subroutine and store the return address in LR
BLX	Branch with link and change state (BLX <reg> only)
CBZ	Compare and branch if zero (architecture v7)
CBNZ	Compare and branch if nonzero (architecture v7)
IT	If-THEN (architecture v7)

Table 4.4 16-Bit Load and Store Instructions

Instruction	Function
LDR	Load word from memory to register
LDRH	Load half-word from memory to register
LDRB	Load byte from memory to register
LDRSH	Load half-word from memory, sign extend it, and put it in register
LDRSB	Load byte from memory, sign extend it, and put it in register
STR	Store word from register to memory
STRH	Store half word from register to memory
STRB	Store byte from register to memory
LDMIA	Load multiple increment after
STMIA	Store multiple increment after
PUSH	Push multiple registers
POP	Pop multiple registers

Table 4.5 Other 16-Bit Instructions

Instruction	Function
SVC	System service call
BKPT	Breakpoint; if debug is enabled, will enter debug mode (halted), or if debug monitor exception is enabled, will invoke the debug exception; otherwise it will invoke a fault exception
NOP	No operation
CPSIE	Enable PRIMASK (CPSIE i)/FAULTMASK (CPSIE f) register (set the register to 0)
CPSID	Disable PRIMASK (CPSID i)/ FAULTMASK (CPSID f) register (set the register to 1)

32-bit Instruction List

Table 4.6 32-Bit Data Processing Instructions

Instruction	Function
ADC	Add with carry
ADD	Add
ADDW	Add wide (#immed_12)
AND	Logical AND
ASR	Arithmetic shift right
BIC	Bit clear (logical AND one value with the logic inversion of another value)
BFC	Bit field clear
BFI	Bit field insert
CMN	Compare negative (compare one data with two's complement of another data and update flags)
CMP	Compare (compare two data and update flags)
CLZ	Count lead zero
EOR	Exclusive OR
LSL	Logical shift left
LSR	Logical shift right
MLA	Multiply accumulate
MLS	Multiply and subtract
MOV	Move
MOVW	Move wide (write a 16-bit immediate value to register)
MOVT	Move top (write an immediate value to the top half word of destination reg)
MVN	Move negative
MUL	Multiply
ORR	Logical OR
ORN	Logical OR NOT
RBIT	Reverse bit
REV	Byte reverse word
REV16/REV16	Byte reverse packed half word
REVSH	Byte reverse signed half word
ROR	Rotate right register
RSB	Reverse subtract
RRX	Rotate right extended
SBFX	Signed bit field extract
SDIV	Signed divide
SMLAL	Signed multiply accumulate long
SMULL	Signed multiply long
SSAT	Signed saturate
SBC	Subtract with carry
SUB	Subtract
SUBW	Subtract wide (#immed_12)
SXTB	Sign extend byte
TEQ	Test equivalent (use as logical exclusive OR; flags are updated but result is not stored)
TST	Test (use as logical AND; Z flag is updated but AND result is not stored)
UBFX	Unsigned bit field extract
UDIV	Unsigned divide
UMLAL	Unsigned multiply accumulate long
UMULL	Unsigned multiply long
USAT	Unsigned saturate
UXTB	Unsigned extend byte
UXTH	Unsigned extend half word

32-bit Instruction List

Table 4.7 32-Bit Load and Store Instructions

Instruction	Function
LDR	Load word data from memory to register
LDRB	Load byte data from memory to register
LDRH	Load half word data from memory to register
LDRSB	Load byte data from memory, sign extend it, and put it to register
LDRSH	Load half word data from memory, sign extend it, and put it to register
LDM	Load multiple data from memory to registers
LDRD	Load double word data from memory to registers
STR	Store word to memory
STRB	Store byte data to memory
STRH	Store half word data to memory
STM	Store multiple words from registers to memory
STRD	Store double word data from registers to memory
PUSH	Push multiple registers
POP	Pop multiple registers

Table 4.9 Other 32-Bit Instructions

Instruction	Function
LDREX	Exclusive load word
LDREXH	Exclusive load half word
LDREXB	Exclusive load byte
STREX	Exclusive store word
STREXH	Exclusive store half word
STREXB	Exclusive store byte
CLREX	Clear the local exclusive access record of local processor
MRS	Move special register to general-purpose register
MSR	Move to special register from general-purpose register
NOP	No operation
SEV	Send event
WFE	Sleep and wake for event
WFI	Sleep and wake for interrupt
ISB	Instruction synchronization barrier
DSB	Data synchronization barrier
DMB	Data memory barrier

Table 4.8 32-Bit Branch Instructions

Instruction	Function
B	Branch
BL	Branch and link
TBB	Table branch byte; forward branch using a table of single byte offset
TBH	Table branch half word; forward branch using a table of half word offset

Moving data within the processor

Between register and register

Between special register and register

Moving an immediate data value into a register

For the Cortex-M4 processor with the floating point unit

Between a register in the core register bank and a register in the floating point unit register bank

Between a floating point system register (such as the FPSCR - Floating point Status and Control Register) and a core register

Move immediate data into a floating point register

Between Register and Register

MOV

Move data between registers

```
MOV R8, R3
```

MVN

generate the negative value of the original data and move to the destination register

MRS and **MSR**

Use the instructions MRS and MSR to access special registers

```
MRS R0, PSR      ; Read Processor status word into R0
MSR CONTROL, R1 ; Write value of R1 into control register
```

APSR can be accessed with user access level with other special registers only being accessed in privileged mode

Between Immediate data to Register

MOV can be used for small values (8 bits or less)

```
MOV  R0, #0x12 ; Set R0 to 0x12
```

Thumb-2 instructions **MOVW** and **MOVT** for larger values

```
MOVW.W  R0,#0x789A ; Set R0 lower half to 0x789A
MOVT.W  R0,#0x3456 ; Set R0 upper half to 0x3456. Now
                   ; R0=0x3456789A
```

LDR (LDR pseudo-instruction, not the LDR instruction)

A pseudo instruction provided in ARM assembler can be used to load a register with either a 32-bit constant

```
LDR  R0, =0x3456789A
```

Between Immediate data to Register

LDR can also used to load a register with an address

If the address is a program address value, it will automatically set the LSB to 1

```
      LDR  R0, =address1 ; R0 set to 0x4001
      ...
address1
0x4000: MOV  R0, R1 ; address1 contains program code
      ...
```

If the address is a data address, LSB will not be changed

```
      LDR R0, =address1 ; R0 set to 0x4000
      ...
address1
0x4000: DCD 0x0 ; address1 contains data
      ...
```

Memory Access Instructions

LDR and **STR**

LDR transfers data from memory to registers, and STR transfers data from registers to memory

Be aware of the position of the destination operand

Example	Description
LDRB Rd, [Rn, #offset]	Read byte from memory location Rn + offset
LDRH Rd, [Rn, #offset]	Read half-word from memory location Rn + offset
LDR Rd, [Rn, #offset]	Read word from memory location Rn + offset
LDRD Rd1,Rd2, [Rn, #offset]	Read double word from memory location Rn + offset
STRB Rd, [Rn, #offset]	Store byte to memory location Rn + offset
STRH Rd, [Rn, #offset]	Store half word to memory location Rn + offset
STR Rd, [Rn, #offset]	Store word to memory location Rn + offset
STRD Rd1,Rd2, [Rn, #offset]	Store double word to memory location Rn + offset

Memory Access Instructions

LDM (Load Multiple) and **STM** (Store Multiple)

Multiple Load and Store operations can be combined into single instructions

Example	Description
LDMIA Rd!,<reg list>	Read multiple words from memory location specified by *Rd*. Address Increment After (IA) each transfer (16-bit Thumb instruction).
STMIA Rd!,<reg list>	Store multiple words to memory location specified by *Rd*. Address Increment After (IA) each transfer (16-bit Thumb instruction).
LDMIA.W Rd(!),<reg list>	Read multiple words from memory location specified by *Rd*. Address increment after each read (.W specified it is a 32-bit Thumb-2 instruction).
LDMDB.W Rd(!),<reg list>	Read multiple words from memory location specified by *Rd*. Address Decrement Before (DB) each read (.W specified it is a 32-bit Thumb-2 instruction).
STMIA.W Rd(!),<reg list>	Write multiple words to memory location specified by *Rd*. Address increment after each read (.W specified it is a 32-bit Thumb-2 instruction).
STMDB.W Rd(!),<reg list>	Write multiple words to memory location specified by *Rd*. Address Decrement Before each read (.W specified it is a 32-bit Thumb-2 instruction).

Memory Access Instructions

"**!**" in the instruction specifies whether the register **Rd** should be updated after the instruction is completed

If R8 equals 0x8000

```
STMIA.W  R8!, {R0-R3} ; R8 changed to 0x8010 after store
                      ; (increment by 4 words)
STMIA.W  R8 , {R0-R3} ; R8 unchanged after store
```

Memory access with **pre-indexing**

The register holding the memory address is adjusted first and then the memory transfer then takes place with the updated address.

```
LDR.W  R0,[R1, #offset]! ; Read memory[R1+offset], with R1
                         ; update to R1+offset
```

Memory Access Instructions

Memory access with **post-indexing**

The memory transfer is carried out using the base address specified by the register and then update the address register afterward

```
LDR.W R0,[R1], #offset ; Read memory[R1], with R1
                       ; updated to R1+offset
```

PUSH and **POP**

```
  PUSH {R0, R4-R7, R9} ; Push R0, R4, R5, R6, R7, R9 into
                       ; stack memory
  POP  {R2,R3}         ; Pop R2 and R3 from stack
```

Processing Data

Many data operation instructions can have multiple instruction formats

```
ADD   R0, R1     ; R0 = R0+R1
ADD   R0, #0x12  ; R0 = R0 + 0x12
ADD.W R0, R1, R2 ; R0 = R1+R2
```

Arithmetic functions, logical operations, reserving bytes in a register, and bit operations

Note that when 16-bit Thumb arithmetic instructions are used the flags in the PSR will be automatically affected.

For 32-bit Thumb-2 instructions, changes of PSR depends on whether the ***S*** suffix is used

Examples of Arithmetic Instructions

Instruction		Operation

ADD Rd, Rn, Rm
ADD Rd, Rm
ADD Rd, #immed
	; Rd = Rn + Rm
	; Rd = Rd + Rm
	; Rd = Rd + #immed
		ADD operation

ADC Rd, Rn, Rm ADC Rd, Rm ADC Rd, #immed	; Rd = Rn + Rm + carry ; Rd = Rd + Rm + carry ; Rd = Rd + #immed + carry	ADD with carry
ADDW Rd, Rn,#immed	; Rd = Rn + #immed	ADD register with 12-bit immediate value

SUB Rd, Rn, Rm
SUB Rd, #immed
SUB Rd, Rn,#immed
	; Rd = Rn – Rm
	; Rd = Rd – #immed
	; Rd = Rn –#immed
		SUBTRACT

RSB.W Rd, Rn, #immed RSB.W Rd, Rn, Rm MUL Rd, Rm MUL.W Rd, Rn, Rm	; Rd = #immed –Rn ; Rd = Rm - Rn ; Rd = Rd * Rm Multiply ; Rd = Rn * Rm	Reverse subtract Multiply
UDIV Rd, Rn, Rm SDIV Rd, Rn, Rm	; Rd = Rn /Rm ; Rd = Rn /Rm	Unsigned and signed divide

Call and Unconditional Branch

B and **BX**

The most basic branch instructions

```
B  label ; Branch to a labeled address
BX reg   ; Branch to an address specified by a register
```

Note: In BX instructions, the LSB of the value contained in the register determines the next state (Thumb/ARM) of the processor. In Cortex-M3, since it is always in Thumb state, this bit should be set to 1; otherwise, the program will cause a usage fault exception.

BL and **BLX**

To call a function

```
BL  label    ; Branch to a labeled address and save return
             ; address in LR
BLX reg      ; Branch to an address specified by a register and
             ; save return
             ; address in LR.
```

Making a Branch

Branch using **MOV**, **LDR** and **POP** instructions

```
MOV R15,  R0     ; Branch to an address inside R0
LDR R15, [R0]    ; Branch to an address in memory location
                 ; specified by R0
POP {R15}        ; Do a stack pop operation, and change the
                 ; program counter value
                 ; to the result value.
```

Note:
1) To make sure that the LSB of the new program counter value is 0x1 (indicating the state of the next instruction is Thumb state)

2) BL instruction will destroy the content of LR. If you need the LR later, you should save the LR before using BL. For example, PUSH {LR}

Save the LR if You Need to Call a Subroutine

The BL instruction will destroy the current content of your LR register. So, if your program code needs the LR register later, you should save your LR before you use BL. The common method is to push the LR to stack in the beginning of your subroutine. For example:

```
main
      ...
      BL functionA
      ...
functionA
      PUSH {LR}  ; Save LR content to stack
      ...
      BL functionB
      ...
      POP  {PC}  ; Use stacked LR content to return to main
functionB
      PUSH {LR}
      ...
      POP  {PC}  ; Use stacked LR content to return to functionA
```

In addition, if the subroutine you call is a C function, you might also need to save the contents in R0-R3 and R12 if these values will be needed at a later stage. According to *AAPCS* (Ref 5), the contents in these registers could be changed by a C function.

Decisions and Conditional Branches

In the APSR, four of five flags are used for branch decisions

- Z (Zero) flag: This flag is set when the result of an instruction has a zero value or when a comparison of two data returns an equal result.
- N (Negative) flag: This flag is set when the result of an instruction has a negative value (bit 31 is 1).
- C (Carry) flag: This flag is for unsigned data processing—for example, in add (ADD) it is set when an overflow occurs; in subtract (SUB) it is set when a borrow did not occur (borrow is the invert of carry).
- V (Overflow) flag: This flag is for signed data processing; for example, in an add (ADD), when two positive values added together produce a negative value, or when two negative values added together produce a positive value.

Decisions and Conditional Branches

With combinations of the four flags (N, Z, C, and V), 15 branch conditions are defined

Symbol	Condition	Flag
EQ	Equal	Z set
NE	Not equal	Z clear
CS/HS	Carry set/unsigned higher or same	C set
CC/LO	Carry clear/unsigned lower	C clear
MI	Minus/negative	N set
PL	Plus/positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N set or V set, or N clear and V clear (N == V)
LT	Signed less than	N set and V clear, or N clear and V set (N != V)
GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
AL	Always (unconditional)	–

Decisions and Conditional Branches

Conditional branch instructions

```
BEQ label ; Branch to address 'label' if Z flag is set
```

Used in IF-THEN-ELSE structure

```
  CMP R0, R1    ; Compare R0 and R1
  ITTEE GT      ; If R0 > R1 Then (first 2 statements execute
                ; if true,
                ; other 2 statements execute if false)
  MOVGT R2, R0 ;       R2 = R0
  MOVGT R3, R1 ;       R3 = R1
  MOVLE R2, R0 ; Else R2 = R1
  MOVLE R3, R1 ;       R3 = R0
```

Several Useful Instructions

MRS and MSR

Symbol	Description
IPSR	Interrupt status register
EPSR	Execution status register (read as zero)
APSR[3]	Flags from previous operation
IEPSR	A composite of IPSR and EPSR
IAPSR	A composite of IPSR and APSR
EAPSR	A composite of EPSR and APSR
PSR	A composite of APSR, EPSR and IPSR
MSP	Main stack pointer
PSP	Process stack pointer
PRIMASK	Normal exception mask register
BASEPRI	Normal exception priority mask register
BASEPRI_MAX	Same as normal exception priority mask register, with conditional write (new priority level must be higher than the old level)
FAULTMASK	Fault exception mask register (also disables normal interrupts)
CONTROL	Control register

```
LDR R0,=0x20008000 ; new value for Process Stack Pointer (PSP)
MSR PSP, R0
```

Several Useful Instructions

IF-THEN-ELSE

```
IT<x>          <cond>
IT<x><y>       <cond>
IT<x><y><z>    <cond>
```

where:

- <x> specifies the execution condition for the second instruction
- <y> specifies the execution condition for the third instruction
- <z> specifies the execution condition for the fourth instruction
- <cond> specifies the base condition of the instruction block; the first instruction following IT executes if <cond> is true

```
if (R0 equal R1) then {
  R3 = R4 + R5
  R3 = R3 / 2
  } else {
  R3 = R6 + R7
  R3 = R3 / 2
  }
```

```
CMP   R0, R1      ; Compare R0 and R1
ITTEE EQ          ; If R0 equal R1, Then-Then-Else-Else
ADDEQ R3, R4, R5  ; Add if equal
ASREQ R3, R3, #1  ; Arithmetic shift right if equal
ADDNE R3, R6, R7  ; Add if not equal
ASRNE R3, R3, #1  ; Arithmetic shift right if not equal
```

Several Useful Instructions

CBZ and CBNZ

```
CBZ  <Rn>, <label>

CBNZ <Rn>, <label>
```

```
while (R0 != 0) {
   function1();
   }
```

```
loop
   CBZ R0, loopexit
   BL  function1
   B   loop
loopexit
```

Refer to the reference book for information of other instructions

The Cortex-M3/M4 Embedded Systems:

Cortex-M3/M4 Exceptions and Interrupts

Refer to Chapter 7, 8, 9 in the reference book
"The Definitive Guide the ARM Cortex-M3"
Refer to Chapter 7, 8, 10, 12 in the reference book
"The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

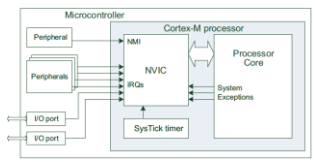
Exceptions

Exceptions are numbered 1 to 15 for system exceptions and the rest 240 for external interrupt inputs.(Total 256 entries in vector table.)

Most of the exceptions have programmable priority, and a few have fixed priority.

The value of the current running exception is indicated by the special register **IPSR** or from the NVIC's **Interrupt Control State Register**.

An enabled exception can be pended (which means it cannot be carried out immediately due to some reasons)



List of Exceptions

Table 7.1 List of System Exceptions

Exception Number	Exception Type	Priority	Description
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Nonmaskable interrupt (external NMI input)
3	Hard Fault	−1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage Fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus Fault	Programmable	Bus error; occurs when AHB interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access)
6	Usage Fault	Programmable	Exceptions due to program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7–10	Reserved	NA	–
11	SVCall	Programmable	System Service call
12	Debug Monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	–
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System Tick Timer

Table 7.2 List of External Interrupts

Exception Number	Exception Type	Priority
16	External Interrupt #0	Programmable
17	External Interrupt #1	Programmable
…	…	…
255	External Interrupt #239	Programmable

Note that here the interrupt number (e.g., Interrupt #0) refers to the interrupt inputs to the Cortex-M3 NVIC

Exception Priority

Whether and when an exception can be carried out can be affected by the priority of the exception

A higher-priority (smaller number in priority level) exception can preempt (抢占) a lower-priority exception.

Reset, NMI, and hard fault have fixed highest-priority levels

The Cortex-M3 supports 256 levels of programmable priority.

The reduction of priority levels can be implemented by cutting out several lowest bits of the priority configuration register.

Priority-Level Configuration Register

3 bits of priority level

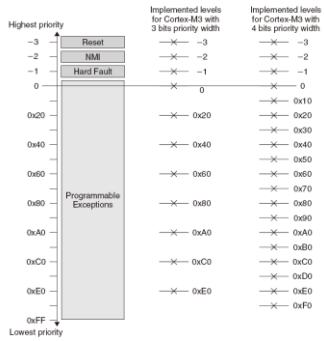
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented			Not implemented, read as zero				

4 bits of priority level

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented				Not implemented, read as zero			

The minimum width of the implemented priority register is 3 bits.

Available Priority Levels with 3-Bit or 4-Bit Priority Width



Priority-Level Configuration Register Organization

This register is further divided into two parts: **preempt priority** and **subpriority**.

The priority-level configuration register organization can be defined using a **Priority Group** register in the NVIC

- **Group (preempt) priority**: an interrupt or exception with a higher preempt priority can preempt one with a lower preempt priority
- **Subpriority**: defines the order when multiple interrupts or exceptions with the same preempt priority occur at the same time

Table 7.5 Application Interrupt and Reset Control Register (Address 0xE000ED0C)

Bits	Name	Type	Reset Value	Description
31:16	VECTKEY	R/W	–	Access key; 0x05FA must be written to this field to write to this register, otherwise the write will be ignored; the read-back value of the upper-half word is 0xFA05
15	ENDIANNESS	R	–	Indicates endianness for data: 1 for big endian (BE8) and 0 for little endian; this can only change after a reset
10:8	PRIGROUP	R/W	0	Priority group
2	SYSRESETREQ	W	–	Requests chip control logic to generate a reset
1	VECTCLRACTIVE	W	–	Clears all active state information for exceptions; typically used in debug or OS to allow system to recover from system error (Reset is safer)
0	VECTRESET	W	–	Resets the Cortex-M3 processor (except debug logic), but this will not reset circuits outside the processor

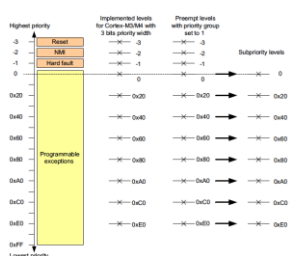
Priority Group	Preempt Priority Field	Subpriority Field
0	Bit [7:1]	Bit [0]
1	Bit [7:2]	Bit [1:0]
2	Bit [7:3]	Bit [2:0]
3	Bit [7:4]	Bit [3:0]
4	Bit [7:5]	Bit [4:0]
5	Bit [7:6]	Bit [5:0]
6	Bit [7]	Bit [6:0]
7	None	Bit [7:0]

When two interrupts with exactly the same group priority level and sub-priority level, the interrupt with the smaller exception number has higher priority (e.g., IRQ #0 has higher priority than IRQ #1)

Deciding Preempt priority and Subpriority Levels

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Preempt priority[5:3]			Preempt priority[2:0] (always 0)			Sub-priority [1:0] (always 0)	

Definition of priority fields in a 3-bit priority-level register with priority group set to 1



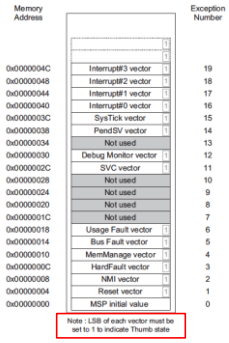
Available priority levels with 3-bit priority width and priority group set to 1

Vector Table

The processor will need to locate the starting address of the exception handler when an exception is being handled. This information is stored in the vector table.

In a minimal setup, the vector table needs to provide the initial MSP value and the Reset vector for the system to boot up.

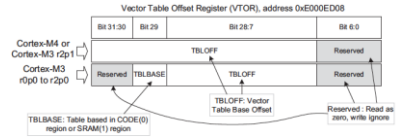
In addition, depending on your application, you might also need to include the NMI vector and the HardFault vector for error handling.



Vector Table Relocation

In Cortex-M3 r2p0 or older versions, the vector table can only be in the CODE region or the SRAM region. This restriction is removed from Cortex-M3 r2p1 and Cortex-M4.

The base address of the new vector table must be aligned to the size of the vector table extended to the next larger power of 2.



Example 1: 32 interrupt sources in the microcontroller

The vector table size is (32 (for interrupts) +16 (for system exception space)) x 4 (bytes for each vector) = 192 (0xC0). Extending it to the next power of two makes it 256 bytes. So the vector table base address can be programmed as 0x00000000, 0x00000100, 0x00000200, and so on.

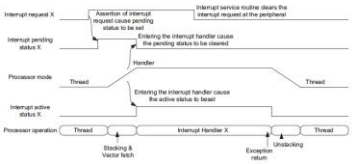
Interrupt Inputs and Pending Behaviors

An interrupt request can be accepted by the processor if:

- *The pending status (hold by a register) of this interrupt is set*
- *The interrupt is enabled (controlled by an interrupt masking register)*
- *The priority of the interrupt is higher than the current level*

Otherwise, the interrupt would be pended until the other interrupt handler is finished, or when the interrupt masking is cleared

When the interrupt is being served, the pending status of the interrupt is cleared automatically and it is in the active state (hold by a register).

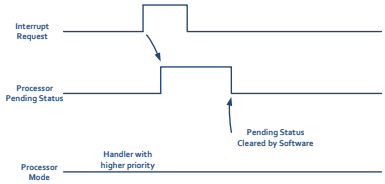


With the pending status of an interrupt, NVIC supports both pulsed interrupt requests and high level interrupt request

Interrupt Inputs and Pending Behaviors

you can clear a pending interrupt or use software to pend a new interrupt by setting the pending register

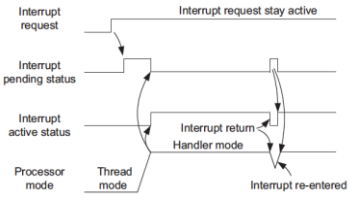
If the pending status is cleared (the pending status of the interrupt can be accessed in the NVIC and is writable) before the processor starts responding to the pended interrupt, the interrupt can be canceled



Interrupt Pending Cleared Before Processor Takes Action

Interrupt Inputs and Pending Behaviors

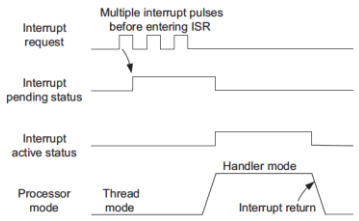
If an interrupt source continues to hold the interrupt request signal active, the interrupt will be pended again at the end of the interrupt service routine.



Continuous Interrupt Request Pends Again After Interrupt Exit

Interrupt Inputs and Pending Behaviors

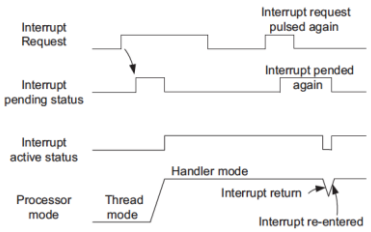
If an interrupt is pulsed several times before the processor starts processing it, it will be treated as one single interrupt request.



Interrupt Pending Only Once, Even with Multiple Pulses Before the Handler

Interrupt Inputs and Pending Behaviors

If an interrupt is de-asserted and then pulsed again during the interrupt service routine, it will be pended again.



Interrupt Pending Occurs Again During the Handler

Interrupt Control

Registers in NVIC related to interrupt control

- Most of the interrupt control/ status registers are accessible only in privileged mode (access via MRS and MSR) in word, half word, or byte transfers
- Except the ***Software Trigger Interrupt Register*** **(STIR)**, which can be set up to be accessible in user mode

In addition, a few other interrupt-masking registers are also involved in the interrupt control.

Address	Register
0xE000E100 to 0xE000E11C	Interrupt Set Enable Registers
0xE000E180 to 0xE000E19C	Interrupt Clear Enable Registers
0xE000E200 to 0xE000E21C	Interrupt Set Pending Registers
0xE000E280 to 0xE000E29C	Interrupt Clear Pending Registers
0xE000E300 to 0xE000E31C	Interrupt Active Bit Registers
0xE000E400 to 0xE000E4EF	Interrupt-Priority Registers
0xE000EF00	Software Trigger Interrupt Register

Interrupt Enable and Clear Enable

The Interrupt Enable register is programmed via two addresses.

- To set the enable bit, write 1 to the SETENA register address;
  - Write 1 to set bit to 1; write 0 has no effect; read value indicates the current status
- To clear the enable bit, write 1 to the CLRENA register address.
  - Write 1 to clear bit to 0; write 0 has no effect; read value indicates the current status
- SETENA/CLRENA registers are 32-bit, each bit represents one interrupt input

If there are more than 32 external interrupts (up to 240), there would be more than one SETENA/CLRENA registers.

- SETENA: 0xE000E100--0xE000E11C
- CLRENA: 0xE000E180-0xE000E19C

Interrupt Enable and Clear Enable

Interrupt Set Enable Registers and Interrupt Clear Enable Registers

Name	Type	Address	Reset Value	Description
SETENA0	R/W	0xE000E100	0	Enable for external interrupt #0-31
SETENA1	R/W	0xE000E104	0	Enable for external interrupt #32-63
…	…	…	…	…
SETENA7	R/W	0xE000E11C	0	Enable for external interrupt #224-239
CLRENA0	R/W	0xE000E180	0	Clear Enable for external interrupt #0-31
CLRENA1	R/W	0xE000E184	0	Clear Enable for external interrupt #32-63
…	…	…	…	
CLRENA7	R/W	0xE000E19C	0	Clear Enable for external interrupt #224-239

Interrupt Pending and Clear Pending

The interrupt-pending status can be programmed via two addresses.

- To set the pending bit, write 1 to the Interrupt Set Pending (SETPEND) register;
  - Write 1 to set bit to 1; write 0 has no effect; read value indicates the current status
- To clear the pending bit, write 1 to the Interrupt Clear Pending (CLRPEND) register.
  - Write 1 to clear bit to 0; write 0 has no effect; read value indicates the current status
- SETPEND/ CLRPEND registers are 32-bit, each bit represents one interrupt input
- User can set the certain bit of SETPEND to enter its handler by software.

If there are more than 32 external interrupts (up to 240), there would be more than one SETPEND/ CLRPEND registers.

- SETPEND: 0xE000E200-0xE000E21C
- CLRPEND: 0xE000E280-0xE000E29C

Interrupt Pending and Clear Pending

Interrupt Set Pending Registers and Interrupt Clear Pending Registers

Name	Type	Address	Reset Value	Description
SETPEND0	R/W	0xE000E200	0	Pending for external interrupt #0-31
SETPEND1	R/W	0xE000E204	0	Pending for external interrupt #32-63
…	…	…	…	…
SETPEND7	R/W	0xE000E21C	0	Pending for external interrupt #224-239
CLRPEND0	R/W	0xE000E280	0	Clear pending for external interrupt #0-31
CLRPEND1	R/W	0xE000E284	0	Clear pending for external interrupt #32-63
…	…	…	…	
CLRPEND7	R/W	0xE000E29C	0	Clear pending for external interrupt #224-239

Priority

Each external interrupt has an associated priority-level register (a width of 3-8 bits)

To find out the number of bits implemented for interrupt priority-level registers, write 0xFF to one of the priority-level registers, then read it back and see how many bits are set.

Interrupt Priority-Level Registers (0xE000E400-0xE000E4EF)

Name	Type	Address	Reset Value	Description
PRI_0	R/W	0xE000E400	0 (8-bit)	Priority-level external interrupt #0
PRI_1	R/W	0xE000E401	0 (8-bit)	Priority-level external interrupt #1
…	…	…	…	..
PRI_239	R/W	0xE000E4EF	0 (8-bit)	Priority-level external interrupt #239

Active Status

Each external interrupt has an active status bit.

When the processor starts the interrupt handler, the bit is set to 1 and cleared when the interrupt return is executed.

Interrupt Active Status Registers (0xE000E300-0xE000E31C)

Name	Type	Address	Reset Value	Description
ACTIVE0	R	0xE000E300	0	Active status for external interrupt #0-31
ACTIVE1	R	0xE000E304	0	Active status for external interrupt #32-63
…	…	…	…	…
ACTIVE7	R	0xE000E31C	0	Active status for external interrupt #224-239

Software Interrupts

Software interrupts can be generated by using:

1. The SETPEND register
2. The Software Trigger Interrupt Register (STIR)

Software Trigger Interrupt Register (0xE000EF00)

Bits	Name	Type	Reset Value	Description
8:0	INTID	W	–	Writing the interrupt number sets the pending bit of the interrupt; for example, write 0 to pend external interrupt #0

System exceptions (NMI, faults, PendSV, and so on) cannot be pended using STIR.

STIR can be accessed in user level when the bit 1 (USERSETMPEND) of the NVIC Configuration Control Register is set.

Fault Exceptions: Bus Faults

Bus faults are produced when an error response is received during a transfer on the AHB interfaces.

Bus fault due to:(BFSR Register records the status)

- *Prefetch abort* (Instruction prefetch)
- *Data abort* (data read/write)
- *Stacking error* (stack PUSH in the beginning of interrupt processing)
- *Unstacking error* (stack POP at the end of interrupt processing)
- *Reading of an interrupt vector address error* (when the processor starts the interrupt-handling sequence)

Handling Bus Faults

If

1. The bus fault handler is enabled (BUSFAULTENA bit in the System Handler Control and State register in the NVIC).
2. No other exceptions with the same or higher priority are running.

Then

The bus fault handler will be executed.

Else if

At the same time the core receives another exception handler with higher priority.

Then

The bus fault exception will be pending.

Note: if the bus fault handler is not enabled or when the bus fault happens in an exception handler that has the same or higher priority than the bus fault handler, the hard fault handler will be executed instead

Checking on Bus Faults

The NVIC has a number of fault status registers. One of them is the ***Bus Fault Status Register*** (**BFSR**).

Bus Fault Status Register (0xE000ED29)

Bits	Name	Type	Reset Value	Description
7	BFARVALID	—	0	Indicates BFAR is valid
6:5	—	—	—	—
4	STKERR	R/Wc	0	Stacking error
3	UNSTKERR	R/Wc	0	Unstacking error
2	IMPREISERR	R/Wc	0	Imprecise data access violation
1	PRECISERR	R/Wc	0	Precise data access violation
0	IBUSERR	R/Wc	0	Instruction access violation

Fault Exceptions: Memory Management Faults

Common memory manage faults include:

- *Access to memory regions not defined in MPU setup.*
- *Execute code from nonexecutable memory regions.*
- *Writing to read-only regions.*
- *An access in the user state to a region defined as privileged access only.*

Handling Memory Management Faults

if

1. The memory manage fault handler is enabled (set the MEMFAULTENA bit in the System Handler Control and State register in the NVIC).
2. No other exceptions with the same or higher priority are running.

Then

The memory manage fault handler will be executed.

Else if

At the same time the core receives another exception handler with higher priority.

Then

The memory manage fault exception will be pending.

Note: If the processor is already running an exception handler with same or higher priority or if the memory management fault handler is not enabled, the hard fault handler will be executed instead.

Checking on Memory Management Faults

The NVIC contains a ***Memory Management Fault Status Register*** (**MFSR**) to indicate the cause of the memory management fault.

Memory Management Fault Status Register (0xE000ED28)

Bits	Name	Type	Reset Value	Description
7	MMARVALID	—	0	Indicates the MMAR is valid
6:5	—	—	—	—
4	MSTKERR	R/Wc	0	Stacking error
3	MUNSTKERR	R/Wc	0	Unstacking error
2	—	—	—	—
1	DACCVIOL	R/Wc	0	Data access violation
0	IACCVIOL	R/Wc	0	Instruction access violation

Fault Exceptions: Usage Faults

Usage faults can be caused by:

- *Undefined instructions*
- *Coprocessor instructions* (the Cortex-M3 processor does not support a coprocessor)
- *Trying to switch to the ARM state* (This can happen if you load a new value to PC with the LSB equal to 0)
- *Invalid interrupt return* (Link Register contains invalid/incorrect values)
- *Unaligned memory accesses using multiple load or store instructions*

It is possible, by setting up certain control bits in the NVIC, to generate usage faults for:

1. *Divide by zero*
2. *Any unaligned memory accesses*

Handling Usage Faults

if

1. The usage fault handler is enabled (set the USGFAULTENA bit in the System Handler Control and State register in the NVIC).
2. No other exceptions with the same or higher priority are running.

Then

The usage fault handler will be executed.

Else if

At the same time the core receives another exception handler with higher priority.

Then

The usage fault exception will be pending.

Note: If the processor is already running an exception handler with same or higher priority or if the usage fault handler is not enabled, the hard fault handler will be executed instead.

Checking on Usage Faults

The NVIC provides a ***Usage Fault Status Register*** (**UFSR**) for the usage fault handler to determine the cause of the fault.

Usage Fault Status Register (0xE000ED2A)

Bits	Name	Type	Reset Value	Description
9	DIVBYZERO	R/Wc	0	Indicates a divide by zero has taken place (can be set only if DIV_0_TRP is set)
8	UNALIGNED	R/Wc	0	Indicates that an unaligned access fault has taken place
7:4	—	—	—	—
3	NOCP	R/Wc	0	Attempts to execute a coprocessor instruction
2	INVPC	R/Wc	0	Attempts to do an exception with a bad value in the EXC_RETURN number
1	INVSTATE	R/Wc	0	Attempts to switch to an invalid state (e.g., ARM)
0	UNDEFINSTR	R/Wc	0	Attempts to execute an undefined instruction

Fault Exceptions: Hard Faults

The hard fault handler can be caused by:

1. Usage faults, bus faults, and memory management faults if their handler cannot be executed

2. A bus fault during vector fetch

Hard Fault Status Register (0xE000ED2C)

Bits	Name	Type	Reset Value	Description
31	DEBUGEVT	R/Wc	0	Indicates hard fault is triggered by debug event
30	FORCED	R/Wc	0	Indicates hard fault is taken because of bus fault, memory management fault, or usage fault
29:2	—	—	—	—
1	VECTBL	R/Wc	0	Indicates hard fault is caused by failed vector fetch
0	—	—	—	—

Dealing with Faults

In a real system, after the cause of a fault is determined (use the Fault Status Registers [FSRs]), the software will have to decide what to do next.

If OS exists, the OS could terminates the offending tasks;

Otherwise,

1. **Reset:** Using the VECTRESET control bit in the Application Interrupt and Reset Control register in the NVIC (reset the processor core but not the whole chip [using the SYSRESETREQ instead]).
2. **Recover:** It is possible to resolve the problem that caused the fault exception.

Note: The FSRs retain their status until they are cleared manually by using a write-to-clear mechanism (clear by writing 1 to the bits that need to be cleared) in fault handlers

SVC and PendSV

SVC (System Service Call) and ***PendSV*** (Pended System Call) are two exceptions targeted at software and operating systems.

SVC

SVC is for generating system function calls

SVC is exception type 11, and has a programmable priority level

SVC is generated using the SVC instruction

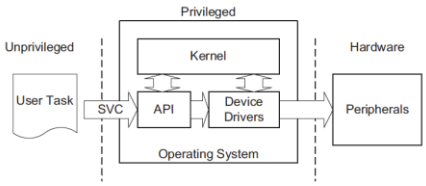
```
SVC   #0x3   ; Call SVC function 3
```

SVC cannot be pended (The SVC handler must be executed after the SVC instruction)

Example:

- Instead of allowing user programs to directly access hardware, operating system may provide access to hardware via an SVC.
- SVC can make software more portable because the user application does not need to know the programming details of the hardware.

SVC



SVC as a Gateway for OS Functions

Note: if the SVC instruction is accidentally used in an exception handler that has the same or higher priority than the SVC exception itself, it will cause the HardFault exception handler to execute. For example, you cannot use SVC inside an SVC handler.

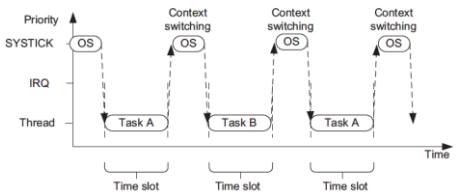
PendSV

PendSV (Pended System Call) works with SVC in the OS.

PendSV is generated by setting its pending status by writing to the Interrupt Control and State Register (ICSR)

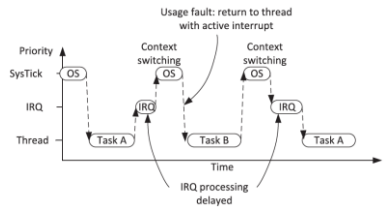
Example: A typical use of PendSV is context switching (switching between tasks).

- A system with only two tasks, and a context switch can be triggered by SYSTICK exceptions.



A Simple Scenario Using SYSTICK to Switch Between Two Tasks

Problems at Context Switching at the IRQ

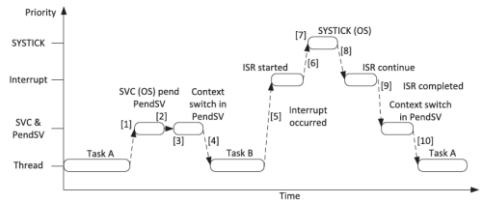


If an interrupt request takes place before the SYSTICK exception, the SYSTICK exception will preempt the IRQ handler. ***What's the problem?***

One solution: context switching can happen only if the OS detects that none of the IRQ handlers are being executed. ***What's the problem?***

Using PendSV for Context Switching

Using PendSV which is programmed at lowest priority level



1. Task A calls SVC for task switching (for example, waiting for some work to complete).
6. While running the interrupt handler routine, a SYSTICK exception (for OS tick) takes place.
7. The OS carries out the essential operation, then pends the PendSV exception and gets ready for the context switch.

PRIMASK and FAULTMASK Special Registers

The PRIMASK register disables all exceptions except NMI and hard fault by changing the current priority level to 0

- When PRIMASK is set, if a fault takes place, the hard fault handler will be executed)

The FAULTMASK register changes the effective current priority level to -1 so that even the hard fault handler is blocked

- FAULTMASK is cleared automatically upon exiting the exception handler.

They are programmable using MRS and MSR instructions in privileged modes.

Example:

```
MOV R0, #1
MSR PRIMASK, R0 ; Write 1 to PRIMASK to disable all interrupts

MOV R0, #0
MSR PRIMASK, R0 ; Write 0 to PRIMASK to allow interrupts
```

The BASEPRI Special Register

The BASEPRI register disables interrupts with priority lower than a certain level.

Example:

```
MOV R0, #0x60
MSR BASEPRI, R0 ;Disable interrupts with priority 0x60-0xFF
```

Write 0 to BASEPRI to turn off the masking:

```
MOV R0, #0x0
MSR BASEPRI, R0 ; Turn off BASEPRI masking
```

The BASEPRI register can also be accessed using the BASEPRI_MAX register name.

The BASEPRI Special Register

Using BASEPRI_MAX as a register, it can only be changed to a higher priority level.

Example:

```
MOV   R0, #0x60
MSR   BASEPRI_MAX, R0 ; Disable interrupts with priority 0x60,
                      ; 0x61,..., etc
MOV   R0, #0xF0
MSR   BASEPRI_MAX, R0 ; This write will be ignored because
                      ; it is lower level than 0x60
MOV   R0,   #0x40
MSR   BASEPRI_MAX, R0 ; This write is allowed and change
                      ; the masking level to 0x40
```

To change to a lower masking level or disable the masking, the BASEPRI register name should be used

Configuration Registers for Other Exceptions

Usage faults, memory management faults, and bus fault exceptions are enabled by the ***System Handler Control and State Register***.

Table 8.5 The System Handler Control and State Register (0xE000ED24)

Bits	Name	Type	Reset Value	Description
18	USGFAULTENA	R/W	0	Usage fault handler enable
17	BUSFAULTENA	R/W	0	Bus fault handler enable
16	MEMFAULTENA	R/W	0	Memory management fault enable
15	SVCALLPENDED	R/W	0	SVC pended; SVCall was started but was replaced by a higher-priority exception
14	BUSFAULTPENDED	R/W	0	Bus fault pended; bus fault handler was started but was replaced by a higher-priority exception
13	MEMFAULTPENDED	R/W	0	Memory management fault pended; memory management fault started but was replaced by a higher-priority exception
12	USGFAULTPENDED	R/W	0	Usage fault pended; usage fault started but was replaced by a higher-priority exception
11	SYSTICKACT	R/W	0	Read as 1 if SYSTICK exception is active
10	PENDSVACT	R/W	0	Read as 1 if PendSV exception is active
8	MONITORACT	R/W	0	Read as 1 if debug monitor exception is active
7	SVCALLACT	R/W	0	Read as 1 if SVCall exception is active
3	USGFAULTACT	R/W	0	Read as 1 if usage fault exception is active
1	BUSFAULTACT	R/W	0	Read as 1 if bus fault exception is active
0	MEMFAULTACT	R/W	0	Read as 1 if memory management fault is active

Note: Bit 12 (USGFAULTPENDED) is not available on revision 0 of Cortex-M3.

Interrupt Control and State Register (0xE000ED04)

Pending for NMI, the SYSTICK timer, and PendSV is programmable via the ***Interrupt Control and State Register***

Table 8.6 Interrupt Control and State Register (0xE000ED04)

Bits	Name	Type	Reset Value	Description
31	NMIPENDSET	R/W	0	NMI pended
28	PENDSVSET	R/W	0	Write 1 to pend system call Read value indicates pending status
27	PENDSVCLR	W	0	Write 1 to clear PendSV pending status
26	PENDSTSET	R/W	0	Write 1 to pend SYSTICK exception Read value indicates pending status
25	PENDSTCLR	W	0	Write 1 to clear SYSTICK pending status
23	ISRPREEMPT	R	0	Indicates that a pending interrupt is going to be active in the next step (for debug)
22	ISRPENDING	R	0	External interrupt pending (excluding system exceptions such as NMI for fault)
21:12	VECTPENDING	R	0	Pending ISR number
11	RETTOBASE	R	0	Set to 1 when the processor is running an exception handler; will return to Thread level if interrupt return and no other exceptions pending
9:0	VECTACTIVE	R	0	Current running interrupt service routine

Example Procedures in Setting Up an Interrupt

1. Set up the priority group register (group 0 by default).
2. *Setup the hard fault and NMI handlers to a new vector table location if vector table relocation is required.*
3. *Set up the Vector Table Offset register if needed.*
4. Set up the interrupt vector for the interrupt：[read the Vector Table Offset register and] calculate the correct memory location for the interrupt handler.
5. Set up the priority level for the interrupt.
6. Enable the interrupt.

Example in assembly:

```
LDR R0, =0xE000ED0C ; Application Interrupt and Reset
                    ; Control Register (see Table 7.5)
LDR R1, =0x05FA0500 ; Priority Group 5 (2/6)
STR R1, [R0]        ; Set Priority Group
```

Example Procedures in Setting Up an Interrupt

```
...
MOV R4, #8           ; Vector Table in ROM
LDR R5, =(NEW_VECT_TABLE+8)
LDMIA R4!, {R0-R1}   ; Read vectors address for NMI and Hard Fault
STMIA R5!, {R0-R1}   ; Copy vectors to new vector table
...
LDR R0, =0xE000ED08  ; Vector Table Offset Register
LDR R1, =NEW_VECT_TABLE
STR R1, [R0]         ; Set vector table to new location
...
LDR R0, =IRQ7_Handler ; Get starting address of IRQ#7 handler
LDR R1, =0xE000ED08  ; Vector Table Offset Register
LDR R1, [R1]
ADD R1, R1, #(4*(7+16)) ; Calculate IRQ#7 handler vector address
STR R0, [R1]         ; Setup vector for IRQ#7
...
LDR R0, =0xE000E400 ; External IRQ priority base
```

Example Procedures in Setting Up an Interrupt

```
MOV R1, #0xC0
STRB R1, [R0,#7]     ; Set IRQ#7 priority to 0xC0
...
LDR R0, =0xE000E100  ; SETEN register
MOV R1, #(1<<7)      ; IRQ#7 enable bit (value 0x1 shifted by 7 bits)
STR R1, [R0]         ; Enable the interrupt
```

Interrupt Behavior

When an exception takes place, a number of things happen:

- Stacking (pushing eight registers' contents to stack)
- Vector fetch (reading the exception handler starting address from the vector table)
- Update of the stack pointer, link register, and program counter

Stacking

When an exception takes place, the registers PC, PSR, R0–R3, R12, and LR are pushed to the stack

If the code that is running uses the PSP, the process stack will be used; if the code that is running uses the MSP, the main stack will be used

The main stack will always be used during the handler

Address	Data	Push Order
Old SP (N) ->	(Previously pushed data)	-
(N-4)	PSR	2
(N-8)	PC	1
(N-12)	LR	8
(N-16)	R12	7
(N-20)	R3	6
(N-24)	R2	5
(N-28)	R1	4
New SP (N-32) ->	R0	3

Vector Fetches

While the data bus is busy stacking the registers, the instruction bus fetches the exception vector (the starting address of the exception handler) from the vector table.

Since the stacking and vector fetch are performed on separate bus interfaces, they can be carried out at the same time

Register Updates

After the stacking and vector fetch are completed, the exception vector will start to execute.

On entry of the exception handler, a number of registers will be updated:

- **SP**: The Stack Pointer (either the MSP or the PSP)
- **PSR**: The IPSR (the lowest part of the PSR) will be updated to the new exception number
- **PC**: This will change to the vector handler as the vector fetch completes
- **LR**: The LR will be updated to a special value called EXC_RETURN, used to store the states of the interrupted program (i.e., which stack used, instruction state, thread/handler mode)

LR Update

When entering an exception handler, the LR is updated to a special value called EXC_RETURN (updated automatically)

Bits	31:4	3	2	1	0
Descriptions	0xFFFFFFF	Return mode (Thread/handler)	Return stack	Reserved; must be 0	Process state (Thumb/ARM)

Value	Condition
0xFFFFFFF1	Return to handler mode
0xFFFFFFF9	Return to Thread mode and on return use the main stack
0xFFFFFFFD	Return to Thread mode and on return use the process stack

Therefore, the 0xFFFFFFF0–0xFFFFFFFF memory range is reserved only for EXC_RETURN (it is in a non-executable region anyway)

Exception Exits

At the end of the exception handler, an exception exit is required to restore the system status so that the interrupted program can resume normal execution.

When the EXC_RETURN is loaded into the PC at the end of the exception handler execution, the processor performs an exception return sequence

There are three ways to trigger the interrupt return sequence:

Return Instruction	Description
BX <*reg*>	If the EXC_RETURN value is still in LR, we can use the *BX LR* instruction to perform the interrupt return.
POP {PC}, or POP {...., PC}	Very often the value of LR is pushed to the stack after entering the exception handler. We can use the POP instruction, either a single POP or multiple POPs, to put the EXC_RETURN value to the program counter. This will cause the processor to perform the interrupt return.
LDR, or LDM	It is possible to produce an interrupt return using the LDR instruction with PC as the destination register.

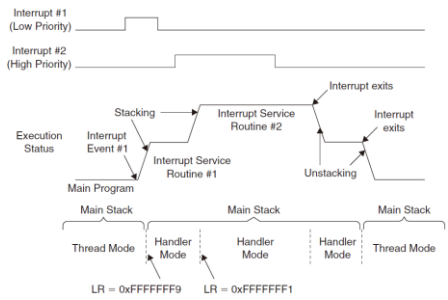
Exception Exits

When the interrupt return instruction is executed, the following processes are carried out:

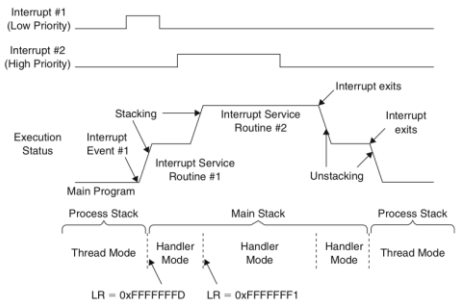
Un-stacking: The registers pushed to the stack will be restored

NVIC register update: The active bit of the exception will be cleared. For external interrupts, if the interrupt input is still asserted, the pending bit will be set again, causing it to reenter the interrupt handler

LR Set to EXC_RETURN Example 1



LR Set to EXC_RETURN Example 2



Nested Interrupts

Being built into the Cortex-M3 processor core and the NVIC :

- When the processor is handling an exception, all other exceptions with the same or lower priority will be blocked (pended)
- The automatic hardware stacking and un-stacking allow the nested interrupt handler to execute without risk of losing data in registers

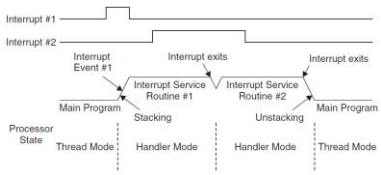
Be Aware of:

- Make sure that there is enough space in the main stack if lots of nested interrupts are allowed
- Reentrant exceptions are not allowed in the Cortex-M3, e.g., SVC instructions cannot be used inside an SVC handler

Tail-Chaining Interrupts

Used to improve interrupt latency:

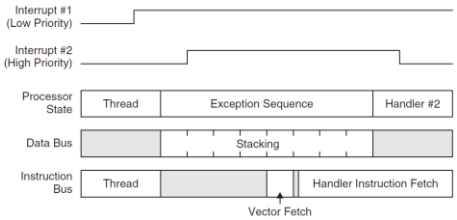
- When the processor is handling an exception, all other exceptions with the same or lower priority will be blocked (pended)
- When the processor has finished executing the current exception handler, the un-stacking for this handler and the stacking for the next handler are skipped (reducing the latency of 12 cycles to 6)



Late Arrival Exception Handling

Used to improve interrupt latency:

- When an exception takes place and the processor has started the stacking process, if a new exception arrives with higher preemption priority, the late arrival exception will be processed first.



Faults Related to Interrupts

Stacking faults: a bus fault exception will be triggered or pended

Un-stacking faults: a bus fault exception will be triggered or pended

Vector fetch faults: the hard fault handler will be executed

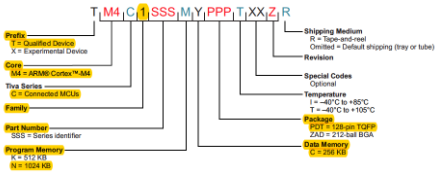
Invalid return faults: If the EXC_RETURN number is invalid or does not match the state of the processor, the usage fault handler will be triggered or pended

The Cortex-M3/M4 Embedded Systems:

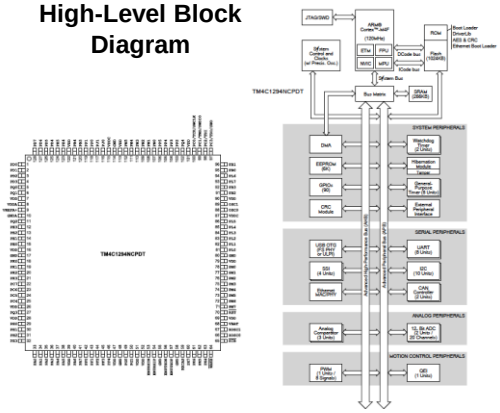
Cortex-M4 Implementation

Chapter 1, 2, 3, 4 in the reference book
"Tiva™ TM4C1294NCPDT Microcontroller DATA SHEET"

Case Study: An Implementation of Cortex-M4 – Tiva™ C Series TM4C1294NCPDT



High-Level Block Diagram



TM4C1294NCPDT Microcontroller Features

Made by Texas Instruments

Tiva™ C Series is a microcontroller family based on ARM Cortex-M4 with FPU and the Thumb-2 instruction set

The TM4C1294NCPDT microcontroller has the following features:

- ARM Cortex-M4F Processor Core
- On-Chip Memory
- External Peripheral Interface (EPI)
- Cyclical Redundancy Check (CRC)
- Serial Communications Peripherals
- System Integration
- Advanced Motion Control
- Analog
- JTAG and ARM Serial Wire Debug (SWD)
- Packaging and Temperature

ARM Cortex-M4F Processor Core

Processor Core

- 120-MHz operation; 150 DMIPS performance
- Thumb-2 mixed 16-/32-bit instruction set: high performance in a compact memory size
  - Single-cycle multiply instruction and hardware divide operation
  - Atomic bit manipulation (bit-banding)
  - Unaligned data access
- The processor accesses code and data in little-endian format
- Single-precision FPU &16-bit SIMD vector processing unit
- Harvard architecture: separate buses for instruction and data
- Deterministic, high-performance interrupt handling for time-critical apps.
- Memory protection unit (MPU) to provide a privileged mode
- Enhanced system debug with extensive breakpoint and trace capabilities
- Ultra-low power consumption with integrated sleep modes

ARM Cortex-M4F Processor Core

System Timer (SysTick)

- a simple, 24-bit, clear-on-write, decrementing, wrap-on-zero counter
- The counter can be used in several ways:
  - An RTOS tick timer that fires at a programmable rate and invokes a SysTick exception handler
  - A high-speed alarm timer using the system clock
  - A simple counter used to measure time to completion and time used

Nested Vectored Interrupt Controller (NVIC)

- Automatic stacking and unstacking of the processor state on an exception
- **Vectored:** efficient interrupt entry with the interrupt vector table
- **Nested:** 8 priority levels on 7 exceptions (system handlers) and 106 interrupts
- **Deterministic and fast interrupt processing:** always 12 cycles, or just 6 cycles with tail-chaining
- NMI
- You can only fully access the NVIC from privileged mode except the STIR (used to pend interrupts in user-mode by software) when the bit 1 (USERSETMPEND) of the NVIC Configuration Control Register is set

On-Chip Memory

SRAM

- 256 KB of single-cycle on-chip SRAM, located at offset 0x2000.0000
- A bit-band region in SRAM
- Data can be transferred to and from SRAM by bus masters
  - Micro Direct Memory Access controller (µDMA)
  - USB
  - Ethernet controller

Flash Memory

- 1024 KB of single-cycle on-chip Flash memory
- Organized as 4 banks of 16K x 128 bits
  - Different levels of code protection: read-only, execute-only (read by instruction fetch mechanism)

On-Chip Memory

ROM

preprogrammed with the following software and programs:

- TivaWare Peripheral Driver Library
- TivaWare Boot Loader
- SafeRTOS™ preemptive real-time kernel
- Advanced Encryption Standard (AES) cryptography tables
- Cyclic Redundancy Check (CRC) error-detection functionality

EEPROM

6Kbytes

External Peripheral Interface (EPI)

The External Peripheral Interface (EPI) provides access to external devices using a parallel path, like a bus

- 8/16/32-bit dedicated parallel bus for external peripherals and memory
- Memory interface supports contiguous memory access independent of data bus width
- Separates processor from timing details through use of an internal write FIFO
- Efficient transfers using Micro Direct Memory Access Controller (µDMA)
- Supports three primary functional modes: SDRAM, Host-bus, and General Purpose

Serial Communications Peripherals

The LM3S9B96 controller supports both asynchronous and synchronous serial communications

- 10/100 Ethernet MAC and PHY
- 2 CAN 2.0 A/B Controllers
- USB 2.0 (full speed and low speed) OTG/Host/Device
- 8 UARTs
- 10 I^2C modules
- 4 QuadSynchronous Serial Interface modules (QSSI)

System Integration

The TM4C1294NCPDT controller provides a variety of standard system functions integrated into the device:

- Micro Direct Memory Access Controller (µDMA)
- System control and clocks including on-chip precision 16-MHz oscillator
- Eight 32-bit timers (each of which can be configured as two 16-bit timers)
- Lower-power battery-backed Hibernation module
- Real-Time Clock
- Two Watchdog Timers
- Up to 90 GPIOs

Advanced Motion Control

The LM3S9B96 controller provides motion control functions integrated into the device, including:

Key features:

- Eight advanced PWM outputs for motion and energy applications
- One Quadrature Encoder Inputs (QEI)

Analog

The LM3S9B96 controller provides analog functions integrated into the device

- Two 12-bit Analog-to-Digital Converters (ADC) with 20 analog input channels and sample rate of two million samples/second
- Three analog comparators
- On-chip voltage regulator

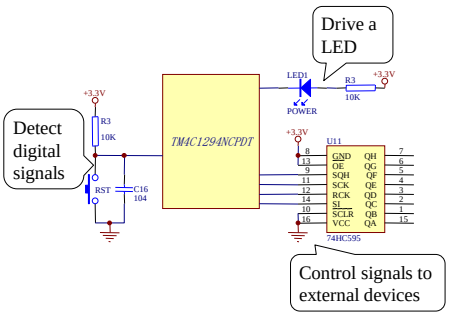
The Cortex-M3/M4 Embedded Systems:

Tiva™ TM4C1294NCPDT Microcontroller – General-Purpose Input/Outputs

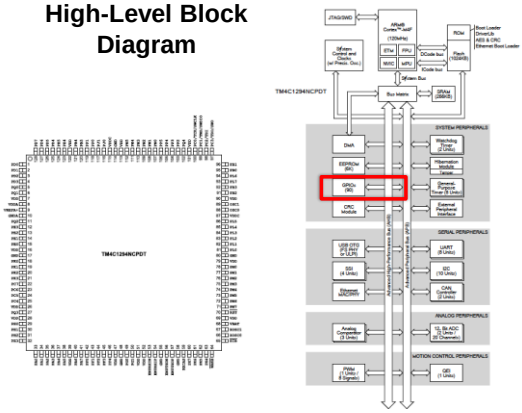
Refer to Chapter 10 in the reference book

"Tiva™ TM4C1294NCPDT Microcontroller - DATA SHEET"

What is General-Purpose Input/Output?



High-Level Block Diagram

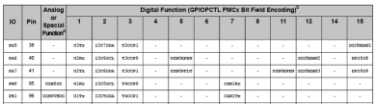


GPIO Module

15 physical GPIO blocks, corresponding to 15 GPIO ports
Supports up to 90 programmable input/output pins
Highly flexible pin muxing
Via AHB
Programmable control for GPIO interrupts: masking, condition
Bit masking in both read and write operations
Programmable control for GPIO pad configuration
- Weak pull-up or pull-down resistors
- 2-, 4-, 8-, 10-, 12-mA pad drive for digital communication
- Slew rate control for the 8-, 10-, 12mA drive
- Open drain

Pin Muxing

GPIO pins may convey analog and digital signals



Important: The table below shows special consideration GPIO pins. Most GPIO pins are configured as GPIOs and tri-stated by default (**GPIOAFSEL**=0, **GPIODEN**=0, **GPIOPDR**=0, **GPIOPUR**=0, and **GPIOPCTL**=0). Special consideration pins may be programed to a non-GPIO function or may have special commit controls out of reset. In addition, a Power-On-Reset (POR) returns these GPIO to their original special consideration state.

GPIO Pins	Default Reset State	GPIOAFSEL	GPIODEN	GPIOPDR	GPIOPUR	GPIOPCTL	GPIOCR
PC[3:0]	JTAG/SWD	1	1	0	1	0x1	0
PD[7]	GPIO[a]	0	0	0	0	0x0	0
PE[7]	GPIO[a]	0	0	0	0	0x0	0

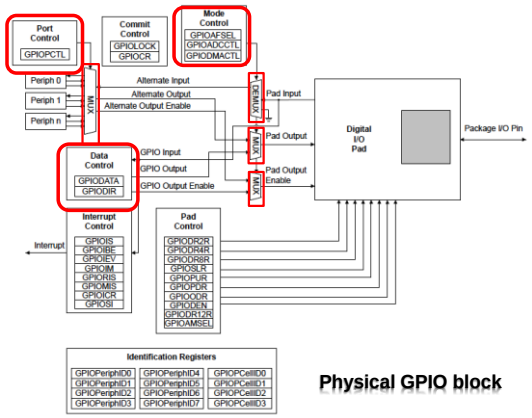
a. This pin is configured as a GPIO by default but is locked and can only be reprogrammed by unlocking the pin in the **GPIOLOCK** register and uncommitting it by setting the **GPIOCR** register.

Functional Description

One physical block for each port (PA, PB … PP, PQ)

For digital signals, GPIO pins can be controlled by GPIO or other on-chip peripherals (i.e., pin muxing) :

- GPIO (software): programming on GPIO data register and direction control register to control the pin
- Other peripherals (hardware): some GPIO pins can function as I/O signals for the on-chip peripheral modules
- Mode selection: controlled by the **GPIO Alternate Function Select (GPIOAFSEL)** register: 0 means GPIO and 1 means other peripherals
- Peripheral selection: provided through the **GPIO Port Control (GPIOPCTL)** register which selects one of several peripheral functions for each muxed pin



Physical GPIO block

Mode Control

The GPIO pins can be controlled by either GPIO or other on-chip peripherals

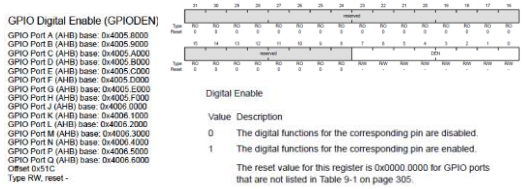
GPIO is the default for most GPIO pins, where the GPIODATA register is used to read or write the corresponding pins

When hardware control is enabled via the **GPIO Alternate Function Select (GPIOAFSEL)** register, the pin state is controlled by one of its alternate functions (that is, one of the connected peripherals)

Further pin muxing options are provided through the **GPIO Port Control (GPIOPCTL)** register (i.e., to select which peripheral)

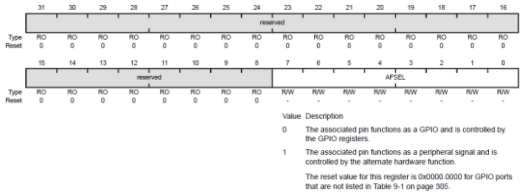
Register 18: GPIO Digital Enable (GPIODEN), offset 0x51C

The **GPIODEN** register is the digital enable register. By default, all GPIO signals except those listed below are configured out of reset to be undriven (tristate). Their digital function is disabled. To use the pin as a digital input or output (either GPIO or alternate functions), the corresponding GPIODEN bit must be set.



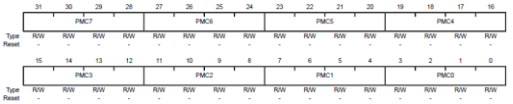
Register 10: GPIO Alternate Function Select (GPIOAFSEL), offset 0x420

The **GPIOAFSEL** register is the mode control select register. If a bit is clear, the pin is used as a GPIO and is controlled by the GPIO registers. Setting a bit in this register configures the corresponding GPIO line to be controlled by an associated peripheral. Several possible peripheral functions are multiplexed on each GPIO. The **GPIO Port Control (GPIOPCTL)** register is used to select one of the possible functions.



Register 22: GPIO Port Control (GPIOPCTL), offset 0x52C

The **GPIOPCTL** register is used in conjunction with the **GPIOAFSEL** register and selects the specific peripheral signal for each GPIO pin when using the alternate function mode. Most bits in the **GPIOAFSEL** register are cleared on reset, therefore most GPIO pins are configured as GPIOs by default. For information on the defined encodings for the bit fields in this register, refer to Table 26-5 on page 1808.



GPIO Mode: Data Control

Data direction operation

- The **GPIO Direction** (**GPIODIR**) register is used to configure each individual pin as an input or output
- If cleared (i.e., 0), input: the corresponding data register bit captures and stores the value on the GPIO pin; otherwise, output: the data bit is driven out on the pin

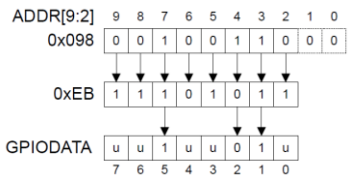
Data register operation

- Modification of individual bits in the **GPIO Data** (**GPIODATA**) register is allowed
- Using bits[9:2] of the address bus as a mask and, therefore, consume 256 locations for operating GPIODATA

Write Operation to GPIODATA

If the address bit associated with that data bit is set, the value of the **GPIODATA** register is altered

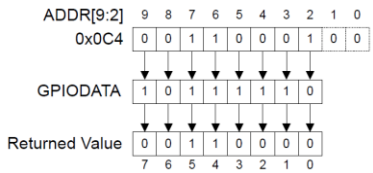
- For example, writing a value of 0xEB to the address GPIODATA + 0x098



Read Operation from GPIODATA

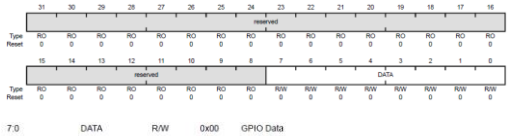
If the address bit associated with that data bit is set, the value of the **GPIODATA** register is read; otherwise, RAZ

For example, reading address GPIODATA + 0x0C4



Register 1: GPIO Data (GPIODATA), offset 0x000

The **GPIODATA** register is the data register. In software control mode, values written in the **GPIODATA** register are transferred onto the GPIO port pins if the respective pins have been configured as outputs through the **GPIO Direction (GPIODIR)** register.

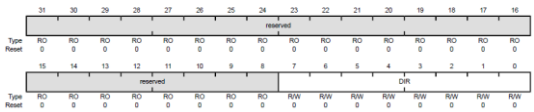


Bit/Field	Name	Type	Reset	Description
7:0	DATA	R/W	0x00	GPIO Data

This register is virtually mapped to 256 locations in the address space. To facilitate the reading and writing of data to these registers by independent drivers, the data read from and written to the registers are masked by the eight address lines [9:2]. Reads from this register return its current state. Writes to this register only affect bits that are not masked by ADDR[9:2] and are configured as outputs. See "Data Register Operation" on page 311 for examples of reads and writes.

Register 2: GPIO Direction (GPIODIR), offset 0x400

The **GPIODIR** register is the data direction register. Setting a bit in the **GPIODIR** register configures the corresponding pin to be an output, while clearing a bit configures the corresponding pin to be an input. All bits are cleared by a reset, meaning all GPIO pins are inputs by default.



Bit/Field	Name	Type	Reset	Description
7:0	DIR	R/W	0x00	GPIO Data Direction

Value	Description
0	Corresponding pin is an input.
1	Corresponding pins is an output.

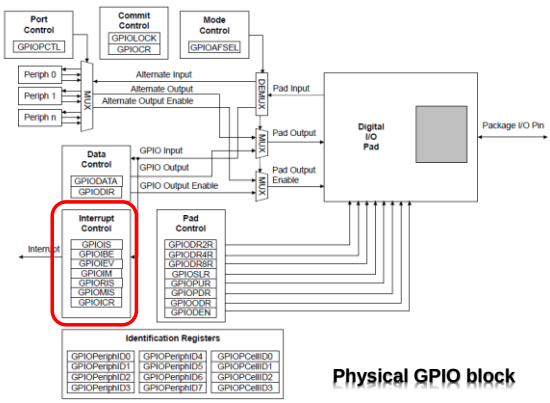
Interrupt Control

The interrupt capabilities of each GPIO port are controlled by a set of seven registers

When one or more GPIO inputs cause an interrupt, a single interrupt output is sent to the NVIC for the entire GPIO port

For edge-triggered interrupts, software must clear the interrupt to enable any further interrupts

For a level-sensitive interrupt, the external source must hold the level constant for the interrupt to be recognized by the controller



Physical GPIO block

Define Interrupt Conditions

GPIO Interrupt Sense (GPIOIS) register: setting a bit, detect levels on the pin; otherwise, detect edges

GPIO Interrupt Both Edges (GPIOIBE) register: when GPIOIS is set to detect edges, setting a bit in GPIOIBE enables the pin to detect both rising and falling edges; otherwise, the pin is controlled by the GPIOIEV register

GPIO Interrupt Event (GPIOIEV) register: setting a bit, detect rising edges (or high levels); otherwise, detect falling edges (or low levels), depending on the settings of GPIOIS

Other Interrupt Control Registers

GPIO Interrupt Mask (GPIOIM) register: setting a bit, allows the pin to generate interrupts; otherwise, disable interrupts

GPIO Raw Interrupt Status (GPIORIS) register: A bit is set when an interrupt condition occurs on the corresponding GPIO pin; otherwise, RAZ.

GPIO Masked Interrupt Status (GPIOMIS) register: If a bit is set, the corresponding interrupt has triggered an interrupt to the interrupt controller; otherwise, either no interrupt has been generated, or the interrupt is masked

GPIO Interrupt Clear (GPIOICR) register: Writing a 1 to a bit in this register clears the corresponding interrupt bit in the GPIORIS and GPIOMIS registers

Register 3: GPIO Interrupt Sense (GPIOIS), offset 0x404

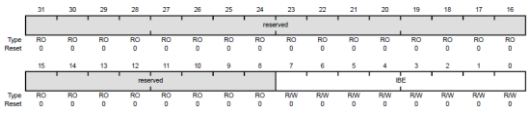
The **GPIOIS** register is the interrupt sense register. Setting a bit in the **GPIOIS** register configures the corresponding pin to detect levels, while clearing a bit configures the corresponding pin to detect edges. All bits are cleared by a reset.

7:0	IS	R/W	0x00	GPIO Interrupt Sense

Value	Description
0	The edge on the corresponding pin is detected (edge-sensitive).
1	The level on the corresponding pin is detected (level-sensitive).

Register 4: GPIO Interrupt Both Edges (GPIOIBE), offset 0x408

The **GPIOIBE** register allows both edges to cause interrupts. When the corresponding bit in the **GPIOIS** register (see page 320) is set to detect edges, setting a bit in the **GPIOIBE** register configures the corresponding pin to detect both rising and falling edges. Clearing a bit configures the pin to be controlled by the **GPIOIEV** register. All bits are cleared by a reset.

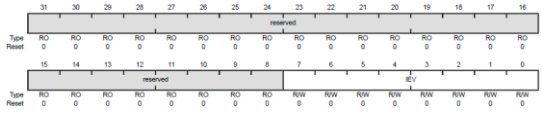


7:0	IBE	R/W	0x00	GPIO Interrupt Both Edges

Value	Description
0	Interrupt generation is controlled by the **GPIO Interrupt Event (GPIOIEV)** register (see page 322).
1	Both edges on the corresponding pin trigger an interrupt.

Register 5: GPIO Interrupt Event (GPIOIEV), offset 0x40C

Setting a bit in the **GPIOIEV** register configures the corresponding pin to detect rising edges or high levels (clearing a bit configures the pin to detect falling edges or low levels), depending on the corresponding bit value in the **GPIO Interrupt Sense (GPIOIS)** register. All bits are cleared by a reset.

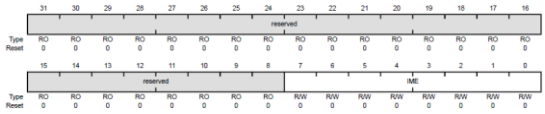


7:0	IEV	R/W	0x00	GPIO Interrupt Event

Value	Description
0	A falling edge or a Low level on the corresponding pin triggers an interrupt.
1	A rising edge or a High level on the corresponding pin triggers an interrupt.

Register 6: GPIO Interrupt Mask (GPIOIM), offset 0x410

The **GPIOIM** register is the interrupt mask register. Setting a bit in the **GPIOIM** register allows interrupts that are generated by the corresponding pin to be sent to the interrupt controller on the combined interrupt signal. All bits are cleared by a reset.

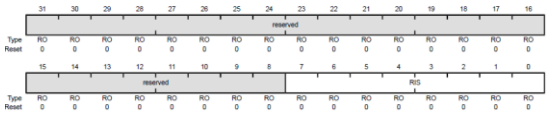


7:0	IME	R/W	0x00	GPIO Interrupt Mask Enable

Value	Description
0	The interrupt from the corresponding pin is masked.
1	The interrupt from the corresponding pin is sent to the interrupt controller.

Register 7: GPIO Raw Interrupt Status (GPIORIS), offset 0x414

The **GPIORIS** register is the raw interrupt status register. A bit in this register is set when an interrupt condition occurs on the corresponding GPIO pin. A bit in this register can be cleared by writing a 1 to the corresponding bit in the **GPIO Interrupt Clear (GPIOICR)** register.



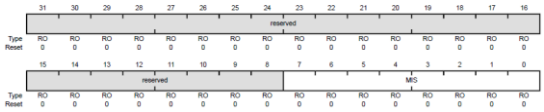
7:0	RIS	RO	0x00	GPIO Interrupt Raw Status

Value	Description
1	An interrupt condition has occurred on the corresponding pin.
0	An interrupt condition has not occurred on the corresponding pin.

A bit is cleared by writing a 1 to the corresponding bit in the **GPIOICR** register.

Register 8: GPIO Masked Interrupt Status (GPIOMIS), offset 0x418

The **GPIOMIS** register is the masked interrupt status register. If a bit is set in this register, the corresponding interrupt has triggered an interrupt to the interrupt controller.



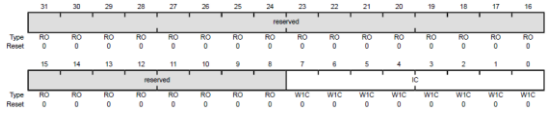
7:0 MIS RO 0x00 GPIO Masked Interrupt Status

Value	Description
1	An interrupt condition on the corresponding pin has triggered an interrupt to the interrupt controller.
0	An interrupt condition on the corresponding pin is masked or has not occurred.

A bit is cleared by writing a 1 to the corresponding bit in the **GPIOICR** register.

Register 9: GPIO Interrupt Clear (GPIOICR), offset 0x41C

The **GPIOICR** register is the interrupt clear register. Writing a 1 to a bit in this register clears the corresponding interrupt bit in the **GPIORIS** and **GPIOMIS** registers. Writing a 0 has no effect.



7:0 IC W1C 0x00 GPIO Interrupt Clear

Value	Description
1	The corresponding interrupt is cleared.
0	The corresponding interrupt is unaffected.

Pad Control

The pad control registers allow software to configure the GPIO pads based on the application requirements

e.g., drive strength, open-drain configuration, pull-up and pull-down resistors, slew-rate control and digital input enable

The **GPIODR2R**, **GPIODR4R** and **GPIODR8R** registers: setting a bit, the corresponding GPIO pin has 2-, 4-, and 8-mA drive

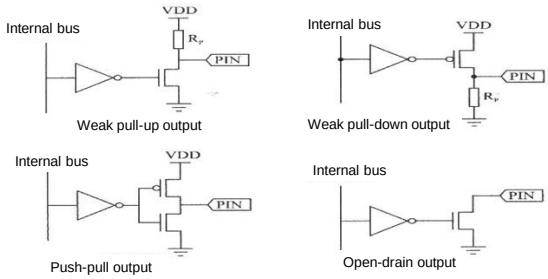
GPIO Open Drain Select (GPIOODR) register: setting a bit, the pin is configured as open drain

GPIO Pull-Up Select (GPIOPUR) register: setting a bit, a weak pull-up resistor on the pin is enabled

GPIO Pull-Down Select (GPIOPDR) register: setting a bit, a weak pull-down resistor on the pin is enabled

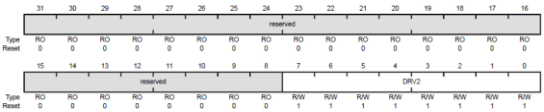
GPIO Slew Rate Control Select (GPIOSLR) register: only available when using 8-mA drive; setting a bit, enables the slew rate control for the pin

Extend drive enable: how to get 6-, 10-, 12-mA drive? See details about GPIO Peripheral Property Register and GPIO Peripheral Configuration Register



Register 11: GPIO 2-mA Drive Select (GPIODR2R), offset 0x500

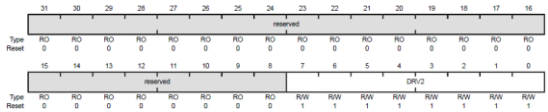
Each GPIO signal in the port can be individually configured without affecting the other pads. When setting the DRV2 bit for a GPIO signal, the corresponding DRV4 bit in the **GPIODR4R** register and DRV8 bit in the **GPIODR8R** register are automatically cleared by hardware. By default, all GPIO pins have 2-mA drive.



Register 12: GPIO 4-mA Drive Select (GPIODR4R), offset 0x504
Register 13: GPIO 8-mA Drive Select (GPIODR8R), offset 0x508

Register 11: GPIO 2-mA Drive Select (GPIODR2R), offset 0x500

Each GPIO signal in the port can be individually configured without affecting the other pads. When setting the DRV2 bit for a GPIO signal, the corresponding DRV4 bit in the **GPIODR4R** register and DRV8 bit in the **GPIODR8R** register are automatically cleared by hardware. By default, all GPIO pins have 2-mA drive.



Register 12: GPIO 4-mA Drive Select (GPIODR4R), offset 0x504
Register 13: GPIO 8-mA Drive Select (GPIODR8R), offset 0x508
Register 14: GPIO Open Drain Select (GPIOODR), offset 0x50C
Register 15: GPIO Pull-Up Select (GPIOPUR), offset 0x510
Register 16: GPIO Pull-Down Select (GPIOPDR), offset 0x514

Initialization and Configuration

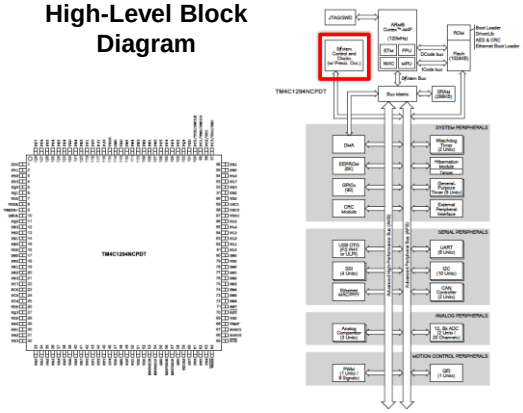
1. To use the pins in a particular GPIO port, enable the clock to the port by setting the appropriate bits in the **RCGCGPIO** register (see page 382). In addition, the SCGCGPIO and DCGCGPIO registers can be programmed in Sleep and Deep-Sleep modes.
2. Set up the individual GPIO pins for your purpose:
   - Set up **GPIODEN**, if the pin is to used for digital signals
   - Set **GPIOAFSEL** (and **GPIOPCTL** if needed)
   - Set interrupt, direction, data, pad control etc.
3. See Chapter 10.4 for details.

The Cortex-M3/M4 Embedded Systems:

Tiva™ TM4C1294NCPDT Microcontroller – System Control

Refer to Chapter 5 in the reference book
"Tiva™ TM4C1294NCPDT Microcontroller - DATA SHEET"

High-Level Block Diagram



System Control

System control configures the overall operation of the device and provides information about the device.

- **reset control**
- NMI operation
- **power control**
- **clock control**
- low-power modes

Reset Sources

The TM4C1294NCPDT microcontroller has 8 sources of reset

- Power-on reset (POR)
  - The internal POR circuit monitors the power supply voltage VDD and generates a reset signal to all of the internal logic including JTAG when the power supply ramp reaches a threshold value.
- External reset input pin (RST) assertion
  - The external reset pin (RST) resets the microcontroller including the core and all the on-chip peripherals except the JTAG TAP controller
- Internal brown-out (BOR) detector
  - brown-out detection circuit that triggers if the power supply VDD drops below a brown-out threshold voltage

Reset Sources

- Software-initiated reset
  - The entire microcontroller including the core can be reset by software by setting the SYSRESETREQ bit in the Cortex-M4 Application Interrupt and Reset Control register
  - On-chip peripherals can be individually reset by software via three registers (see the SRCRn registers)
- A watchdog timer reset condition violation
  - A watchdog timer can be configured to generate an interrupt to the microcontroller on its first time-out and to generate a reset on its second time-out
- MOSC failure
  - The TM4C1294NCPDT microcontroller provides a main oscillator verification circuit that generates an error condition if the oscillator is running too fast or too slow

Reset Sources

Hibernation module event

When the Hibernation module has been configured and powered by an initial "cold" POR and is subsequently put into hibernation mode, a wake event causes the module to generate a system reset.

On-chip peripherals can be individually reset by software via three registers (see the SRCRn registers)

A software restart initiated through a Hardware System Service Request (HSSR)

A successful write to the Hardware System Service Request (HSSR) register initiates a system reset.

Boot Configuration

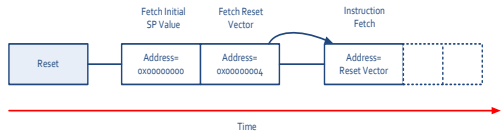
After the processor exits POR, it will read the BOOTCFG register and two words from memory:

Address 0x00000000: default value of R13 (MSP)

Address 0x00000004:

If the data at 0x00000004 is not 0xFFFFFFFF and the EN bit in BOOTCFG register is set, load the Reset vector (the starting address of startup program)

Otherwise, the stack pointer and reset vector pointer are loaded from ROM at address 0x0100.0000 and 0x0100.0004, respectively. The core executes the ROM Boot Loader.

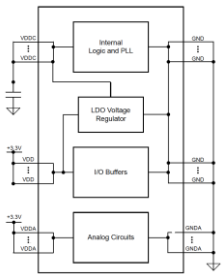


Power Control

Within the MCU, an integrated LDO regulator is used to provide power to the majority of the MCU's internal logic

The voltage output has a maximum voltage of 1.2 V

VDDA must be supplied with 3.3 V, or the microcontroller does not function properly. VDDA is the supply for all of the analog circuitry on the device, including the clock circuitry.



Clock Control

Fundamental Clock Sources

Precision Internal Oscillator (PIOSC): on-chip clock source, 16MHz ± 1%, used by the microcontroller during POR

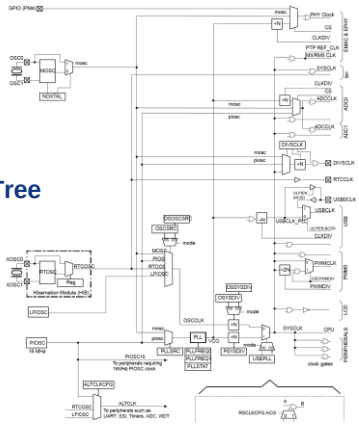
Main Oscillator (MOSC): an external crystal is connected across the `OSC0` input and `OSC1` output pins; if PLL is being used, crystal frequency range from 5 MHz to 25 MHz (inclusive); if not, between 4 MHz and 25 MHz

Low-Frequency Internal Oscillator (LFIOSC): on-chip clock source, 33kHz, used during Deep-Sleep power-saving modes

Hibernation Module RTC Oscillator (RTCOSC) Clock Source: an external 32.768-kHz clock connected to the XOSC0 pin or a low-frequency clock (HIB LFIOSC)

The internal system clock can be derived from all above clock sources and the output of PLL

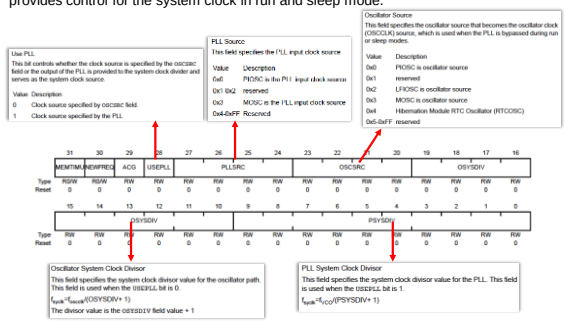
Main Clock Tree



Register 11: Run and Sleep Mode Configuration Register (RSCLKCFG), offset 0x0

Base 0x400F.E000
Offset 0x0B0
Type RW, reset 0x0000.0000

The Run and Sleep Mode Configuration Register (RSCLKCFG) provides control for the system clock in run and sleep mode.



PLL Configuration

The PLL is disabled by default during power-on reset and is enabled later by software if required.

The PLL is controlled using the **PLLFREQ0**, **PLLFREQ1** and **PLLSTAT** registers

Changes made to these registers do not become active until after the NEWFREQ bit in the RSCLKCFG register is enabled

$SysClk = f_{VCO} / (PSYSDIV + 1)$

$f_{VCO} = f_{IN} * MDIV$

where

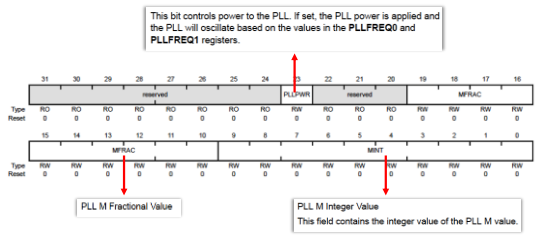
$f_{IN} = f_{XTAL} / (Q+1) (N+1)$ or $f_{PIOSC} / (Q+1)(N+1)$

$MDIV = MINT + (MFRAC / 1024)$

Register 19: PLL Frequency 0 (PLLFREQ0), offset 0x160

This register always contains the variables used to configure the PLL.

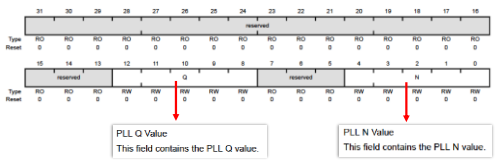
Base 0x400F.E000
Offset 0x160
Type RW, reset 0x0000.0000



Register 20: PLL Frequency 1 (PLLFREQ1), offset 0x164

This register always contains the variables used to configure the PLL.

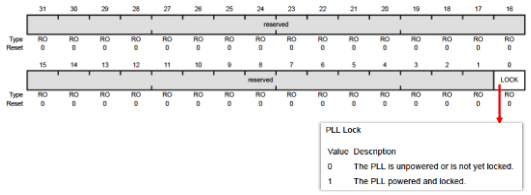
Base 0x400F.E000
Offset 0x164
Type RW, reset 0x0000.0000



Register 21: PLL Status (PLLSTAT), offset 0x168

This register shows the direct status of the PLL lock.

Base 0x400F.E000
Offset 0x168
Type RO, reset 0x0000.0000



PLL Initialization and Configuration

The steps for initializing the system clock from POR to use the PLL from the main oscillator are as follows:

1. Once POR has completed, the PIOSC is acting as the system clock.
2. Power up the MOSC by clearing the NOXTAL bit in the **MOSCCTL** register.
3. If single-ended MOSC mode is required, the MOSC is ready to use. If crystal mode is required, clear the PWRDN bit and wait for the MOSCPUPRIS bit to be set in the **Raw Interrupt Status (RIS)**, indicating MOSC crystal mode is ready.
4. Set the OSCSRC field to 0x3 in the **RSCLKCFG** register at offset 0x0B0.
5. If the application also requires the MOSC to be the deep-sleep clock source, then program the DSOSCSRC field in the **DSCLKCFG** register to 0x3.
6. Write the **PLLFREQ0** and **PLLFREQ1** registers with the values of Q, N, MINT, and MFRAC to the configure the desired VCO frequency setting.
7. Write the **MEMTIM0** register to correspond to the new system clock setting.
8. Wait for the **PLLSTAT** register to indicate the PLL has reached lock at the new operating point (or that a timeout period has passed and lock has failed, in which case an error condition exists and this sequence is abandoned and error processing is initiated).
9. Write the **RSCLKCFG** register's PSYSDIV value, set the USEPLL bit to enabled, and MEMTIMU bit.

System Control

For power-savings purposes, the **RCGCx** (e.g., **RCGCGPIO**), **SCGCx**, and **DCGCx** registers control the clock gating logic for each peripheral or block in the system while the microcontroller is in Run, Sleep, and Deep-Sleep mode, respectively.

Run mode: the microcontroller actively executes code. The processor and all of the peripherals that are currently enabled by the **RCGCx** registers operate normally. The **Run and Sleep Clock Configuration** (**RSCLKCFG**) register specifies the source of SysClk.

Sleep mode: (entered by executing a `WFI` (Wait for Interrupt) instruction), peripherals enabled by the **SCGCx** (when auto-clock gating is enabled) registers are clocked, but the processor and the memory subsystem are not clocked. The **RSCLKCFG** register specifies the source of SysClk.

System Control

Deep Sleep mode: (entered by setting the SLEEPDEEP bit in the **System Control** (**SYSCTRL**) register and then executing a `WFI` instruction), only peripherals enabled by the **DCGCx** (when auto-clock gating is enabled) registers are clocked. The system clock source is specified in the **DSCLKCFG** register.

Hibernation Mode: the power supplies are turned off to the main part of the microcontroller and only the Hibernation module's circuitry is active. An external wake event or RTC event is required to bring the microcontroller back to Run mode.

System Clock Configuration with PDL

For TM4C129 devices, `SysCtlClockFreqSet()` is used to configure the system clock; the return value from `SysCtlClockFreqSet()` indicates the system clock frequency.

```
//Use PLL to generate frequency of 480MHz, and divide the PLL frequency to get
//system clock of 20MHz, the divisor must be an integer
SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |SYSCTL_OSC_MAIN |SYSCTL_USE_PLL
|SYSCTL_CFG_VCO_480), 20000000);

//Use MOSC as the system clock of 25MHz
SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ |SYSCTL_OSC_MAIN |SYSCTL_USE_OSC),
25000000);
```

System Control with PDL

To enable a peripheral in run mode.

```
//Enable GPIO PortF in run mode
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
```

The Cortex-M3/M4 Embedded Systems:

TM4C1294NCPDT Microcontroller – Inter-Integrated Circuit (I2C) Interface

Refer to Chapter 18 in the reference book
"Tiva™ TM4C1294NCPDT Microcontroller- DATA SHEET"

Inter-Integrated Circuit (I^2C) Interface

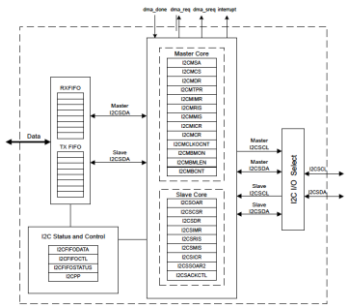
The Inter-Integrated Circuit (I^2C) bus :

- bi-directional data transfer via a two-wire design: a serial data line SDA and a serial clock line SCL
- interfaces to external I^2C devices: serial memory, networking devices, LCDs, tone generators

The TM4C1294NCPDT microcontroller includes 10 I^2C modules:

- Supports both transmitting and receiving data as either a master or a slave
- Four I^2C modes: Master/Slave transmit/receive
- Two 8-entry FIFOs for receive and transmit data
- Four transmission speeds: Standard (100 Kbps) and Fast (400 Kbps) Fast-mode plus (1 Mbps) High-speed mode (3.33 Mbps)
- Master and slave interrupt generation
- Master with arbitration and clock synchronization, multi-master support, and 7-bit addressing mode

Block Diagram



Functional Description

The I^2C bus uses only two signals: SDA and SCL, named I2CSDA and I2CSCL on *Tiva* microcontrollers

The bus is considered idle when both lines are High

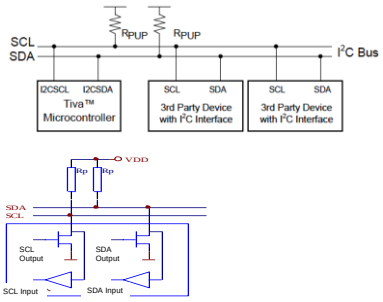
Every transaction on the I2C bus is nine bits long, i.e., 8 data bits (MSB first) and 1 single acknowledge bit

The number of bytes in one transfer is unrestricted

When a receiver cannot receive another complete byte, it can hold the clock line SCL Low and force the transmitter into a wait state

The data transfer continues when the receiver releases the clock SCL

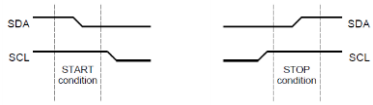
I²C Bus Configuration



START and STOP Conditions

The protocol of the I²C bus defines two states to begin and end a transaction: START and STOP

- A High-to-Low transition on the SDA line while the SCL is High is defined as a START condition
- A Low-to-High transition on the SDA line while SCL is High is defined as a STOP condition



Data Format with 7-Bit Address

After the START condition, a slave address is transmitted

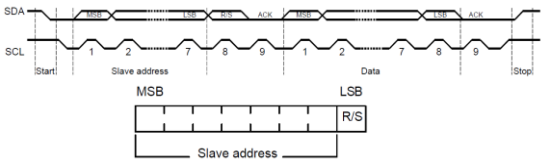
The address is 7 bits long and followed by a direction bit (R/S bit in the **I2C Master Slave Address (I2CMSA)** register)

- If R/S is cleared, a transmit operation (send)
- If R/S is set, a request for data (receive)

A data transfer is always terminated by a STOP condition generated by the master

A master can initiate communications with another device on the bus by generating a repeated START condition and addressing another slave without first generating a STOP condition.

Data Format with 7-Bit Address

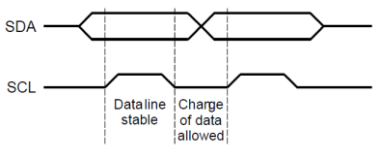


A zero in the R/S position of the first byte means that the master transmits (sends) data to the selected slave, and a one in this position means that the master receives data from the slave

Data Validity

The data on the SDA line must be stable during the high period of the clock

The data line can only change when SCL is Low



Acknowledge

All bus transactions have a required acknowledge clock cycle

During the acknowledge cycle, the transmitter (master or slave) releases the SDA line

To acknowledge the transaction, the receiver must pull down SDA during the acknowledge clock cycle

- When a slave receiver does not ack the slave address, the master can generate a STOP condition and abort the current transfer.
- When the master acting as a receiver, it is responsible for acknowledging each transfer made by the slave
- The master receiver controls the number of bytes in the transfer by not generating an acknowledge on the last data byte; slave transmitter will give up the data line

Standard, Fast, and Fast Plus Speed Modes

The I²C bus can run in either Standard mode (100 kbps) or Fast mode (400 kbps) Fast mode plus (1 Mbps)

The selected mode should match the speed of the other I²C devices on the bus

The mode is selected by using a value in the **I2C Master Timer Period (I2CMTPR)** register

The I²C clock rate is determined by the parameters:

- CLK_PRD is the system clock period
- SCL_LP is the low phase of SCL (fixed at 6)
- SCL_HP is the high phase of SCL (fixed at 4)
- TIMER_PRD is the programmed value in the **I2CMTPR** register

Standard, Fast, and Fast Plus Speed Modes

SCL_PERIOD = 2 × (1 + TIMER_PRD) × (SCL_LP + SCL_HP) × CLK_PRD

For example: If *CLK_PRD = 50 ns, TIMER_PRD = 2, SCL_LP=6, SCL_HP=4,* then, *SCL_PERIOD = 3000 ns*

Therefore, the SCL frequency is : *1/SCL_PERIOD = 333 Khz*

System Clock	Timer Period	Standard Mode	Timer Period	Fast Mode	Timer Period	Fast Mode Plus
4 MHz	0x01	100 Kbps	-	-	-	-
6 MHz	0x02	100 Kbps	-	-	-	-
12.5 MHz	0x06	89 Kbps	0x01	312 Kbps	-	-
16.7 MHz	0x08	93 Kbps	0x02	278 Kbps	-	-
20 MHz	0x09	100 Kbps	0x02	333 Kbps	-	-
25 MHz	0x0C	96.2 Kbps	0x03	312 Kbps	-	-
33 MHz	0x10	97.1 Kbps	0x04	330 Kbps	-	-
40 MHz	0x13	100 Kbps	0x04	400 Kbps	0x01	1000 Kbps
50 MHz	0x18	100 Kbps	0x06	357 Kbps	0x02	833 Kbps
80 MHz	0x27	100 Kbps	0x09	400 Kbps	0x03	1000 Kbps
100 MHz	0x31	100 Kbps	0x0C	385 Kbps	0x04	1000 Kbps
120 MHz	0x3B	100 Kbps	0xE	400 Kbps	0x5	1000 Kbps

High-Speed Mode

High-Speed mode is configured by setting the `HS` bit in the **I2C Master Control/Status** (**I2CMCS**) register.

The I²C clock rate is determined by the parameters:

- CLK_PRD is the system clock period
- SCL_LP is the low phase of SCL (fixed at 2)
- SCL_HP is the high phase of SCL (fixed at 1)
- TIMER_PRD is the programmed value and the `HS` bit should be set in the **I2CMTPR** register

High-Speed Mode

SCL_PERIOD = 2 × (1 + TIMER_PRD) × (SCL_LP + SCL_HP) × CLK_PRD

For example: *CLK_PRD = 25 ns, TIMER_PRD = 1, SCL_LP=2, SCL_HP=1*

Therefore, the SCL frequency is : *1/SCL_PERIOD = 3.33 MHz*

System Clock	Timer Period	Transmission Mode
40 MHz	0x01	3.33 Mbps
50 MHz	0x02	2.77 Mbps
80 MHz	0x03	3.33 Mbps

The master is responsible for sending a master code byte in either Standard (100 Kbps) or Fast-mode (400 Kbps) before it begins transferring in High-speed mode.

Interrupts

The I2C can generate interrupts when certain conditions are observed:

- See Chapter 18.3.3 for interrupt conditions

The I2C master and I2C slave modules have separate interrupt signals

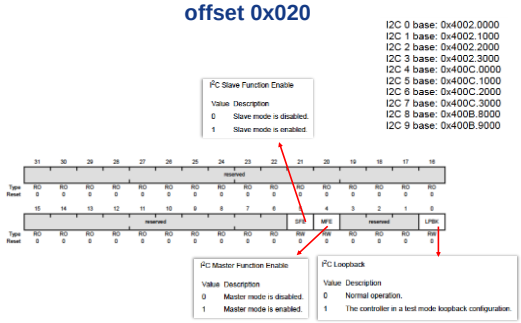
A module can generate interrupts for multiple conditions but only one single interrupt signal is sent to the interrupt controller

Loopback Operation

The I²C modules can be placed into an internal loopback mode for diagnostic or debug work

- Set the LPBK bit in the **I2C Master Configuration (I2CMCR)** register
- SDA and SCL signals from the master and slave modules are tied together

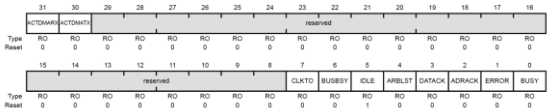
Register 9: I2C Master Configuration (I2CMCR), offset 0x020



Register 2: I2C Master Control/Status (I2CMCS), offset 0x004

I2C 0 base: 0x4002.0000
I2C 1 base: 0x4002.1000
I2C 2 base: 0x4002.2000
I2C 3 base: 0x4002.3000
I2C 4 base: 0x400C.0000
I2C 5 base: 0x400C.1000
I2C 6 base: 0x400C.2000
I2C 7 base: 0x400C.3000
I2C 8 base: 0x400B.8000
I2C 9 base: 0x400B.9000

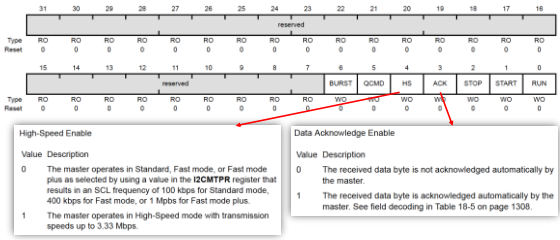
Read-Only Status Register



Register 2: I2C Master Control/Status (I2CMCS), offset 0x004

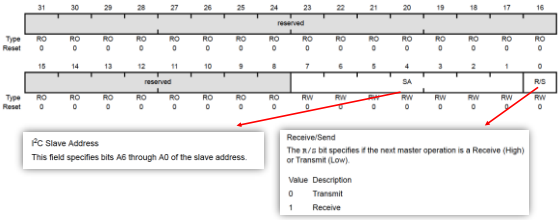
I2C 0 base: 0x4002.0000
I2C 1 base: 0x4002.1000
I2C 2 base: 0x4002.2000
I2C 3 base: 0x4002.3000
I2C 4 base: 0x400C.0000
I2C 5 base: 0x400C.1000
I2C 6 base: 0x400C.2000
I2C 7 base: 0x400C.3000
I2C 8 base: 0x400B.8000
I2C 9 base: 0x400B.9000

Write-Only Control Register



Register 1: I2C Master Slave Address (I2CMSA), offset 0x000

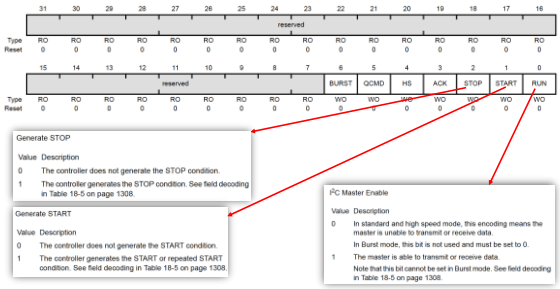
I2C 0 base: 0x4002.0000
I2C 1 base: 0x4002.1000
I2C 2 base: 0x4002.2000
I2C 3 base: 0x4002.3000
I2C 4 base: 0x400C.0000
I2C 5 base: 0x400C.1000
I2C 6 base: 0x400C.2000
I2C 7 base: 0x400C.3000
I2C 8 base: 0x400B.8000
I2C 9 base: 0x400B.9000



Register 2: I2C Master Control/Status (I2CMCS), offset 0x004

I2C 0 base: 0x4002.0000
I2C 1 base: 0x4002.1000
I2C 2 base: 0x4002.2000
I2C 3 base: 0x4002.3000
I2C 4 base: 0x400C.0000
I2C 5 base: 0x400C.1000
I2C 6 base: 0x400C.2000
I2C 7 base: 0x400C.3000
I2C 8 base: 0x400B.8000
I2C 9 base: 0x400B.9000

Write-Only Control Register



Initialization and Configuration

The following example shows how to configure the I²C module to transmit a single byte as a master:

1. Enable the I²C clock using the **RCGCI2C** register in the System Control module
2. Enable the clock to the appropriate GPIO module via the **RCGCGPIO** register in the System Control module
3. In the GPIO module, enable the appropriate pins for their alternate function using the **GPIOAFSEL** register
4. Enable the `I2CSDA` pin for Open Drain operation
5. Configure the PMCn fields in the **GPIOPCTL** register to assign the I2C signals to the appropriate pins
6. Initialize the I2C Master by writing the **I2CMCR** register with a value of 0x00000010

Initialization and Configuration

7. Set the desired SCL clock speed of 100 Kbps by writing the **I2CMTPR** register with the correct value:

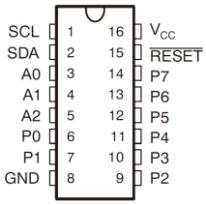
```
TPR = (System Clock/(2*(SCL_LP + SCL_HP)*SCL_CLK))-1;
TPR = (20MHz/(2*(6+4)*100000))-1;
TPR = 9
```

8. Given the slave address 0x3B, specify the slave address of the master and that the next operation is a Transmit by writing the **I2CMSA** register with a value of 0x00000076
9. Place data (byte) to be transmitted in the data register by writing the **I2CMDR** register with the desired data
10. Initiate a single byte transmit of the data from Master to Slave by writing the **I2CMCS** register with a value of 0x00000007 (STOP, START, RUN)
11. Wait until the transmission completes by polling the **I2CMCS** register's BUSBSY bit until it has been cleared.
12. Check the `ERROR` bit in the **I2CMCS** register to confirm the transmit was acknowledged

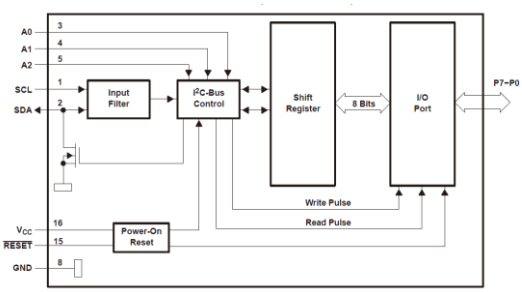
PCA9557: REMOTE 8-BIT I²C AND SMBus LOW-POWER I/O EXPANDER

Chip Description and Key Features

- 8-bit I/O expander for I²C bus
  - **I²C** to **parallel port** expander
  - 400-kHz fast I²C bus
  - 3 hardware address pins allow for use of up to 8 devices on I²C
  - Noise filter on SCL/SDA inputs
- General-purpose remote I/O expansion for most microcontroller families
- At power on, the I/Os are configured as inputs
- The I/Os can be programmed by I²C master to be as either inputs or outputs
  - Latched outputs with high current drive maximum capability for directly driving LEDs
  - Data for each input or output is kept in the corresponding input or output register
  - The polarity of the input port register can be inverted with the polarity inversion register



Logic Diagram



I²C Protocol Review

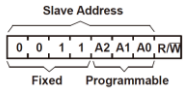
- Two lines: the serial clock (SCL) and serial data (SDA)
  - Connected to a positive supply through a pull-up resistor
- Communication is initiated by a master sending a start condition
  - A high-to-low transition on the SDA & the SCL input is high
- After the start condition, the device address byte is sent
  - Most-significant bit (MSB) first
  - Followed by the data direction bit (R/W)
- After receiving the valid address byte, the device having this address responds with an ACK
  - A low on the SDA during the high of the ACK-related clock pulse

I²C Protocol Review Cont.

- Data transfer begins
  - Only one data bit is transferred during each clock pulse
  - Any number of data bytes can be transferred from the transmitter to the receiver
  - Each byte of eight bits is followed by one ACK bit
    - The transmitter must release the SDA line
    - The receiver must pull down the SDA line during the ACK clock pulse
- Master will stop the transmission by issuing a stop condition
  - A low-to-high transition on the SDA & the SCL input is high

Device Address

The address of the PCA9557 is shown as follows



R/~W: a high (1) indicates a read operation, while a low (0) selects a write operation, from the perspective of the Master.

INPUTS			I²C BUS SLAVE ADDRESS
A2	A1	A0	
L	L	L	24 (decimal), 18 (hexadecimal)
L	L	H	25 (decimal), 19 (hexadecimal)
L	H	L	26 (decimal), 1A (hexadecimal)
L	H	H	27 (decimal), 1B (hexadecimal)
H	L	L	28 (decimal), 1C (hexadecimal)
H	L	H	29 (decimal), 1D (hexadecimal)
H	H	L	30 (decimal), 1E (hexadecimal)
H	H	H	31 (decimal), 1F (hexadecimal)

Control Register & Command Byte

After the successful ACK of the address type, the bus master sends a command byte that is stored in the control register in the PCA9557

Command bytes are used to specify the operation and the internal registers that will be affected

0	0	0	0	0	0	B1	B0

Control Register Bits

Command Byte

CONTROL REGISTER BITS		COMMAND BYTE (HEX)	REGISTER	PROTOCOL	POWER-UP DEFAULT
B1	B0				
0	0	0x00	Input Port	Read byte	xxxx xxxx
0	1	0x01	Output Port	Read/write byte	0000 0000
1	0	0x02	Polarity Inversion	Read/write byte	1111 0000
1	1	0x03	Configuration	Read/write byte	1111 1111

Configuration Register

Configures the directions of the I/O pins

- If a bit in this register is set to 1, the corresponding port pin is enabled as an input
- If a bit in this register is cleared, the corresponding port pin is enabled as an output

Register 3 (Configuration Register)

BIT	C7	C6	C5	C4	C3	C2	C1	C0
DEFAULT	1	1	1	1	1	1	1	1

Input Port Register

Reflects the incoming logic levels of the pins

- All pins no matter whether a pin is an input or an output
- The default value is determined by the externally applied logic level

Register 0 (Input Port Register)

BIT	I7	I6	I5	I4	I3	I2	I1	I0
DEFAULT	X	X	X	X	X	X	X	X

Output Port Register

Shows the outgoing logic levels of the pins defined as outputs

- Bit values in this register have no effect on pins defined as inputs
- reads from this register reflect the value that is in the flip-flop controlling the output selection, not the actual pin value

Register 1 (Output Port Register)

BIT	O7	O6	O5	O4	O3	O2	O1	O0
DEFAULT	0	0	0	0	0	0	0	0

Polarity Inversion Register

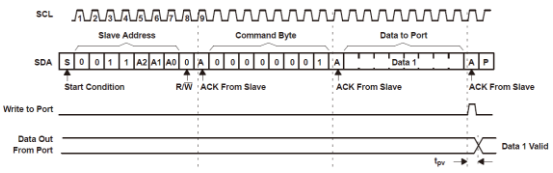
Inverts the polarity of input pins

- If a bit in this register is set, the corresponding port pin's polarity is inverted
- If a bit in this register is cleared, the corresponding port pin's original polarity is retained

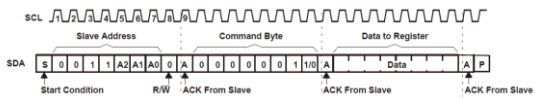
Register 2 (Polarity Inversion Register)

BIT	N7	N6	N5	N4	N3	N2	N1	N0
DEFAULT	1	1	1	1	0	0	0	0

Write Operation

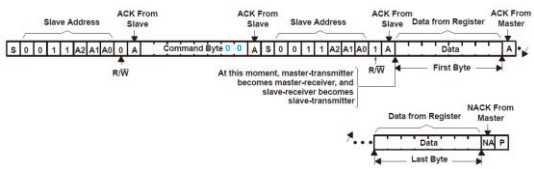


Write to Output Port Register



Write to Configuration or Polarity Inversion Registers

Read Operation



Read From Register

TCA6424: Low-Voltage 24-Bit I2C AND SMBus I/O Expander

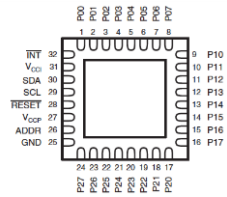
Chip Description and Key Features

I2C to 24-bit Parallel Port Expander
- 400-kHz fast I²C bus
- Input/Output Configuration Register
- Polarity Inversion Register

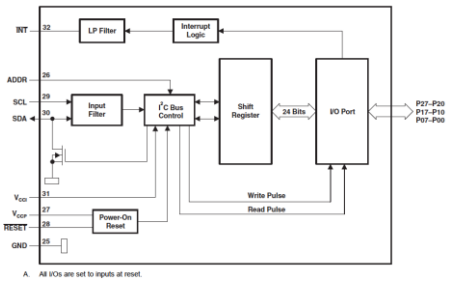
General-purpose remote I/O expansion for most microcontroller families

At power on, the I/Os are configured as inputs

The I/Os can be programmed by I²C master to be as either inputs or outputs
- Latched outputs with high current drive maximum capability for directly driving LEDs
- Data for each input or output is kept in the corresponding input or output register
- The polarity of the input port register can be inverted with the polarity inversion register



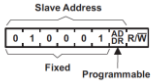
Logic Diagram



A. All I/Os are set to inputs at reset.

Device Address

The address of the TCA6424 is shown as follows



R/~W: a high (1) indicates a read operation, while a low (0) selects a write operation, from the perspective of the Master.

ADDR	I²C BUS SLAVE ADDRESS
L	34 (decimal), 22 (hexadecimal)
H	35 (decimal), 23 (hexadecimal)

Control Register & Command Byte

After the successful ACK of the address type, the bus master sends a command byte that is stored in the control register in the PCA9557

Command bytes are used to specify the operation and the internal registers that will be affected

AI	B6	B5	B4	B3	B2	B1	B0

Figure 24. Control Register Bits

Table 4. Command Byte

CONTROL REGISTER BITS								AUTO-INCREMENT SETTING	COMMAND BYTE (HEX)	REGISTER	PROTOCOL	POWER-UP DEFAULT
AI	B6	B5	B4	B3	B2	B1	B0					
0	0	0	0	0	0	0	0	Disable	00	Input Port 0	Read byte	xxxx xxxx[1]
1	0	0	0	0	0	0	0	Enable	80			
0	0	0	0	0	0	0	1	Disable	01	Input Port 1	Read byte	xxxx xxxx[1]
1	0	0	0	0	0	0	1	Enable	81			
0	0	0	0	0	0	1	0	Disable	02	Input Port 2	Read byte	xxxx xxxx[1]
1	0	0	0	0	0	1	0	Enable	82			
0	0	0	0	0	0	1	1	Disable	03	Reserved	Reserved	Reserved
1	0	0	0	0	0	1	1	Enable	83			
0	0	0	0	0	1	0	0	Disable	04	Output Port 0	Read/write byte	1111 1111
1	0	0	0	0	1	0	0	Enable	84			
0	0	0	0	0	1	0	1	Disable	05	Output Port 1	Read/write byte	1111 1111
1	0	0	0	0	1	0	1	Enable	85			
0	0	0	0	0	1	1	0	Disable	06	Output Port 2	Read/write byte	1111 1111
1	0	0	0	0	1	1	0	Enable	86			
0	0	0	0	0	1	1	1	Disable	07	Reserved	Reserved	Reserved
1	0	0	0	0	1	1	1	Enable	87			
0	0	0	0	1	0	0	0	Disable	08	Polarity Inversion Port 0	Read/write byte	0000 0000
1	0	0	0	1	0	0	0	Enable	88			
0	0	0	0	1	0	0	1	Disable	09	Polarity Inversion Port 1	Read/write byte	0000 0000
1	0	0	0	1	0	0	1	Enable	89			
0	0	0	0	1	0	1	0	Disable	0A	Polarity Inversion Port 2	Read/write byte	0000 0000
1	0	0	0	1	0	1	0	Enable	8A			
0	0	0	0	1	0	1	1	Disable	0B	Reserved	Reserved	Reserved
1	0	0	0	1	0	1	1	Enable	8B			
0	0	0	0	1	1	0	0	Disable	0C	Configuration Port 0	Read/write byte	1111 1111
1	0	0	0	1	1	0	0	Enable	8C			
0	0	0	0	1	1	0	1	Disable	0D	Configuration Port 1	Read/write byte	1111 1111
1	0	0	0	1	1	0	1	Enable	8D			
0	0	0	0	1	1	1	0	Disable	0E	Configuration Port 2	Read/write byte	1111 1111
1	0	0	0	1	1	1	0	Enable	8E			
0	0	0	0	1	1	1	1	Disable	0F	Reserved	Reserved	Reserved
1	0	0	0	1	1	1	1	Disable	8F			

Configuration Register

The Configuration registers (registers 12, 13 and 14) configure the directions of the I/O pins
- If a bit in such a register is set to 1, the corresponding port pin is enabled as an input
- If a bit in such a register is cleared, the corresponding port pin is enabled as an output

BIT	C-07	C-06	C-05	C-04	C-03	C-02	C-01	C-00
DEFAULT	1	1	1	1	1	1	1	1
BIT	C-17	C-16	C-15	C-14	C-13	C-12	C-11	C-10
DEFAULT	1	1	1	1	1	1	1	1
BIT	C-27	C-26	C-25	C-24	C-23	C-22	C-21	C-20
DEFAULT	1	1	1	1	1	1	1	1

Input Port Registers

The Input Port registers (registers 0, 1 and 2) reflect the incoming logic levels of the pins

- All pins no matter whether a pin is an input or an output
- The default value is determined by the externally applied logic level

BIT	I-07	I-06	I-05	I-04	I-03	I-02	I-01	I-00
DEFAULT	X	X	X	X	X	X	X	X
BIT	I-17	I-16	I-15	I-14	I-13	I-12	I-11	I-10
DEFAULT	X	X	X	X	X	X	X	X
BIT	I-27	I-26	I-25	I-24	I-23	I-22	I-21	I-20
DEFAULT	X	X	X	X	X	X	X	X

Output Port Registers

The Output Port registers (registers 4, 5 and 6) show the outgoing logic levels of the pins defined as outputs

- Bit values in this register have no effect on pins defined as inputs
- reads from this register reflect the value that is in the flip-flop controlling the output selection, not the actual pin value

BIT	O-07	O-06	O-05	O-04	O-03	O-02	O-01	O-00
DEFAULT	1	1	1	1	1	1	1	1
BIT	O-17	O-16	O-15	O-14	O-13	O-12	O-11	O-10
DEFAULT	1	1	1	1	1	1	1	1
BIT	O-27	O-26	O-25	O-24	O-23	O-22	O-21	O-20
DEFAULT	1	1	1	1	1	1	1	1

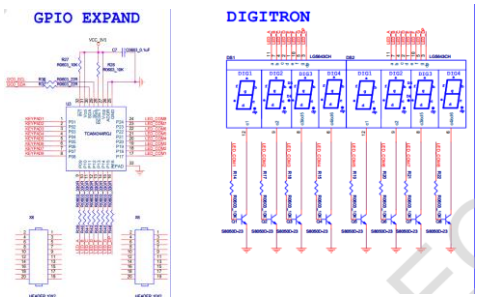
Polarity Inversion Registers

The Polarity Inversion registers (registers 8, 9 and 10) invert the polarity of input pins

- If a bit in such a register is set, the corresponding port pin's polarity is inverted
- If a bit in such a register is cleared, the corresponding port pin's original polarity is retained

BIT	P-07	P-06	P-05	P-04	P-03	P-02	P-01	P-00
DEFAULT	0	0	0	0	0	0	0	0
BIT	P-17	P-16	P-15	P-14	P-13	P-12	P-11	P-10
DEFAULT	0	0	0	0	0	0	0	0
BIT	P-27	P-26	P-25	P-24	P-23	P-22	P-21	P-20
DEFAULT	0	0	0	0	0	0	0	0

I²C on S800



I²C Programming with PDL and Expender Chips (1)

Slave address of PCA9557 and TCA6424, and the command byte for internal registers

```
#define TCA6424_I2CADDR                 0x22
#define PCA9557_I2CADDR                 0x18

#define PCA9557_INPUT                   0x00
#define PCA9557_OUTPUT                  0x01
#define PCA9557_POLINVERT               0x02
#define PCA9557_CONFIG                  0x03

#define TCA6424_CONFIG_PORT0            0x0c
#define TCA6424_CONFIG_PORT1            0x0d
#define TCA6424_CONFIG_PORT2            0x0e

#define TCA6424_INPUT_PORT0             0x00
#define TCA6424_INPUT_PORT1             0x01
#define TCA6424_INPUT_PORT2             0x02

#define TCA6424_OUTPUT_PORT0            0x04
#define TCA6424_OUTPUT_PORT1            0x05
#define TCA6424_OUTPUT_PORT2            0x06
```

I²C Programming with PDL and Expender Chips (2)

Write data to the output/configuration register of PCA9557/TCA6424

```
uint8_t I2C0_WriteByte(uint8_t DevAddr, uint8_t RegAddr, uint8_t WriteData){
    uint8_t rop;

    while(I2CMasterBusy(I2C0_BASE)){};

    I2CMasterSlaveAddrSet(I2C0_BASE, DevAddr, false);
    I2CMasterDataPut(I2C0_BASE, RegAddr);
    I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_START);
    while(I2CMasterBusy(I2C0_BASE)){};
    rop = (uint8_t)I2CMasterErr(I2C0_BASE);
    I2CMasterDataPut(I2C0_BASE, WriteData);
    I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_BURST_SEND_FINISH);
    while(I2CMasterBusy(I2C0_BASE)){};
    rop = (uint8_t)I2CMasterErr(I2C0_BASE);
    return rop;
}
```

I²C Programming with PDL and Expender Chips (3)

Read data from the input/inversion register of PCA9557/TCA6424

```
uint8_t I2C0_ReadByte(uint8_t DevAddr, uint8_t RegAddr)
{
    uint8_t value,rop;
    while(I2CMasterBusy(I2C0_BASE)){};
    I2CMasterSlaveAddrSet(I2C0_BASE, DevAddr, false);
    I2CMasterDataPut(I2C0_BASE, RegAddr);

    I2CMasterControl(I2C0_BASE, I2C_MASTER_CMD_SINGLE_SEND);
    while(I2CMasterBusBusy(I2C0_BASE));
    rop = (uint8_t)I2CMasterErr(I2C0_BASE);
    Delay(1);
    //receive data
    I2CMasterSlaveAddrSet(I2C0_BASE, DevAddr, true);
    I2CMasterControl(I2C0_BASE,I2C_MASTER_CMD_SINGLE_RECEIVE);
    while(I2CMasterBusBusy(I2C0_BASE));
    value=I2CMasterDataGet(I2C0_BASE);
    Delay(1);
    return value;
}
```

I²C Programming with PDL and Expender Chips (4)

Enable I2C0 and configure PB2 and PB3 as I2C0_SCL and I2C0_SDA

```
void S800_I2C0_Init(void){
    uint8_t result;
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    GPIOPinConfigure(GPIO_PB2_I2C0SCL);
    GPIOPinConfigure(GPIO_PB3_I2C0SDA);
    GPIOPinTypeI2CSCL(GPIO_PORTB_BASE, GPIO_PIN_2);
    GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_3);
    I2CMasterInitExpClk(I2C0_BASE,ui32SysClock, true);  //config I2C0 400k
    I2CMasterEnable(I2C0_BASE);
    result = I2C0_WriteByte(TCA6424_I2CADDR,TCA6424_CONFIG_PORT0,0x0ff);
    //config port 0 as input
    result = I2C0_WriteByte(TCA6424_I2CADDR,TCA6424_CONFIG_PORT1,0x0);
    //config port 1 as output
    result = I2C0_WriteByte(TCA6424_I2CADDR,TCA6424_CONFIG_PORT2,0x0);
    //config port 2 as output
    result = I2C0_WriteByte(PCA9557_I2CADDR,PCA9557_CONFIG,0x00);
    //config port as output
    result = I2C0_WriteByte(PCA9557_I2CADDR,PCA9557_OUTPUT,0x0ff);
    //turn off the LED1-8
}
```

The Cortex-M3/M4 Embedded Systems:

TM4C1294NCPDT Microcontroller – SysTick

Refer to Chapter 3 in the reference book
"Tiva™ TM4C1294NCPDT Microcontroller- DATA SHEET"

SysTick: System Timer

Provides a simple, 24-bit clear-on-write, decrementing, wrap-on-zero counter

- An RTOS tick timer which fires at a programmable rate (for example, 100 Hz) and invokes a SysTick handler routine
- A high-speed alarm timer using the system clock
- A variable rate alarm with other reference clock
- A simple counter used to measure time to completion and time used

Functional Description

SysTick consists of three registers:

- **SysTick Control and Status Register:** a control and status counter to configure its clock, enable the counter, enable the SysTick interrupt, and determine counter status
- **SysTick Reload Value Register:** the reload value for the counter, used to provide the counter's wrap value
- **SysTick Current Value Register:** the current value of the counter

Note: the SysTick Calibration Value Register, is not implemented

Functional Description

When enabled:

- SysTick operates in a multi-shot way: reloads the Reload value from the **SysTick Reload Value register** and counts down on each clock to zero

When counting **from 1 to 0**:

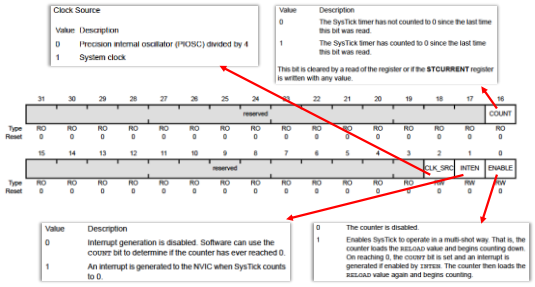
- the **COUNT** status bit in **SYSTICK Control and Status Register** is set (clear on reads)
- An interrupt is generated if enabled by the **INTEN** bit in **SYSTICK Control and Status Register**
- Reloads (wraps) the Reload value on the next clock edge

During counting down:

- Clearing the **SysTick Reload Value register** disables the counter on the next wrap
- Get the current count by reading the **SysTick Current Value register**
- Writing to the **SysTick Current Value register** clears the register and the **COUNT** status bit

If the core is in debug state (halted), the counter does not decrement

SysTick Control and Status Register, 0xE000E010



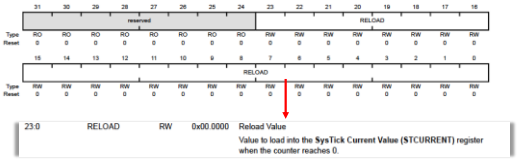
SysTick Reload Value Register, 0xE000E014

The initial count *N* can be between 1 and 0x00FF.FFFF

SysTick fires on every *N*+1 clock.

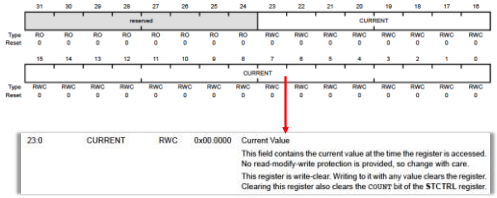
For example, if a tick interrupt is required every 100 clock pulses, 99 must be written into the RELOAD field

When reading this register, the system clock must be faster than 8MHz



SysTick Current Value Register, 0xE000E018

The SysTick Current Value Register contains the current value of the counter.



Initialization and Configuration

Write the appropriate initial count to the **SysTick Reload Value Register**

Configure the SysTick timer by programming the **SysTick Control and Status Register**

- Set **ENABLE** and **CLK_SRC** (if use system clock) as 1
- If you are using interrupts:
  - Set **INTEN** as 1
  - Write the corresponding handler for SysTick exceptions
- Otherwise, you are using software pulling:
  - Set **INTEN** as 0
  - To check whether there is a tick: polling the **COUNT** status and process accordingly when the value is 1
  - To measure a time duration: read the **SysTick Current Value Register** when job is done and calculate the passed time

SysTick Programming with PDL (1)

Software pulling

```
ui32SysClock = SysCtlClockFreqSet((SYSCTL_XTAL_16MHZ |SYSCTL_OSC_INT
|SYSCTL_USE_OSC), 16000000);

SysTickPeriodSet(ui32SysClock/1000);//the tick duration is 1mS
SysTickEnable();                    //enable the SysTick
ui32Value = SysTickValueGet();      //read the current count of SysTick
```

SysTick Programming with PDL (2)

Interrupt driven

```
ui32SysClock = SysCtlClockFreqSet((SYSCTL_XTAL_16MHZ |SYSCTL_OSC_INT
|SYSCTL_USE_OSC), 16000000);

SysTickPeriodSet(ui32SysClock/1000); );//The tick duration is 1mS
SysTickEnable();          //Enable the SysTick
SysTickIntEnable();       //Enable SysTick exceptions
IntMasterEnable();        //Enables the processor interrupt


void SysTick_Handler(void)
{
   //SysTick handler
}
```

The Cortex-M3/M4 Embedded Systems:
TM4C1294NCPDT Microcontroller – Interrupts

Refer to Chapter 2.5 and 3.1 in the reference book
"Tiva™ TM4C1294NCPDT Microcontroller- DATA SHEET"

Exceptions

Exceptions are numbered 1 to 15 for system exceptions and the rest 240 for external interrupt inputs
- For TM4C1294NCPDT, 10 system exceptions and 97 interrupts

Most of the exceptions have programmable priority, and a few have fixed priority.
- Reset, NMI, and Hard Fault have fixed priorities of -3, -2, and -1, respectively
- 3-bit priority registers are implemented for all the programmable priorities
- Eight user programmable priority levels; 0 is the highest and the default priority is 0 for all the programmable priorities

List of Exceptions

Exception Type	Vector Number	Priority[d]	Vector Address or Offset[c]	Activation
-	0	-	0x0000.0000	Stack top is loaded from the first entry of the vector table on reset.
Reset	1	-3 (highest)	0x0000.0004	Asynchronous
Non-Maskable Interrupt (NMI)	2	-2	0x0000.0008	Asynchronous
Hard Fault	3	-1	0x0000.000C	-
Memory Management	4	programmable[b]	0x0000.0010	Synchronous
Bus Fault	5	programmable[b]	0x0000.0014	Synchronous when precise and asynchronous when imprecise
Usage Fault	6	programmable[b]	0x0000.0018	Synchronous
-	7-10	-	-	Reserved
SVCall	11	programmable[b]	0x0000.002C	Synchronous
Debug Monitor	12	programmable[b]	0x0000.0030	Synchronous
-	13	-	-	Reserved
PendSV	14	programmable[b]	0x0000.0038	Asynchronous
SysTick	15	programmable[b]	0x0000.003C	Asynchronous
Interrupts	16 and above	programmable[b]	0x0000.0040 and above	Asynchronous

a. 0 is the default priority for all the programmable priorities.
b. See "Vector Table" on page 119.
c. See **SYSPRI1** on page 177.
d. See **PRIn** registers on page 159.

List of Interrupts

Vector Number	Interrupt Number (Bit in Interrupt Registers)	Vector Address or Offset	Description
0-15	-	0x0000.0000 - 0x0000.003C	Processor exceptions
16	0	0x0000.0040	GPIO Port A
17	1	0x0000.0044	GPIO Port B
18	2	0x0000.0048	GPIO Port C
19	3	0x0000.004C	GPIO Port D
20	4	0x0000.0050	GPIO Port E
21	5	0x0000.0054	UART0
22	6	0x0000.0058	UART1
23	7	0x0000.005C	SSI0
24	8	0x0000.0060	I²C0
25	9	0x0000.0064	PWM Fault
26	10	0x0000.0068	PWM Generator 0
27	11	0x0000.006C	PWM Generator 1
28	12	0x0000.0070	PWM Generator 2
29	13	0x0000.0074	QEI0
30	14	0x0000.0078	ADC0 Sequence 0
31	15	0x0000.007C	ADC0 Sequence 1
32	16	0x0000.0080	ADC0 Sequence 2
33	17	0x0000.0084	ADC0 Sequence 3
34	18	0x0000.0088	Watchdog Timers 0 and 1
35	19	0x0000.008C	16/32-Bit Timer 0A
36	20	0x0000.0090	16/32-Bit Timer 0B
37	21	0x0000.0094	16/32-Bit Timer 1A

List of Interrupts

Vector Number	Interrupt Number (Bit in Interrupt Registers)	Vector Address or Offset	Description
38	22	0x0000.0098	16/32-Bit Timer 1B
39	23	0x0000.009C	16/32-Bit Timer 2A
40	24	0x0000.00A0	16/32-Bit Timer 2B
41	25	0x0000.00A4	Analog Comparator 0
42	26	0x0000.00A8	Analog Comparator 1
43	27	0x0000.00AC	Analog Comparator 2
44	28	0x0000.00B0	System Control
45	29	0x0000.00B4	Flash Memory Control
46	30	0x0000.00B8	GPIO Port F
47	31	0x0000.00BC	GPIO Port G
48	32	0x0000.00C0	GPIO Port H
49	33	0x0000.00C4	UART2
50	34	0x0000.00C8	SSI1
51	35	0x0000.00CC	16/32-Bit Timer 3A
52	36	0x0000.00D0	16/32-Bit Timer 3B
53	37	0x0000.00D4	I²C1
54	38	0x0000.00D8	CAN 0
55	39	0x0000.00DC	CAN1
56	40	0x0000.00E0	Ethernet MAC
57	41	0x0000.00E4	HIB
58	42	0x0000.00E8	USB MAC
59	43	0x0000.00EC	PWM Generator 3
60	44	0x0000.00F0	uDMA 0 Software
61	45	0x0000.00F4	uDMA 0 Error
62	46	0x0000.00F8	ADC1 Sequence 0
63	47	0x0000.00FC	ADC1 Sequence 1
64	48	0x0000.0100	ADC1 Sequence 2
65	49	0x0000.0104	ADC1 Sequence 3
66	50	0x0000.0108	EPI 0
67	51	0x0000.010C	GPIO Port J
68	52	0x0000.0110	GPIO Port K
69	53	0x0000.0114	GPIO Port L
70	54	0x0000.0118	SSI 2
71	55	0x0000.011C	SSI 3
72	56	0x0000.0120	UART 3
73	57	0x0000.0124	UART 4
74	58	0x0000.0128	UART 5
75	59	0x0000.012C	UART 6
76	60	0x0000.0130	UART 7
77	61	0x0000.0134	I²C 2
78	62	0x0000.0138	I²C 3

List of Interrupts

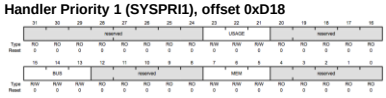
Vector Number	Interrupt Number (Bit in Interrupt Registers)	Vector Address or Offset	Description
79	63	0x0000.013C	Timer 4A
80	64	0x0000.0140	Timer 4B
81	65	0x0000.0144	Timer 5A
82	66	0x0000.0148	Timer 5B
83	67	0x0000.014C	Floating-Point Exception (imprecise)
84-85	68-69	-	Reserved
86	70	0x0000.0158	I²C 4
87	71	0x0000.015C	I²C 5
88	72	0x0000.0160	GPIO Port M
89	73	0x0000.0164	GPIO Port N
90	74	-	Reserved
91	75	0x0000.016C	Tamper
92	76	0x0000.017	GPIO Port P (Summary or P0)
93	77	0x0000.0174	GPIO Port P1
94	78	0x0000.0178	GPIO Port P2
95	79	0x0000.017C	GPIO Port P3
96	80	0x0000.0180	GPIO Port P4
97	81	0x0000.0184	GPIO Port P5
98	82	0x0000.0188	GPIO Port P6
99	83	0x0000.018C	GPIO Port P7
100	84	0x0000.0190	GPIO Port Q (Summary or Q0)
101	85	0x0000.0194	GPIO Port Q1
102	86	0x0000.0198	GPIO Port Q2
103	87	0x0000.019C	GPIO Port Q3
104	88	0x0000.01A0	GPIO Port Q4
105	89	0x0000.01A4	GPIO Port Q5
106	90	0x0000.01A8	GPIO Port Q6
107	91	0x0000.01AC	GPIO Port Q7
108-113	92-97	-	Reserved
114	98	0x0000.01C8	16/32-Bit Timer 6A
115	99	0x0000.01CC	16/32-Bit Timer 6B
116	100	0x0000.01D0	16/32-Bit Timer 7A
117	101	0x0000.01D4	16/32-Bit Timer 7B
118	102	0x0000.01D8	I²C 6
119	103	0x0000.01DC	I²C 7
120-124	104-108	-	Reserved
125	109	0x0000.01F4	I²C 8
126	110	0x0000.01F8	I²C 9
127-129	111-113	-	Reserved

Setup Exceptions at SCB and NVIC

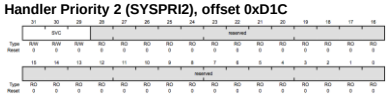
Set Exception Priorities

For system exceptions, priorities are set with **System Handler Priority n** (**SYSPRIn**) registers in **SCB**

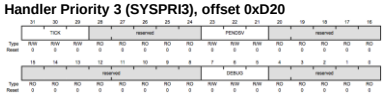
System Handler Priority 1 (SYSPRI1), offset 0xD18



System Handler Priority 2 (SYSPRI2), offset 0xD1C



System Handler Priority 3 (SYSPRI3), offset 0xD20



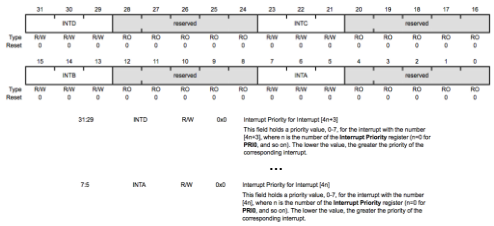
Set Exception Priorities

For interrupts, priorities are set with **Interrupt Priority n** (**PRIn**) registers in **NVIC**

Interrupt 0-3 Priority (PRI0), offset 0x400

…

Interrupt 112-113 Priority (PRI28), offset 0x470



Bit/Field	Name	Type	Reset	Description
31:29	INTD	R/W	0x0	Interrupt Priority for Interrupt [4n+3] This field holds a priority value, 0-7, for the interrupt with the number [4n+3], where n is the number of the **Interrupt Priority** register (n=0 for **PRI0**, and so on). The lower the value, the greater the priority of the corresponding interrupt.
…				
7:5	INTA	R/W	0x0	Interrupt Priority for Interrupt [4n] This field holds a priority value, 0-7, for the interrupt with the number [4n], where n is the number of the **Interrupt Priority** register (n=0 for **PRI0**, and so on). The lower the value, the greater the priority of the corresponding interrupt.

Exception Priority Grouping

8 exception priorities can be further grouped defined by the **Application Interrupt and Reset Control** (**APINT**) register in **SCB**

Table 7.5 Application Interrupt and Reset Control Register (Address 0xE000ED0C)

Bits	Name	Type	Reset Value	Description
31:16	VECTKEY	R/W	-	Access key; 0x05FA must be written to this field to write to this register, otherwise the write will be ignored; the read-back value of the upper half-word is 0xFA05
15	ENDIANESS	R	-	Indicates endianness for data; 1 for big endian (BE8), and 0 for little endian; this can only change after a reset
10:8	PRIGROUP	R/W	0	Priority group
2	SYSRESETREQ	W	-	Requests chip control logic to generate a reset
1	VECTCLRACTIVE	W	-	Clears all active state information for exceptions; typically used in debug or OS to allow system to recover from system error (Reset is safer)
0	VECTRESET	W	-	Resets the Cortex-M3 processor (except debug logic), but this will not reset circuits outside the processor

Priority Group	Preempt Priority Field	Subpriority Field
0	Bit [7:1]	Bit [0]
1	Bit [7:2]	Bit [1:0]
2	Bit [7:3]	Bit [2:0]
3	Bit [7:4]	Bit [3:0]
4	Bit [7:5]	Bit [4:0]
5	Bit [7:6]	Bit [5:0]
6	Bit [7]	Bit [6:0]
7	None	Bit [7:0]

Only the group priority determines preemption of interrupt exceptions

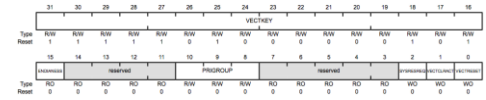
If multiple pending interrupts have the same group priority, the subpriority field determines the order in which they are processed

If multiple pending interrupts have the same group priority and subpriority, the interrupt with the lowest IRQ number is processed first

Application Interrupt and Reset Control (APINT), offset 0xD0C

Base 0xE000.E000

Provides priority grouping control, endian status for data accesses, and reset control of the system; accessed from privileged mode

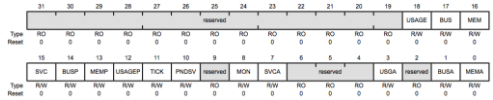


Bit/Field	Name	Type	Reset	Description
31:16	VECTKEY	R/W	0xFA05	Register Key This field is used to guard against accidental writes to this register. 0x05FA must be written to this field in order to change the bits in this register. On a read, 0xFA05 is returned.
15	ENDIANESS	RO	0	Data Endianess The Stellaris implementation uses only little-endian mode so this is cleared to 0.
10:8	PRIGROUP	R/W	0x0	Interrupt Priority Grouping This field determines the split of group priority from subpriority (see Table 3-8 on page 157 for more information).
2	SYSRESREQ	WO	0	System Reset Request

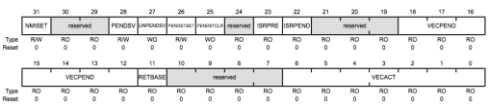
Enable and Disable Exceptions

For system exceptions, the **SYSHNDCTRL** register enables the system handlers, and indicates the pending and active status of the system handlers; the **INCTRL** register provides a set-pending bit for the NMI exception, and set-pending and clear-pending bits for the PendSV and SysTick exceptions in **SCB**

System Handler Control and State (SYSHNDCTRL), offset 0xD24



Interrupt Control and State (INTCTRL), offset 0xD04



Enable and Disable Exceptions

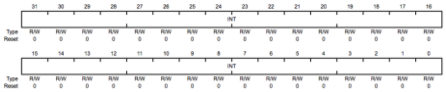
For interrupts, the **ENn** register enable interrupts and show which interrupts are enabled in **NVIC**.

Interrupt 0-31 Set Enable (EN0), offset 0x100

Interrupt 32-63 Set Enable (EN1), offset 0x104

Interrupt 64-95 Set Enable (EN2), offset 0x108

Interrupt 96-113 Set Enable (EN3), offset 0x10C



Value	Description
0	On a read, indicates the interrupt is disabled. On a write, no effect.
1	On a read, indicates the interrupt is enabled. On a write, enables the interrupt.

A bit can only be cleared by setting the corresponding `INT[n]` bit in the **DISn** register.

Enable and Disable Exceptions

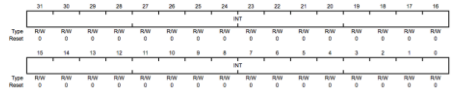
For interrupts, the **DISn** register disable interrupts and show which interrupts are disabled in **NVIC**.

Interrupt 0-31 Clear Enable (DIS0), offset 0x180

Interrupt 32-63 Clear Enable (DIS1), offset 0x184

Interrupt 64-95 Clear Enable (DIS2), offset 0x188

Interrupt 96-113 Clear Enable (DIS3), offset 0x18C



0	On a read, indicates the interrupt is disabled. On a write, no effect.
1	On a read, indicates the interrupt is enabled. On a write, clears the corresponding `INT[n]` bit in the **EN1** register, disabling interrupt [n].

Interrupt Pending Status

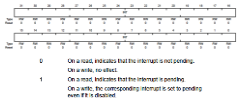
If an interrupt takes place but cannot be executed immediately, it will be pended.

The interrupt-pending status can be accessed through the **Interrupt Set Pending** (**PEND**) and **Interrupt Clear Pending** (**UNPEND**) registers in **NVIC**.

User can set the certain bit of **PEND** to enter its handler by software.

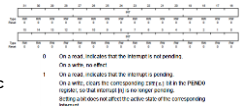
For each interrupt, the NVIC has a 1-bit PEND and a 1-bit UNPEND register.

Interrupt 0-31 Set Pending (PEND0), offset 0x200
Interrupt 32-63 Set Pending (PEND1), offset 0x204
Interrupt 64-95 Set Pending (PEND2), offset 0x208
Interrupt 96-113 Set Pending (PEND3), offset 0x20C



0	On a read, indicates that the interrupt is not pending. On a write, no effect.
1	On a read, indicates that the interrupt is pending. On a write, the corresponding interrupt is set to pending even if it is disabled.

Interrupt 0-31 Clear Pending (UNPEND0), offset 0x280
Interrupt 32-63 Clear Pending (UNPEND1), offset 0x284
Interrupt 64-95 Clear Pending (UNPEND2), offset 0x288
Interrupt 96-113 Clear Pending (UNPEND3), offset 0x28C



0	On a read, indicates that the interrupt is not pending. On a write, no effect.
1	On a read, indicates that the interrupt is pending. On a write, clears the corresponding `INT[n]` bit in the PEND0 register, so that interrupt [n] is no longer pending. Writing a 0 does not affect the active state of the corresponding interrupt.

Interrupt Active Status

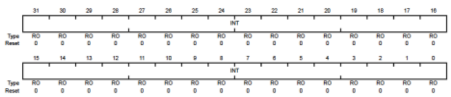
Similarly, for each interrupt the NVIC has a Active status register in **NVIC**.

Interrupt 0-31 Active Bit (ACTIVE0), offset 0x300

Interrupt 32-63 Active Bit (ACTIVE1), offset 0x304

Interrupt 64-95 Active Bit (ACTIVE2), offset 0x308

Interrupt 96-127 Active Bit (ACTIVE3), offset 0x30C



Value	Description
0	The corresponding interrupt is not active.
1	The corresponding interrupt is active, or active and pending.

Software Interrupts

Software interrupts can be generated by using:

1. The PEND register
2. The Software Trigger Interrupt Register (STIR) in **SCB**

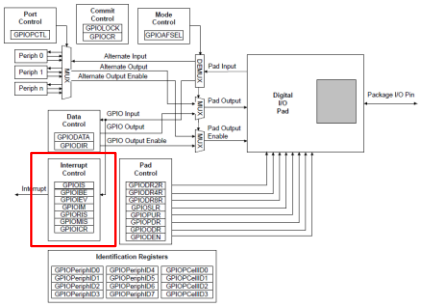
Software Trigger Interrupt Register (0xE000EF00)

Bits	Name	Type	Reset Value	Description
8:0	INTID	W	–	Writing the interrupt number sets the pending bit of the interrupt; for example, write 0 to pend external interrupt #0

System exceptions (NMI, faults, PendSV, and so on) cannot be pended using STIR.

Setup Exceptions at a Peripheral

Example: Setting up Interrupt for GPIO



Define Interrupt Conditions

GPIO Interrupt Sense (GPIOIS) register: setting a bit, detect levels on the pin; otherwise, detect edges

GPIO Interrupt Both Edges (GPIOIBE) register: when GPIOIS is set to detect edges, setting a bit in GPIOIBE enables the pin to detect both rising and falling edges; otherwise, the pin is controlled by the GPIOIEV register

GPIO Interrupt Event (GPIOIEV) register: setting a bit, detect rising edges (or high levels); otherwise, detect falling edges (or low levels), depending on the settings of GPIOIS

Other Interrupt Control Registers

GPIO Interrupt Mask (GPIOIM): setting a bit, allows the pin to generate interrupts; otherwise, disable interrupts

Enable pins to generate interrupts

GPIO Raw Interrupt Status (GPIORIS): A bit is set when an interrupt condition occurs on the corresponding GPIO pin; otherwise, RAZ.

GPIO Masked Interrupt Status (GPIOMIS): If a bit is set, the corresponding interrupt has triggered an interrupt to the interrupt controller; otherwise, either no interrupt has been generated, or the interrupt is masked

In the handler, use these registers to find out the specific interrupt source

GPIO Interrupt Clear (GPIOICR): Writing a 1 to a bit in this register clears the corresponding interrupt bit in the GPIORIS and GPIOMIS registers

In the handler, after finding out the interrupt source, use this register to clear interrupt request and status

How to configure an Interrupt?

At the processor

- Program on PRIMASK, FAULTMASK, BASEPRI registers to mask exceptions and interrupts
- All exceptions are unmasked by default

```
//clear PRIMASK to enable exceptions
IntMasterEnable();

//set PRIMASK to disable exceptions
//with priority not higher than 0
IntMasterDisable();

//set BASEPRI to disable exceptions
//with priority not higher than 0x80
IntPriorityMaskSet(0x80);

//Get current interrupt priority mask
Mask = IntPriorityMaskGet();
```

At the SCB and NVIC side

- Set up the priority group register (Group 0 by default)
- Set up the priority level for the interrupt
- Enable the interrupt in NVIC

```
// Set the priority grouping for the
// interrupt controller to 2 bits
IntPriorityGroupingSet(2);

Group = IntPriorityGroupingGet();

// Set the USB0 interrupt priority to the
// highest priority.
IntPrioritySet(INT_USB0, 0);

Pri = IntPriorityGet(INT_UART0);

// Enable the UART0 interrupt in NVIC
IntEnable(INT_UART0);
// Disable the UART0 interrupt in NVIC
IntDisable(INT_UART0);
```

How to configure an Interrupt?

At the peripheral side

- Enable the peripheral (setup the **RCGCXXX**)
- Configure the interrupt type for the peripheral
- Enable to generate interrupt in the peripheral

```
//set the interrupt type for PJ0
GPIOIntTypeSet(GPIO_PORTJ_BASE,
GPIO_PIN_0, GPIO_FALLING_EDGE);

//Enables the specified GPIO interrupts
GPIOIntEnable(GPIO_PORTJ_BASE,
GPIO_INT_PIN_0)
```

Write an interrupt service routine (ISR)

- Identify the interrupt source
- Clear the interrupt request

```
//Gets interrupt status for the specified
//GPIO port, true for masked status, false
//for raw status
Status = GPIOIntStatus(GPIO_PORTJ_BASE,
true);

//Clears the specified interrupt sources
GPIOIntClear(GPIO_PORTJ_BASE,
GPIO_INT_PIN_0)
```

Register the ISR in the interrupt vector table

The Cortex-M3/M4 Embedded Systems:

TM4C1294NCPDT Microcontroller – Universal Asynchronous Receivers/Transmitters

Refer to Chapter 16 in the reference book

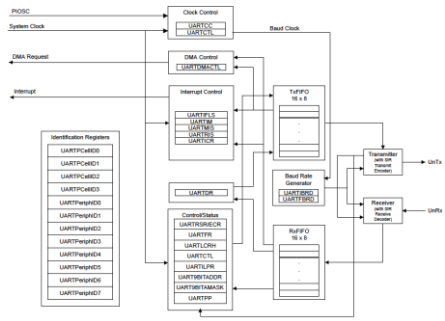
"Tiva™ TM4C1294NCPDT Microcontroller- DATA SHEET"

Universal Asynchronous Receivers/Transmitters (UARTs)

The TM4C1294NCPDT controller includes eight UARTs with the following features:

- Programmable baud-rate generator (up to 7.5Mbps for regular speed or 15 Mbps for high speed)
- Separate 16-entry transmit (TX) and receive (RX) FIFOs
- Programmable FIFO length
- FIFO trigger levels of 1/8, 1/4, 1/2, 3/4, and 7/8
- False-start bit detection
- Line-break generation and detection
- Fully programmable serial interface characteristics (data bits, parity bit, stop bits)
- Standard FIFO-level and End-of-Transmission interrupts

Block Diagram



Functional Description

Each UART performs the functions of parallel-to-serial and serial-to-parallel conversions

The UART is configured to transmit and/or receive via the **TXE** and **RXE** bits of the **UART Control (UARTCTL)** register, enabled by default

the UART must be disabled by clearing the **UARTEN** bit in **UARTCTL** before being programmed

Transmit/Receive Logic

The transmit logic performs parallel-to-serial conversion on the data read from the transmit FIFO

- beginning with a start bit
- the data bits (LSB first)
- parity bit, and the stop bits (according to the programmed configuration)

The receive logic performs serial-to-parallel conversion on the received bit stream

- after detecting a valid start pulse
- Overrun, parity, frame error checking, and line-break detection
- Status and data are written to the receive FIFO

Baud-Rate Generation

The baud-rate divisor (BRD) is a 22-bit number

- a 16-bit integer: can be loaded through the **UART Integer Baud-Rate Divisor (UARTIBRD)** register
- a 6-bit fractional part: can be loaded with the **UART Fractional Baud-Rate Divisor (UARTFBRD)** register
- BRD = BRDI + BRDF = UARTSysClk / (ClkDiv * Baud Rate)
  - UARTSysClk is the system clock
  - ClkDiv (baud factor) is either 16 (if HSE in **UARTCTL** is clear) or 8 (if HSE is set)
- BRDI is loaded to **UARTIBRD**
- integer(BRDF * 64 + 0.5) is loaded to **UARTFBRD**

Baud-Rate Generation

The UART generates an internal baud-rate *reference clock* at 8x (*Baud8*) or 16x (*Baud16*) the baud-rate (depending on the setting of the HSE bit (bit 5) in **UARTCTL**)

This reference clock is divided by 8 or 16 to generate the transmit clock and for error detection during receive operations

any changes to the baud-rate divisor must be followed by a write to a **UARTLCRH** register for the changes to take effect

- **UARTIBRD** write / **UARTFBRD** write, and **UARTLCRH** write

Data Transmission

Data received or transmitted is stored in two 16-byte FIFOs

The receive FIFO has an extra four bits per character for status information

For transmitter:

- data is written into the transmit FIFO
- If the UART is enabled, a data frame starts transmitting with the parameters indicated in the **UARTLCRH** register
- The BUSY bit in the **UART Flag (UARTFR)** register is asserted as soon as there is data in the transmit FIFO
- The BUSY bit remains asserted while data is being transmitted until the transmit FIFO is empty

Data Transmission

For receiver:

- Checks a start bit: When data input goes low (a start bit) from high (idle) for eight cycles of Baud16 (HSE clear) or four cycles of Baud8 (HSE set)
- Samples a data bit: the receive counter begins running and data is sampled on the eighth cycle of Baud16 or fourth cycle of Baud8
- Detected invalid start bit is ignored
- The parity bit is then checked if parity mode is enabled
- Frame is defined in the **UARTLCRH** register
- Lastly, a valid stop bit is confirmed if the UnRx signal is High, otherwise a framing error has occurred

FIFO Operation

The UART has two 16-entry FIFOs: one for Rx, one for Tx

Both FIFOs are accessed via the **UART Data (UARTDR)** register

- Read operations return a 12-bit value consisting of 8 data bits and 4 error flags
- Write operations place 8-bit data in the transmit FIFO
- Out of reset, both FIFOs are disabled and act as 1-byte-deep holding registers (the bottom word of the FIFOs))
- The FIFOs are enabled by setting the FEN bit in **UARTLCRH**
- FIFO status can be monitored via the **UART Flag (UARTFR)** register and the **UART Receive Status (UARTRSR)** register: the **UARTFR** register contains empty and full flags (TXFE, TXFF, RXFE, and RXFF bits), and the **UARTRSR** register shows overrun status via the OE bit

FIFO Operation

The trigger points at which the FIFOs generate interrupts is controlled via the **UART Interrupt FIFO Level Select (UARTIFLS)** register

- Both FIFOs can be individually configured
- Available configurations include ⅛, ¼, ½, ¾, and ⅞
- For example, if ¾ is selected for the receive FIFO, the UART generates an interrupt after ¾ X 16 = 12 bytes are received
- Out of reset, both FIFOs are configured to trigger an interrupt at the ½ mark

Interrupts

The UART can generate interrupts when the following conditions are observed:

- Overrun Error
- Break Error
- Parity Error
- Framing Error
- Receive Timeout
- Transmit (when condition defined in the TXIFLSEL bit in the **UARTIFLS** register is met, or when the last bit of all transmitted data leaves the serializer, i.e., EOT bit in **UARTCTRL**)
- Receive (when condition defined in the RXIFLSEL bit in the **UARTIFLS** register is met)

Interrupts

All of the interrupt events are **OR**ed together before being sent to the interrupt controller

Software can service multiple interrupt events in a single ISR by reading the **UART Masked Interrupt Status (UARTMIS)** register

The interrupt events can be masked via the **UART Interrupt Mask (UARTIM)** register

If interrupts are not used, the raw interrupt status is always visible via the **UART Raw Interrupt Status (UARTRIS)** register

Interrupts

Interrupts are always cleared for both the **UARTMIS and UARTRIS** registers, when writing a **1** to the corresponding bit in the **UART Interrupt Clear (UARTICR)** register

When the receive FIFO is not empty, and no further data is received over a 32-bit period, a receive timeout interrupt is asserted

The receive timeout interrupt is cleared:

- when the FIFO becomes empty through reading all the data
- when a **1** is written to the corresponding bit in the **UARTICR** register

Loopback Operation

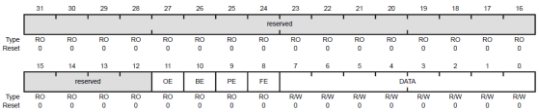
The UART can be placed into an internal loopback mode for diagnostic or debug work

- Set the LBE bit in the **UARTCTL** register
- Data transmitted on the UnTx output is received on the UnRx input
- Note that the LBE bit should be set before the UART is enabled

Register Description: UARTDR, offset 0x000

This register is the data register (the interface to the FIFOs).

- A write to this register initiates a transmission from the UART
- The received data can be retrieved by reading this register



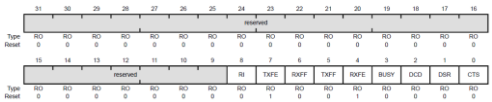
Overrun Error: New data was received when the FIFO was full

Break Error: the receive data input was held Low for longer than a full-frame transmission time

Framing Error: The received character does not have a valid stop bit

Register Description: UARTFR, offset 0x018

This register is the flag register.



TXFE: UART Transmit FIFO Empty

RXFF: UART Receive FIFO Full

TXFF: UART Transmit FIFO Full

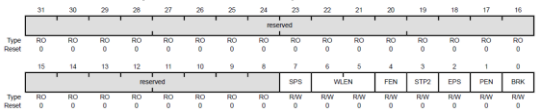
RXFE: UART Receive FIFO Empty

BUSY: UART is busy transmitting data (This bit is set as soon as the transmit FIFO becomes non-empty)

Register Description: UARTLCRH, offset 0x02C

This register is the line control register

- Set up serial parameters such as data length, parity, and stop bit selection
- When updating the baud-rate divisor (**UARTIBRD** and/or **UARTIFRD),** the **UARTLCRH** register must also be written



WLEN: UART Word Length, 5, 6, 7, 8 bits

FEN: UART Enable FIFOs

STP2: UART Two Stop Bits Select

EPS: UART Even Parity Select

PEN: UART Parity Enable

BRK: UART Send Break

Register Description: UARTCTL, offset 0x030

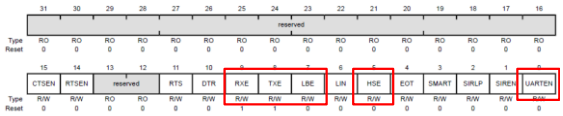
This register is the control register

If software requires a configuration change in the module, the **UARTEN** bit must be cleared before the configuration changes are written

Sequence for making changes:

1. Disable the UART.
2. Wait for the end of transmission or reception of the current character
3. Flush the transmit FIFO by clearing bit 4 (FEN) in the line control register (**UARTLCRH)**
4. Reprogram the control register
5. Enable the UART

Register Description: UARTCTL



Register Description: Others

UART Integer Baud-Rate Divisor (UARTIBRD)
UART Fractional Baud-Rate Divisor (UARTFBRD)
UART Interrupt FIFO Level Select (UARTIFLS)
UART Interrupt Mask (UARTIM)
UART Raw Interrupt Status (UARTRIS)
UART Masked Interrupt Status (UARTMIS)
UART Interrupt Clear (UARTICR)

Initialization and Configuration

To enable and initialize the UART, the following steps are necessary:

1. Enable the UART module using the **RCGCUART** register
2. The clock to the appropriate GPIO module must be enabled via the **RCGCGPIO** register in the System Control module
3. Set the GPIO AFSEL bits for the appropriate pins
4. Configure the GPIO current level and/or slew rate as specified for the mode selected
5. Configure the PMCn fields in the **GPIOPCTL** register to assign the UART signals to the appropriate pins

Initialization and Configuration

Using UART with an example: *the UART clock is 20MHz, 115200 baud rate, data length of 8 bits, one stop bit, no parity, FIFOs disabled, no interrupts*

Get the appropriate baud-rate divisor (BRD) first since the **UARTIBRD** and **UARTFBRD** registers must be written before the UARTLCRH register

$$BRD = 20{,}000{,}000 / (16 * 115{,}200) = 10.8507$$

Therefore,

UARTIBRD = 10,

UARTFBRD = INTEGER(0.8507 * 64 + 0.5) = 54

Initialization and Configuration

With the BRD values, the UART configuration is written to the module in the following order:

1. Disable the UART (clear the UARTEN bit in the **UARTCTL** register)
2. Write the integer portion of the BRD to the **UARTIBRD** register
3. Write the fractional portion of the BRD to the **UARTFBRD** register
4. Write the desired serial parameters to the **UARTLCRH** register (0x0000 0060 in this case)
5. Enable the UART (set the UARTEN bit in the **UARTCTL** register)

NOTE 1: the UART module clock must be enabled before the registers can be programmed

NOTE 2: The UART must be disabled before any of the control registers are reprogrammed

UART Programming with PDL (1)

Initialization of UART

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0); //Clocking UART0
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA); //Clocking GPIO Port A
while(!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOA));
GPIOPinConfigure(GPIO_PA0_U0RX); // Set PA0 and PA1 as UART pins.
GPIOPinConfigure(GPIO_PA1_U0TX);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

//disables UART0, waits for the end of transmission of the current
//character, and flushes the transmit FIFO
UARTDisable(UART0_BASE);

//selects the baud clock source, UART_CLOCK_SYSTEM or UART_CLOCK_PIOSC
UARTClockSourceSet(UART0_BASE, UART_CLOCK_SYSTEM)

//Sets the baud rate, 8-bit byte, no parity, one stop bit
//then enables UART
UARTConfigSetExpClk(UART0_BASE, ui32SysClock, 115200,
                    (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
                    UART_CONFIG_PAR_NONE));
```

UART Programming with PDL (2)

Data transmission with UART

Blocking transmission:

```
//gets a character from the receive FIFO of UART0. If no character
//available, it waits until a character is received.
value = UARTCharGet(UART0_BASE);
//sends a character to the transmit FIFO of UART0. If FIFO is full,
//it waits until there is space available.
UARTCharPut(UART0_BASE, 'a');
```

Non-blocking transmission:

```
//gets a character from the receive FIFO of UART0. If no character
//available, return -1. Use UARTCharsAvail() to check receive FIFO.
value = UARTCharGetNonBlocking(UART0_BASE);
//sends a character to the transmit FIFO of UART0. Returns true if
//sent; otherwise, returns false.
status = UARTCharPutNonBlocking(UART0_BASE, 'a');
```

UART Programming with PDL (3)

Interrupt-driven programming with UART

```
//Enables individual UART interrupt sources
UARTIntEnable(UART0_BASE, UART_INT_RX | UART_INT_RT);

//In the ISR, checks the interrupt source, true for masked interrupt
//status and false for raw interrupt status
uart0_int_status = UARTIntStatus(UART0_BASE, true);
// In the ISR, clears the interrupt source
UARTIntClear(UART0_BASE, uart0_int_status);
// In the ISR, checks the interrupt source and process the event
If (uart0_int_status & UART_INT_RX) {
    //read character from the receive FIFO
} else {
    //send the next character to the transmit FIFO
}
```

NOTE: Do not use blocking transmission in an ISR