LECTURE NOTES
ON

MICRO CONTROLLERS AND PROGRAMMABLE DIGITAL SIGNAL PROCESSING

(BESB02)

I M.TECH I semester ECE (ES)

(AUTONOMOUS-R18)

Presented By:

Mr.K.Chaitanya

(Assistant Professor)

ELECRTONICS AND COMMUNICATION ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

DUNDIGAL, HYDERABAD - 500043

UNIT-I
ARM Cortex-M3 processor

UNIT-I
ARM Cortex-M3 processor

The ARM Cortex-M3 processor, the first of the Cortex generation of proces sors released by ARM in 2006, was primarily designed to target the 32-bit microcontroller market. The Cortex-M3 processor provides excellent performance at low gate count and comes with many new features previously available only in high-end processors.

Applications:

i) Low-cost microcontrollers:

The cortex M3 processor is ideally suited for low-cost micro controllers, which are commonly used in consumer products. Low power, high performance, ease-of-use are the advantages.

ii) Automotive:

The cortex M3 has high performance efficiency and low interrupts latency, allowing to be used in real time systems.

iii) Data communication:

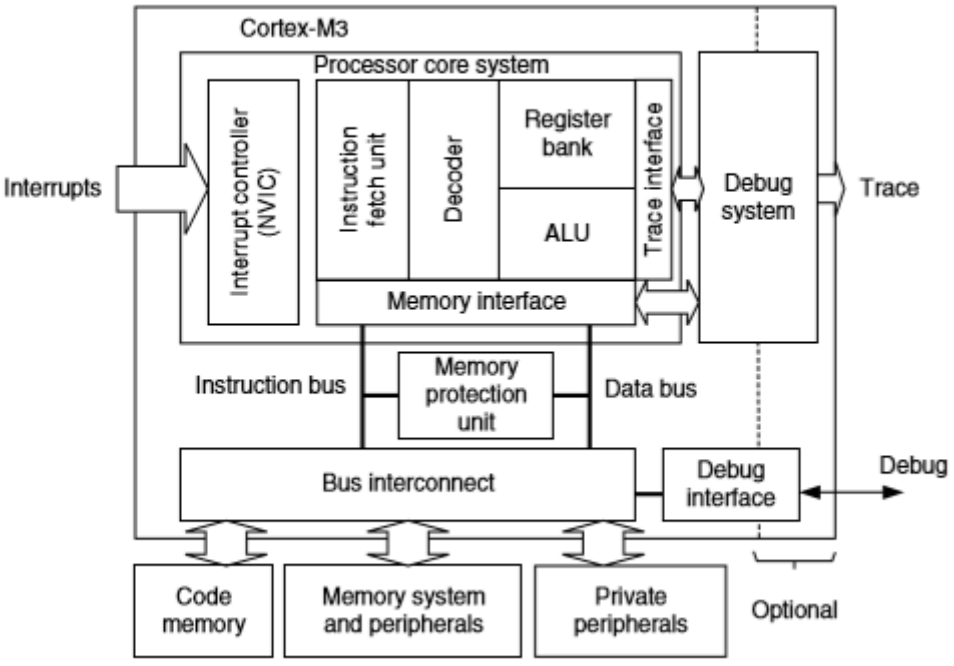
The processor's low power and high efficiency, coupled with Thumb-2 instructions, make cortex M3 ideal for many communication applications. (Bluetooth, Zigbee)

iv) Industrial control:

In industrial control applications simplicity, fast response and reliability are key factors. Cortex M3 has low interrupt latency so is best suited.

v) Consumer products:

The cortex M3 is a small processor and is highly efficient and low in power and supports an MPU enabling complex software to execute while providing robust memory protection.



A Simplified View of the Cortex-M3

Programming model:

REGISTERS

The Cortex-M3 processor has registers R0 through R15. R13 (the stack poin ter) is banked, with only one copy of the R13 visible at a time.

R0–R12: General-Purpose Registers

R0–R12 are 32-bit general-purpose registers for data operations. Some 16-bit Thumb instructions can only access a subset of these registers (low register, R0–R7).

R13: Stack Pointers

The Cortex-M3 contains two stack pointers (R13). They are banked so that only one is visible at a time.

The two stack pointers are as follows:

- **Main Stack Pointer (MSP):**

The default stack pointer, used by the operating system (OS) kernel and exception handlers

- **Process Stack Pointer (PSP):** Used by user application code the lowest 2 bits of the stack pointers are always 0, which means they are always word aligned.

R14: The Link Register

When a subroutine is called, the return address is stored in the link register.

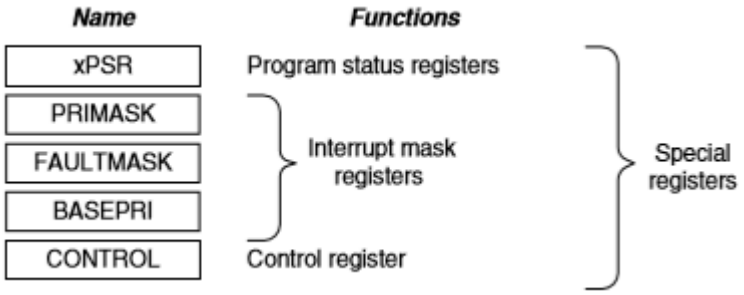
R15: The Program Counter The program counter is the current program address. This register can be written to control the program flow.

Special Registers

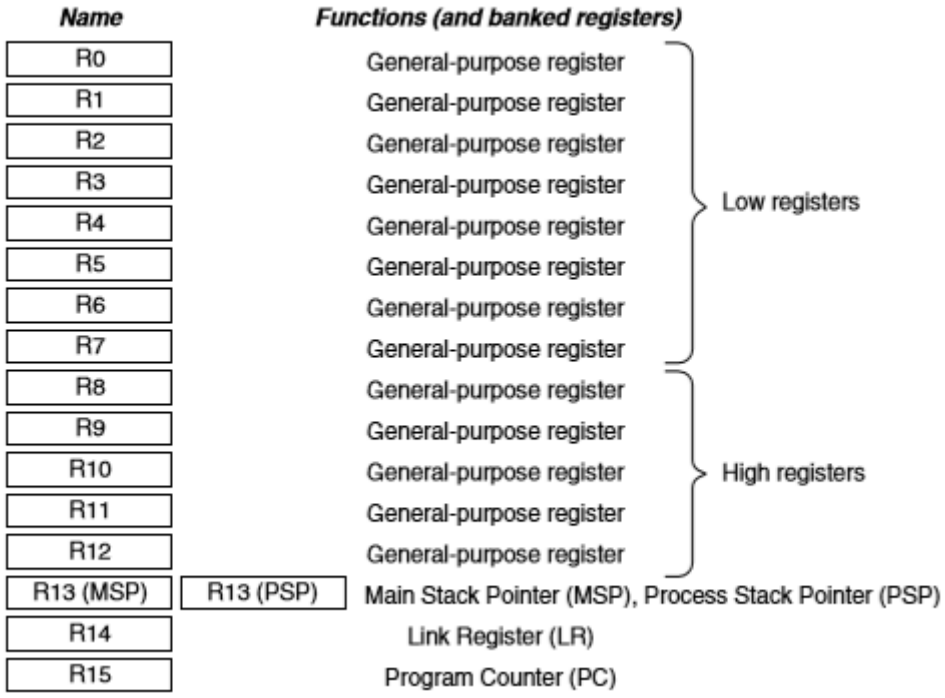
The Cortex-M3 processor also has a number of special registers.They are as follows:

Program Status registers (PSRs)

- Interrupt Mask registers (PRIMASK, FAULTMASK, and BASEPRI)
- Control register (CONTROL)
- These registers have special functions and can be accessed only by special instructions. They cannot be used for normal data processing.



Special Registers in the Cortex-M3



Registers in the Cortex-M3

Table 2.1 Special Registers and Their Functions

Register	Function
xPSR	Provide arithmetic and logic processing flags (zero flag and carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and hard fault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

Special Registers and Their Functions

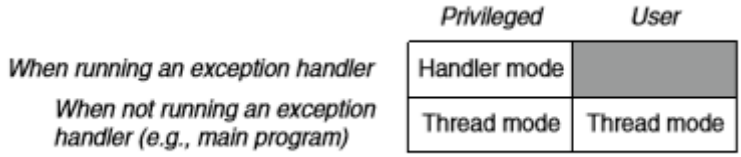
OPERATION MODES

The Cortex-M3 processor has two modes and two privilege levels. The operation modes (thread mode and handler mode) determine whether the processor is running a normal program or running an exception handler like an interrupt handler or system exception handler (see Figure 2.4). The privilege levels (privileged level and user level) provide a mechanism for safeguarding memory accesses to critical regions as well as providing a basic security model.

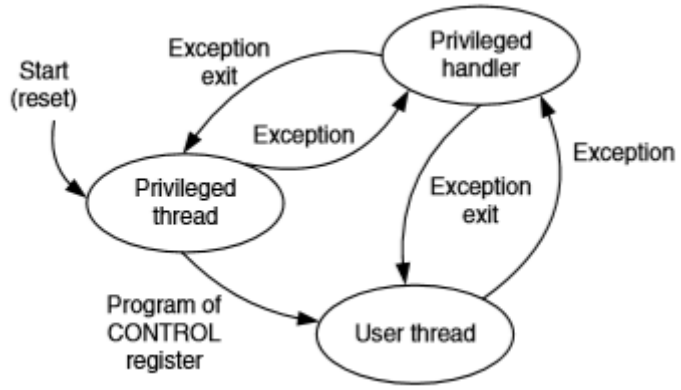
When the processor is running a main program (thread mode), it can be either in a privileged state or a user state, but exception handlers can only be in a privileged state. When the processor exits reset, it is in thread mode, with privileged access rights. In the privileged state, a program has access to all memory ranges (except when prohibited by MPU settings) and can use all supported instructions. Software in the privileged access level can switch the program into the user access level using the control register.

When an exception takes place, the processor will always switch back to the privileged state and return to the previous state when exiting the exception handler. A user program cannot change back to the privileged state by writing to the control register (see Figure 2.5). It has to go through an exception handler that programs the control register to switch the processor back into the privileged access level when returning to thread mode.

The separation of privilege and user levels improves system reliability by preventing system configuration registers from being accessed or changed by some untrusted programs. If an MPU is available, it can be used in conjunction with privilege levels to protect critical memory locations, such as programs and data for OSs. For example, with privileged accesses, usually used by the OS kernel, all memory locations can be accessed (unless prohibited by MPU setup). When the OS launches a user application, it is likely to be executed in the user access level to protect the system from failing due to a crash of untrusted user programs.



Operation Modes and Privilege Levels in Cortex-M3



Allowed Operation Mode Transitions

INTERRUPTS AND EXCEPTIONS

The Cortex-M3 processor implements a new exception model, introduced in the ARMv7-M architecture. This exception model differs from the traditional ARM exception model, enabling very efficient exception handling. It has a number of system exceptions plus a number of external Interrupt Request (IRQs) (external interrupt inputs). There is no fast interrupt (FIQ) (fast interrupt in ARM7/ARM9/ ARM10/ARM11) in the Cortex-M3; however, interrupt priority handling and nested interrupt support are now included in the interrupt architecture. Therefore, it is easy to set up a system that supports nested interrupts (a higher-priority interrupt can override or preempt a lower-priority interrupt handler) and that behaves just like the FIQ in traditional ARM processors. The interrupt features in the Cortex-M3 are implemented in the

NVIC. Aside from supporting external interrupts, the Cortex-M3 also supports a number of internal exception sources, such as system fault handling. As a result, the Cortex-M3 has a number of predefined exception types, as shown in Table 2.2.

Low Power and High Energy Efficiency

The Cortex-M3 processor is designed with various features to allow designers to develop low power and high energy efficient products. First, it has sleep mode and deep sleep mode supports, which can work with various system-design methodologies to reduce power consumption during idle period.Second, its low gate count and design techniques reduce circuit activities in the processor to allow active power to be reduced. In addition, since Cortex-M3 has high code density, it has lowered the program size requirement.

At the same time, it allows processing tasks to be completed in a short time, so that the processor can return to sleep modes as soon as possible to cut down energy use. As a result, the energy efficiency of Cortex-M3 is better than many 8-bit or 16-bit microcontrollers.

Starting from Cortex-M3 revision 2, a new feature called Wakeup Interrupt Controller (WIC) is available. This feature allows the whole processor core to be powered down, while processor states are retained and the processor can be returned to active state almost immediately when an interrupt takes place. This makes the Cortex-M3 even more suitable for many ultra-low power applications that previously could only be implemented with 8-bit or 16-bit microcontrollers.

Table 2.2 Cortex-M3 Exception Types

Exception Number	Exception Type	Priority (Default to 0 if Programmable)	Description
0	NA	NA	No exception running
1	Reset	–3 (Highest)	Reset
2	NMI	–2	NMI (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 (prefetch abort or data abort)
6	Usage fault	Programmable	Program error
7–10	Reserved	NA	Reserved
11	SVCall	Programmable	Supervisor call
12	Debug monitor	Programmable	Debug monitor (break points, watchpoints, or external debug request)
13	Reserved	NA	Reserved
14	PendSV	Programmable	Pendable request for system service
15	SYSTICK	Programmable	System tick timer
16	IRQ #0	Programmable	External interrupt #0
17	IRQ #1	Programmable	External interrupt #1
…	…	…	…
255	IRQ #239	Programmable	External interrupt #239

The number of external interrupt inputs is defined by chip manufacturers. A maximum of 240 external interrupt inputs can be supported. In addition, the Cortex-M3 also has an NMI interrupt input. When it is asserted, the NMI-ISR is executed unconditionally.

Reset Sequence Instruction Set

The Cortex-M3 supports the Thumb-2 instruction set. This is one of the most important features of the Cortex-M3 processor because it allows 32-bit instructions and 16-bit instructions to be used together for high code density and high efficiency. It is flexible and powerful yet easy to use. In previous ARM processors, the central processing unit (CPU) had two operation states: a 32-bit ARM state and a 16-bit Thumb state.

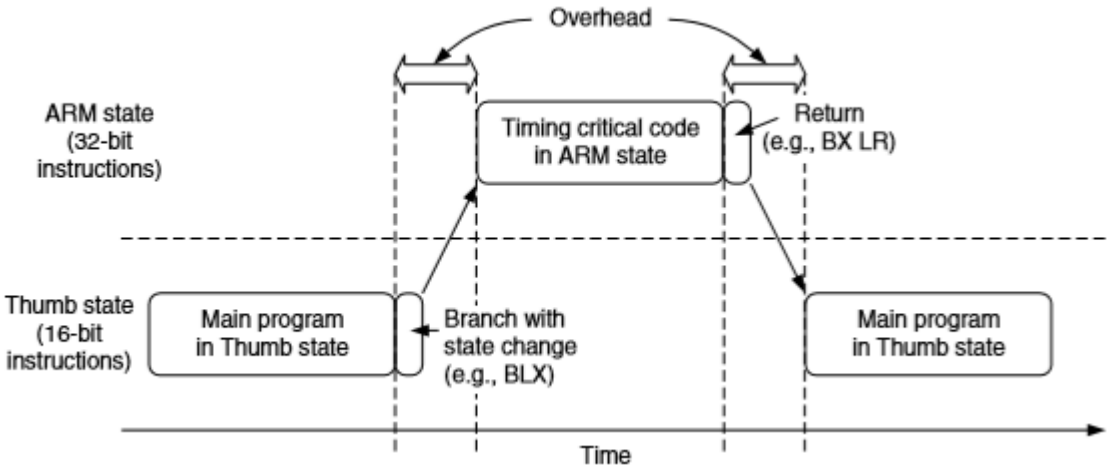
In the ARM state, the instructions are 32 bits and can execute all supported instructions with very high performance. In the Thumb state, the instructions are 16 bits, so there is a much higher instruction code density, but the Thumb state does not have all the functionality of ARM instructions and may require more instructions to complete certain types of operations. To get the best of both worlds, many applications have mixed ARM and Thumb codes. However, the mixed-code arrangement does not always work best. There is overhead (in terms of execution time and instruction space, see Figure 2.7) to switch between the states, and ARM and Thumb codes might need to be compiled separately in different files.

This increases the complexity of software development and reduces maximum efficiency of the CPU core. With the introduction of the Thumb-2 instruction set, it is now possible to handle all processing requirements in one operation state. There is no need to switch between the two. In fact, the Cortex-M3 does not support the ARM code. Even interrupts are now handled with the Thumb state. (Previously, the ARM core entered interrupt handlers in the ARM state.) Since there is no need to switch between states, the Cortex-M3 processor has a number of advantages over traditional ARM processors, such as:

- No state switching overhead, saving both execution time and instruction space
- No need to separate ARM code and Thumb code source files, making software development andmaintenance easier

It's easier to get the best efficiency and performance, in turn making it easier to write software,

Because there is no need to worry about switching code between ARM and Thumb to try to get the best density/performance



Switching between ARM Code and Thumb Code in Traditional ARM Processors Such as the ARM7

The Cortex-M3 processor has a number of interesting and powerful instructions. Here are a few examples:

- UFBX, BFI, and BFC: Bit field extract, insert, and clear instructions • UDIV and SDIV: Unsigned and signed divide instructions
- WFE, WFI, and SEV: Wait-For-Event, Wait-For-Interrupts, and Send-Event; these allow the processor to enter sleep mode and to handle task synchronization on

multiprocessor systems

- MSR and MRS: Move to special register from general-purpose register and move special register to general-purpose register; for access to the special registers

Since the Cortex-M3 processor supports the Thumb-2 instruction set only, existing program code for ARM needs to be ported to the new architecture. Most C applications simply need to be recompiled using new compilers that support the Cortex-M3. Some assembler codes need modification and porting to use the new architecture and the new unified assembler framework.

Note that not all the instructions in the Thumb-2 instruction set are implemented on the Cortex-M3. The ARMv7-M Architecture Application Level Reference Manual only requires a subset of the Thumb-2 instructions to be implemented. For example, coprocessor instructions are not supported on the Cortex-M3 (external data processing engines can be added), and Single Instruction–Multiple Data (SIMD) is not implemented on the Cortex-M3. In addition, a few Thumb instructions are not supported, such as Branch with Link and Exchange (BLX) with immediate (used to switch processor state from Thumb to ARM), a couple of change process state (CPS) instructions, and the SETEND (Set Endian) instructions, which were introduced in architecture v6.

Assembler Language: Unified Assembler Language

To support and get the best out of the Thumb®-2 instruction set, the Unified Assembler Language (UAL) was developed to allow selection of 16-bit and 32-bit instructions and to make it easier to port applications between ARM code and Thumb code by using the same syntax for both. (With UAL, the syntax of Thumb instructions is now the same as for ARM instructions.)

```
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. The choice between whether the instructions are interpreted as traditional Thumb code or the new UAL syntax is normally defined by the directive in the assembly file. For example, with ARM assembler tool, a program code header with “CODE16” directive implies the code is in the traditional Thumb syntax, and “THUMB” directive implies the code is in the new UAL syntax.

One thing you need to be careful with reusing traditional Thumb is that some instructions change the flags in APSR, even if the S suffix is not used. However, when the UAL syntax is used, whether the instruction changes the flag depends on the S suffix. For example,

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

With the new instructions in Thumb-2 technology, some of the operations can be handled by either a Thumb instruction or a Thumb-2 instruction. For example, R0 = R0 + 1 can be implemented as a 16-bit Thumb instruction or a 32-bit Thumb-2 instruction. With UAL, you can specify which instruction you want by adding suffixes:

```
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)
```

The .W (wide) suffix specifies a 32-bit instruction. If no suffix is given, the assembler tool can choose either instruction but usually defaults to 16-bit Thumb code to get a smaller size. Depending on tool support, you may also use the .N (narrow) suffix to specify a 16-bit Thumb instruction. Again, this syntax is for ARM assembler tools. Other assemblers might have slightly different syntax. If no suffix is given, the assembler might choose the instruction for you, with the minimum code size.

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. However, when the immediate data exceed a certain range or when the operation can be better handled with a 32-bit Thumb-2 instruction, the 32-bit instruction will be used.

The 32-bit Thumb-2 instructions can be half word aligned. For example, you can have a 32-bit instruction located in a half word location.

```
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–R7; 32-bit Thumb-2 instructions do not have this limitation. However, use of PC (R15) might not be allowed in some of the instructions. Refer to the ARM v7-M Architecture Application Level Reference Manual

MEMORY MAPS

The Cortex-M3 processor has a fixed memory map (see Figure 5.1). This makes it easier to port software from one Cortex-M3 product to another. For example, components described in previous sections, such as Nested Vectored Interrupt Controller (NVIC) and Memory Protection Unit (MPU), have the same memory locations in all Cortex-M3 products. Nevertheless, the memory map definition allows great flexibility so that manufacturers can differentiate their Cortex-M3-based product from others.

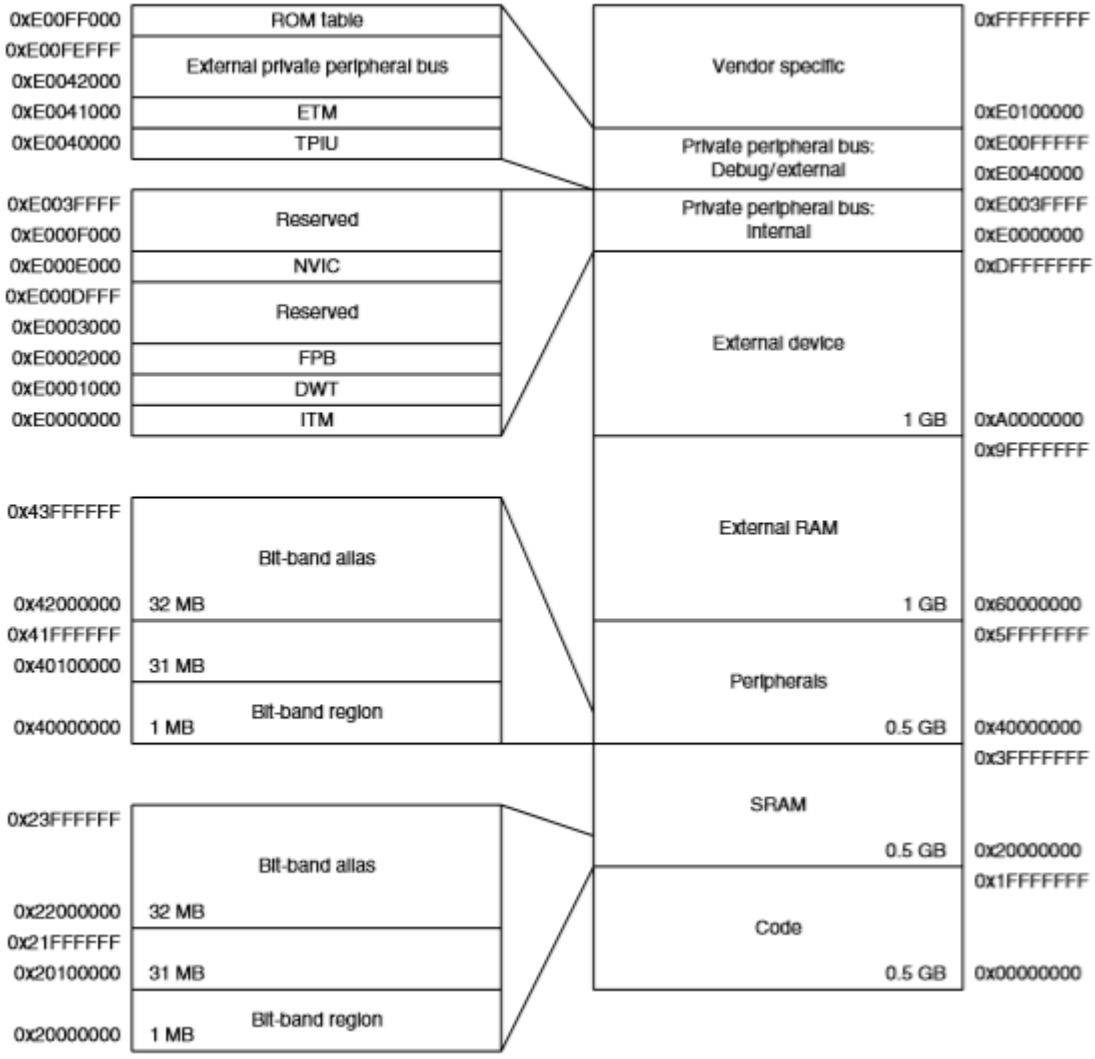
Some of the memory locations are allocated for private peripherals such as debugging components. They are located in the private peripheral memory region. These debugging components include the following:

Fetch Patch and Breakpoint Unit (FPB)
Data Watchpoint and Trace Unit (DWT)
Instrumentation Trace Macrocell (ITM)
Embedded Trace Macrocell (ETM)
Trace Port Interface Unit (TPIU)
ROM table

The details of these components are discussed in later chapters on debugging features. The Cortex-M3 processor has a total of 4 GB of address space. Program code can be located in the code region, the Static Random Access Memory (SRAM) region, or the external RAM region. However, it is best to put the program code in the code region because with this arrangement, the instruction fetches and data accesses are carried out simultaneously on two

separate bus interfaces. The SRAM memory range is for connecting internal SRAM. Access to this region is carried out via the system interface bus.

In this region, a 32-MB range is defined as a bit-band alias. Within the 32-bit-band alias memory range, each word address represents a single bit in the 1-MB bit-band region. A data write access to this bit-band alias memory range will be converted to an atomic READ MODIFY-WRITE operation to the bit-band region so as to allow a program to set or clear individual data bits in the memory. The bit-band operation applies only to data accesses not instruction fetches. By putting Boolean information (single bits) in the bit-band region, we can pack multiple Boolean data in a single word while still allowing them to be accessible individually via bit-band alias, thus saving memory space without the need for handling READ-MODIFY-WRITE in software. More details on bit-band alias can be found later in this chapter. Another 0.5-GB block of address range is allocated to on-chip peripherals.



Cortex-M3 Predefined Memory Map

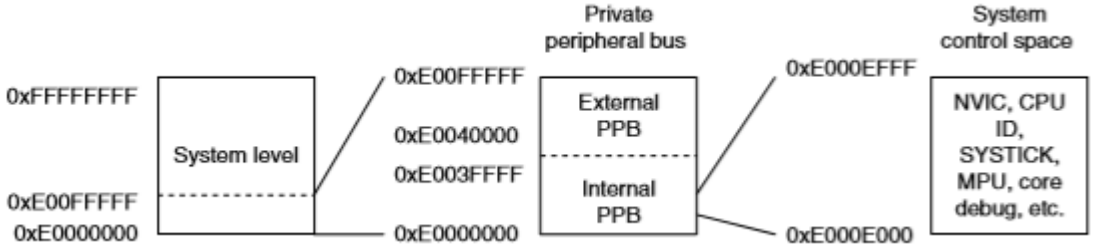
Similar to the SRAM region, this region supports bit-band alias and is accessed via the system bus interface. However, instruction execution in this region is not allowed. The bit-band support in the peripheral region makes it easy to access or change control and status

bits of peripherals, making it easier to program peripheral control. Two slots of 1-GB memory space are allocated for external RAM and external devices. The difference between the two is that program execution in the external device region is not allowed, and there are some differences with the caching behaviors. The last 0.5-GB memory is for the system-level components, internal peripheral buses, external peripheral bus, and vendor-specific system peripherals. There are two segments of the private peripheral bus (PPB):

Advanced High-Performance Bus (AHB) PPB, for Cortex-M3 internal AHB peripherals only; this includes NVIC, FPB, DWT, and ITM

Advance Peripheral Bus (APB) PPB, for Cortex-M3 internal APB devices as well as external peripherals (external to the Cortex-M3 processor); the Cortex-M3 allows chip vendors to add additional on-chip APB peripherals on this private peripheral bus via an APB interface

The NVIC is located in a memory region called the system control space (SCS) (see Figure 5.2). Besides providing interrupt control features, this region also provides the control registers for SYSTICK, MPU, and code debugging control. The remaining unused vendor-specific memory range can be accessed via the system bus interface. However, instruction execution in this region is not allowed. The Cortex-M3 processor also comes with an optional MPU. Chip manufacturers can decide whether to include the MPU in their products The System Control Space.



What we have shown in the memory map is merely a template; individual semiconductor vendors provide detailed memory maps including the actual location and size of ROM, RAM, and peripheral memory locations.

MEMORY ACCESS ATTRIBUTES

The memory map shows what is included in each memory region. Aside from decoding which memory block or device is accessed, the memory map also defines the memory attributes of the access. The memory attributes you can find in the Cortex-M3 processor include the following:

- *Bufferable*: Write to memory can be carried out by a write buffer while the processor continues on next instruction execution.
- *Cacheable*: Data obtained from memory read can be copied to a memory cache so that next time it is accessed the value can be obtained from the cache to speed up the program execution.
- *Executable*: The processor can fetch and execute program code from this memory region.

- *Sharable*: Data in this memory region could be shared by multiple bus masters. Memory system needs to ensure coherency of data between different bus masters in shareable memory region.

The Cortex-M3 bus interfaces output the memory access attributes information to the memory system for each instruction and data transfer. The default memory attribute settings can be overridden if MPU is present and the MPU region configurations are programmed differently from the default. Though the Cortex-M3 processor does not have a cache memory or cache controller, a cache unit can be added on the microcontroller which can use the memory attribute information to define the memory access behaviors. In addition, the cache attributes might also affect the operation of memory controllers for on-chip memory and off-chip memory, depending on the memory controllers used by the chip manufacturers.

The memory access attributes for each memory region are as follows:

- *Code memory region (0x00000000–0x1FFFFFFF):* This region is executable, and the cache attributes iswrite through (WT). You can put data memory in this region as well. If data operations are carried out for this region, they will take place via the data bus interface. Write transfers to this region are bufferable.
  - *SRAM memory region (0x20000000–0x3FFFFFFF):* This region is intended for on-chip RAM. Write transfers to this region are bufferable, and the cache attribute is write back, write allocated (WB-WA). This region is executable, so you can copy program code here and execute it.
  - *Peripheral region (0x40000000–0x5FFFFFFF):* This region is intended for peripherals. The accesses are noncacheable. You cannot execute instruction code in this region (Execute Never, or XN in ARM documentation, such as the Cortex-M3 TRM).
  - *External RAM region (0x60000000–0x7FFFFFFF):* This region is intended for either on-chip or off-chip memory. The accesses are cacheable (WB-WA), and you can execute code in this region.
  - *External RAM region (0x80000000–0x9FFFFFFF):* This region is intended for either on-chip or off-chip memory. The accesses are cacheable (WT), and you can execute code in this region.
  - *External devices (0xA0000000–0xBFFFFFFF):* This region is intended for external devices and/or shared memory that needs ordering/nonbuffered accesses. It is also a nonexecutable region.
  - *External devices (0xC0000000–0xDFFFFFFF):* This region is intended for external devices and/or shared memory that needs ordering/nonbuffered accesses. It is also a nonexecutable region.
  - *System region (0xE0000000–0xFFFFFFFF):* This region is for private peripherals and vendor-specific devices. It is nonexecutable. For the PPB memory range, the accesses are strongly ordered (noncacheable, nonbufferable). For the vendor-specific memory region, the accesses are bufferable and noncacheable.

Note that the Cortex-M3, the code region memory attribute export to external memory system is hardwired to cacheable and nonbufferable. This cannot be overridden by MPU configuration. This update only affects the memory system outside the processor (e.g., level 2 cache and certain types of memory controllers with cache features). Within the processor, the internal write buffer can still be used for write transfers accessing the code region.

DEFAULT MEMORY ACCESS PERMISSIONS

The Cortex-M3 memory map has a default configuration for memory access permissions. This prevents user programs (non-privileged) from accessing system control memory spaces such as the NVIC. The default memory access permission is used when either no MPU is present or MPU is present but disabled. If MPU is present and enabled, the access permission in the MPU setup will determine whether user accesses are allowed.
The default memory access permissions are shown in Table.

Table 5.1 Default Memory Access Permissions

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

Table 5.1 Default Memory Access Permissions *Continued*

Memory Region	Address	Access in User Program
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

When a user access is blocked, the fault exception takes place immediately.

BIT-BAND OPERATIONS

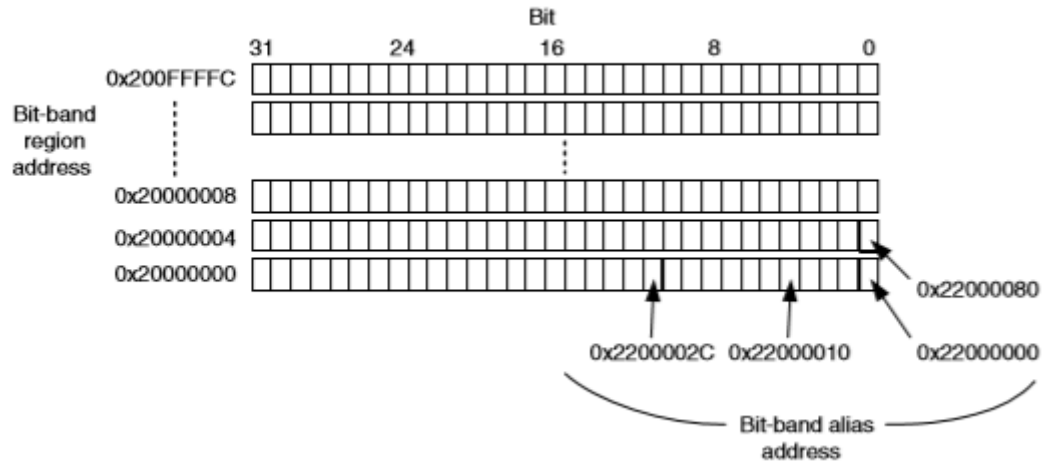
Bit-band operation support allows a single load/store operation to access (read/write) to a single data bit. In the Cortex-M3, this is supported in two predefined memory regions called bit-band regions. One of them is located in the first 1 MB of the SRAM region, and the other is located in the first 1 MB of the peripheral region. These two memory regions can be accessed like normal memory, but they can also be accessed via a separate memory region called the bit-band alias (see Figure 5.3). When the bit-band alias address is used, each individual bit can be accessed separately in the least significant bit (LSB) of each word-aligned address.

For example, to set bit 2 in word data in address 0x20000000, instead of using three instructions to read the data, set the bit, and then write back the result, this task can be carried out by a single instruction (see Figure 5.4). The assembler sequence for these two cases could be like the one shown in Figure 5.5.

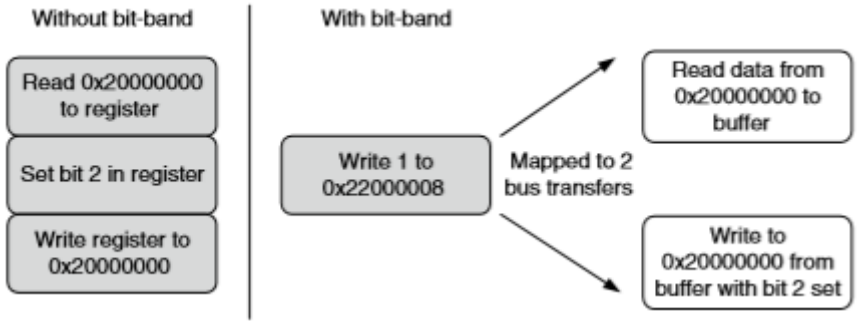
Similarly, bit-band support can simplify application code if we need to read a bit in a memory location. For example, if we need to determine bit 2 of address 0x20000000, we use the steps outlined in Figure 5.6. The assembler sequence for these two cases could be like the one shown in Figure 5.7. Bit-band operation is not a new idea; in fact, a similar feature has existed for more than 30 years on 8-bit microcontrollers such as the 8051. Although the Cortex-M3 does not have special instructions for bit operation, special memory regions are defined so that data accesses to these regions are automatically converted into bit-band operations.

Note that the Cortex-M3 uses the following terms for the bit-band memory addresses:

- Bit-band region: This is a memory address region that supports bit-band operation.
- Bit-band alias: Access to the bit-band alias will cause an access (a bit-band operation) to the bit-band region. (Note: A memory remapping is performed.



Bit Accesses to Bit-Band Region via the Bit-Band Alias.



Write to Bit-Band Alias

Without bit-band

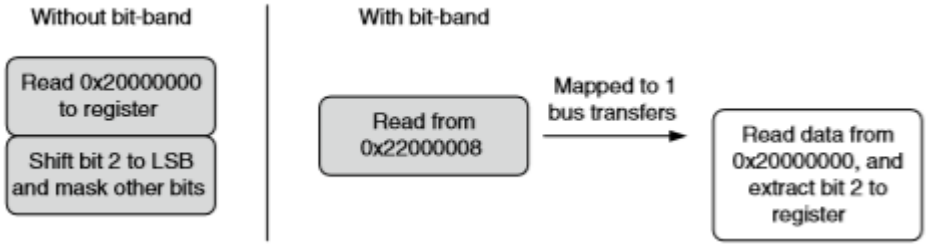
```
LDR   R0,=0x20000000 ; Setup address
LDR   R1, [R0]       ; Read
ORR.W R1, #0x4       ; Modify bit
STR   R1, [R0]       ; Write back result
```

With bit-band

```
LDR   R0,=0x22000008 ; Setup add
MOV   R1, #1         ; Setup dat
STR   R1, [R0]       ; Write
```

Example Assembler Sequence to Write a Bit with and without Bit-Band

Within the bit-band region, each word is represented by an LSB of 32 words in the bit-band alias address range. What actually happens is that when the bit-band alias address is accessed, the address is remapped into a bit-band address. For read operations, the word is read and the chosen bit location is shifted to the LSB of the read return data. For write operations, the written bit data are shifted to the required bit position, and a READ-MODIFY-WRITE is performed.



Read from the Bit-Band Alias.

Without bit-band

```
LDR    R0,=0x20000000 ; Setup address
LDR    R1, [R0]       ; Read
UBFX.W R1, R1, #2, #1 ; Extract bit[2]
```

With bit-band

```
LDR   R0,=0x22000008 ; Setup address
LDR   R1, [R0]       ; Read
```

Read from the Bit-Band Alias

Table 5.2 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]

There are two regions of memory for bit-band operations:
0x20000000–0x200FFFFF (SRAM, 1 MB)
0x40000000–0x400FFFFF (peripherals, 1 MB)

For the SRAM memory region, the remapping of the bit-band alias is shown in Table 5.2. Similarly, the bit-band region of the peripheral memory region can be accessed via bit-band aliased addresses, as shown in Table 5.3.

Table 5.3 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]

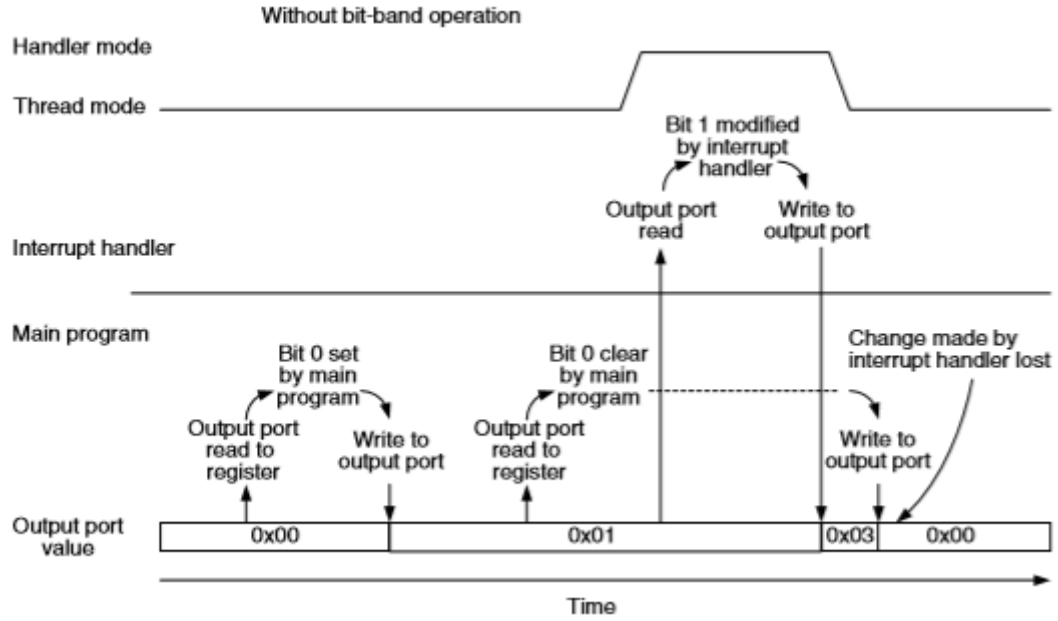
Advantages of Bit-Band Operations

So, what are the uses of bit-band operations? We can use them to, for example, implement serial data transfers in general-purpose input/output (GPIO) ports to serial devices. The application code can be implemented easily because access to serial data and clock signals can be separated.Bit-band operation can also be used to simplify branch decisions. For example, if a branch should be carried out based on 1 single bit in a status register in a peripheral, instead of

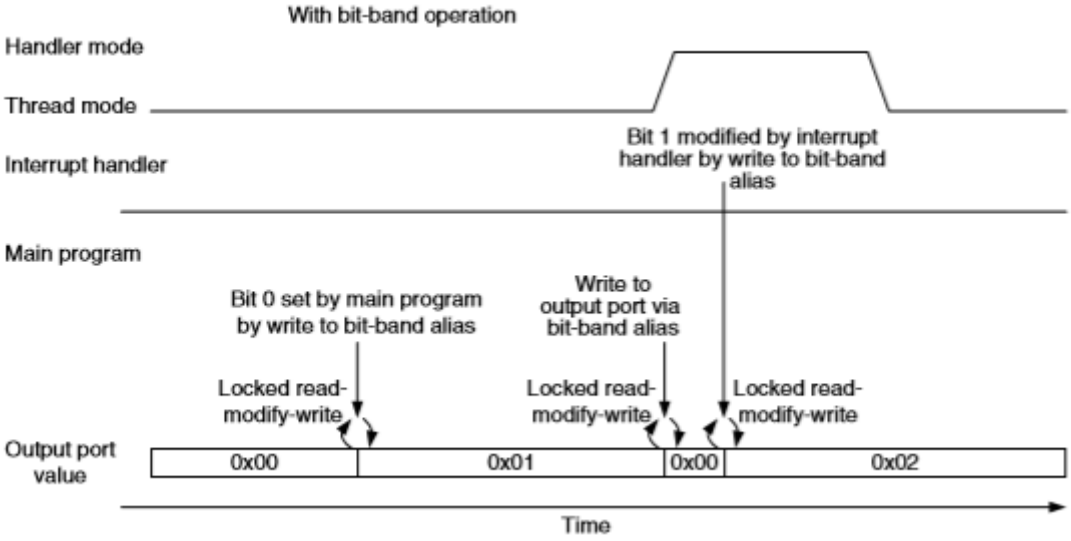
- Reading the whole register
- Masking the unwanted bits
- Comparing and branching
- Reading the status bit via the bit-band alias (get 0 or 1)
- Comparing and branching

Besides providing faster bit operations with fewer instructions, the bit-band feature in the Cortex-M3 is also essential for situations in which resources are being shared by more than one process. One of the most important advantages or properties of a bit-band operation is that it is atomic. In other words, the READ-MODIFY-WRITE sequence cannot be interrupted by other bus activities. Without this behavior in, for example, using a software READ-MODIFY-WRITE sequence, the following problem can occur: consider a simple output port with bit 0 used by a main program and bit 1 used by an interrupt handler. A software-based READ-MODIFY-WRITE operation can cause data conflicts, as shown in Figure.

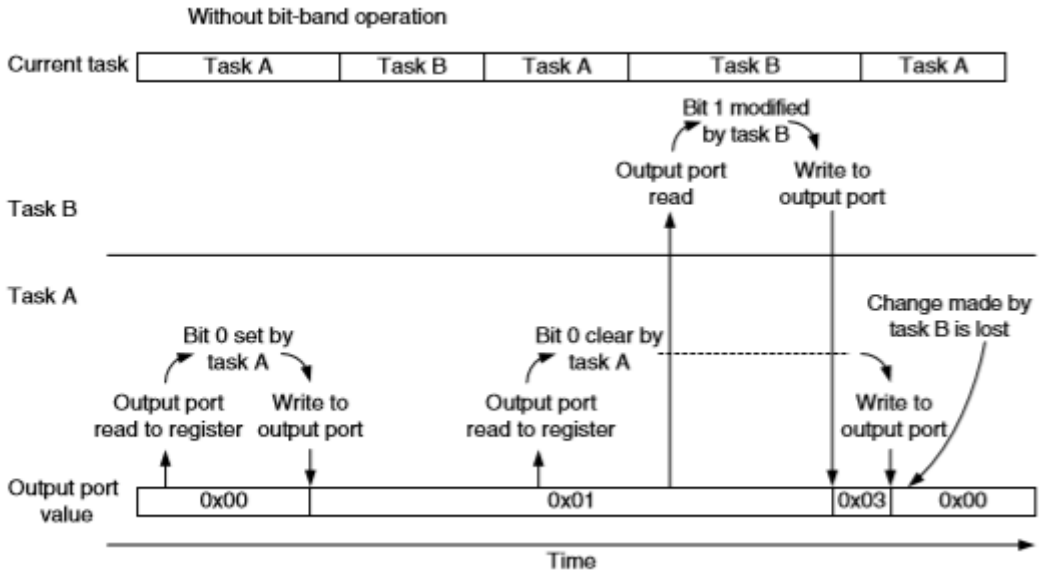
With the Cortex-M3 bit-band feature, this kind of race condition can be avoided because the READMODIFY-WRITE is carried out at the hardware level and is atomic (the two transfers cannot be pulled apart) and interrupts cannot take place between them (see Figure 5.9). Similar issues can be found in multitasking systems. For example, if bit 0 of the output port is used by Process A and bit 1 is used by Process B, a data conflict can occur in software-based READMODIFY-WRITE.



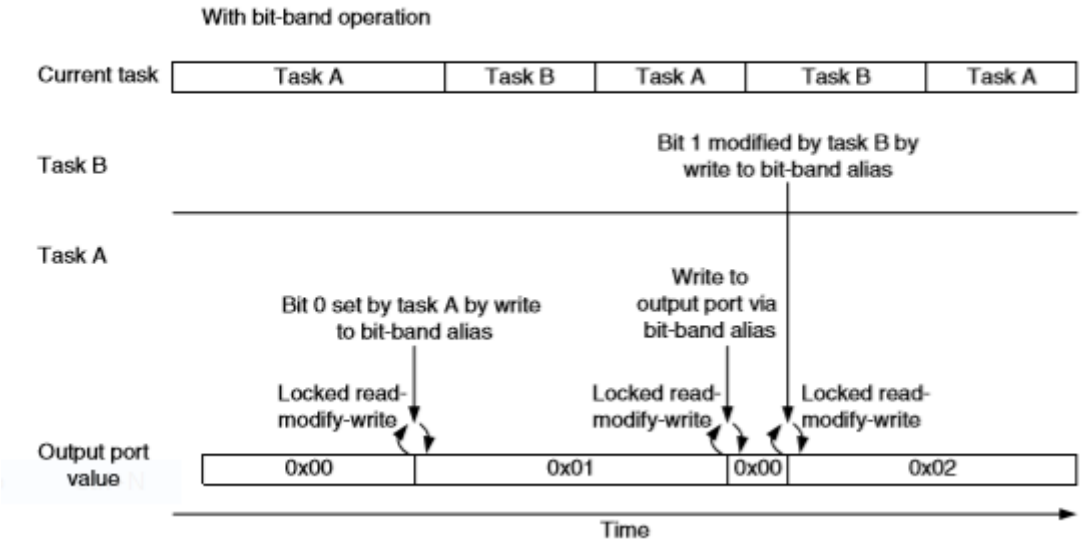
Data Are Lost When an Exception Handler Modifies a Shared Memory Location



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



Data Are Lost When a Different Task Modifies a Shared Memory Location



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

Again, the bit-band feature can ensure that bit accesses from each task are separated so that no data conflicts occur (see Figure 5.11). Besides I/O functions, the bit-band feature can be used for storing and handling Boolean data in the SRAM region. For example, multiple Boolean variables can be packed into one single memory location to save memory space, whereas the access to each bit is still completely separated when the access is carried out via the bit-band alias address range. For system-on-chip (SoC) designers designing a bit-band-capable device, the device's memory address should be located within the bit-band memory, and the lock (HMASTLOCK) signal from the AHB interface must be checked to make sure that writable register contents will not be changed except by the bus when a locked transfer is carried out.

Bit-Band Operation of Different Data Sizes

Bit-band operation is not limited to word transfers. It can be carried out as byte transfers or half word transfers as well. For example, when a byte access instruction

(LDRB/STRB) is used to access a bit-band alias address range, the accesses generated to the bit-band region will be in byte size. The same applies to half word transfers (LDRH/STRH). When you use nonword transfers to bit-band alias addresses, the address value should still be word aligned

UNALIGNED AND EXCLUSIVE TRANSFERS

Unaligned Transfers

The Cortex-M3 supports unaligned transfers on single accesses. Data memory accesses can be defined as aligned or unaligned. Traditionally, ARM processors (such as the RM7/ARM9/ARM10) allow only aligned transfers. That means in accessing memory, a word transfer must have address bit [1] and bit [0] equal to 0, and a half word transfer must have address bit[0] equal to 0. For example, word data can be located at 0x1000 or 0x1004, but it cannot be located in 0x1001, 0x1002, or 0x1003. For half word data, the address can be 0x1000 or 0x1002, but it cannot be 0x1001. So, what does an unaligned transfer look like? Figures 5.12 through 5.16 shows some examples. Assuming that the memory infrastructure is 32-bit (4 bytes) wide, an unaligned transfer can be any word size read/write such that the address is not a multiple of 4, as shown in Figures 5.12–5.14, or when the transfer is in half word size, and the address is not a multiple of 2, as shown in Figures 5.15 and 5.16. All the byte-size transfers are aligned on the Cortex-M3 because the minimum address step is 1 byte.

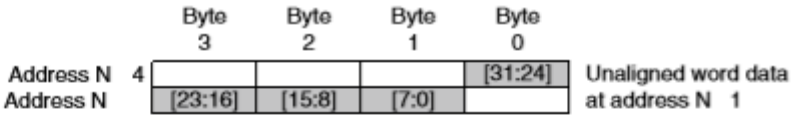


FIGURE 5.12

Unaligned Transfer Example 1.

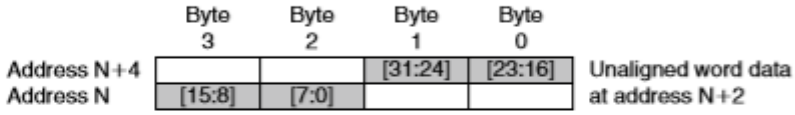


FIGURE 5.13

Unaligned Transfer Example 2.

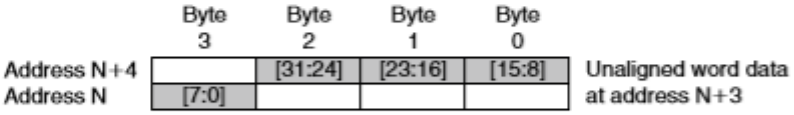


FIGURE 5.14

Unaligned Transfer Example 3.

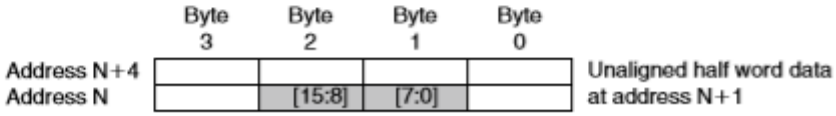


FIGURE 5.15

Unaligned Transfer Example 4.

In the Cortex-M3, unaligned transfers are supported in normal memory accesses (such as LDR, LDRH, STR, and STRH instructions). There are a number of limitations:

- Unaligned transfers are not supported in Load/Store multiple instructions.
- Stack operations (PUSH/POP) must be aligned.
- Exclusive accesses (such as LDREX or STREX) must be aligned; otherwise, a fault exception (usage fault) will be triggered.
- Unaligned transfers are not supported in bit-band operations. Results will be unpredictable if you attempt to do so.

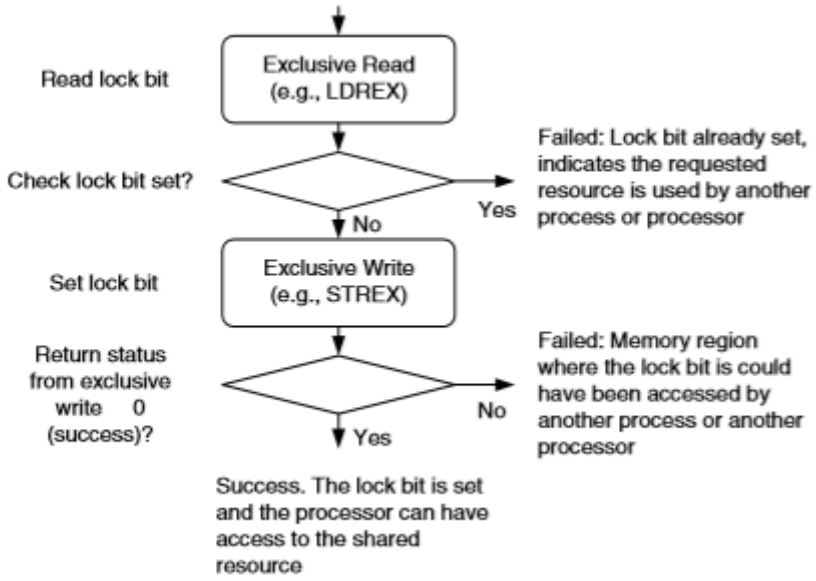
When unaligned transfers are used, they are actually converted into multiple aligned transfers by the processor's bus interface unit. This conversion is transparent, so application programmers do not have to worry about it. However, when an unaligned transfer takes place, it is broken into separate transfers, and as a result, it takes more clock cycles for a single data access and might not be good for situations in which high performance is required. To get the best performance, it's worth making sure that data are aligned properly.

It is also possible to set up the NVIC so that an exception is triggered when an unaligned transfer takes place. This is done by setting the UNALIGN_TRP (unaligned trap) bit in the configuration control register in the NVIC (0xE000ED14). In this way, the Cortex-M3 generates usage fault exceptions when unaligned transfers take place. This is useful during software development to test whether an application produces unaligned transfers.

EXCLUSIVE ACCESSES

You might have noticed that the Cortex-M3 has no SWP instruction (swap), which was used for semaphore operations in traditional ARM processors like ARM7TDMI. This is now being replaced by exclusive access operations. Exclusive accesses were first supported in architecture v6 (for example, in the ARM1136). Semaphores are commonly used for allocating shared resources to applications. When a shared resource can only service one client or application processor, we also call it Mutual Exclusion (MUTEX). In such cases, when a resource is being used by one process, it is locked to that process and cannot serve another process until the lock is released. To set up a MUTEX semaphore, a memory location is defined as the lock flag to indicate whether a shared resource is locked by a process. When a process or application wants to use the resource, it needs to check whether the resource has been locked first. If it is not being used, it can set the lock flag to indicate that the resource is now locked. In traditional ARM processors, the access to the lock flag is carried out by the SWP instruction. It allows the lock flag read and write to be atomic, preventing the resource from being locked by two processes at the same time.

In newer ARM processors, the read/write access can be carried out on separated buses. In such situations, the SWP instructions can no longer be used to make the memory access atomic because the read and write in a locked transfer sequence must be on the same bus. Therefore, the locked transfers are replaced by exclusive accesses. The concept of exclusive access operation is quite simple but different from SWP; it allows the possibility that the memory location for a semaphore could be accessed by another bus master or another process running on the same processor.



To allow exclusive access to work properly in a multiple processor environment, an additional hardware called "exclusive access monitor" is required. This monitor checks the transfers toward shared address locations and replies to the processor if an exclusive access is success. The processor bus interface also provides additional control signals1 to this monitor to indicate if the transfer is an exclusive access.

If the memory device has been accessed by another bus master between the exclusive read and the exclusive write, the exclusive access monitor will flag an exclusive failed through the bus system when the processor attempts the exclusive write. This will cause the return status of the exclusive write to be 1. In the case of failed exclusive write, the exclusive access monitor also blocks the write transfer from getting to the exclusive access address.

Exclusive access instructions in the Cortex-M3 include LDREX (word), LDREXB (byte), LDREXH (half word), STREX (word), STREXB (byte), and STREXH (half word). A simple example of the syntax is as follows:

LDREX <Rxf>, [Rn, #offset]
STREX <Rd>, <Rxf>,[Rn, #offset]

Where Rd is the return status of the exclusive write (0 = success and 1 = failure). Example code for exclusive accesses can be found in Chapter 10. You can also access exclusive access instructions in C using intrinsic functions provided in Cortex Microcontroller Software Interface Standard (CMSIS) compliant device driver libraries from microcontroller vendors: __LDREX, __LEDEXH, __LDREXB, __STREX, __STREXH, __STREXB. More details of these functions are covered in Appendix G.

When exclusive accesses are used, the internal write buffers in the Cortex-M3 bus interface will be bypassed, even when the MPU defines the region as bufferable. This ensures that semaphore information on the physical memory is always up to date and coherent between bus masters. SoC designers using Cortex-M3 on multiprocessor systems should ensure that the memory system enforces data coherency when exclusive transfers occur.

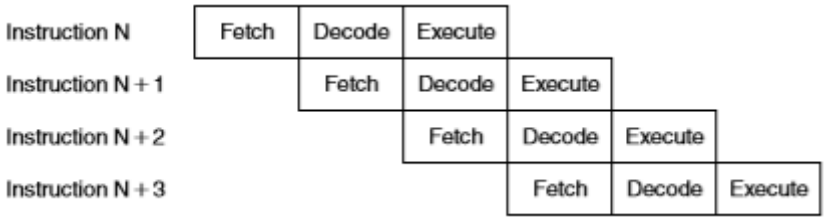
PIPELINE

The Cortex-M3 processor has a three-stage pipeline. The pipeline stages are instruction fetches, instruction decode, and instruction execution (see Figure 6.1). Some people might argue that there are four stages because of the pipeline behavior in the bus interface when it accesses

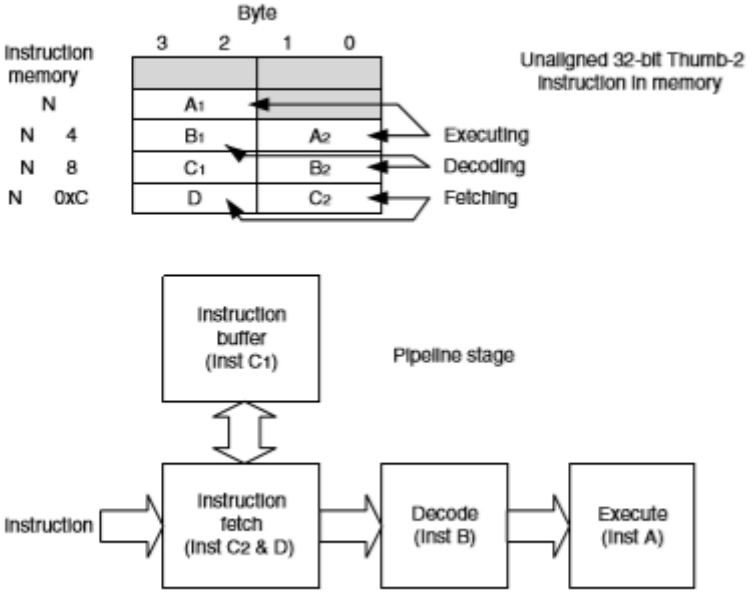
memory, but this stage is outside the processor, so the processor itself still has only three stages.

When running programs with mostly 16-bit instructions, you will find that the processor might not fetch instructions in every cycle. This is because the processor fetches up to two instructions (32-bit) in one go, so after one instruction is fetched, the next one is already inside the processor. In this case, the processor bus interface may try to fetch the instruction after the next or, if the buffer is full, the bus interface could be idle. Some of the instructions take multiple cycles to execute; in this case, the pipeline will be stalled.

In executing a branch instruction, the pipeline will be flushed. The processor will have to fetch instructions from the branch destination to fill up the pipeline again. However, the Cortex-M3 processor supports a number of instructions in v7-M architecture, so some of the short-distance branches can be avoided by replacing them with conditional execution codes.



The Three-Stage Pipeline in the Cortex-M3.



Use of a Buffer in the Instruction Fetch Unit to Improve 32-Bit Instruction Handling

Because of the pipeline nature of the processor and to ensure that the program is compatible with Thumb® codes, the read value will be the address of the instruction plus 4, when the program counter is read during instruction execution. If the program counter is used for address generation for memory accesses, the word aligned value of the instruction address plus 4 would be used. This offset is constant, independent of the combination of 16-bit Thumb instructions and 32-bit Thumb-2 instructions. This ensures consistency between Thumb and Thumb-2.

Inside the instruction prefetch unit of the processor core, there is also an instruction

buffer (see Figure 6.2). This buffer allows additional instructions to be queued before they are needed. This buffer prevents the pipeline being stalled when the instruction sequence contains 32-bit Thumb-2 instructions that are not word aligned. However, this buffer does not add an extra stage to the pipeline, so it does not increase the branch penalty.

BUS INTERFACES ON THE CORTEX-M3

Unless you are designing an SoC product using the Cortex-M3 processor, it is unlikely that you can directly access the bus interface signals described here. Normally, the chip manufacturer will hook up all the bus signals to memory blocks and peripherals, and in a few cases, you might find that the chip manufacturer connected the bus to a bus bridge and allows external bus systems to be connected offchip. The bus interfaces on the Cortex-M3 processor are based on AHB-Lite and APB protocols.

The I-Code Bus

The I-Code bus is a 32-bit bus based on the AHB-Lite bus protocol for instruction fetches in memory regions from 0x00000000 to 0x1FFFFFFF. Instruction fetches are performed in word size, even for 16-bit Thumb instructions. Therefore, during execution, the CPU core could fetch up to two Thumb instructions at a time.

The D-Code Bus

The D-Code bus is a 32-bit bus based on the AHB-Lite bus protocol; it is used for data access in memory regions from 0x00000000 to 0x1FFFFFFF. Although the Cortex-M3 processor supports unaligned transfers, you won't get any unaligned transfer on this bus, because the bus interface on the processor core converts the unaligned transfers into aligned transfers for you. Therefore, devices (such as memory) that attach to this bus need only support AHB-Lite (AMBA 2.0) aligned transfers.

The System Bus

The system bus is a 32-bit bus based on the AHB-Lite bus protocol; it is used for instruction fetch and data access in memory regions from 0x20000000 to 0xDFFFFFFF and 0xE0100000 to 0xFFFFFFFF. Similar to the D-Code bus, all the transfers on the system bus are aligned.

The External PPB

The External PPB is a 32-bit bus based on the APB bus protocol. This is intended for private peripheral accesses in memory regions 0xE0040000 to 0xE00FFFFF. However, since some part of this APB memory is already used for TPIU, ETM, and the ROM table, the memory region that can be used for attaching extra peripherals on this bus is only 0xE0042000 to 0xE00FF000. Transfers on this bus are word aligned.

The DAP Bus

The DAP bus interface is a 32-bit bus based on an enhanced version of the APB specification. This is for attaching debug interface blocks such as SWJ-DP or SW-DP. Do not use this bus for other purposes. More information on this interface can be found in Chapter 15, or in the ARM document CoreSight Technology System Design Guide.

UNIT – II
EXCEPTIONS AND INTERRUPTS

This chapter describes the interrupt and exception-handling mechanism when operating in protected mode on an Intel 64 or IA-32 processor. Most of the information provided here also applies to interrupt and exception mechanisms used in real-address, virtual-8086 mode, and 64-bit mode.

OVERVIEW OF INTERRUPT AND EXCEPTION:

Interrupts and exceptions are events that indicate that a condition exists somewhere in the system, the processor, or within the currently executing program or task that requires the attention of a processor. They typically result in a forced transfer of execution from the currently running program or task to a special software routine or task called an interrupt handler or an exception handler. The action taken by a processor in response to an interrupt or exception is referred to as servicing or handling the interrupt or exception.

Interrupts occur at random times during the execution of a program, in response to signals from hardware. System hardware uses interrupts to handle events external to the processor, such as requests to service peripheral devices. Software can also generate interrupts by executing the INT n instruction.

Exceptions occur when the processor detects an error condition while executing an instruction, such as division by zero. The processor detects a variety of error conditions including protection violations, page faults, and internal machine faults. The machine-checks architecture of the Pentium 4, Intel Xeon, P6 family, and Pentium processors also permits a machine- check exception to be generated when internal hardware errors and bus errors are detected.

When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler. When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task. The resumption of the interrupted procedure or task happens without loss of program continuity, unless recovery from an exception was not possible or an interrupt caused the currently running program to be terminated.

This chapter describes the processor's interrupt and exception- handling mechanism, when operating in protected mode. A description of the exceptions and the conditions that cause them to be generated is given at the end of this chapter.

EXCEPTION AND INTERRUPTVECTORS

To aid in handling exceptions and interrupts, each architecturally defined exception and each interrupt condition requiring special handling by the processor is assigned a unique identification number, called a vector number. The processor uses the vector number assigned to an exception or interrupt as an index into the interrupt descriptor table (IDT). The table provides the entry point to an exception or interrupt handler ("Interrupt Descriptor Table (IDT)").

The allowable range for vector numbers is 0 to 255. Vector numbers in the range 0 through 31 are reserved by the Intel 64 and IA-32 architectures for architecture- defined exceptions and interrupts. Not all of the vector numbers in this range have a currently defined function. The unassigned vector numbers in this range are reserved. Do not use the reserved vectornumbers.

Vector numbers in the range 32 to 255 are designated as user- defined interrupts and are not reserved by the Intel 64 and IA- 32 architecture. These interrupts are generally assigned to

external I / O devices to enable those devices to send interrupts to the processor through one of the external hardware interrupt mechanisms (“Sources of Iinterrupts”).

Table shows vector number assignments for architecturally defined exceptions and for the NMI interrupt. This table gives the exception type (“Exception Classifications”) and indicates whether an error code is saved on the stack for the exception. The source of each predefined exception and the NMI interrupt is also given.

SOURCES OF INTERRUPTS

The processor receives interrupts from two sources:

- External (hardware generated) interrupts.
- Software-generated interrupts.

External Interrupts

External interrupts are received through pins on the processor or through the local API C. The primary interrupt pins on Pentium 4, I ntel Xeon, P6 family, and Pentium processors are the LINT [1: 0] pins, which are connected to the local API C (see Chapter 10, “Advanced Programmable I nterrupt Controller (API C)”). When the local API C is enabled, the LI NT[1: 0] pins can be programmed through the APIC’s local vector table (LVT) to be associated with any of the processor’s exception or interrupt vectors.

Vector No.	Mne-monic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	RESERVED	Fault/ Trap	No	For Intel use only.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[2]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	

16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[3]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[4]
19	#XM	SIMD Floating-Point Exception	Fault	No	SSE/SSE2/SSE3 floating-point instructions[5]
20	#VE	Virtualization Exception	Fault	No	EPT violations[6]
21-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT*n* instruction.

Table -Protected-Mode Exceptions and Interrupts

When the local APIC is global/ hardware disabled, these pins are configured as I NTR and NMI pins, respectively. Asserting the I NTR pin signals the processor that an external interrupt has occurred. The processor reads from the system bus the interrupt vector number provided by an external interrupt controller, such as an 8259A ("Exception and Interrupt Vectors"). Asserting the NMI pin signals a non- maskable interrupt (NMI), which is assigned to interrupt vector 2.

The processor's local APIC is normally connected to a system-based I/O APIC. Here, external interrupts received at the I/O APIC's pins can be directed to the local APIC through the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel® Atom™, and I ntel Xeon processors) or the API C serial bus (P6 family and Pentium processors). The I / O APIC determines the vector number of the interrupt and sends this number to the local APIC. When a system contains multiple processors, processors can also send interrupts to one another by means of the system bus (Pentium 4, Intel Core Duo, Intel Core 2, I ntel Atom, and Intel Xeon processors) or the APIC serial bus (P6 family and Pentium processors).

The LI NT[1: 0] pins are not available on the I ntel486 processor and earlier Pentium processors that do not contain an on- chip local API C. These processors have dedicated NMI and I NTR pins. With these processors, external inter- rupts are typically generated by a system- based interrupt controller (8259A), with the interrupts being signaled through the INTR pin.

Note that several other pins on the processor can cause a processor interrupt to occur. However, these interrupts are not handled by the interrupt and exception mechanism described in this chapter. These pins include the RESET#, FLUSH#, STPCLK#, SMI#, R/ S#, and INIT# pins. Whether they are included on a particular processor is implementation dependent. Pin functions are described in the data books for the individual processors. The SMI# pin is described in Chapter 34, "System Management Mode."

Maskable Hardware Interrupts

Any external interrupt that is delivered to the processor by means of the I NTR pin or through the local API C is called a maskable hardware interrupt. Maskable hardware interrupts that can be delivered through the INTR pin include all IA- 32 architecture defined interrupt vectors from 0 through 255; those that can be delivered through the local API C include interrupt vectors 16 through 255.The IF flag in the EFLAGS register permits all maskable hardware interrupts to be masked as a group ("Masking Maskable Hardware Interrupts"). Note that when interrupts 0 through 15 are delivered through the local API C, the API C indicates the receipt of an illegal vector.

Software-Generated Interrupts

The INT n instruction a permit interrupts to be generated from within software by supplying an interrupt vector number as an operand.

For example, the INT 35 instruction forces an implicit call to the interrupt handler for interrupt 35.

Any of the interrupt vectors from 0 to 255 can be used as a parameter in this instruction. I f the processor's predefined NMI vector is used, however, the response of the processor will not be the same as it would be from an NMI interrupt generated in the normal manner. I f vector number 2 (the NMI vector) is used in this instruction, the NMI interrupt handler is called, but the processor's NMI- handling hardware is not activated.

Interrupts generated in software with the I NT n instruction cannot be masked by the IF flag in the EFLAGS register.

SOURCES OF EXCEPTIONS

The processor receives exceptions from three sources:

- Processor-detected program-error exceptions.
- Software-generated exceptions.
- Machine-check exceptions.

Program-Error Exceptions

The processor generates one or more exceptions when it detects program errors during the execution in an application program or the operating system or executive. Intel 64 and IA-32 architectures define a vector number for each processor- detectable exception. Exceptions are classified as **faults**, **traps**, and **aborts** (see Section 6.5, "Exception Classifications").

Software-Generated Exceptions

The INTO, INT 3, and BOUND instructions permit exceptions to be generated in software. These instructions allow checks for exception conditions to be performed at points in the instruction stream. For example, INT 3 causes a breakpoint exception to be generated.

The INT n instruction can be used to emulate exceptions in software; but there is a limitation. If INT n provides a vector for one of the architecturally- defined exceptions, the processor generates an interrupt to the correct vector (to access the exception handler) but does not push an error code on the stack. This is true even if the associated hardware- generated exception normally produces an error code. The exception handler will still attempt to pop an

error code from the stack while handling the exception. Because no error code was pushed, the handler will pop off and discard the EI P instead (in place of the missing error code). This sends the return to the wrong location.

Machine-Check Exceptions

The P6 family and Pentium processors provide both internal and external machine-check mechanisms for checking the operation of the internal chip hardware and bus transactions. These mechanisms are implementation depen- dent. When a machine- check error is detected, the processor signals a machine-check exception (vector 18) and returns an error code.("Interrupt 18—Machine- Check Exception (#MC)" and Chapter 15, "Machine-Check Architecture," for more information about the machine-check mechanism.

EXCEPTION CLASSIFICATIONS

E

xceptions are classified as **faults**, **traps**, or **aborts** depending on the way they are reported and whether the instruction that caused the exception can be restarted without loss of program or task continuity.

- **Faults**— A fault is an exception that can generally be corrected and that, once corrected, allows the program to be restarted with no loss of continuity. When a fault is reported, the processor restores the machine state to the state prior to the beginning of execution of the faulting instruction. The return address (saved contents of the CS and EIP registers) for the fault handler points to the faulting instruction, rather than to the instruction following the faulting instruction.
- **Traps** — A trap is an exception that is reported immediately following the execution of the trapping instruction. Traps allow execution of a program or task to be continued without loss of program continuity. The return address for the trap handler points to the instruction to be executed after the trapping instruction.
- **Aborts**— An abort is an exception that does not always report the precise location of the instruction causing the exception and does not allow a restart of the program or task that caused the exception. Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.

NOTE:

One exception subset normally reported as a fault is not restartable. Such exceptions result in loss of some processor state. For example, executing a POPAD instruction where the stack frame crosses over the end of the stack segment causes a fault to be reported. I n this situation, the exception handler sees that the instruction pointer (CS: EI P) has been restored as if the POPAD instruction had not been executed. However, internal processor state (the general-purpose registers) will have been modified. Such cases are considered programming errors. An application causing this class of exceptions should be terminated by the operating system.

When a page- fault exception occurs, the exception handler can load the page into memory and resume execution of the program or task by restarting the faulting instruction. To insure that the restart is handled trans- parently to the currently executing program or task, the processor saves the necessary registers and stack pointers to allow a restart to the state prior to the execution of the faulting instruction.

For trap-class exceptions, the return instruction pointer points to the instruction following the trapping instruction. I f a trap is detected during an instruction which transfers execution, the return instruction pointer reflects the transfer. For example, if a trap is detected while executing a JMP instruction, the return instruction pointer points to the destination of the JMP instruction, not to the next address past the JMP instruction. All trap exceptions allow program or task restart with no loss of continuity. For example, the overflow exception is a trap exception. Here, the return instruction pointer points to the instruction following the INTO instruction that tested EFLAGS OF (over- flow) flag. The trap handler for this exception resolves the overflow condition. Upon return from the trap handler, program or task execution continues at the instruction following the INTO instruction.

The abort-class exceptions do not support reliable restarting of the program or task. Abort handlers are designed to collect diagnostic information about the state of the processor when the abort exception occurred and then shut down the application and system as gracefully as possible.

Interrupts rigorously support restarting of interrupted programs and tasks without loss of continuity. The return instruction pointer saved for an interrupt points to the next instruction to be executed at the instruction boundary where the processor took the interrupt. I f the instruction just executed has a repeat prefix, the interrupt is taken at the end of the current iteration with the registers set to execute the next iteration.

The ability of a P6 family processor to speculatively execute instructions does not affect the taking of interrupts by the processor. interrupts are taken at instruction boundaries located during the retirement phase of instruction execution; so they are always taken in the "in-order" instruction stream. See Chapter 2, "Intel® 64 and IA-32 Architectures," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1, for more information about the P6 family processors' micro architecture and its support for out-of-order instruction execution.

Note that the Pentium processor and earlier I A-32 processors also perform varying amounts of perfecting and preliminary decoding. With these processors as well, exceptions and interrupts are not signaled until actual "in-order" execution of the instructions. For a given code sample, the signaling of exceptions occurs uniformly when the code is executed on any family of IA-32 processors (except where new exceptions or new opcodes have been defined).

NONMASKABLE INTERRUPT (NMI)

The nonmaskable interrupt (NMI) can be generated in either of two ways:

- External hardware asserts the NMI pin.
- The processor receives a message on the system bus (Pentium 4, Intel Core Duo, Intel Core 2, Intel Atom, and Intel Xeon processors) or the API C serial bus (P6 family and Pentium processors) with a delivery mode NMI

When the processor receives a NMI from either of these sources, the processor handles it immediately by calling the NMI handler pointed to by interrupt vector number 2. The processor also invokes certain hardware conditions to insure that no other interrupts, including NMI interrupts, are received until the NMI handler has completed executing (see Section 6.7.1, "Handling Multiple NMIs").

Also, when an NMI is received from either of the above sources, it cannot be masked by the IF flag in the EFLAGS register.

It is possible to issue a maskable hardware interrupt (through the INTR pin) to vector 2 to invoke the NMI interrupt handler; however, this interrupt will not truly be an NMI interrupt. A true NMI interrupt that activates the processor's NMI - handling hardware can only be delivered through one of the mechanisms listed above.

Handling Multiple NMIs

While an NMI interrupt handler is executing, the processor blocks delivery of subsequent NMI s until the next execution of the I RET instruction. This blocking of NMI s prevents nested execution of the NMI handler. I t is recommended that the NMI interrupt handler be accessed through an interrupt gate to disable maskable hardware interrupts ("Masking Maskable Hardware Interrupts")

An execution of the I RET instruction unblocks NMIs even if the instruction causes a fault. For example, if the I RET instruction executes with EFLAGS.VM = 1 and IOPL of less than 3, a general-protection exception is generated ("Sensitive Instructions"). In such a case, NMIs are unmasked before the exception handler is invoked.

ENABLING AND DISABLINGINTERRUPTS

The processor inhibits the generation of some interrupts, depending on the state of the processor and of the I F and RF flags in the EFLAGS register, as described in the following sections.

Masking Maskable Hardware Interrupts

The IF flag can disable the servicing of maskable hardware interrupts received on the processor's INTR pin or through the local APIC (see Section 6.3.2, "Maskable Hardware Interrupts"). When the IF flag is clear, the processor inhibits interrupts delivered to the I NTR pin or through the local API C from generating an internal inter- rupt request; when the IF flag is set, interrupts delivered to the I NTR or through the local API C pin are processed as normal external interrupts.

The I F flag does not affect non- maskable interrupts (NMI s) delivered to the NMI pin or delivery mode NMI messages delivered through the local API C, nor does it affect processor generated exceptions. As with the other flags in the EFLAGS register, the processor clears the I F flag in response to hardware reset.

The fact that the group of maskable hardware interrupts includes the reserved interrupt and exception vectors 0 through 32 can potentially cause confusion. Architecturally, when the IF flag is set, an interrupt for any of the vectors from 0 through 32 can be delivered to the processor through the I NTR pin and any of the vectors from 16 through 32 can be delivered through the local API C. The processor will then generate an interrupt and call the interrupt or exception handler pointed to by the vector number. So for example, it is possible to invoke the page- fault handler through the I NTR pin (by means of vector 14); however, this is not a true page- fault exception. I t is an interrupt. As with the I NT n instruction (see Section 6.4.2, "Software- Generated Exceptions"), when an inter- rupt is generated through the I NTR pin to an exception vector, the processor does not push an error code on the stack, so the exception handler may not operate correctly.

The IF flag can be set or cleared with the STI (set interrupt- enable flag) and CLI (clear interrupt- enable flag) instructions, respectively. These instructions may be executed only if the CPL is equal to or less than the I OPL. A general- protection exception (#GP) is generated if they are executed when the CPL is greater than the IOPL. (The effect of the IOPL on these instructions is modified slightly when the virtual mode extension is enabled by setting the VME flag in control register CR4: The IF flag is also affected by the following operations:

- The PUSHF instruction stores all flags on the stack, where they can be examined and modified. The POPF instruction can be used to load the modified flags back into the EFLAGS register.
- Task switches and the POPF and IRET instructions load the EFLAGS register; therefore, they can be used to modify the setting of the IF flag.
- When an interrupt is handled through an interrupt gate, the I F flag is automatically cleared, which disables maskable hardware interrupts. (I f an interrupt is handled through a trap gate, the IF flag is not cleared.)

Masking Instruction Breakpoints

The RF (resume) flag in the EFLAGS register controls the response of the processor to instruction- breakpoint conditions (see the description of the RF flag in Section 2.3, "System Flags and Fields in the EFLAGS Register").

When set, it prevents an instruction breakpoint from generating a debug exception (#DB); when clear, instruction breakpoints will generate debug exceptions. The primary function of the RF flag is to prevent the processor from going into a debug exception loop on an instruction- breakpoint. See Section 17.3.1.1, "Instruction-Breakpoint Exception Condition," for more information on the use of this flag.

Masking Exceptions and Interrupts When Switching Stacks

To switch to a different stack segment, software often uses a pair of instructions, for example:

```
MOV SS, AX
MOV ESP, Stack Top
```

If can interrupt or exception occurs after the segment selector has been loaded into the SS register but before the ESP register has been loaded, these two parts of the logical address into the stack space are inconsistent for the duration of the interrupt or exception handler.

To prevent this situation, the processor inhibits interrupts, debug exceptions, and single- step trap exceptions after either a MOV to SS instruction or a POP to SS instruction, until the instruction boundary following the next instruction is reached. All other faults may still be generated. I f the LSS instruction is used to modify the contents of the SS register (which is the recommended method of modifying this register), this problem does not occur.

PRIORITY AMONG SIMULTANEOUS EXCEPTIONS AND INTERRUPTS

I f more than one exception or interrupt is pending at an instruction boundary; the processor services them in a predictable order. Table shows the priority among classes of exception and interrupt sources.

Table-Priority among Simultaneous Exceptions and Interrupts

Priority	Description
1 (Highest)	Hardware Reset and Machine Checks - RESET - Machine Check
2	Trap on Task Switch - T flag in TSS is set
3	External Hardware Interventions - FLUSH - STOPCLK - SMI - INIT
4	Traps on the Previous Instruction - Breakpoints - Debug Trap Exceptions (TF flag set or data/I-O breakpoint)
5	Nonmaskable Interrupts (NMI) 1
6	Maskable Hardware Interrupts 1
7	Code Breakpoint Fault
8	Faults from Fetching Next Instruction - Code-Segment Limit Violation - Code Page Fault
9	Faults from Decoding the Next Instruction - Instruction length > 15 bytes - Invalid Opcode - Coprocessor Not Available

10 (Lowest)	Faults on Executing an Instruction - Overflow - Bound error - Invalid TSS - Segment Not Present - Stack fault - General Protection - Data Page Fault - Alignment Check - x87 FPU Floating-point exception - SIMD floating-point exception - Virtualization exception

NOTE

1. The Intel® 486 processor and earlier processors group nonmaskable and maskable interrupts in the same priority class.

While priority among these classes listed in Table 6- 2 is consistent throughout the architecture, exceptions within each class are implementation- dependent and may vary from processor to processor. The processor first services pending exception or interrupt from the class which has the highest priority, transferring execution to the first instruction of the handler. Lower priority exceptions are discarded; lower priority interrupts are held pending.

Discarded exceptions are re- generated when the interrupt handler returns execution to the point in the program or task where the exceptions and/ or interrupts occurred.

INTERRUPT DESCRIPTOR TABLE (IDT)

The interrupt descriptor table (IDT) associates each exception or interrupt vector with a gate descriptor for the procedure or task used to service the associated exception or interrupt. Like the GDT and LDTs, the IDT is an array of 8- byte descriptors (in protected mode). Unlike the GDT, the first entry of the I DT may contain a descriptor. To form an index into the I DT, the processor scales the exception or interrupt vector by eight (the number of bytes in a gate descriptor). Because there are only 256 interrupt or exception vectors, the IDT need not contain more than 256 descriptors. I t can contain fewer than 256 descriptors, because descriptors are required only for the interrupt and exception vectors that may occur. All empty descriptor slots in the I DT should have the present flag for the descriptor set to 0.

The base addresses of the IDT should be aligned on an 8- byte boundary to maximize performance of cache line fills. The limit value is expressed in bytes and is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly 1 valid byte. Because IDT entries are always eight bytes long, the limit should always be one less than an integral multiple of eight (that is, 8N – 1).

The IDT may reside anywhere in the linear address space. As shown in Figure 6- 1, the processor locates the IDT using the IDTR register. This register holds both a 32- bit base address and 16-bit limit for the IDT.

The LIDT (load IDT register) and SI DT (store I DT register) instructions load and store the contents of the IDTR register, respectively. The LIDT instruction loads the IDTR register with the base address and limit held in a memory operand. This instruction can be executed only when the CPL is 0. I t normally is used by the initialization code of an operating system when creating an IDT. An operating system also may use it to change from one I DT to another. The SI DT instruction copies the base and limit value stored in I DTR to memory. This instruction can be executed at any privilege level.

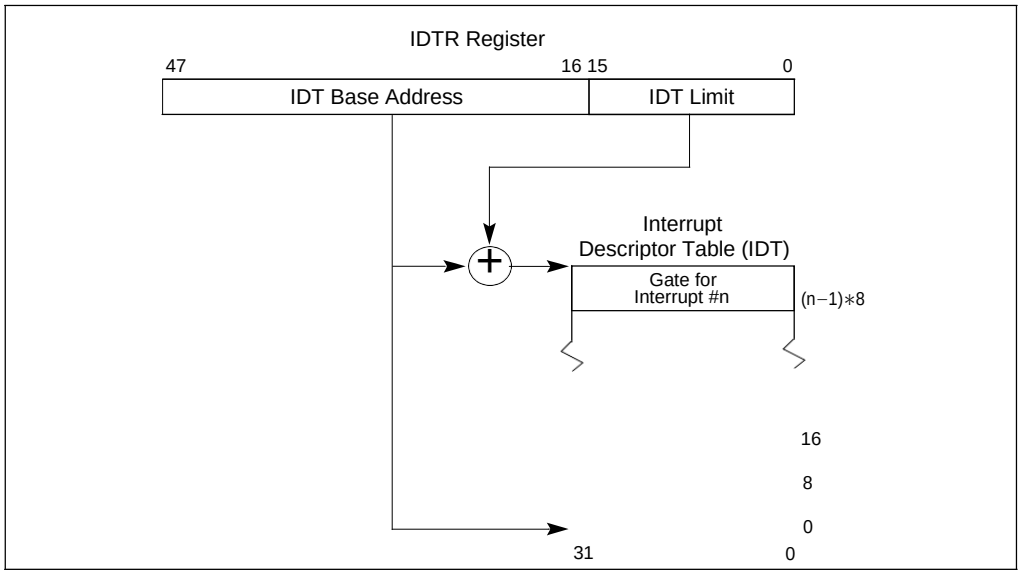
If a vector references a descriptor beyond the limit of the IDT, a general- protection exception (#GP) is generated.

NOTE

Because interrupts are delivered to the processor core only once, an incorrectly configured IDT could result in incomplete interrupt handling and/ or the blocking of interrupt delivery.

IA- 32 architecture rules need to be followed for setting up IDTR base/ limit/ access fields and each field in the gate descriptors. The same apply for the Intel 64 architecture. This includes implicit referencing of the destination code segment through the GDT or LDT and accessing the stack.

Relationship of the IDTR and IDT



IDT DESCRIPTORS

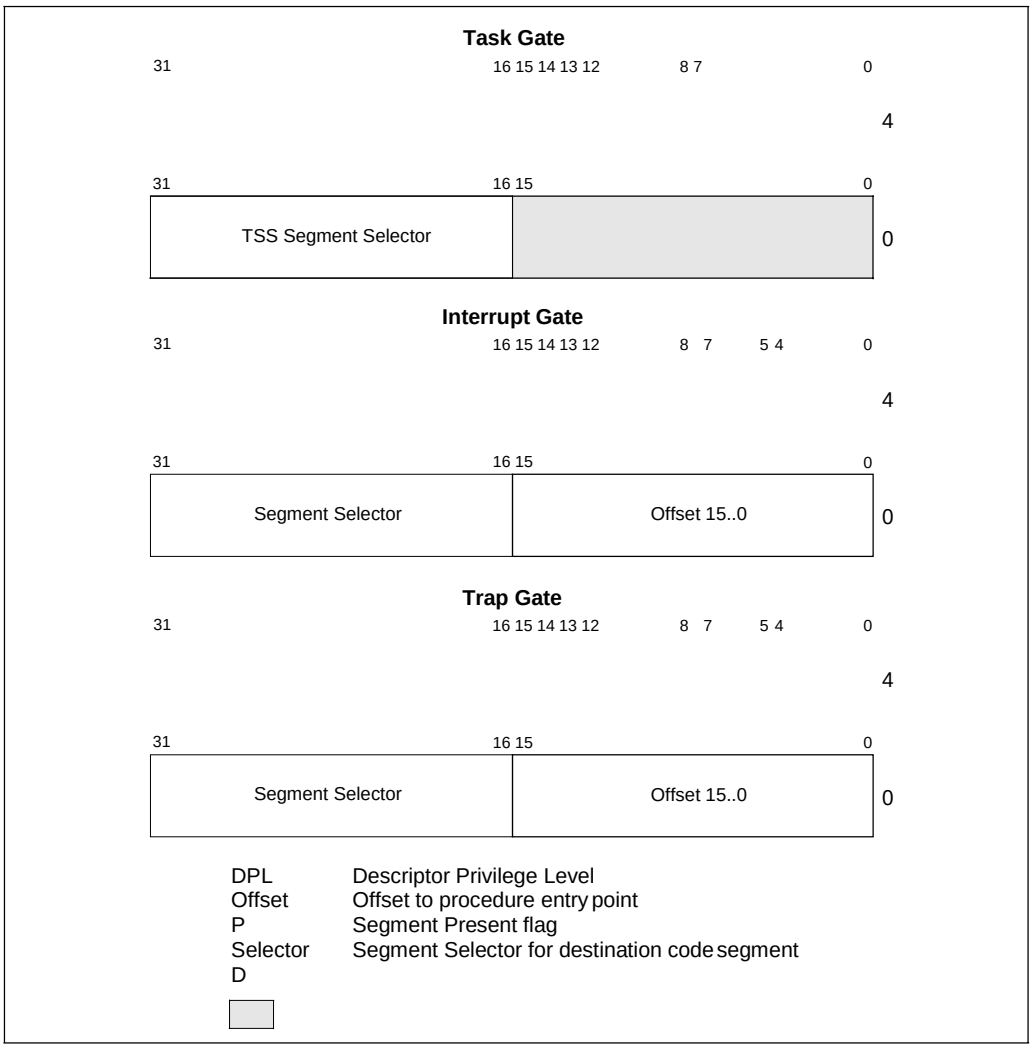
The IDT may contain any of three kinds of gate descriptors:

- Task- gate descriptor
- Interrupt- gate descriptor
- Trap- gate descripto

Figure shows the formats for the task- gate, interrupt- gate, and trap- gate descriptors. The format of a task gate used in an I DT is the same as that of a task gate used in the GDT or an LDT ("Task- Gate Descriptor"). The task gate contains the segment selector for a TSS for an exception and/ or interrupt handler task.

Interrupt and trap gates are very similar to call gates. They contain a far pointer (segment selector and offset) that the processor uses to transfer program execution to a handler procedure in an exception- or interrupt- handler code segment. These gates differ in the way the processor handles the I F flag in the EFLAGS register.

IDT Gate Descriptors



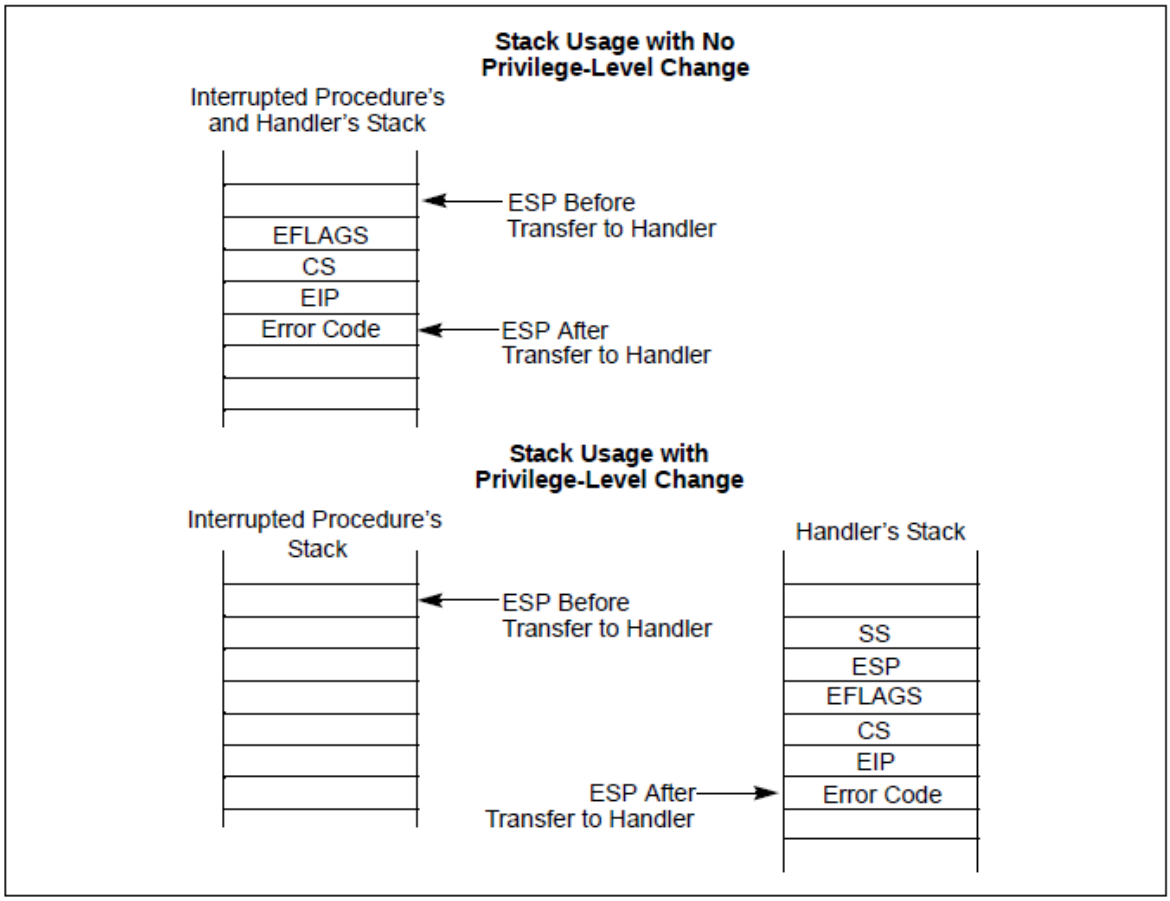
EXCEPTION AND INTERRUPT HANDLING

The processor handles calls to exception- and interrupt- handlers similar to the way it handles calls with a CALL instruction to a procedure or a task. When responding to an exception or interrupt, the processor uses the exception or interrupt vector as an index to a descriptor in the I DT. I f the index points to an interrupt gate or trap gate, the processor calls the exception or interrupt handler in a manner similar to a CALL to a call gate. If index points to a task gate, the processor executes a task switch to the exception- or interrupt- handler task in a manner similar to a CALL to a task gate.

Exception- or Interrupt-Handler Procedures

An interrupt gate or trap gate references an exception- or interrupt- handler procedure that runs in the context of the currently executing task (see Figure 6- 3). The segment selector for the gate points to a segment descriptor for an executable code segment in either the GDT or the current LDT. The offset field of the gate descriptor points to the beginning of the exception- or interrupt- handling procedure.
When the processor performs a call to the exception- or interrupt- handler procedure:

- I f the handler procedure is going to be executed at a numerically lower privilege level, a stack switch occurs. When the stack switch occurs:
- The segment selector and stack pointer for the stack to be used by the handler are obtained from the TSS for the currently executing task. On this new stack, the processor pushes the stack segment selector and stack pointer of the interrupted procedure.
- The processor then saves the current state of the EFLAGS, CS, and EI P registers on the new stack (see Figures 6- 4).
- I f an exception causes an error code to be saved; it is pushed on the new stack after the EI P value.
- I f the handler procedure is going to be executed at the same privilege level as the interrupted procedure:
- The processor saves the current state of the EFLAGS, CS, and EIP registers on the current stack (see Figures 6- 4).
- I f an exception causes an error code to be saved, it is pushed on the current stack after the EIP value.


Stack Usage with No
Privilege-Level Change
Interrupted Procedure's
and Handler's Stack
ESP Before
Transfer to Handler
EFLAGS
CS
EIP
Error Code
ESP After
Transfer to Handler
Stack Usage with
Privilege-Level Change
Interrupted Procedure's
Stack
Handler's Stack
ESP Before
Transfer to Handler
SS
ESP
EFLAGS
CS
EIP
ESP After
Transfer to Handler
Error Code

Interrupt Procedure Call

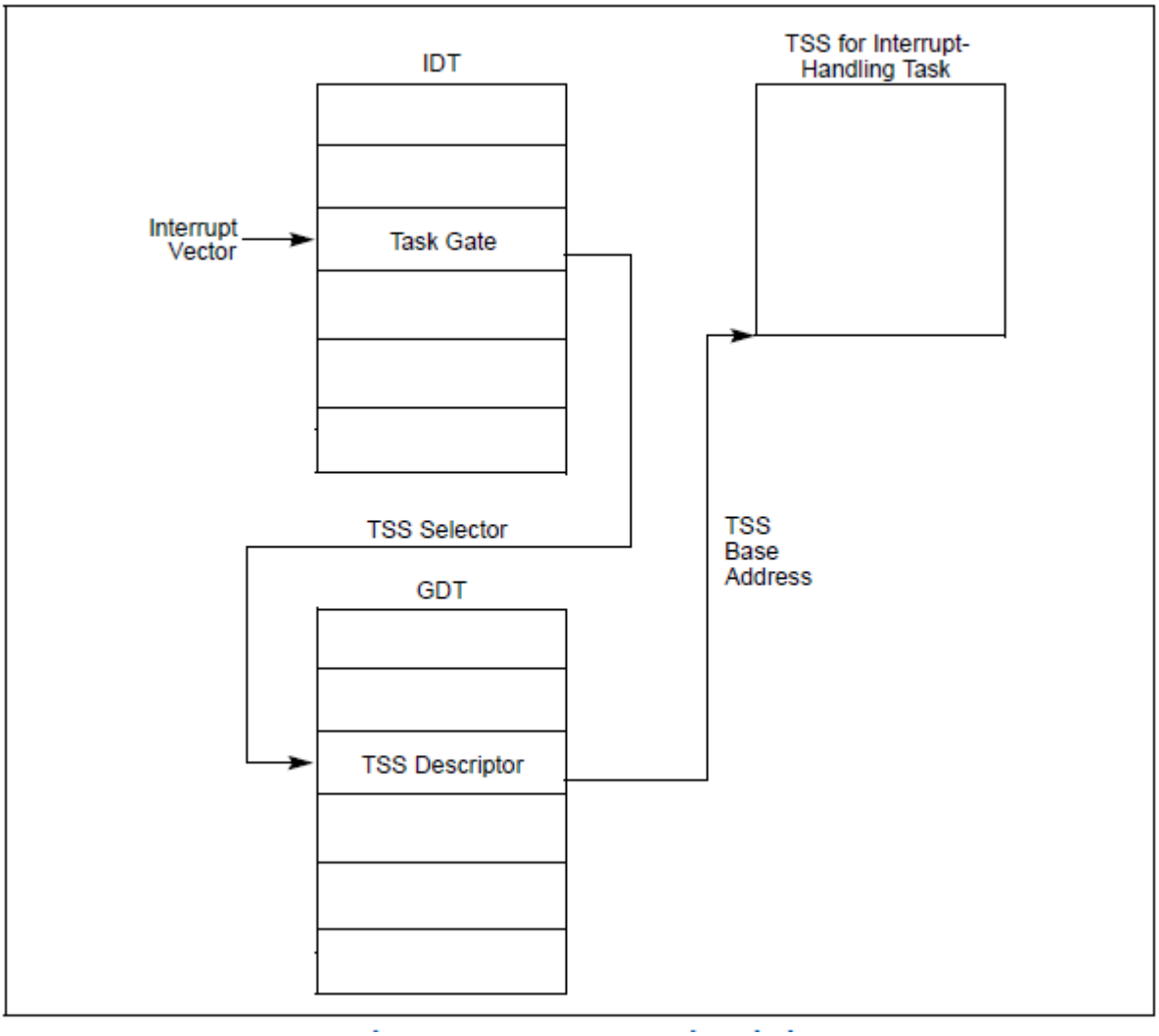
To return from an exception- or interrupt- handler procedure, the handler must use the I RET (or I RETD) instruction. The IRET instruction is similar to the RET instruction except that it restores the saved flags into the EFLAGS register. The IOPL field of the EFLAGS register is restored only if the CPL is 0. The I F flag is changed only if the CPL is less than or equal to the I OPL. If a stack switch occurred when calling the handler procedure, the I RET instruction switches back to the interrupted procedure's stack on the return.

Protection of Exception- and Interrupt-Handler Procedures

The privilege- level protection for exception- and interrupt- handler procedures is similar to that used for ordinary procedure calls when called through a call gate. The processor does not permit transfer of execution to an exception- or interrupt- handler procedure in a less privileged code segment (numerically greater privilege level) than the CPL.

An attempt to violate this rule results in a general- protection exception (#GP). The protection mechanism for exception- and interrupt- handler procedures is different in the following ways:

- Because interrupt and exception vectors have no RPL, the RPL is not checked on implicit calls to exception and interrupt handlers.
- The processor checks the DPL of the interrupt or trap gate only if an exception or interrupt is generated with an INT n, I NT 3, or INTO instruction. Here, the CPL must be less than or equal to the DPL of the gate. This restriction prevents application programs or procedures running at privilege level 3 from using software interrupt to access critical exception handlers, such as the page- fault handler, providing that those handlers are placed in more privileged code segments (numerically lower privilege level). For hardware- generated interrupts and processor- detected exceptions, the processor ignores the DPL of interrupt and trap gates.



Interrupt Task Switch

ERROR CODE

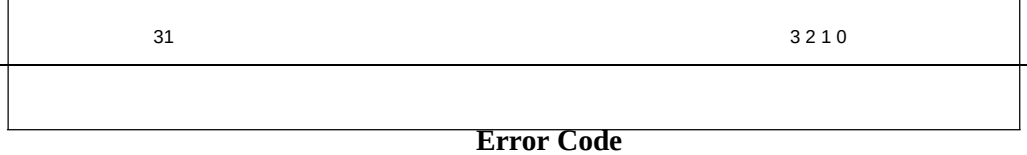
When an exception condition is related to a specific segment selector or IDT vector, the processor pushes an error code onto the stack of the exception handler (whether it is a procedure or task). The error code has the format shown in Figure. The error code resembles a segment selector; however, instead of a TI flag and RPL field, the error code contains 3 flags:

EXT External event (bit 0) — When set, indicates that the exception occurred during delivery of an event external to the program, such as an interrupt or an earlier exception.

I DT Descriptor location (bit 1) — When set, indicates that the index portion of the error code refers to a gate descriptor in the I DT; when clear, indicates that the index refers to a descriptor in the GDT or the current LDT.

TI GDT/ LDT (bit 2) — only used when the IDT flag is clear. When set, the TI flag indicates that the index portion of the error code refers to a segment or gate descriptor in the LDT; when clear, it indicates that the index refers to a descriptor in the current GDT

Reserved	Segment Selector Index	TI	IDT	EXT



Error Code

The segment selector index field provides an index into the IDT, GDT, or current LDT to the segment or gate selector being referenced by the error code. In some cases the error code is null (all bits are clear except possibly EXT). A null error code indicates that the error was not caused by a reference to a specific segment or that a null segment descriptor was referenced in an operation.

The format of the error code is different for page- fault exceptions (#PF). See the "Interrupt 14—Page- Fault Exception (#PF)" section in this chapter.

The error code is pushed on the stack as a double word or word (depending on the default interrupt, trap, or task gate size). To keep the stack aligned for double word pushes, the upper half of the error code is reserved. Note that the error code is not popped when the IRET instruction is executed to return from an exception handler, so the handler must remove the error code before executing a return.
Error codes are not pushed on the stack for exceptions that are generated externally (with the INTR or LINT [1: 0] pins) or the I NT n instruction, even if an error code is normally produced for those exceptions.

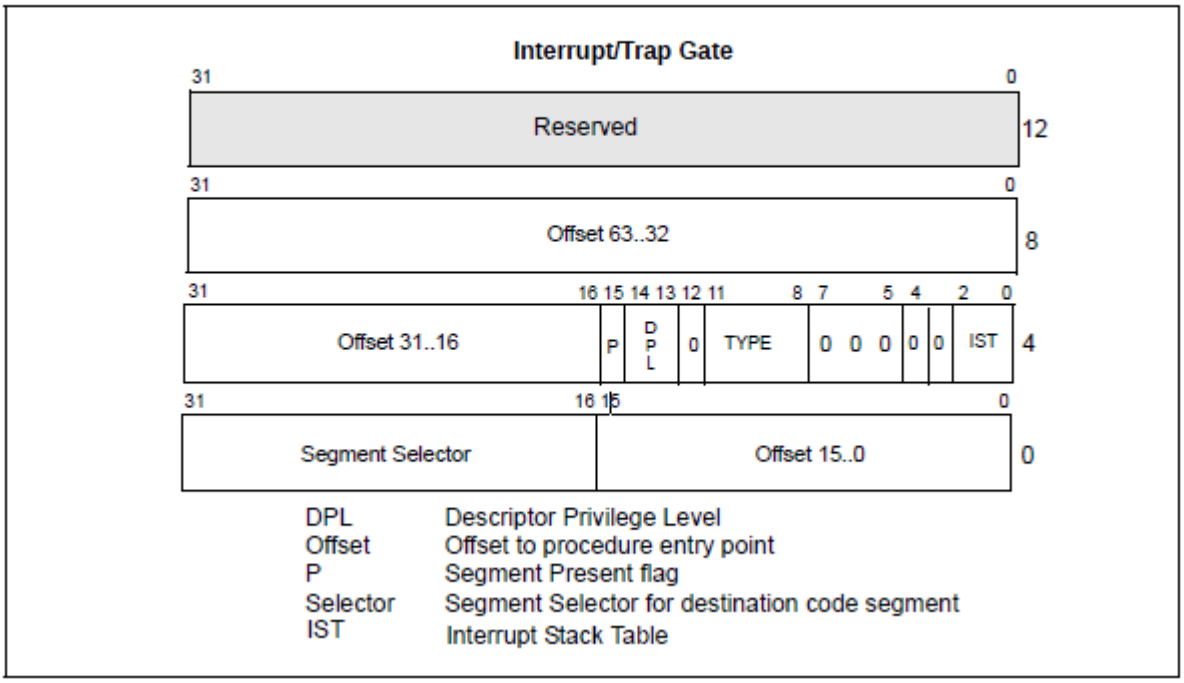
EXCEPTION AND INTERRUPT HANDLING IN 64-BIT MODE

In 64- bit mode, interrupt and exception handling is similar to what has been described for non-64- bit modes. The following are the exceptions:

- All interrupt handlers pointed by the IDT are in 64- bit code (this does not apply to the SMI handler).
- The size of interrupt- stack pushes is fixed at 64 bits; and the processor uses 8- byte, zero extended stores.
- The stack pointer (SS: RSP) is pushed unconditionally on interrupts. I n legacy modes, this push is conditional and based on a change in current privilege level (CPL).
- The new SS is set to NULL if there is a change in CPL.
- IRET behavior changes.
- There is a new interrupt stack- switch mechanism.
- The alignment of interrupt stack frame is different.

64-Bit Mode IDT

Interrupt and trap gates are 16 bytes in length to provide a 64- bit offset for the instruction pointer (RIP). The 64- bit RIP referenced by interrupt- gate descriptors allows an interrupt service routine to be located anywhere in the linear-address space. See Figure.



64-Bit IDT Gate Descriptors

In 64- bit mode, the IDT index is formed by scaling the interrupt vector by 16. The first eight bytes (bytes 7: 0) of a 64- bit mode interrupt gate are similar but not identical to legacy 32- bit interrupt gates. The type field (bits 11: 8 in bytes 7: 4) is described in Table 3- 2. The Interrupt Stack Table (IST) field (bits 4: 0 in bytes 7: 4) is used by the stack switching mechanisms described in Section 6.14.5, “Interrupt Stack Table.” Bytes 11: 8 hold the upper 32 bits of the target RIP (interrupt segment offset) in canonical form. A general- protection exception (#GP) is generated if software attempts to reference an interrupt gate with a target RI P that is not in canonical form.

The target code segment referenced by the interrupt gate must be a 64- bit code segment (CS.L = 1, CS.D = 0). I f the target is not a 64-bit code segment; a general- protection exception (#GP) is generated with the I DT vector number reported as the error code.

Only 64- bit interrupt and trap gates can be referenced in IA- 32e mode (64- bit mode and compatibility mode). Legacy 32- bit interrupt or trap gate types (0EH or 0FH) are redefined in IA- 32e mode as 64- bit interrupt and trap gate types. No 32- bit interrupt or trap gate type exists in IA- 32e mode. I f a reference is made to a 16-bit interrupt or trap gate (06H or 07H), a general- protection exception (#GP (0)) is generated.

64-Bit Mode Stack Frame

In legacy mode, the size of an IDT entry (16 bits or 32 bits) determines the size of interrupt-stack- frame pushes. SS: ESP is pushed only on a CPL change. I n 64- bit mode, the size of interrupt stack- frame pushes is fixed at eight bytes. This is because only 64- bit mode gates can be referenced. 64- Bit mode also pushes SS: RSP unconditionally, rather than only on a CPL change.

Aside from error codes pushing SS: RSP unconditionally presents operating systems with a consistent interrupt- stack frame size across all interrupts. Interrupt service- routine entry points that handle interrupts generated by the INTn instruction or external I NTR# signal can push an additional error code place- holder to maintain consistency.

In legacy mode, the stack pointer may be at any alignment when an interrupt or exception causes a stack frame to be pushed. This causes the stack frame and succeeding pushes done by an interrupt handler to be at arbitrary alignments. In IA- 32e mode, the RSP is aligned to a 16- byte boundary before pushing the stack frame. The stack frame itself is aligned on a 16- byte boundary when the interrupt handler is called. The processor can arbitrarily realign the new RSP on interrupts because the previous (possibly unaligned) RSP is unconditionally saved on the newly aligned stack. The previous RSP will be automatically restored by a subsequent I RET.

Aligning the stack permits exception and interrupt frames to be aligned on a 16- byte boundary before interrupts are re- enabled. This allows the stack to be formatted for optimal storage of 16- byte XMM registers, which enables the interrupt handler to use faster 16- byte aligned loads and stores (MOVAPS rather than MOVUPS) to save and restore XMM registers.

Although the RSP alignment is always performed when LMA = 1, it is only of consequence for the kernel- mode case where there is no stack switch or I ST used. For a stack switch or I ST, the OS would have presumably put suitably aligned RSP values in the TSS.

IRET in IA-32e Mode

In IA- 32e mode, IRET executes with an 8- byte operand size. There is nothing that forces this requirement. The stack is formatted in such a way that for actions where IRET is required, the 8- byte IRET operand size works correctly.

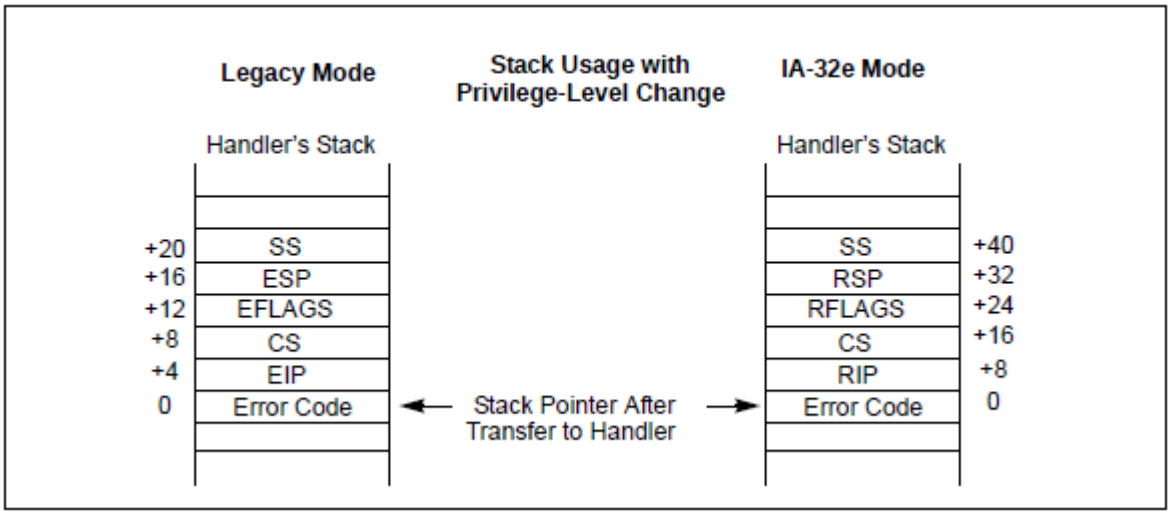
Because interrupt stack- frame pushes are always eight bytes in IA- 32e mode, an IRET must pop eight byte items off the stack. This is accomplished by preceding the IRET with a 64- bit operand-size prefix. The size of the pop is determined by the address size of the instruction. The SS/ ESP/ RSP size adjustment is determined by the stack size.

I RET pops SS: RSP unconditionally off the interrupt stack frame only when it is executed in 64-bit mode. In compat- ibility mode, I RET pops SS: RSP off the stack only if there is a CPL change. This allows legacy applications to execute properly in compatibility mode when using the IRET instruction. 64- bit interrupt service routines that exit with an I RET unconditionally pop SS: RSP off of the interrupt stack frame, even if the target code segment is running in 64- bit mode or at CPL = 0. This is because the original interrupt always pushes SS: RSP.

In IA- 32e mode, IRET is allowed to load a NULL SS under certain conditions. I f the target mode is 64- bit mode and the target CPL <> 3, I RET allows SS to be loaded with a NULL selector. As part of the stack switch mechanism, an interrupt or exception sets the new SS to NULL, instead of fetching a new SS selector from the TSS and loading the corresponding descriptor from the GDT or LDT. The new SS selector is set to NULL in order to properly handle returns from subsequent nested far transfers. I f the called procedure itself is interrupted, the NULL SS is pushed on the stack frame. On the subsequent IRET, the NULL SS on the stack acts as a flag to tell the processor not to load a new SS descriptor.

The I A- 32 architecture provides a mechanism to automatically switch stack frames in response to an interrupt. The 64- bit extensions of I ntel 64 architecture implement a modified version of the legacy stack- switching mechanism and an alternative stack- switching mechanism called the interrupt stack table (IST).

In I A- 32 modes, the legacy I A- 32 stack- switch mechanism is unchanged. In I A- 32e mode, the legacy stack- switch mechanism is modified. When stacks are switched as part of a 64- bit mode privilege- level change (resulting from an interrupt), a new SS descriptor is not loaded. IA- 32e mode loads only an inner- level RSP from the TSS. The new SS selector is forced to NULL and the SS selector's RPL field is set to the new CPL. The new SS is set to NULL in order to handle nested far transfers (far CALL, I NT, interrupts and exceptions). The old SS and RSP are saved on the new stack (Figure 6- 8). On the subsequent I RET, the old SS is popped from the stack and loaded into the SS register. In summary, a stack switch in IA- 32e mode works like the legacy stack switch, except that a new SS selector is not loaded from the TSS. Instead, the new SS is forced to NULL.



IA-32e Mode Stack Usage after Privilege Level Change

Interrupt Stack Table

In IA- 32e mode, a new interrupt stack table (IST) mechanism is available as an alternative to the modified legacy stack- switching mechanism described above. This mechanism unconditionally switches stacks when it is enabled. I t can be enabled on an individual interrupt-vector basis using a field in the I DT entry. This means that some inter- rupt vectors can use the modified legacy mechanism and others can use the IST mechanism.

The IST mechanism is only available in IA- 32e mode. I t is part of the 64- bit mode TSS. The motivation for the IST mechanism is to provide a method for specific interrupts (such as NMI , double-fault, and machine- check) to always execute on a known good stack. I n legacy mode, interrupts can use the task- switch mechanism to set up a known- good stack by accessing the interrupt service routine through a task gate located in the IDT. However, the legacy task- switch mechanism is not supported in I A- 32e mode.

The IST mechanism provides up to seven IST pointers in the TSS. The pointers are referenced by an interrupt- gate descriptor in the interrupt- descriptor table (IDT); see Figure 6- 7. The gate descriptor contains a 3- bit I ST index field that provides an offset into the IST section of the TSS. Using the IST mechanism, the processor loads the value pointed by an IST pointer into the RSP.

When an interrupt occurs, the new SS selector is forced to NULL and the SS selector's RPL field is set to the new CPL. The old SS, RSP, RFLAGS, CS, and RIP are pushed onto the new stack. Interrupt processing then proceeds as normal. I f the IST index is zero, the modified legacy stack- switching mechanism described above is used.

EXCEPTION AND INTERRUPT REFERENCE

The following sections describe conditions which generate exceptions and interrupts. They are arranged in the order of vector numbers. The information contained in these sections are as follows:

- **Exception Class** — indicates whether the exception class is a fault, trap, or abort type. Some exceptions can be either a fault or trap type, depending on when the error condition is detected. (This section is not applicable to interrupts.)
- **Description** — Gives a general description of the purpose of the exception or interrupts type. I t also describes how the processor handles the exception or interrupt.
- **Exception Error Code** — indicates whether an error code is saved for the exception. I f one is saved, the contents of the error code are described. (This section is not applicable to interrupts.)
- **Saved I nstruction Pointer —** Describes which instruction the saved (or return) instruction pointer points to. It also indicates whether the pointer can be used to restart a faulting instruction.

Program State Change — describes the effects of the exception or interrupt on the state of the currently running program or task and the possibilities of restarting the program or task without loss of continuity.

Because exceptions and interrupts generally do not occur at predictable times, these privilege rules effectively impose restrictions on the privilege levels at which exception and interrupt- handling procedures can run. Either of the following techniques can be used to avoid privilege- level violations.

- The exception or interrupt handler can be placed in a conforming code segment. This technique can be used for handlers that only need to access data available on the stack (for example, divide error exceptions). I f the handler needs data from a data segment, the data segment needs to be accessible from privilege level 3, which would make it unprotected.
- The handler can be placed in a nonconforming code segment with privilege level 0. This handler would always run, regardless of the CPL that the interrupted program or task is running at.

Flag Usage By Exception- or Interrupt-Handler Procedure

When accessing an exception or interrupt handler through either an interrupt gate or a trap gate, the processor clears the TF flag in the EFLAGS register after it saves the contents of the EFLAGS register on the stack. (On calls to exception and interrupt handlers, the processor also clears the VM, RF, and NT flags in the EFLAGS register, after they are saved on the stack.) Clearing the TF flag prevents instruction tracing from affecting interrupt response. A subsequent I RET instruction restores the TF (and VM, RF, and NT) flags to the values in the saved contents of the EFLAGS register on the stack.

The only difference between an interrupt gate and a trap gate is the way the processor handles the I F flag in the EFLAGS register. When accessing an exception- or interrupt- handling procedure through an interrupt gate, the processor clears the IF flag to prevent other interrupts from interfering with the current interrupt handler. A subse- quent I RET instruction restores the IF flag to its value in the saved contents of the EFLAGS register on the stack. Accessing a handler procedure through a trap gate does not affect the IF flag.

Interrupt Tasks

When an exception or interrupt handler is accessed through a task gate in the IDT, a task switch results. Handling an exception or interrupt with a separate task offers several advantages:

- The entire context of the interrupted program or task is saved automatically.
- A new TSS permits the handler to use a new privilege level 0 stack when handling the exception or interrupt. I f an exception or interrupt occurs when the current privilege level 0 stack is corrupted, accessing the handler through a task gate can prevent a system crash by providing the handler with a new privilege level 0 stack.
- The handler can be further isolated from other tasks by giving it a separate address space. This is done by giving it a separate LDT.

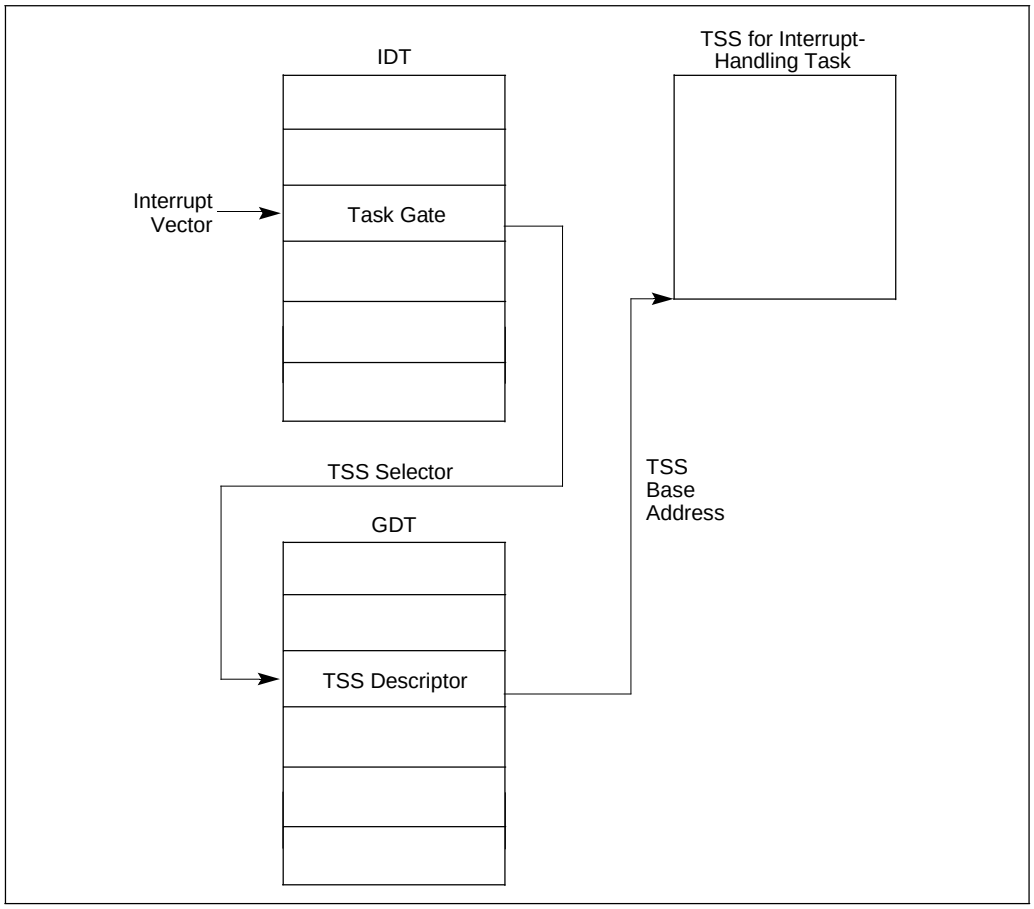
The disadvantage of handling an interrupt with a separate task is that the amount of machine state that must be saved on a task switch makes it slower than using an interrupt gate, resulting in increased interrupt latency.

A task gate in the IDT references a TSS descriptor in the GDT (see Figure 6-5). A switch to the handler task is handled in the same manner as an ordinary task switch (see Section 7.3, “Task Switching”). The link back to the interrupted task is stored in the previous task link field of the handler task’s TSS. I f an exception caused an error code to be generated, this error code is copied to the stack of the new task.

When exception- or interrupt- handler tasks are used in an operating system, there are actually two mechanisms that can be used to dispatch tasks: the software scheduler (part of the operating system) and the hardware sched- uler (part of the processor's interrupt mechanism). The software scheduler needs to accommodate interrupt tasks that may be dispatched when interrupts are enabled.

NOTE

Because IA- 32 architecture tasks are not re- entrant, an interrupt- handler task must disable interrupts between the time it completes handling the interrupt and the time it executes the IRET instruction. This action prevents another interrupt from occurring while the interrupt task’s TSS is still marked busy, which would cause a general- protection (#GP) exception.



Protected-Mode Exceptions and Interrupts (Contd.)

Exception Handling

The trouble with programmed I/O is that it both wastes CPU resources and it has potential for incorrect operation.

What we really want:

1. (Since most I/O devices are slow), have I/O devices signal the CPU when they have a change in status. This would be more efficient than polling the devices at the devices' maximum operating speed.
2. The I/O devices tell the processor that they become "ready."

In order to do this we need:

1. Hardware (wires) from devices to the CPU.
2. A way for special software to be invoked when the a device signals on the wire.

The modern solution bundles the software to deal with these signals (interrupts) and other situations into an **exception handler**. (Effectively part of the OS.)

Exceptions

There are 2 categories of exceptions: **interrupts** and **traps**. All processors use the same mechanism (a combination of hardware and software) to deal with exceptions.

Note: The Java and C++ languages have overloaded the term **exception**. Processors have used this term since the 1950s. OO languages developed many years later. For this class, the term **exception** does *not* refer to Java or C++ exceptions.

The 2 categories:

1. **interrupts**
   - ✓ initiated outside the instruction stream
   - ✓ arrive asynchronously (at no specific time), with respect to the timing of the fetch and execute cycle

examples:

- ✓ I/O device status change
- ✓ I/O device error condition
- ✓ thermal override shutdown
- ✓ internal error detection

When should the interrupt be dealt with? Answer: as soon as conveniently possible, and before the condition that caused the interrupt might cause yet-another interrupt (losing the first!)

2. **traps**
   - ✓ occur due to something in instruction stream
   - ✓ arrive synchronously (while instruction is executing). A good test: if program was re-run (with the same input), the trap would occur in precisely the same place in the code.

Examples:

- unaligned address error
- arithmetic overflow
- syscall

When should the trap be dealt with? Answer: right now! The user program cannot continue until whatever caused the trap is dealt with.

Exception Handling

The mechanism for dealing with exceptions is simple; its implementation can get complex. The implementation varies among architectures.

Situation: a user program (also called an application) is running (executing), and a device generates an interrupt request.

Mechanism to respond: the hardware temporarily suspends the user program, and instead runs code called an **exception handler**. After the handler is finished doing whatever it needs to, the hardware returns control to the user program. The user program continues from where it left off.

Limitations of exception handler:
Since it is being invoked (potentially) in the middle of a user program, the handler must take extra care not to change the state of the user program.

- it cannot change register values
- it cannot change the stack

So, how can it do anything at all?
-- The key to this answer is that any portion of the processor state that the handler wishes to change must be saved before the change and restored before returning to the user program.
-- The handler often uses the system stack to temporarily save register values.

When to handle an interrupt -- 2 possibilities:

1. Right now! Note that this could be in the middle of an instruction. In order to do this, the hardware must be able to know where the instruction is in its execution and be able to "take up where it left off". This is very difficult to do, because the hardware is complex. But, it has been done in simpler forms on a few machines. Example: arbitrary memory to memory copy on IBM 360.
2. Wait until the currently executing instruction finishes, then handle. *THIS IS THE METHOD OF CHOICE.* It handles interrupts in between 2 instructions.

The instruction fetch/execute cycle must be expanded to

1. If an interrupt is pending, handle it.
2. instruction fetch
3. PC update
4. decode
5. operand load
6. operation
7. store results

The MIPS R2000 exception handling mechanism

When an exception occurs, the hardware does the following things. Note that there is no inherent ordering of #1-4. They all happen "between" instructions, and before #5.

- ✓ processor sets state giving cause of exception

within the **Cause register** -- coprocessor C0, register $13, a 32-bit register
bits 6..2 (5 bits) specify the type of the exception, called the ExcCode.
Here are some mappings of encodings to causes.

Examples:

00000 (0) Interrupt
00100 (4) load from an illegal address
01000 (8) syscall instruction
01100 (12) arithmetic overflow

- ✓ changes to kernel mode, saving the previous mode in a hardware stack (3 levels deep)

The mode is in the **Status register** -- coprocessor C0, register 12, bit 1.
user mode = 1
kernel mode = 0
defined in the processor's architecture are 2 modes,

- ✓ **user** -- the mode that user programs run under. Certain instructions are not available, like those that can write to the control registers (Status register and Cause register).
- ✓ **kernel** -- the operating system mode. Allows the OS to retain control over "vital" system aspects. All instructions are available.

1. disable further interrupts bit 0 of the Status register the field is called IEc (Interrupt Enable, Current) determines whether interrupts are currently enabled = 1 disabled = 0

If interrupts are disabled, then the hardware is not checking to see if there are further interrupts to handle. Disabling makes sure that the handling of an interrupt is not interrupted.

2. save current PC coprocessor C0, register 14, called the **Exception Program Counter**.

Gives return address within user program. Where to return to when done handling the exception.

3. jumps to hardwired address 0x8000 0080. This is where the exception handler code is.

Then, the instruction fetch and execute cycle starts up again, only now the code within the exception handler is being executed.

This handler code does the following:

1. Save some registers (on system stack).
2. The handler needs to use registers too! It may not change (clobber, overwrite) the register contents of the user program. So, it saves them (on stack or in memory).
3. Figure out exception type. (in ExcCode)
4. mfc0 $k0, $13 # get Cause register
5. andi $k0, $k0, 0x3c # Mask out all but ExcCode
6. Use ExcCode in combination with a **jump table** to jump to the correct location within the exception handler.
7. Handle the exception (whatever it is!)
8. Restore registers saved in step 1.
9. atomically: (as if done in 1 step, not 3)

See Useful diagrams for the MIPS R2000, as distributed in class!
The **EPC** (Exception Program Counter)
The **Status Register**
The **Cause Register**
some terms

- **Interrupt request** -- the activation of hardware somewhere that signals the initial request for an interrupt.
- **pending interrupt** -- an interrupt that has not been handled yet, but needs to be
- **Kernel** -- the exception handler. In most minds, when people think of a kernel, they think of critical portions of an operating system. The exception handler *is* a critical portion of an operating system!
- **Handler** -- the code of the exception handler.
- **nonreentrant** -- what we talk about (mostly) in 354. While running an exception handler (the kernel), further pending interrupts are ignored.
- **reentrant** -- An exception handler that is carefully crafted such that the handling of one exception can be interrupted to handle a second exception.

about Jump Tables
A clever mechanism for implementing a switch statement. A jump to one of many locations
Keep a table of addresses (case1, case2, and case3):

```
JumpTable:  .word case0   # different syntax!
            .word case1   # the address assigned to these labels
            .word case2   #   becomes the contents of an array element


 sll  $8, $8, 2         # case number shifted left 2 bits

          # (need a word offset into table, not byte)
 lw   $9, JumpTable($8)  # load address into $9
 jr   $9                 # jump to address contained in $9

 .
 .
 .

case0:   #code for case0 here
 .
 .
 .
case1:   #code for case1 here
 .
 .
 .
case2:   #code for case2 here
```

Note that the cases do not have to go in any specific order.
not-yet-seen Addressing mode: label($rb)
Effective address is gotten by: label + ($rb)
label does not fit into 16 bit displacement field of load/store instruction. So, the MAL->TAL synthesis of this must be something like:

```
la  $1, label
add $1, $1, $rb
```

Then use 0 ($1) as addressing mode in load/store instructions

some advanced topics

Priorities

A problem: Multiple interrupt requests can arrive simultaneously. Which one should get handled first?

Possible solutions:

- ✓ FCFS -- the first one to arrive gets handled first.

Difficulty 1) This might allow a malicious/recalcitrant device or program to gain control of the processor.

Difficulty 2) There must be hardware that maintains an ordering of pending exceptions. (a queue)

- ✓ Prioritize all exceptions -- the one with the highest priority gets handled first. This is a common method for solving the problem.

Priorities for various exceptions are assigned either by the manufacturer, or by a system administrator through software. The priorities are normally defined (and fixed) when a machine is booted (the OS is started up).

Difficulty 1) Exceptions with the same priority must still be handled in some order. Example of same priority exceptions might be all keyboard interrupts. Consider a machine with many terminals hooked up.

The instructions fetch/execute cycle becomes:

1. If any interrupts with a higher priority than whatever is currently running pending, handle them now
2. instruction fetch
3. PC update
4. decode
5. get operands
6. do the instruction's operation
7. put result away

NOTE: This implies that there is some hardware notion of the priority for whatever code is currently running (application, keyboard interrupt handler, clock interrupt handler, etc.).

Priorities are a matter of which is most urgent, and therefore cannot wait, and how long it takes to process the interrupt.

- clock is urgent, clock interrupts occur frequently, and handling takes little processing, maybe only a variable increment.
- power failure is very urgent, but takes a lot or processing, because the machine will be stopped.
- overflow is urgent to the program which caused it, because the program cannot continue.
- keyboard is urgent because we do not want to lose a second key press before the first is handled.

So, what ordering ought to be imposed?

Reentrant Exception Handlers

The best solution combines priorities with an exception handler that can itself be interrupted. There are many details to get right to make this possible.

- The instruction fetch/execute cycle remains the same. At the beginning of *every* instruction (even those within the exception handler), a check is made to see
  - ✓ if there are pending interrupts
  - ✓ if listening for interrupts is enabled

Only those with higher priorities than whatever is currently running will be processed.

- The exception handler must be modified so that it can be interrupted. Its own state must be saved (safely).

Within the handler:

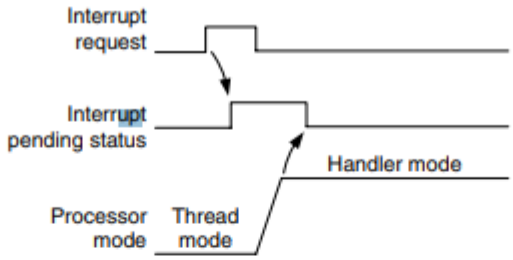
1. While interrupts are disabled, save important state that cannot get clobbered. (EPC, current priority level, maybe registers $26 and $27). Question: Where do these things get saved?
2. Re-enable interrupts for devices with higher priorities than current level. If the priority level

checking is done in hardware, then all interrupts can be re-enabled.
3. This invocation of the exception handler eventually finishes.

INTERRUPT INPUTS AND PENDING BEHAVIOUR

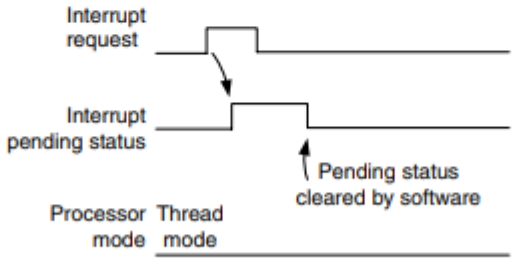
This section describes the behaviour of IRQ inputs and pending behaviour. It also applies to NMI input, except that an NMI will be executed immediately in most cases, unless the core is already executing an NMI handler, halted by a debugger, or locked up because of some serious system error. When an interrupt input is asserted, it will be pended, which means it is put into a state of waiting for the processor to process the request. Even if the interrupt source deasserts the interrupt, the pended interrupt status will still cause the interrupt handler to be executed when the priority is allowed.

Once the interrupt handler is started, the pending status is cleared automatically. This is shown in Figure. However, if the pending status is cleared before the processor starts responding to the pended interrupt (for example, the interrupt was not taken immediately because PRIMASK/FAULTMASK is set to 1, and the pending status was cleared by software writing to NVIC interrupt control registers), the interrupt can be cancelled (Figure 7.10). The pending status of the interrupt can be accessed in the NVIC and is writable, so you can clear a pending interrupt or use software to pend a new interrupt by setting the pending register.



FIGURE

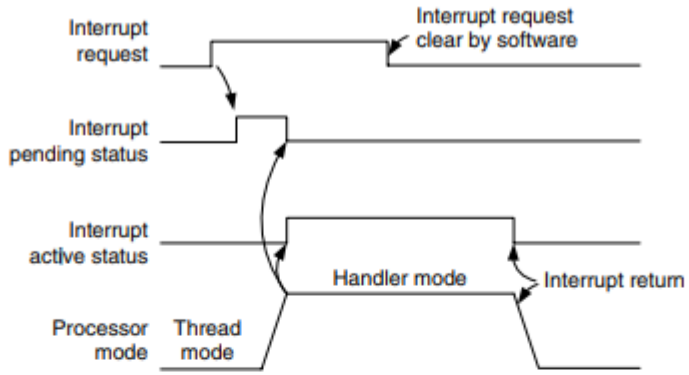
Interrupt Pending.



FIGURE

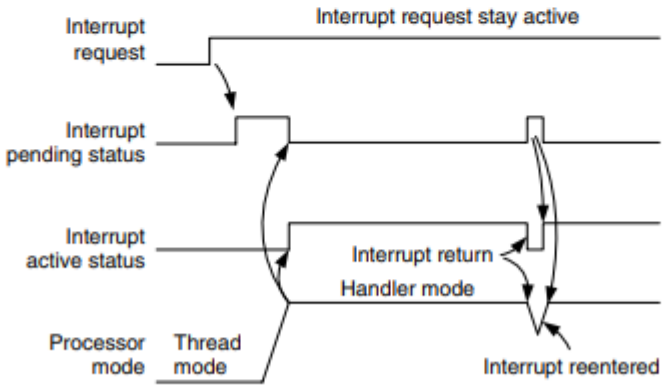
Interrupt Pending Cleared Before Processor Takes Action.

When the processor starts to execute an interrupt, the interrupt becomes active and the pending bit will be cleared automatically (Figure 7.11). When an interrupt is active, you cannot start processing the same interrupt again, until the interrupt service routine is terminated with an interrupt return (also called an exception exit, as discussed in Chapter 9). Then the active status is cleared, and the interrupt can be processed again if the pending status is 1. It is possible to repend an interrupt before the end of the interrupt service routine. 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 as shown in Figure. This is just like the traditional ARM7TDMI. If an interrupt is pulsed several times before the processor starts processing it, it will be treated as one single interrupt request as illustrated in Figure. If an interrupt is disserted and then pulsed again during the interrupt service routine, it will be pended again as shown in Figure.



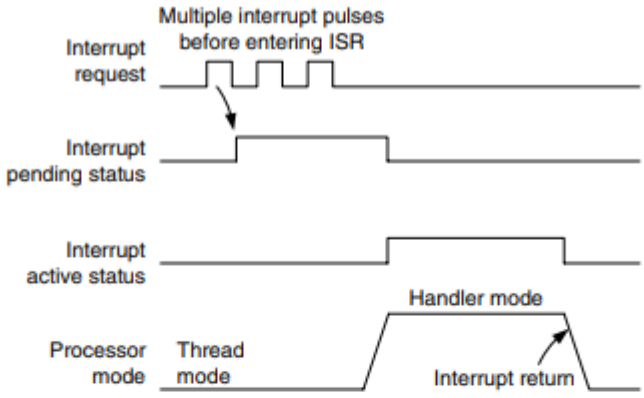
FIGURE

Interrupt Active Status Set as Processor Enters Handler.



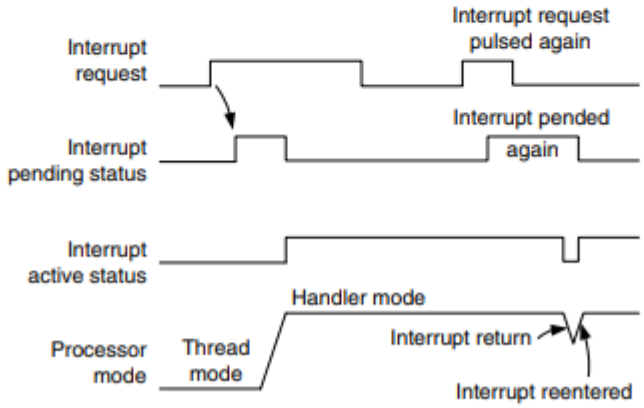
FIGURE

Continuous Interrupt Request Pends Again After Interrupt Exit.



FIGURE

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



FIGURE

Interrupt Pending Occurs Again during the Handler.

Pending of an interrupt can happen even if the interrupt is disabled; the pended interrupt can then trigger the interrupt sequence when the enable is set later. As a result, before enabling an interrupt, it could be useful to check whether the pending register has been set. The interruptsource might have been activated previously and have set the pending status. If necessary, you can clear the pending status before you enable an interrupt.

FAULT EXCEPTIONS:

Bus faults are produced when an error response is received during a transfer on the AHB interfaces. It can happen at these stages: • Instruction fetch, commonly called prefetch abort • Data read/write, commonly called data abort In the Cortex-M3, bus faults can also occur during the following: • Stack PUSH in the beginning of interrupt processing, called a stacking error • Stack POP at the end of interrupt processing, called an unstacking error • Reading of an interrupt vector address (vector fetch) when the processor starts the interrupthandling sequence (a special case classified as a hard fault) When these types of bus faults (except vector fetches) take place and if the bus fault handler is enabled and no other exceptions with the same or higher priority are running, the bus fault handler will be executed. If the bus fault handler is enabled but at the same time the core receives another exception handler with higher priority, the bus fault exception will be pending. Finally, 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. If another bus fault takes place when running the hard fault handler, the core will enter a lockup state.

To enable the bus fault handler, you need to set the BUSFAULTENA bit in the System Handler Control and State register in the NVIC. Before doing that, make sure that the bus fault handler starting address is set up in the vector table if the vector table has been relocated to RAM. Hence, how do you

find out what went wrong when the processor entered the bus fault handler? The NVIC has a number of Fault Status registers (FSRs). One of them is the Bus Fault Status register (BFSR). From this register, the bus fault handler can find out if the fault was caused by data/instruction access or an interrupt stacking or unstacking operation.

For precise bus faults, the offending instruction can be located by the stacked program counter, and if the BFARVALID bit in BFSR is set, it is also possible to determine the memory location that caused the bus fault. This is done by reading another NVIC register called the Bus Fault Address register (BFAR). However, the same information is not available for imprecise bus faults because by the time the processor receives the error, the processor could have already executed a number of other instructions.

Table 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	IMPRECISERR	R/Wc	0	Imprecise data access violation
1	PRECISERR	R/Wc	0	Precise data access violation
0	IBUSERR	R/Wc	0	Instruction access violation

Supervisor Call and Pendable Service Call:

Supervisor Call (SVC) and Pendable Service Call (PendSV) are two exceptions targeted at software and operating systems. SVC is for generating system function calls. For example, instead of allowing user programs to directly access hardware, an operating system may provide access to hardware through an SVC. So when a user program wants to use certain hardware, it generates the SVC exception using SVC instructions, and then the software exception handler in the operating system is executed and provides the service the user application requested.

In this way, access to hardware is under the control of the OS, which can provide a more robust system by preventing the user applications from directly accessing the hardware. SVC can also make software more portable because the user application does not need to know the programming details of the hardware. The user program will only need to know the application programming interface (API) function ID and parameters; the actual hardware-level programming is handled by device drivers. SVC exception is generated using the SVC instruction. An immediate value is required for this instruction, which works as a parameter-passing method. The SVC exception handler can then extract the parameter and determine what action it needs to perform.

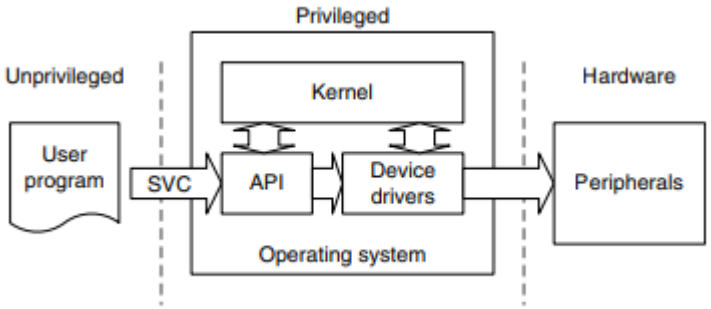
For example,

SVC #0x3; Call SVC function 3

The traditional syntax for SVC is also acceptable (without the "#"):

SVC 0x3; Call SVC function 3

For C language development, the SVC instruction can be generated using __svc function (for ARM Real View C Compiler or KEIL Microcontroller Development Kit for ARM), or using inline assembly in other C compilers.



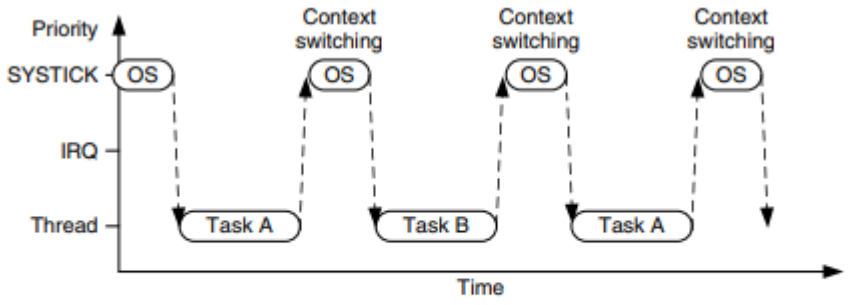
When the SVC handler is executed, you can determine the immediate data value in the SVC instruction by reading the stacked program counter value, then reading the instruction from that address and masking out the unneeded bits. If the system uses a Process Stack Pointer for user applications, you might need to determine which stack was used first. This can be determined from the link register value when the handler is entered.

Because of the interrupt priority model in the Cortex-M3, you cannot use SVC inside an SVC handler (because the priority is the same as the current priority). Doing so will result in a usage fault. For the same reason, you cannot use SVC in an NMI handler or a hard fault handler.

PendSV (Pendable Service Call) works with SVC in the OS. Although SVC (by SVC instruction) cannot be pended (an application calling SVC will expect the required task to be done immediately), PendSV can be pended and is useful for an OS to pend an exception so that an action can be performed after other important tasks are completed. PendSV is generated by writing 1 to the PENDSVSET bit in the NVIC Interrupt Control State register.

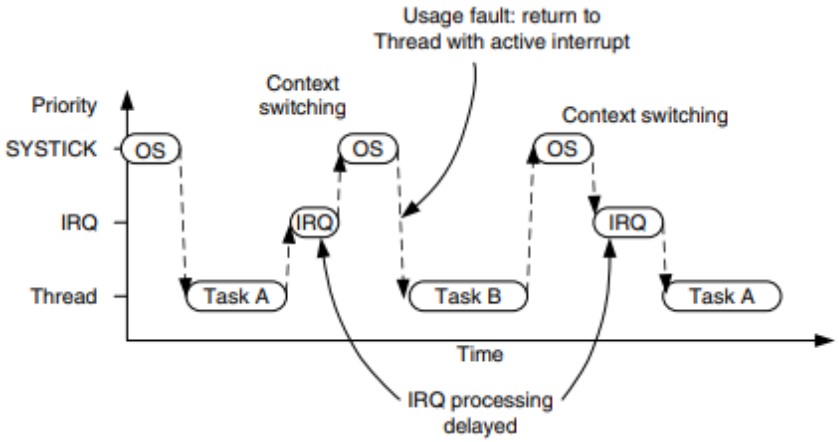
A typical use of PendSV is context switching (switching between tasks). For example, a system might have two active tasks, and context switching can be triggered by the following:

- Calling an SVC function
- The system timer (SYSTICK) Let's look at a simple example of having only two tasks in a system, and a context switch is triggered by SYSTICK exceptions (see Figure). If an interrupt request takes place before the SYSTICK exception, the SYSTICK exception will preempt the IRQ handler. In this case, the OS should not carry out the context switching. Otherwise the



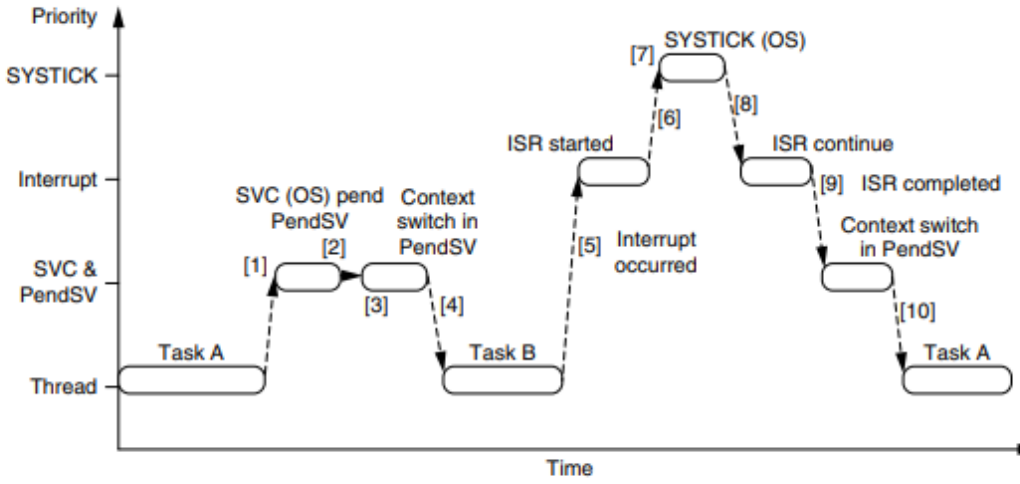
FIGURE

A Simple Scenario Using SYSTICK to Switch between Two Tasks.



FIGURE

Problem with Context Switching at the IRQ.



FIGURE

Example Context Switching with PendSV.

NESTED VECTORED INTERRUPT CONTROLLER AND INTERRUPT CONTROL:

As we've seen, the Nested Vectored Interrupt Controller (NVIC)is an integrated part of the Cortex™-M3 processor. It is closely linked to the Cortex-M3 CPU core logic. Its control registers are accessible as memory-mapped devices. Besides control registers and control logic for interrupt processing, the NVIC unit also contains control registers for the SYSTICK Timer, and debugging controls. In this chapter, we'll examine the control logic for interrupt processing. Memory Protection Unit and debugging control logic are discussed in later chapters. The NVIC supports 1–240 external interrupt inputs (commonly known as interrupt request [IRQs]). The exact number of supported interrupts is determined by the chip manufacturers when they develop their Cortex-M3 chips. In addition, the NVIC also has a Nonmaskable Interrupt (NMI) input. The actual function of the NMI is also decided by the chip manufacturer. In some cases, this NMI cannot be controlled from an external source. The NVIC can be accessed in the System Control Space (SCS) address range, which is memory location 0xE000E000. Most of the interrupt control/status registers are accessible only in privileged mode, except the Software Trigger Interrupt register (STIR), which can be set up to be accessible in user mode. The interrupt control/status register can be accessed in word, half word, or byte transfers. In addition, a few other interrupt-masking registers are also involved in the interrupts. They are the "special registers" covered in Chapter 3 and are accessed through special registers access instructions: move

special register to general-purpose register (MRS) and move to special register from general-purpose register (MSR) instructions.

The Basic Interrupt Configuration

Each external interrupt has several registers associated with it.

- Enable and Clear Enable registers
- Set-Pending and Clear-Pending registers
- Priority level
- Active status In addition, a number of other registers can also affect the interrupt processing:
- Exception-masking registers (PRIMASK, FAULTMASK, and BASEPRI)
- Vector Table Offset register
- STIR
- Priority group

Interrupt Enable and Clear Enable:

The Interrupt Enable register is programmed through two addresses. To set the enable bit, you need to write to the SETENA register address; to clear the enable bit, you need to write to the CLRENA register address. In this way, enabling or disabling an interrupt will not affect other interrupt enable states. The SETENA/CLRENA registers are 32 bits wide; each bit represents one interrupt input. As there could be more than 32 external interrupts in the Cortex-M3 processor, you might find more than one SETENA and CLRENA register—for example, SETENA0, SETENA1, and so on. Only the enable bits for interrupts that exist are implemented. So, if you have only 32 interrupt inputs, you will only have SETENA0 and CLRENA0. The SETENA and CLRENA registers can be accessed as word, half word, or byte. As the first 16 exception types are system exceptions, external Interrupt #0 has a start exception number of 16.

Interrupt Set Pending and Clear Pending:

If an interrupt takes place but cannot be executed immediately (for instance, if another higher-priority interrupt handler is running), it will be pended. The interrupt-pending status can be accessed through the Interrupt Set Pending (SETPEND) and Interrupt Clear Pending (CLRPEND) registers. Similarly to the enable registers, the pending status controls might contain more than one register if there are more than 32 external interrupt inputs.

The values of pending status registers can be changed by software, so you can cancel a current pended exception through the CLRPEND register, or generate software interrupts through the SETPEND register.

Priority Levels:

Each external interrupt has an associated priority-level register, which has a maximum width of 8 bits and a minimum width of 3 bits. As described in the previous chapter, each register can be further divided into preempt priority level and subpriority level based on priority group settings. The prioritylevel registers can be accessed as byte, half word, or word. The number of priority-level registers depends on how many external interrupts the chip contains.

Table Interrupt Set Enable Registers and Interrupt Clear Enable Registers (0xE000E100-0xE000E11C, 0xE000E180-0xE000E19C)

Address	Name	Type	Reset Value	Description
0xE000E100	SETENA0	R/W	0	Enable for external Interrupt #0–31 bit[0] for Interrupt #0 (exception #16) bit[1] for Interrupt #1 (exception #17) … bit[31] for Interrupt #31 (exception #47) Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
0xE000E104	SETENA1	R/W	0	Enable for external Interrupt #32–63 Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
0xE000E108	SETENA2	R/W	0	Enable for external Interrupt #64–95 Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
…	—	—	—	—
0xE000E180	CLRENA0	R/W	0	Clear enable for external Interrupt #0–31 bit[0] for Interrupt #0 bit[1] for Interrupt #1 … bit[31] for Interrupt #31 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current enable status
0xE000E184	CLRENA1	R/W	0	Clear enable for external Interrupt #32–63 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current enable status
0xE000E188	CLRENA2	R/W	0	Clear enable for external Interrupt #64–95 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current enable status
…	—	—	—	—

Table Interrupt Set-Pending Registers and Interrupt Clear-Pending Registers (0xE000E200-0xE000E21C, 0xE000E280-0xE000E29C)

Address	Name	Type	Reset Value	Description
0xE000E200	SETPEND0	R/W	0	Pending for external Interrupt #0–31 bit[0] for Interrupt #0 (exception #16) bit[1] for Interrupt #1 (exception #17) … bit[31] for Interrupt #31 (exception #47) Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
0xE000E204	SETPEND1	R/W	0	Pending for external Interrupt #32–63 Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status

Table Interrupt Set-Pending Registers and Interrupt Clear-Pending Registers (0xE000E200-0xE000E21C, 0xE000E280-0xE000E29C) *Continued*

Address	Name	Type	Reset Value	Description
0xE000E208	SETPEND2	R/W	0	Pending for external Interrupt #64–95 Write 1 to set bit to 1; write 0 has no effect Read value indicates the current status
...	—	—	—	—
0xE000E280	CLRPEND0	R/W	0	Clear pending for external Interrupt #0–31 bit[0] for Interrupt #0 (exception #16) bit[1] for Interrupt #1 (exception #17) ... bit[31] for Interrupt #31 (exception #47) Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current pending status
0xE000E284	CLRPEND1	R/W	0	Clear pending for external Interrupt #32–63 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current pending status
0xE000E288	CLRPEND2	R/W	0	Clear pending for external Interrupt #64–95 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current pending status
...	—	—	—	—

CONFIGURATION REGISTERS FOR OTHER EXCEPTIONS:

Usage faults, memory management faults, and bus fault exceptions are enabled by the System Handler Control and State register (0xE000ED24). The pending status of faults and active status of most system exceptions are also available from this register.

Table 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 handler enable
15	SVCALLPENDED	R/W	0	SVC pended; SVC 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 SVC 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.

SYSTICK TIMER:

The SYSTICK Timer is integrated with the NVIC and can be used to generate a SYSTICK exception (exception type #15). In many operating systems, a hardware timer is used to generate interrupts so that the OS can carry out task management—for example, to allow multiple tasks to run at different time slots and to make sure that no single task can lock up the whole system. To do that, the timer needs to be able to generate interrupts, and if possible, it should be protected from user tasks so that user applications cannot change the timer behaviour.

The Cortex-M3 processor includes a simple timer. Because all Cortex-M3 chips have the same time, porting software between different Cortex-M3 products is simplified. The timer is a 24-bit down counter. It can use the internal free running processor clock signal on the Cortex-M3 processor or an external reference clock (documented as the STCLK signal on the Cortex-M3 TRM). However, the source of the STCLK will be decided by chip designers, so the clock frequency might vary between products. You should check the chip's datasheet carefully when selecting a clock source.

The SYSTICK Timer can be used to generate interrupts. It has a dedicated exception type and exception vector. It makes porting operating systems and software easier because the process will be the same across different Cortex-M3 products. The SYSTICK Timer is controlled by four registers, shown in Tables 8.9–8.12.

The Calibration Value register provides a solution for applications to generate the same SYSTICK interrupt interval when running on various Cortex-M3 products. To use it, just write the value in TENMS to the reload value register. This will give an interrupt interval of about 10 ms. For other interrupt timing intervals, the software code will need to calculate a new suitable value from the calibration value. However, the TENMS field might not be available in all Cortex-M3 products (the calibration input signals to the Cortex-M3 might have been tied low), so check with your manufacturer's datasheets before using this feature.

Aside from being a system tick timer for operating systems, the SYSTICK Timer can be used in a number of ways: as an alarm timer, for timing measurement, and more. Note that the SYSTICK

Table 8.9 SYSTICK Control and Status Register (0xE000E010)

Bits	Name	Type	Reset Value	Description
16	COUNTFLAG	R	0	Read as 1 if counter reaches 0 since last time this register is read; clear to 0 automatically when read or when current counter value is cleared
2	CLKSOURCE	R/W	0	0 = External reference clock (STCLK) 1 = Use processor free running clock
1	TICKINT	R/W	0	1 = Enable SYSTICK interrupt generation when SYSTICK Timer reaches 0 0 = Do not generate interrupt
0	ENABLE	R/W	0	SYSTICK Timer enable

Table 8.10 SYSTICK Reload Value Register (0xE000E014)

Bits	Name	Type	Reset Value	Description
23:0	RELOAD	R/W	0	Reload value when timer reaches 0

Table 8.11 SYSTICK Current Value Register (0xE000E018)

Bits	Name	Type	Reset Value	Description
23:0	CURRENT	R/Wc	0	Read to return current value of the timer. Write to clear counter to 0. Clearing of current value also clears COUNTFLAG in SYSTICK Control and Status register

Table 8.12 SYSTICK Calibration Value Register (0xE000E01C)

Bits	Name	Type	Reset Value	Description
31	NOREF	R	—	1 = No external reference clock (STCLK not available) 0 = External reference clock available
30	SKEW	R	—	1 = Calibration value is not exactly 10 ms 0 = Calibration value is accurate
23:0	TENMS	R/W	0	Calibration value for 10 ms; chip designer should provide this value through Cortex-M3 input signals. If this value is read as 0, calibration value is not available

Timer stops counting when the processor is halted during debugging. Depending on the design of the microcontroller, the SysTick Timer could also be stopped when the processor enters certain type of sleep modes.

To set up the SysTick Timer, the recommended programming sequence is as follows:

- Disable SysTick by writing 0 to the SYSTICK Control and Status register.
- Write new reload value to the SYSTICK Reload Value register.
- Write to the SYSTICK Current Value register to clear the current value to 0.
- Write to the SYSTICK Control and Status register to start the SysTick timer.

This programming sequence can be used on all Cortex-M3 processors.

Interrupt Sequences:
When an exception takes place, a number of things happen, such as

- 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 (LR), and program counter (PC)

Stacking:

When an exception takes place, the registers R0–R3, R12, LR, PC, and Program Status (PSR) are pushed to the stack. If the code that is running uses the Process Stack Pointer (PSP), the process stack will be used; if the code that is running uses the Main Stack Pointer (MSP), the main stack will be used. Afterward, the main stack will always be used during the handler, so all nested interrupts will use the main stack.

The block of eight words of data being pushed to the stack is commonly called a stack frame. Prior to Cortex™-M3 revision 2, the stack frame was started in any word address by default. In Cortex-M3 revision 2, the stack frame is aligned to double word address by default, although the alignment feature can be turned off by programming the STKALIGN bit in Nested Vectored Interrupt Controller (NVIC) Configuration Control register to zero. The stack frame feature is also available in Cortex-M3 revision 1, but it needs to be enabled by writing 1 to the STKALIGN bit. More details on this register can be found in Chapter 12. The data arrangement inside an exception stack frame is shown in Figure 9.1.

The order of stacking is shown in Figure 9.2 (assuming that the stack pointer [SP] value is N after the exception). Due to the pipeline nature of the Advanced High-Performance Bus (AHB) interface, the address and data are offset by one pipeline state. The values of PC and PSR are stacked first so that instruction fetch can be started early (which requires modification of PC) and the Interrupt Program Status register (IPSR) can be updated early. After stacking, SP will be updated, and the stacked data arrangement in the stack memory will look like Figure 9.1. The reason the registers R0–R3, R12, LR, PC, and PSR are stacked is that these are caller-saved registers, according to C standards(C/C++ standard Procedure Call Standard for the ARM Architecture.

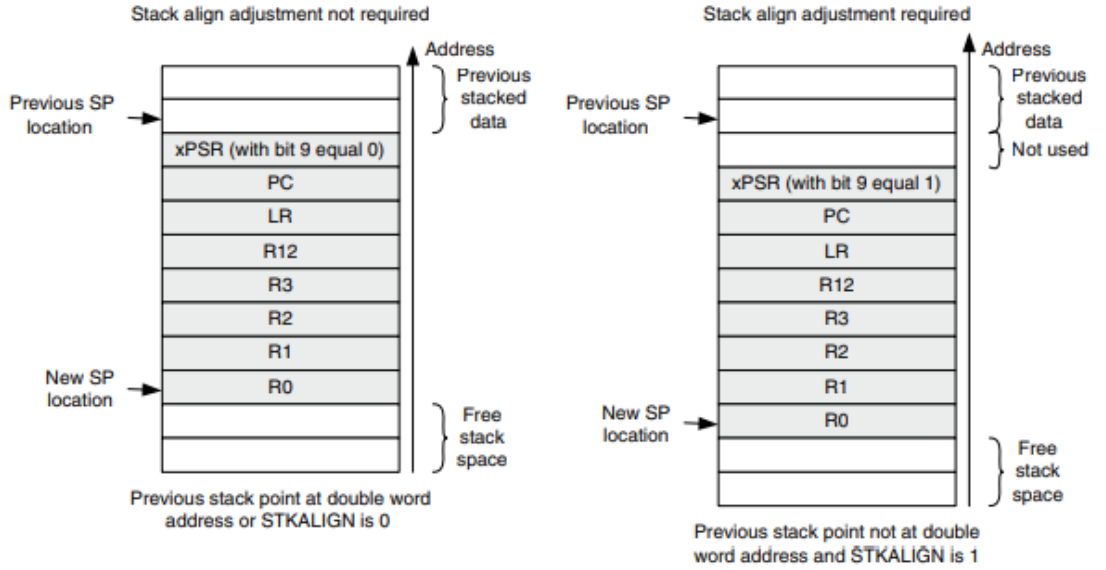


FIGURE 9.1

Exception Stack Frame.

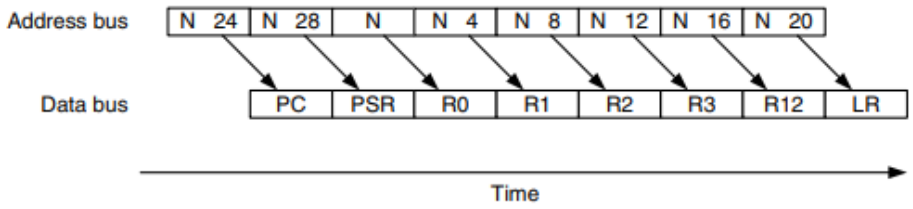


FIGURE 9.2

Stacking Sequence.

Exception Exits:

At the end of the exception handler, an exception exit (known as an interrupt return in some processors) is required to restore the system status so that the interrupted program can resume normal execution. There are three ways to trigger the interrupt return sequence; all of them use the special value stored in the LR in the beginning of the handler (see Table 9.1).

Some microprocessor architectures use special instructions for interrupt returns (for example, reti in 8051). In the Cortex-M3, a normal return instruction is used so that the whole interrupt handler can be implemented as a C subroutine.

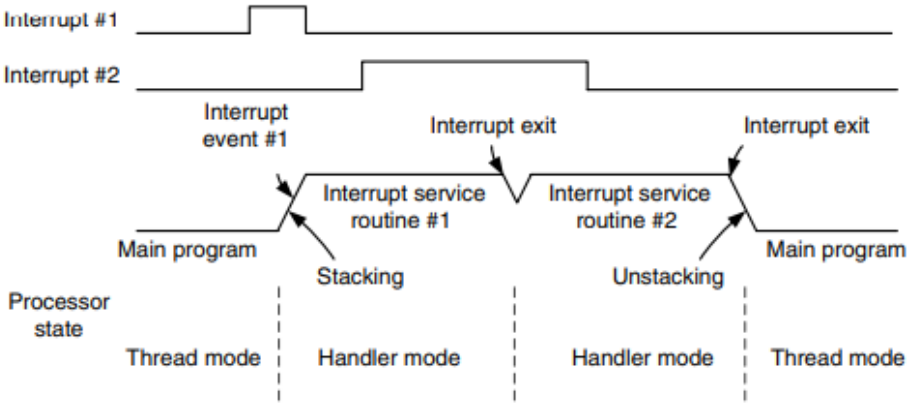
When the interrupt return instruction is executed, the unstacking and the NVIC registers update processes that are listed in Table are carried out.

Table Instructions That Can Be Used for Triggering Exception Return

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.
Load (LDR) or Load multiple (LDM)	It is possible to produce an interrupt return using the LDR or LDM instruction with PC as the destination register.

TAIL-CHAINING INTERRUPTS:

The Cortex-M3 uses a number of methods to improve interrupt latency. The first one we'll look at is tail chaining. When an exception takes place but the processor is handling another exception of the same or higher priority, the exception will enter pending state. When the processor has finished executing the current exception handler, it can then process the pended interrupt. Instead of restoring the registers back from the stack (unstacking) and then pushing them onto the stack again (stacking), the processor skips the unstacking and stacking steps and enters the exception handler of the pended exception as soon as possible. In this way, the timing gap between the two exception handlers is considerably reduced.



FIGURE

Tail Chaining of Exceptions.

INTERRUPT LATENCY:

The term interrupt latency refers to the delay from the start of the interrupt request to the start of interrupt handler execution. In the Cortex-M3 processor, if the memory system has zero latency, and provided that the bus system design allows vector fetch and stacking to happen at the same time, the interrupt latency can be as low as 12 cycles. This includes stacking the registers, vector fetch, and fetching instructions for the interrupt handler. However, this depends on memory access wait states and a few other factors.

For tail-chaining interrupts, since there is no need to carry out stacking operations, the latency of switching from one exception handler to another exception handler can be as low as six cycles. When the processor is executing a multicycle instruction, such as divide, the instruction could be abandoned and restarted after the interrupt handler completes. This also applies to load double (LDRD) and store double (STRD) instructions.

To reduce exception latency, the Cortex-M3 processor allows exceptions in the middle of Multiple Load and Store instructions (LDM/STM). If the LDM/STM instruction is executing, the current memory accesses will be completed, and the next register number will be saved in the stacked xPSR (Interrupt-Continuable Instruction [ICI] bits). After the exception handler completes, the multiple load/ store will resume from the point at which the transfer stopped. There is a corner case: If the multiple load/store instruction being interrupted is part of an IF-THEN (IT) instruction block, the load/store instruction will be cancelled and restarted when the interrupt is completed. This is because the ICI bits and IT execution status bits share the same space in the Execution
Program Status Register (EPSR).

In addition, if there is an outstanding transfer on the bus interface, such as a buffered write, the processor will wait until the transfer is completed. This is necessary to ensure that a bus fault handler preempts the correct process.

Of course, the interrupt could be blocked if the processor is already executing another exception handler of the same or higher priority or if the Interrupt Mask register was masking the interrupt request. In these cases, the interrupt will be pended and will not be processed until the blocking is removed.

UNIT-III
LPC 17XX
MICROCONTROLLER

UNIT-III
LPC 17XX MICROCONTROLLER

General description

The LPC1769/68/67/66/65/64/63 are ARM Cortex-M3 based microcontrollers for embedded applications featuring a high level of integration and low power consumption. The Arm Cortex-M3 is a next generation core that offers system enhancements such as enhanced debug features and a higher level of support block integration.

The LPC1768/67/66/65/64/63 operate at CPU frequencies of up to 100 MHz. The LPC1769 operates at CPU frequencies of up to 120 MHz. The Arm Cortex-M3 CPU incorporates a 3-stage pipeline and uses Harvard architecture with separate local instruction and data buses as well as a third bus for peripherals. The Arm Cortex-M3 CPU also includes an internal prefetch unit that supports speculative branching.

The peripheral complement of the LPC1769/68/67/66/65/64/63 includes up to 512 kB of flash memory, up to 64 kB of data memory, Ethernet MAC, USB Device/Host/OTG interface, 8-channel general purpose DMA controller, 4 UARTs, 2 CAN channels, 2 SSP controllers, SPI interface, 3 I^2C-bus interfaces, 2-input plus 2-output I2S-bus interface,
8-channel 12-bit ADC, 10-bit DAC, motor control PWM, Quadrature Encoder interface, four general purpose timers, 6-output general purpose PWM, ultra-low power Real-Time Clock (RTC) with separate battery supply, and up to 70 general purpose I/O pins.

The LPC1769/68/67/66/65/64/63 is pin-compatible to the 100-pin LPC236x Arm7-based microcontroller series.

Features and benefits

- ✓ Arm Cortex-M3 processor, running at frequencies of up to 100 MHz (LPC1768/67/66/65/64/63) or of up to 120 MHz (LPC1769). A Memory Protection Unit (MPU) supporting eight regions is included.
- ✓ Arm Cortex-M3 built-in Nested Vectored Interrupt Controller (NVIC).
- ✓ Up to 512 kB on-chip flash programming memory. Enhanced flash memory accelerator enables high-speed 120 MHz operation with zero wait states.
- ✓ In-System Programming (ISP) and In-Application Programming (IAP) via on-chip bootloader software.

On-chip SRAM includes:

1. 2/16 kB of SRAM on the CPU with local code/data bus for high-performance CPU access.
2. These SRAM blocks may be used for Ethernet, USB, and DMA memory, as well as for general purpose CPU instruction and data storage.
3. Eight channel General Purpose DMA controller (GPDMA) on the AHB multilayer matrix that can be used with SSP, I^2S-bus, UART, Analog-to-Digital and
4. Digital-to-Analog converter peripherals, timer match signals, and for memory-to-memory transfers.
5. Multilayer AHB matrix interconnect provides a separate bus for each AHB master. AHB masters include the CPU, General Purpose DMA controller, Ethernet MAC, and the USB interface. This interconnect provides communication with no arbitration delays.
6. Split APB bus allows high throughput with few stalls between the CPU and DMA.

Serial interfaces:

a. Ethernet MAC with RMII interface and dedicated DMA controller. (Not available on all parts, see Table 2.)
b. USB 2.0 full-speed device/Host/OTG controller with dedicated DMA controller and on-chip PHY for device, Host, and OTG functions. (Not available on all parts, see Table 2.)
c. Four UARTs with fractional baud rate generation, internal FIFO, and DMA support. One UART has modem control I/O and RS-485/EIA-485 support, and one UART has IrDA support.
d. CAN 2.0B controller with two channels. (Not available on all parts, see Table 2.)
e. SPI controller with synchronous, serial, full duplex communication and programmable data length.
f. Two SSP controllers with FIFO and multi-protocol capabilities. The SSP interfaces can be used with the GPDMA controller.
g. Three enhanced I2C bus interfaces, one with an open-drain output supporting full I2C specification and Fast mode plus with data rates of 1 Mbit/s, two with standard port pins. Enhancements include multiple address recognition and monitor mode.
h. I2S (Inter-IC Sound) interface for digital audio input or output, with fractional rate control. The I2S-bus interface can be used with the GPDMA. The I2S-bus interface supports 3-wire and 4-wire data transmit and receive as well as master clock input/output. (Not available on all parts, see Table 2.)

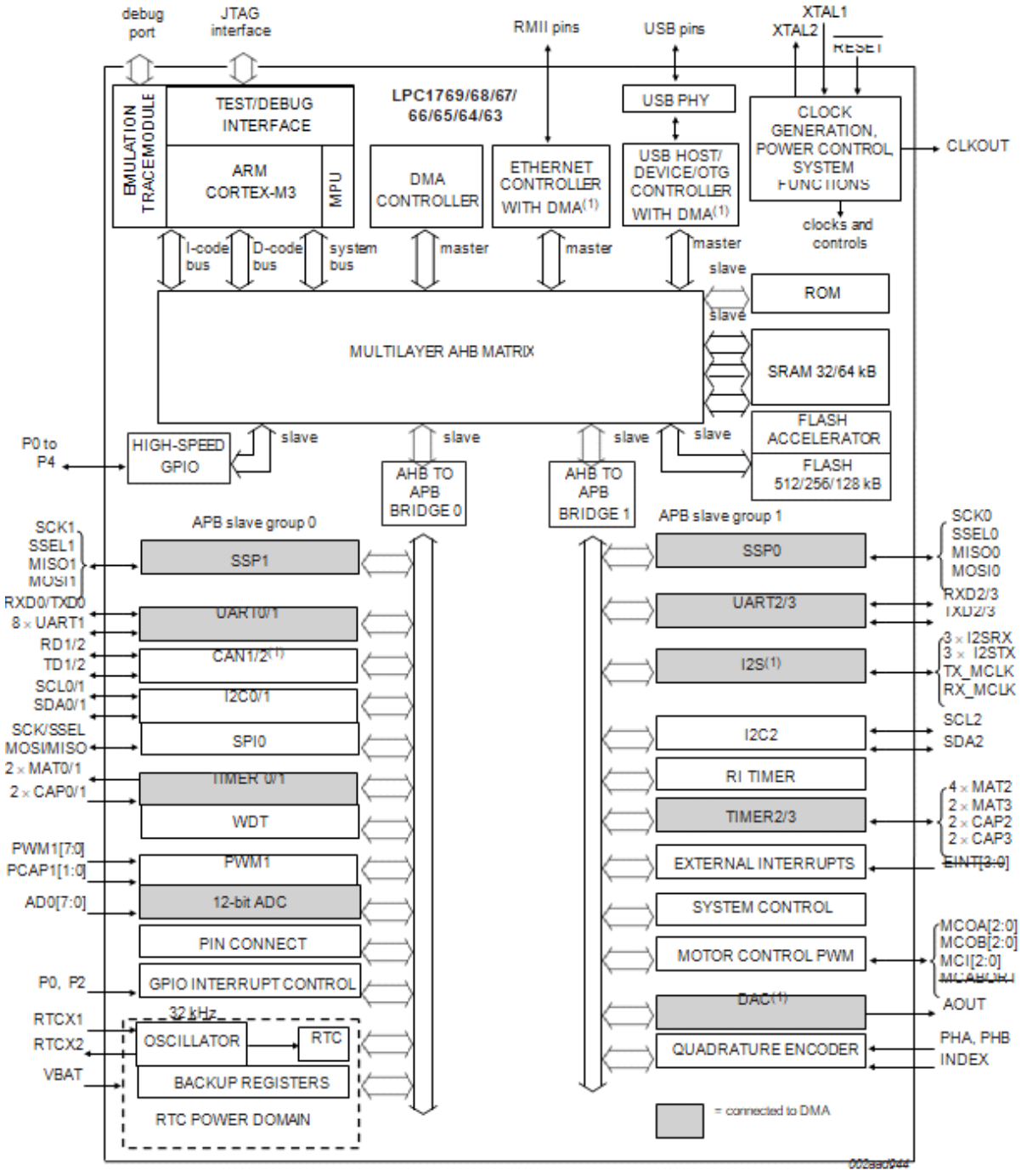
Other peripherals:

- ✓ 70 (100 pin package) General Purpose I/O (GPIO) pins with configurable
- ✓ pull-up/down resistors. All GPIOs support a new, configurable open-drain operating mode. The GPIO block is accessed through the AHB multilayer bus for fast access and located in memory such that it supports Cortex-M3 bit banding and use by the General Purpose DMA Controller.
- ✓ 12-bit Analog-to-Digital Converter (ADC) with input multiplexing among eight pins, conversion rates up to 200 kHz, and multiple result registers. The 12-bit ADC can be used with the GPDMA controller.
- ✓ 10-bit Digital-to-Analog Converter (DAC) with dedicated conversion timer and DMA support. (Not available on all parts, see Table 2)
- ✓ Four general purpose timers/counters, with a total of eight capture inputs and ten compare outputs. Each timer block has an external count input. Specific timer events can be selected to generate DMA requests.
- ✓ One motor control PWM with support for three-phase motor control.
- ✓ Zuadrature encoder interface that can monitor one external quadrature encoder.
- ✓ One standard PWM/timer block with external count input.
- ✓ RTC with a separate power domain and dedicated RTC oscillator. The RTC block includes 20 bytes of battery-powered backup registers.
- ✓ WatchDog Timer (WDT). The WDT can be clocked from the internal RC oscillator, the RTC oscillator, or the APB clock.
- ✓ Arm Cortex-M3 system tick timer, including an external clock input option.
- ✓ Repetitive interrupt timer provides programmable and repeating timed interrupts.
- ✓ Each peripheral has its own clock divider for further power savings.
- ✓ Standard JTAG debug interface for compatibility with existing tools. Serial Wire Debug and Serial Wire Trace Port options. Boundary Scan Description Language (BSDL) is not available for this device.
- ✓ Emulation trace module enables non-intrusive, high-speed real-time tracing of instruction execution.
- ✓ Integrated PMU (Power Management Unit) automatically adjusts internal regulators to minimize power consumption during Sleep, Deep sleep, Power-down, and Deep power-down modes.
- ✓ Four reduced power modes: Sleep, Deep-sleep, Power-down, and Deep power-down.

- ✓ Single 3.3 V power supply (2.4 V to 3.6 V).
- ✓ Four external interrupt inputs configurable as edge/level sensitive. All pins on Port 0 and Port 2 can be used as edge sensitive interrupt sources.
- ✓ Non-maskable Interrupt (NMI) input.
- ✓ Clock output function that can reflect the main oscillator clock, IRC clock, RTC clock, CPU clock, and the USB clock.
- ✓ The Wake-up Interrupt Controller (WIC) allows the CPU to automatically wake up from any priority interrupt that can occur while the clocks are stopped in deep sleep, Power-down, and Deep power-down modes.
- ✓ Processor wake-up from Power-down mode via any interrupt able to operate during Power-down mode (includes external interrupts, RTC interrupt, USB activity, Ethernet wake-up interrupt, CAN bus activity, Port 0/2 pin interrupt, and NMI).
- ✓ Brownout detect with separate threshold for interrupt and forced reset.
- ✓ Power-On Reset (POR).
- ✓ Crystal oscillator with an operating range of 1 MHz to 25 MHz.
- ✓ 4 MHz internal RC oscillator trimmed to 1 % accuracy that can optionally be used as a system clock.
- ✓ PLL allows CPU operation up to the maximum CPU rate without the need for a
- ✓ high-frequency crystal. May be run from the main oscillator, the internal RC oscillator, or the RTC oscillator.
- ✓ USB PLL for added flexibility.
- ✓ Code Read Protection (CRP) with different security levels.
- ✓ Unique device serial number for identification purposes.
- ✓ Available as LQFP100 (14 mm × 14 mm × 1.4 mm), TFBGA100 (9 mm × 9 mm × 0.7 mm), and WLCSP100 (5.07 × 5.07 × 0.53 mm) package.

Applications

- eMetering
- Lighting
- Industrial networking
- Alarm systems
- White goods
- Motor control



BLOCK DIAGRAM OF LPC176XX

PINNING INFORMATION

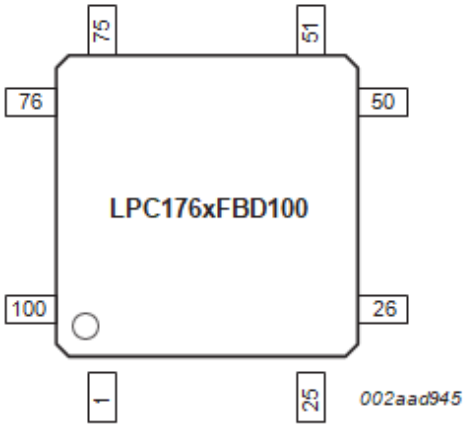
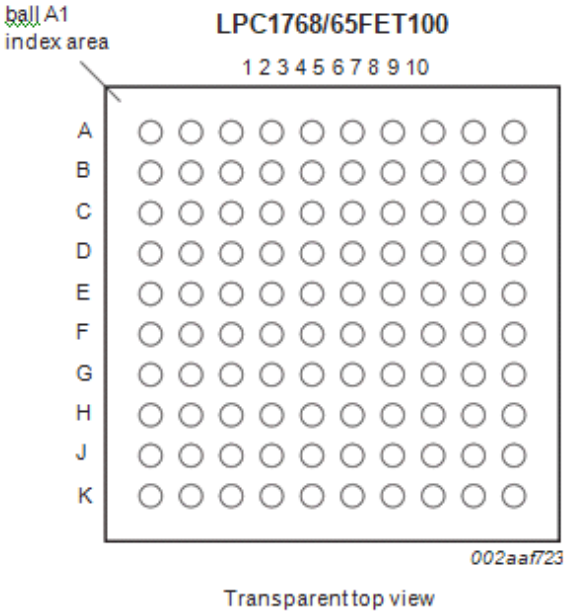
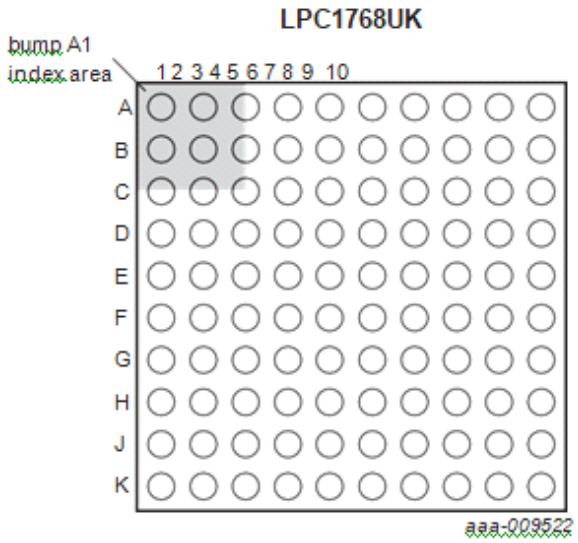


Fig 2. Pin configuration LQFP100 package



Transparent top view

Pin configuration TFBGA100 package



Transparent top view

Pin configuration WLCSP100 package

Table 4. Pin allocation table TFBGA100

Pin	Symbol	Pin	Symbol	Pin	Symbol	Pin	Symbol
Row A							
1	TDO/SWO	2	P0[3]/RXD0/AD0[6]	3	$V_{DD(3V3)}$	4	P1[4]/ENET_TX_EN
5	P1[10]/ENET_RXD1	6	P1[16]/ENET_MDC	7	$V_{DD(REG)(3V3)}$	8	P0[4]/I2SRX_CLK/ RD2/CAP2[0]
9	P0[7]/I2STX_CLK/ SCK1/MAT2[1]	10	P0[9]/I2STX_SDA/ MOSI1/MAT2[3]	11	-	12	-
Row B							
1	TMS/SWDIO	2	RTCK	3	V_{SS}	4	P1[1]/ENET_TXD1
5	P1[9]/ENET_RXD0	6	P1[17]/ ENET_MDIO	7	V_{SS}	8	P0[6]/I2SRX_SDA/ SSEL1/MAT2[0]
9	P2[0]/PWM1[1]/TXD1	10	P2[1]/PWM1[2]/RXD1	11	-	12	-
Row C							
1	TCK/SWDCLK	2	$\overline{TRST}$	3	TDI	4	P0[2]/TXD0/AD0[7]
5	P1[8]/ENET_CRS	6	P1[15]/ ENET_REF_CLK	7	P4[28]/RX_MCLK/ MAT2[0]/TXD3	8	P0[8]/I2STX_WS/ MISO1/MAT2[2]
9	V_{SS}	10	$V_{DD(3V3)}$	11	-	12	-
Row D							
1	P0[24]/AD0[1]/ I2SRX_WS/CAP3[1]	2	P0[25]/AD0[2]/ I2SRX_SDA/TXD3	3	P0[26]/AD0[3]/ AOUT/RXD3	4	n.c.
5	P1[0]/ENET_TXD0	6	P1[14]/ENET_RX_ER	7	P0[5]/I2SRX_WS/ TD2/CAP2[1]	8	P2[2]/PWM1[3]/ CTS1/TRACEDATA[3]
9	P2[4]/PWM1[5]/ DSR1/TRACEDATA[1]	10	P2[5]/PWM1[6]/ DTR1/TRACEDATA[0]	11	-	12	-
Row E							
1	V_{SSA}	2	V_{DDA}	3	VREFP	4	n.c.
5	P0[23]/AD0[0]/ I2SRX_CLK/CAP3[0]	6	P4[29]/TX_MCLK/ MAT2[1]/RXD3	7	P2[3]/PWM1[4]/ DCD1/TRACEDATA[2]	8	P2[6]/PCAP1[0]/ RI1/TRACECLK

Table 4. Pin allocation table TFBGA100 *...continued*

Pin	Symbol	Pin	Symbol	Pin	Symbol	Pin	Symbol
9	P2[7]/RD2/RTS1	10	P2[8]/TD2/TXD2	11	-	12	-
Row F							
1	VREFN	2	RTCX1	3	$\overline{\text{RESET}}$	4	P1[31]/SCK1/ AD0[5]
5	P1[21]/$\overline{\text{MCABORT}}$/ PWM1[3]/SSEL0	6	P0[18]/DCD1/ MOSI0/MOSI	7	P2[9]/USB_CONNECT/ RXD2	8	P0[16]/RXD1/ SSEL0/SSEL
9	P0[17]/CTS1/ MISO0/MISO	10	P0[15]/TXD1/ SCK0/SCK	11	-	12	-
Row G							
1	RTCX2	2	VBAT	3	XTAL2	4	P0[30]/USB_D–
5	P1[25]/MCOA1/ MAT1[1]	6	P1[29]/MCOB2/ PCAP1[1]/MAT0[1]	7	V_{SS}	8	P0[21]/RI1/RD1
9	P0[20]/DTR1/SCL1	10	P0[19]/DSR1/SDA1	11	-	12	-
Row H							
1	P1[30]/V_{BUS}/ AD0[4]	2	XTAL1	3	P3[25]/MAT0[0]/ PWM1[2]	4	P1[18]/USB_UP_LED/ PWM1[1]/CAP1[0]
5	P1[24]/MCI2/ PWM1[5]/MOSI0	6	$V_{DD(REG)(3V3)}$	7	P0[10]/TXD2/ SDA2/MAT3[0]	8	P2[11]/$\overline{\text{EINT1}}$/ I2STX_CLK
9	$V_{DD(3V3)}$	10	P0[22]/RTS1/TD1	11	-	12	-

Table 4. Pin allocation table TFBGA100 *...continued*

Pin	Symbol	Pin	Symbol	Pin	Symbol	Pin	Symbol
Row J							
1	P0[28]/SCL0/ USB_SCL	2	P0[27]/SDA0/ USB_SDA	3	P0[29]/USB_D+	4	P1[19]/MCOA0/ USB_PPWR/ CAP1[1]
5	P1[22]/MCOB0/ USB_PWRD/ MAT1[0]	6	V_{SS}	7	P1[28]/MCOA2/ PCAP1[0]/ MAT0[0]	8	P0[1]/TD1/RXD3/SCL1
9	P2[13]/$\overline{\text{EINT3}}$/ I2STX_SDA	10	P2[10]/$\overline{\text{EINT0}}$/NMI	11	-	12	-
Row K							
1	P3[26]/STCLK/ MAT0[1]/PWM1[3]	2	$V_{DD(3V3)}$	3	V_{SS}	4	P1[20]/MCI0/ PWM1[2]/SCK0
5	P1[23]/MCI1/ PWM1[4]/MISO0	6	P1[26]/MCOB1/ PWM1[6]/CAP0[0]	7	P1[27]/CLKOUT /USB_OVRCR/ CAP0[1]	8	P0[0]/RD1/TXD3/SDA1
9	P0[11]/RXD2/ SCL2/MAT3[1]	10	P2[12]/$\overline{\text{EINT2}}$/ I2STX_WS	11	-	12	-

Pin description

Symbol	Pin/ball				Type	Description
	LQFP100	TFBGA100	WLCSP100			
P0[0] to P0[31]					I/O	**Port 0:** Port 0 is a 32-bit I/O port with individual direction controls for each bit. The operation of port 0 pins depends upon the pin function selected via the pin connect block. Pins 12, 13, 14, and 31 of this port are not available.
P0[0]/RD1/TXD3/SDA1	46	K8	H10	[1]	I/O	**P0[0]** — General purpose digital input/output pin.
					I	**RD1** — CAN1 receiver input. (LPC1769/68/66/65/64 only).
					O	**TXD3** — Transmitter output for UART3.
					I/O	**SDA1** — I^2C1 data input/output. (This is not an I^2C-bus compliant open-drain pin).
P0[1]/TD1/RXD3/SCL1	47	J8	H9	[1]	I/O	**P0[1]** — General purpose digital input/output pin.
					O	**TD1** — CAN1 transmitter output. (LPC1769/68/66/65/64 only).
					I	**RXD3** — Receiver input for UART3.
					I/O	**SCL1** — I^2C1 clock input/output. (This is not an I^2C-bus compliant open-drain pin).
P0[2]/TXD0/AD0[7]	98	C4	B1	[2]	I/O	**P0[2]** — General purpose digital input/output pin.
					O	**TXD0** — Transmitter output for UART0.
					I	**AD0[7]** — A/D converter 0, input 7.
P0[3]/RXD0/AD0[6]	99	A2	C3	[2]	I/O	**P0[3]** — General purpose digital input/output pin.
					I	**RXD0** — Receiver input for UART0.
					I	**AD0[6]** — A/D converter 0, input 6.

Symbol	Pin/ball				Type	Description
	LQFP100	TFBGA100	WLCSP100			
P0[4]/ I2SRX_CLK/ RD2/CAP2[0]	81	A8	G2	[1]	I/O	**P0[4]** — General purpose digital input/output pin.
					I/O	**I2SRX_CLK** — Receive Clock. It is driven by the master and received by the slave. Corresponds to the signal SCK in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
					I	**RD2** — CAN2 receiver input. (LPC1769/68/66/65/64 only).
					I	**CAP2[0]** — Capture input for Timer 2, channel 0.
P0[5]/ I2SRX_WS/ TD2/CAP2[1]	80	D7	H1	[1]	I/O	**P0[5]** — General purpose digital input/output pin.
					I/O	**I2SRX_WS** — Receive Word Select. It is driven by the master and received by the slave. Corresponds to the signal WS in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
					O	**TD2** — CAN2 transmitter output. (LPC1769/68/66/65/64 only).
					I	**CAP2[1]** — Capture input for Timer 2, channel 1.
P0[6]/ I2SRX_SDA/ SSEL1/MAT2[0]	79	B8	G3	[1]	I/O	**P0[6]** — General purpose digital input/output pin.
					I/O	**I2SRX_SDA** — Receive data. It is driven by the transmitter and read by the receiver. Corresponds to the signal SD in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
					I/O	**SSEL1** — Slave Select for SSP1.
					O	**MAT2[0]** — Match output for Timer 2, channel 0.
P0[7]/ I2STX_CLK/ SCK1/MAT2[1]	78	A9	J1	[1]	I/O	**P0[7]** — General purpose digital input/output pin.
					I/O	**I2STX_CLK** — Transmit Clock. It is driven by the master and received by the slave. Corresponds to the signal SCK in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
					I/O	**SCK1** — Serial Clock for SSP1.
					O	**MAT2[1]** — Match output for Timer 2, channel 1.
P0[8]/ I2STX_WS/ MISO1/MAT2[2]	77	C8	H2	[1]	I/O	**P0[8]** — General purpose digital input/output pin.
					I/O	**I2STX_WS** — Transmit Word Select. It is driven by the master and received by the slave. Corresponds to the signal WS in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
					I/O	**MISO1** — Master In Slave Out for SSP1.
					O	**MAT2[2]** — Match output for Timer 2, channel 2.
P0[9]/ I2STX_SDA/ MOSI1/MAT2[3]	76	A10	H3	[1]	I/O	**P0[9]** — General purpose digital input/output pin.
					I/O	**I2STX_SDA** — Transmit data. It is driven by the transmitter and read by the receiver. Corresponds to the signal SD in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
					I/O	**MOSI1** — Master Out Slave In for SSP1.
					O	**MAT2[3]** — Match output for Timer 2, channel 3.
P0[10]/TXD2/ SDA2/MAT3[0]	48	H7	H8	[1]	I/O	**P0[10]** — General purpose digital input/output pin.
					O	**TXD2** — Transmitter output for UART2.
					I/O	**SDA2** — I²C2 data input/output (this is not an open-drain pin).
					O	**MAT3[0]** — Match output for Timer 3, channel 0.

Symbol	Pin/ball				Type	Description
	LQFP100	TFBGA100	WLCSP100			
P0[11]/RXD2/ SCL2/MAT3[1]	49	K9	J10	[1]	I/O	**P0[11]** — General purpose digital input/output pin.
					I	**RXD2** — Receiver input for UART2.
					I/O	**SCL2** — I²C2 clock input/output (this is not an open-drain pin).
					O	**MAT3[1]** — Match output for Timer 3, channel 1.
P0[15]/TXD1/ SCK0/SCK	62	F10	H6	[1]	I/O	**P0[15]** — General purpose digital input/output pin.
					O	**TXD1** — Transmitter output for UART1.
					I/O	**SCK0** — Serial clock for SSP0.
					I/O	**SCK** — Serial clock for SPI.
P0[16]/RXD1/ SSEL0/SSEL	63	F8	J5	[1]	I/O	**P0[16]** — General purpose digital input/output pin.
					I	**RXD1** — Receiver input for UART1.
					I/O	**SSEL0** — Slave Select for SSP0.
					I/O	**SSEL** — Slave Select for SPI.
P0[17]/CTS1/ MISO0/MISO	61	F9	K6	[1]	I/O	**P0[17]** — General purpose digital input/output pin.
					I	**CTS1** — Clear to Send input for UART1.
					I/O	**MISO0** — Master In Slave Out for SSP0.
					I/O	**MISO** — Master In Slave Out for SPI.
P0[18]/DCD1/ MOSI0/MOSI	60	F6	J6	[1]	I/O	**P0[18]** — General purpose digital input/output pin.
					I	**DCD1** — Data Carrier Detect input for UART1.
					I/O	**MOSI0** — Master Out Slave In for SSP0.
					I/O	**MOSI** — Master Out Slave In for SPI.
P0[19]/DSR1/ SDA1	59	G10	K7	[1]	I/O	**P0[19]** — General purpose digital input/output pin.
					I	**DSR1** — Data Set Ready input for UART1.
					I/O	**SDA1** — I²C1 data input/output (this is not an I²C-bus compliant open-drain pin).
P0[20]/DTR1/SCL1	58	G9	J7	[1]	I/O	**P0[20]** — General purpose digital input/output pin.
					O	**DTR1** — Data Terminal Ready output for UART1. Can also be configured to be an RS-485/EIA-485 output enable signal.
					I/O	**SCL1** — I²C1 clock input/output (this is not an I²C-bus compliant open-drain pin).
P0[21]/RI1/RD1	57	G8	H7	[1]	I/O	**P0[21]** — General purpose digital input/output pin.
					I	**RI1** — Ring Indicator input for UART1.
					I	**RD1** — CAN1 receiver input. (LPC1769/68/66/65/64 only).
P0[22]/RTS1/TD1	56	H10	K8	[1]	I/O	**P0[22]** — General purpose digital input/output pin.
					O	**RTS1** — Request to Send output for UART1. Can also be configured to be an RS-485/EIA-485 output enable signal.
					O	**TD1** — CAN1 transmitter output. (LPC1769/68/66/65/64 only).

Symbol	Pin/ball				Type	Description
	LQFP100	TFBGA100	WLCSP100			
P0[23]/AD0[0]/ I2SRX_CLK/ CAP3[0]	9	E5	D5	[2]	I/O	**P0[23]** — General purpose digital input/output pin.
					I	**AD0[0]** — A/D converter 0, input 0.
					I/O	**I2SRX_CLK** — Receive Clock. It is driven by the master and received by the slave. Corresponds to the signal SCK in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
					I	**CAP3[0]** — Capture input for Timer 3, channel 0.
P0[24]/AD0[1]/ I2SRX_WS/ CAP3[1]	8	D1	B4	[2]	I/O	**P0[24]** — General purpose digital input/output pin.
					I	**AD0[1]** — A/D converter 0, input 1.
					I/O	**I2SRX_WS** — Receive Word Select. It is driven by the master and received by the slave. Corresponds to the signal WS in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
					I	**CAP3[1]** — Capture input for Timer 3, channel 1.
P0[25]/AD0[2]/ I2SRX_SDA/ TXD3	7	D2	A3	[2]	I/O	**P0[25]** — General purpose digital input/output pin.
					I	**AD0[2]** — A/D converter 0, input 2.
					I/O	**I2SRX_SDA** — Receive data. It is driven by the transmitter and read by the receiver. Corresponds to the signal SD in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
					O	**TXD3** — Transmitter output for UART3.
P0[26]/AD0[3]/ AOUT/RXD3	6	D3	C5	[3]	I/O	**P0[26]** — General purpose digital input/output pin.
					I	**AD0[3]** — A/D converter 0, input 3.
					O	**AOUT** — DAC output (LPC1769/68/67/66/65/63 only).
					I	**RXD3** — Receiver input for UART3.
P0[27]/SDA0/ USB_SDA	25	J2	C8	[4]	I/O	**P0[27]** — General purpose digital input/output pin. Output is open-drain.
					I/O	**SDA0** — I²C0 data input/output. Open-drain output (for I²C-bus compliance).
					I/O	**USB_SDA** — USB port I²C serial data (OTG transceiver, LPC1769/68/66/65 only).
P0[28]/SCL0/ USB_SCL	24	J1	B9	[4]	I/O	**P0[28]** — General purpose digital input/output pin. Output is open-drain.
					I/O	**SCL0** — I²C0 clock input/output. Open-drain output (for I²C-bus compliance).
					I/O	**USB_SCL** — USB port I²C serial clock (OTG transceiver, LPC1769/68/66/65 only).
P0[29]/USB_D+	29	J3	B10	[5]	I/O	**P0[29]** — General purpose digital input/output pin.
					I/O	**USB_D+** — USB bidirectional D+ line. (LPC1769/68/66/65/64 only).
P0[30]/USB_D−	30	G4	C9	[5]	I/O	**P0[30]** — General purpose digital input/output pin.
					I/O	**USB_D−** — USB bidirectional D− line. (LPC1769/68/66/65/64 only).

Symbol	Pin/ball				Type	Description
	LQFP100	TFBGA100	WLCSP100			
P1[0] to P1[31]					I/O	**Port 1:** Port 1 is a 32-bit I/O port with individual direction controls for each bit. The operation of port 1 pins depends upon the pin function selected via the pin connect block. Pins 2, 3, 5, 6, 7, 11, 12, and 13 of this port are not available.
P1[0]/ ENET_TXD0	95	D5	C1	[1]	I/O	**P1[0]** — General purpose digital input/output pin.
					O	**ENET_TXD0** — Ethernet transmit data 0. (LPC1769/68/67/66/64 only).
P1[1]/ ENET_TXD1	94	B4	C2	[1]	I/O	**P1[1]** — General purpose digital input/output pin.
					O	**ENET_TXD1** — Ethernet transmit data 1. (LPC1769/68/67/66/64 only).
P1[4]/ ENET_TX_EN	93	A4	D2	[1]	I/O	**P1[4]** — General purpose digital input/output pin.
					O	**ENET_TX_EN** — Ethernet transmit data enable. (LPC1769/68/67/66/64 only).
P1[8]/ ENET_CRS	92	C5	D1	[1]	I/O	**P1[8]** — General purpose digital input/output pin.
					I	**ENET_CRS** — Ethernet carrier sense. (LPC1769/68/67/66/64 only).
P1[9]/ ENET_RXD0	91	B5	D3	[1]	I/O	**P1[9]** — General purpose digital input/output pin.
					I	**ENET_RXD0** — Ethernet receive data. (LPC1769/68/67/66/64 only).
P1[10]/ ENET_RXD1	90	A5	E3	[1]	I/O	**P1[10]** — General purpose digital input/output pin.
					I	**ENET_RXD1** — Ethernet receive data. (LPC1769/68/67/66/64 only).
P1[14]/ ENET_RX_ER	89	D6	E2	[1]	I/O	**P1[14]** — General purpose digital input/output pin.
					I	**ENET_RX_ER** — Ethernet receive error. (LPC1769/68/67/66/64 only).
P1[15]/ ENET_REF_CLK	88	C6	E1	[1]	I/O	**P1[15]** — General purpose digital input/output pin.
					I	**ENET_REF_CLK** — Ethernet reference clock. (LPC1769/68/67/66/64 only).
P1[16]/ ENET_MDC	87	A6	F3	[1]	I/O	**P1[16]** — General purpose digital input/output pin.
					O	**ENET_MDC** — Ethernet MIIM clock (LPC1769/68/67/66/64 only).
P1[17]/ ENET_MDIO	86	B6	F2	[1]	I/O	**P1[17]** — General purpose digital input/output pin.
					I/O	**ENET_MDIO** — Ethernet MIIM data input and output. (LPC1769/68/67/66/64 only).

Symbol	Pin/ball				Type	Description
	LQFP100	TFBGA100	WLCSP100			
P1[18]/ USB_UP_LED/ PWM1[1]/ CAP1[0]	32	H4	D9	[1]	I/O	**P1[18]** — General purpose digital input/output pin.
					O	**USB_UP_LED** — USB GoodLink LED indicator. It is LOW when the device is configured (non-control endpoints enabled), or when the host is enabled and has detected a device on the bus. It is HIGH when the device is not configured, or when host is enabled and has not detected a device on the bus, or during global suspend. It transitions between LOW and HIGH (flashes) when the host is enabled and detects activity on the bus. (LPC1769/68/66/65/64 only).
					O	**PWM1[1]** — Pulse Width Modulator 1, channel 1 output.
					I	**CAP1[0]** — Capture input for Timer 1, channel 0.
P1[19]/MCOA0/ USB_PPWR/ CAP1[1]	33	J4	C10	[1]	I/O	**P1[19]** — General purpose digital input/output pin.
					O	**MCOA0** — Motor control PWM channel 0, output A.
					O	**USB_PPWR** — Port Power enable signal for USB port. (LPC1769/68/66/65 only).
					I	**CAP1[1]** — Capture input for Timer 1, channel 1.
P1[20]/MCI0/ PWM1[2]/SCK0	34	K4	E8	[1]	I/O	**P1[20]** — General purpose digital input/output pin.
					I	**MCI0** — Motor control PWM channel 0, input. Also Quadrature Encoder Interface PHA input.
					O	**PWM1[2]** — Pulse Width Modulator 1, channel 2 output.
					I/O	**SCK0** — Serial clock for SSP0.
P1[21]/MCABORT/ PWM1[3]/ SSEL0	35	F5	E9	[1]	I/O	**P1[21]** — General purpose digital input/output pin.
					O	**MCABORT** — Motor control PWM, LOW-active fast abort.
					O	**PWM1[3]** — Pulse Width Modulator 1, channel 3 output.
					I/O	**SSEL0** — Slave Select for SSP0.
P1[22]/MCOB0/ USB_PWRD/ MAT1[0]	36	J5	D10	[1]	I/O	**P1[22]** — General purpose digital input/output pin.
					O	**MCOB0** — Motor control PWM channel 0, output B.
					I	**USB_PWRD** — Power Status for USB port (host power switch, LPC1769/68/66/65 only).
					O	**MAT1[0]** — Match output for Timer 1, channel 0.
P1[23]/MCI1/ PWM1[4]/MISO0	37	K5	E7	[1]	I/O	**P1[23]** — General purpose digital input/output pin.
					I	**MCI1** — Motor control PWM channel 1, input. Also Quadrature Encoder Interface PHB input.
					O	**PWM1[4]** — Pulse Width Modulator 1, channel 4 output.
					I/O	**MISO0** — Master In Slave Out for SSP0.
P1[24]/MCI2/ PWM1[5]/MOSI0	38	H5	F8	[1]	I/O	**P1[24]** — General purpose digital input/output pin.
					I	**MCI2** — Motor control PWM channel 2, input. Also Quadrature Encoder Interface INDEX input.
					O	**PWM1[5]** — Pulse Width Modulator 1, channel 5 output.
					I/O	**MOSI0** — Master Out Slave in for SSP0.

Symbol	Pin/ball				Type	Description
	LQFP100	TFBGA100	WLCSP100			
P1[25]/MCOA1/ MAT1[1]	39	G5	F9	[1]	I/O	**P1[25]** — General purpose digital input/output pin.
					O	**MCOA1** — Motor control PWM channel 1, output A.
					O	**MAT1[1]** — Match output for Timer 1, channel 1.
P1[26]/MCOB1/ PWM1[6]/CAP0[0]	40	K6	E10	[1]	I/O	**P1[26]** — General purpose digital input/output pin.
					O	**MCOB1** — Motor control PWM channel 1, output B.
					O	**PWM1[6]** — Pulse Width Modulator 1, channel 6 output.
					I	**CAP0[0]** — Capture input for Timer 0, channel 0.
P1[27]/CLKOUT /USB_OVRCR/ CAP0[1]	43	K7	G9	[1]	I/O	**P1[27]** — General purpose digital input/output pin.
					O	**CLKOUT** — Clock output pin.
					I	**USB_OVRCR** — USB port Over-Current status. (LPC1769/68/66/65 only).
					I	**CAP0[1]** — Capture input for Timer 0, channel 1.
P1[28]/MCOA2/ PCAP1[0]/ MAT0[0]	44	J7	G10	[1]	I/O	**P1[28]** — General purpose digital input/output pin.
					O	**MCOA2** — Motor control PWM channel 2, output A.
					I	**PCAP1[0]** — Capture input for PWM1, channel 0.
					O	**MAT0[0]** — Match output for Timer 0, channel 0.
P1[29]/MCOB2/ PCAP1[1]/ MAT0[1]	45	G6	G8	[1]	I/O	**P1[29]** — General purpose digital input/output pin.
					O	**MCOB2** — Motor control PWM channel 2, output B.
					I	**PCAP1[1]** — Capture input for PWM1, channel 1.
					O	**MAT0[1]** — Match output for Timer 0, channel 1.
P1[30]/V_{BUS}/ AD0[4]	21	H1	B8	[2]	I/O	**P1[30]** — General purpose digital input/output pin.
					I	**V_{BUS}** — Monitors the presence of USB bus power. (LPC1769/68/66/65/64 only). **Note:** This signal must be HIGH for USB reset to occur.
					I	**AD0[4]** — A/D converter 0, input 4.
P1[31]/SCK1/ AD0[5]	20	F4	C7	[2]	I/O	**P1[31]** — General purpose digital input/output pin.
					I/O	**SCK1** — Serial Clock for SSP1.
					I	**AD0[5]** — A/D converter 0, input 5.
P2[0] to P2[31]					I/O	**Port 2:** Port 2 is a 32-bit I/O port with individual direction controls for each bit. The operation of port 2 pins depends upon the pin function selected via the pin connect block. Pins 14 through 31 of this port are not available.
P2[0]/PWM1[1]/ TXD1	75	B9	K1	[1]	I/O	**P2[0]** — General purpose digital input/output pin.
					O	**PWM1[1]** — Pulse Width Modulator 1, channel 1 output.
					O	**TXD1** — Transmitter output for UART1.
P2[1]/PWM1[2]/ RXD1	74	B10	J2	[1]	I/O	**P2[1]** — General purpose digital input/output pin.
					O	**PWM1[2]** — Pulse Width Modulator 1, channel 2 output.
					I	**RXD1** — Receiver input for UART1.

Symbol	Pin/ball				Type	Description
	LQFP100	TFBGA100	WLCSP100			
P2[2]/PWM1[3]/ CTS1/ TRACEDATA[3]	73	D8	K2	[1]	I/O	**P2[2]** — General purpose digital input/output pin.
					O	**PWM1[3]** — Pulse Width Modulator 1, channel 3 output.
					I	**CTS1** — Clear to Send input for UART1.
					O	**TRACEDATA[3]** — Trace data, bit 3.
P2[3]/PWM1[4]/ DCD1/ TRACEDATA[2]	70	E7	K3	[1]	I/O	**P2[3]** — General purpose digital input/output pin.
					O	**PWM1[4]** — Pulse Width Modulator 1, channel 4 output.
					I	**DCD1** — Data Carrier Detect input for UART1.
					O	**TRACEDATA[2]** — Trace data, bit 2.
P2[4]/PWM1[5]/ DSR1/ TRACEDATA[1]	69	D9	J3	[1]	I/O	**P2[4]** — General purpose digital input/output pin.
					O	**PWM1[5]** — Pulse Width Modulator 1, channel 5 output.
					I	**DSR1** — Data Set Ready input for UART1.
					O	**TRACEDATA[1]** — Trace data, bit 1.
P2[5]/PWM1[6]/ DTR1/ TRACEDATA[0]	68	D10	H4	[1]	I/O	**P2[5]** — General purpose digital input/output pin.
					O	**PWM1[6]** — Pulse Width Modulator 1, channel 6 output.
					O	**DTR1** — Data Terminal Ready output for UART1. Can also be configured to be an RS-485/EIA-485 output enable signal.
					O	**TRACEDATA[0]** — Trace data, bit 0.
P2[6]/PCAP1[0]/ RI1/TRACECLK	67	E8	K4	[1]	I/O	**P2[6]** — General purpose digital input/output pin.
					I	**PCAP1[0]** — Capture input for PWM1, channel 0.
					I	**RI1** — Ring Indicator input for UART1.
					O	**TRACECLK** — Trace Clock.
P2[7]/RD2/ RTS1	66	E9	J4	[1]	I/O	**P2[7]** — General purpose digital input/output pin.
					I	**RD2** — CAN2 receiver input. (LPC1769/68/66/65/64 only).
					O	**RTS1** — Request to Send output for UART1. Can also be configured to be an RS-485/EIA-485 output enable signal.
P2[8]/TD2/ TXD2	65	E10	H5	[1]	I/O	**P2[8]** — General purpose digital input/output pin.
					O	**TD2** — CAN2 transmitter output. (LPC1769/68/66/65/64 only).
					O	**TXD2** — Transmitter output for UART2.
P2[9]/ USB_CONNECT/ RXD2	64	F7	K5	[1]	I/O	**P2[9]** — General purpose digital input/output pin.
					O	**USB_CONNECT** — Signal used to switch an external 1.5 kΩ resistor under software control. Used with the SoftConnect USB feature. (LPC1769/68/66/65/64 only).
					I	**RXD2** — Receiver input for UART2.
P2[10]/EINT0/NMI	53	J10	K9	[8]	I/O	**P2[10]** — General purpose digital input/output pin. A LOW level on this pin during reset starts the ISP command handler.
					I	**EINT0** — External interrupt 0 input.
					I	**NMI** — Non-maskable interrupt input.

Symbol	Pin/ball				Type	Description
	LQFP100	TFBGA100	WLCSP100			
P2[11]/EINT1/ I2STX_CLK	52	H8	J8	[5]	I/O	**P2[11]** — General purpose digital input/output pin.
					I	**EINT1** — External interrupt 1 input.
					I/O	**I2STX_CLK** — Transmit Clock. It is driven by the master and received by the slave. Corresponds to the signal SCK in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
P2[12]/EINT2/ I2STX_WS	51	K10	K10	[5]	I/O	**P2[12]** — General purpose digital input/output pin.
					I	**EINT2** — External interrupt 2 input.
					I/O	**I2STX_WS** — Transmit Word Select. It is driven by the master and received by the slave. Corresponds to the signal WS in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
P2[13]/EINT3/ I2STX_SDA	50	J9	J9	[5]	I/O	**P2[13]** — General purpose digital input/output pin.
					I	**EINT3** — External interrupt 3 input.
					I/O	**I2STX_SDA** — Transmit data. It is driven by the transmitter and read by the receiver. Corresponds to the signal SD in the *I²S-bus specification*. (LPC1769/68/67/66/65/63 only).
P3[0] to P3[31]					I/O	**Port 3:** Port 3 is a 32-bit I/O port with individual direction controls for each bit. The operation of port 3 pins depends upon the pin function selected via the pin connect block. Pins 0 through 24, and 27 through 31 of this port are not available.
P3[25]/MAT0[0]/ PWM1[2]	27	H3	D8	[1]	I/O	**P3[25]** — General purpose digital input/output pin.
					O	**MAT0[0]** — Match output for Timer 0, channel 0.
					O	**PWM1[2]** — Pulse Width Modulator 1, output 2.
P3[26]/STCLK/ MAT0[1]/PWM1[3]	26	K1	A10	[1]	I/O	**P3[26]** — General purpose digital input/output pin.
					I	**STCLK** — System tick timer clock input. The maximum STCLK frequency is 1/4 of the Arm processor clock frequency CCLK.
					O	**MAT0[1]** — Match output for Timer 0, channel 1.
					O	**PWM1[3]** — Pulse Width Modulator 1, output 3.
P4[0] to P4[31]					I/O	**Port 4:** Port 4 is a 32-bit I/O port with individual direction controls for each bit. The operation of port 4 pins depends upon the pin function selected via the pin connect block. Pins 0 through 27, 30, and 31 of this port are not available.
P4[28]/RX_MCLK/ MAT2[0]/TXD3	82	C7	G1	[1]	I/O	**P4[28]** — General purpose digital input/output pin.
					O	**RX_MCLK** — I²S receive master clock. (LPC1769/68/67/66/65 only).
					O	**MAT2[0]** — Match output for Timer 2, channel 0.
					O	**TXD3** — Transmitter output for UART3.
P4[29]/TX_MCLK/ MAT2[1]/RXD3	85	E6	F1	[1]	I/O	**P4[29]** — General purpose digital input/output pin.
					O	**TX_MCLK** — I²S transmit master clock. (LPC1769/68/67/66/65 only).
					O	**MAT2[1]** — Match output for Timer 2, channel 1.
					I	**RXD3** — Receiver input for UART3.

Symbol	Pin/ball				Type	Description
	LQFP100	TFBGA100	WLCSP100			
TDO/SWO	1	A1	A1	[7]	O	**TDO** — Test Data out for JTAG interface.
					O	**SWO** — Serial wire trace output.
TDI	2	C3	C4	[1][8]	I	**TDI** — Test Data in for JTAG interface.
TMS/SWDIO	3	B1	B3	[1][8]	I	**TMS** — Test Mode Select for JTAG interface.
					I/O	**SWDIO** — Serial wire debug data input/output.
TRST	4	C2	A2	[1][8]	I	**TRST** — Test Reset for JTAG interface.
TCK/SWDCLK	5	C1	D4	[7]	I	**TCK** — Test Clock for JTAG interface.
					I	**SWDCLK** — Serial wire clock.
RTCK	100	B2	B2	[7]	O	**RTCK** — JTAG interface control signal.
RSTOUT	14	-	-	-	O	**RSTOUT** — This is a 3.3 V pin. LOW on this pin indicates the microcontroller being in Reset state.
RESET	17	F3	C6	[9]	I	**External reset input:** A LOW-going pulse as short as 50 ns on this pin resets the device, causing I/O ports and peripherals to take on their default states, and processor execution to begin at address 0. TTL with hysteresis, 5 V tolerant.
XTAL1	22	H2	D7	[10][11]	I	Input to the oscillator circuit and internal clock generator circuits.
XTAL2	23	G3	A9	[10][11]	O	Output from the oscillator amplifier.
RTCX1	16	F2	A7	[10][11]	I	Input to the RTC oscillator circuit.
RTCX2	18	G1	B7	[10]	O	Output from the RTC oscillator circuit.
V_{SS}	31, 41, 55, 72, 83, 97	B3, B7, C9, G7, J6, K3	E5, F5, F6, G5, G6, G7	[10]	I	**ground:** 0 V reference.
V_{SSA}	11	E1	B5	[10]	I	**analog ground:** 0 V reference. This should nominally be the same voltage as V_{SS}, but should be isolated to minimize noise and error.
$V_{DD(3V3)}$	28, 54, 71, 96	K2, H9, C10, A3	E4, E6, F7, G4	[10]	I	**3.3 V supply voltage:** This is the power supply voltage for the I/O ports.
$V_{DD(REG)(3V3)}$	42, 84	H6, A7	F4, F10	[10]	I	**3.3 V voltage regulator supply voltage:** This is the supply voltage for the on-chip voltage regulator only.
V_{DDA}	10	E2	A4	[10]	I	**analog 3.3 V pad supply voltage:** This should be nominally the same voltage as $V_{DD(3V3)}$ but should be isolated to minimize noise and error. This voltage is used to power the ADC and DAC. This pin should be tied to 3.3 V if the ADC and DAC are not used.
VREFP	12	E3	A5	[10]	I	**ADC positive reference voltage:** This should be nominally the same voltage as V_{DDA} but should be isolated to minimize noise and error. Level on this pin is used as a reference for ADC and DAC. This pin should be tied to 3.3 V if the ADC and DAC are not used.

Symbol	Pin/ball				Type	Description
	LQFP100	TFBGA100	WLCSP100			
VREFN	15	F1	A6		I	**ADC negative reference voltage:** This should be nominally the same voltage as V_{SS} but should be isolated to minimize noise and error. Level on this pin is used as a reference for ADC and DAC.
VBAT	19	G2	A8	[10][12]	I	**RTC pin power supply:** 3.3 V on this pin supplies the power to the RTC peripheral.
n.c.	13	D4, E4	B6, D6		-	not connected.

- 5 V tolerant pad providing digital I/O functions with TTL levels and hysteresis. This pin is pulled up to a voltage level of 2.3 V to 2.6 V.
- 5 V tolerant pad providing digital I/O functions (with TTL levels and hysteresis) and analog input. When configured as a ADC input, digital section of the pad is disabled and the pin is not 5 V tolerant. This pin is pulled up to a voltage level of 2.3 V to 2.6 V.
- 5 V tolerant pad providing digital I/O with TTL levels and hysteresis and analog output function. When configured as the DAC output, digital section of the pad is disabled. This pin is pulled up to a voltage level of 2.3 V to 2.6 V.
- Open-drain 5 V tolerant digital I/O pad, compatible with I^2C-bus 400 kHz specification. This pad requires an external pull-up to provide output functionality. When power is switched off, this pin connected to the I2C-bus is floating and does not disturb the I2C lines. Open-drain configuration applies to all functions on this pin.
- Pad provides digital I/O and USB functions. It is designed in accordance with the *USB specification, revision 2.0* (Full-speed and Low-speed mode only). This pad is not 5 V tolerant.
- 5 V tolerant pad with 10 ns glitch filter providing digital I/O functions with TTL levels and hysteresis. This pin is pulled up to a voltage level of 2.3 V to 2.6 V.
- 5 V tolerant pad with TTL levels and hysteresis. Internal pull-up and pull-down resistors disabled.
- 5 V tolerant pad with TTL levels and hysteresis and internal pull-up resistor.
- 5 V tolerant pad with 20 ns glitch filter providing digital I/O function with TTL levels and hysteresis.
- Pad provides special analog functionality. A 32 kHz crystal oscillator must be used with the RTC.
- When the system oscillator is not used, connect XTAL1 and XTAL2 as follows: XTAL1 can be left floating or can be grounded (grounding is preferred to reduce susceptibility to noise). XTAL2 should be left floating.
- When the RTC is not used, connect VBAT to $V_{DD(REG)(3V3)}$ and leave RTCX1 floating.

Functional description

Architectural overview

Remark: In the following, the notation LPC17xx refers to all parts: LPC1769/68/67/66/65/64/63.

The Arm Cortex-M3 includes three AHB-Lite buses: the system bus, the I-code bus, and the D-code bus. The I-code and D-code core buses are faster than the system bus and are used similarly to TCM interfaces: one bus dedicated for instruction fetch (I-code) and one bus for data access (D-code). The use of two core buses allows for simultaneous operations if concurrent operations target different devices.

The LPC17xx use a multi-layer AHB matrix to connect the Arm Cortex-M3 buses and other bus masters to peripherals in a flexible manner that optimizes performance by allowing peripherals that are on different slaves ports of the matrix to be accessed simultaneously by different bus masters.

- **Arm Cortex-M3 processor**

The Arm Cortex-M3 is a general purpose, 32-bit microprocessor, which offers high performance and very low power consumption. The Arm Cortex-M3 offers many new features, including a Thumb-2 instruction set, low interrupt latency, hardware divide, interruptible/continuable multiple load and store instructions, automatic state save and restore for interrupts, tightly integrated interrupt controller with wake-up interrupt controller, and multiple core buses capable of simultaneous accesses.

Pipeline techniques are employed so that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory.

On-chip flash program memory

The LPC17xx contain up to 512 kB of on-chip flash memory. A new two-port flash accelerator maximizes performance for use with the two fast AHB-Lite buses.

On-chip SRAM

The LPC17xx contain a total of 64 kB on-chip static RAM memory. This includes the main 32 kB SRAM, accessible by the CPU and DMA controller on a higher-speed bus, and two additional 16 kB each SRAM blocks situated on a separate slave port on the AHB multilayer matrix.

This architecture allows CPU and DMA accesses to be spread over three separate RAMs that can be accessed simultaneously.

Memory Protection Unit (MPU)

The LPC17xx have a Memory Protection Unit (MPU) which can be used to improve the reliability of an embedded system by protecting critical data within the user application.

The MPU allows separating processing tasks by disallowing access to each other's data, disabling access to memory regions, allowing memory regions to be defined as read-only and detecting unexpected memory accesses that could potentially break the system.

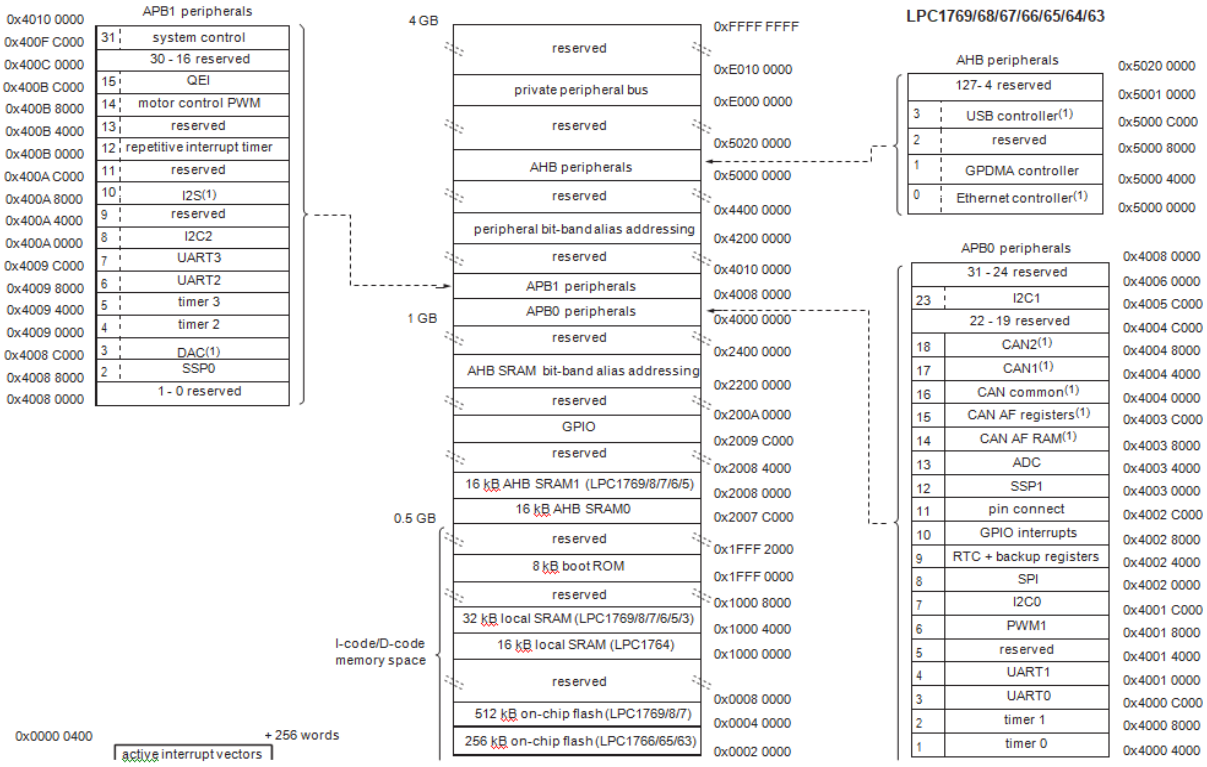
The MPU separates the memory into distinct regions and implements protection by preventing disallowed accesses. The MPU supports up to 8 regions each of which can be divided into 8 subregions. Accesses to memory locations that are not defined in the MPU regions, or not permitted by the region setting, will cause the Memory Management Fault exception to take place.

- **Memory map**

  The LPC17xx incorporates several distinct memory regions, shows the overall map of the entire address space from the user program viewpoint following reset. The interrupt vector area supports address remapping.

  The AHB peripheral area is 2 MB in size and is divided to allow for up to 128 peripherals. The APB peripheral area is 1 MB in size and is divided to allow for up to 64 peripherals. Each peripheral of either type is allocated 16 kB of space. This allows simplifying the address decoding for each peripheral.

LPC17xx memory map



Nested Vectored Interrupt Controller (NVIC)

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

Features

- Controls system exceptions and peripheral interrupts
- In the LPC17xx, the NVIC supports 33 vectored interrupts
- 32 programmable interrupt priority levels, with hardware priority level masking
- Relocatable vector table
- Non-Maskable Interrupt (NMI)
- Software interrupt generation

Interrupt sources

Each peripheral device has one interrupt line connected to the NVIC but may have several interrupt flags. Individual interrupt flags may also represent more than one interrupt source.

Any pin on Port 0 and Port 2 (total of 42 pins) regardless of the selected function, can be programmed to generate an interrupt on a rising edge, a falling edge, or both.

Pin connect block

The pin connect block allows selected pins of the microcontroller to have more than one function. Configuration registers control the multiplexers to allow connection between the pin and the on-chip peripherals.

Peripherals should be connected to the appropriate pins prior to being activated and prior to any related interrupt(s) being enabled. Activity of any enabled peripheral function that is not mapped to a related pin should be considered undefined.

Most pins can also be configured as open-drain outputs or to have a pull-up, pull-down, or no resistor enabled.

General purpose DMA controller

The GPDMA is an AMBA AHB compliant peripheral allowing selected peripherals to have DMA support.

The GPDMA enables peripheral-to-memory, memory-to-peripheral, peripheral-to-peripheral, and memory-to-memory transactions. The source and destination areas can each be either a memory region or a peripheral, and can be accessed through the AHB master. The GPDMA controller allows data transfers between the USB and Ethernet controllers and the various on-chip SRAM areas. The supported APB peripherals are SSP0/1, all UARTs, the I^2S-bus interface, the ADC, and the DAC. Two match signals for each timer can be used to trigger DMA transfers.

Remark: The Ethernet controller is available on parts LPC1769/68/67/66/64. The USB controller is available on parts LPC1769/68/66/65/64. The I^2S-bus interface is available on parts LPC1769/68/67/66/65. The DAC is available on parts LPC1769/68/67/66/65/63.

Features

- Eight DMA channels. Each channel can support an unidirectional transfer.
- 16 DMA request lines.
- Single DMA and burst DMA request signals. Each peripheral connected to the DMA Controller can assert either a burst DMA request or a single DMA request. The DMA burst size is set by programming the DMA Controller.
- Memory-to-memory, memory-to-peripheral, peripheral-to-memory, and peripheral-to-peripheral transfers are supported.
- Scatter or gather DMA is supported through the use of linked lists. This means that the source and destination areas do not have to occupy contiguous areas of memory.
- Hardware DMA channel priority.
- AHB slave DMA programming interface. The DMA Controller is programmed by writing to the DMA control registers over the AHB slave interface.
- One AHB bus master for transferring data. The interface transfers data when a DMA request goes active.
- 32-bit AHB master bus width.

- ❖ Incrementing or non-incrementing addressing for source and destination.
- ❖ Programmable DMA burst size. The DMA burst size can be programmed to more efficiently transfer data.
- ❖ Internal four-word FIFO per channel.
- ❖ Supports 8, 16, and 32-bit wide transactions.
- ❖ Big-endian and little-endian support. The DMA Controller defaults to little-endian mode on reset.
- ❖ An interrupt to the processor can be generated on a DMA completion or when a DMA error has occurred.
- ❖ Raw interrupt status. The DMA error and DMA count raw interrupt status can be read prior to masking.

Fast general purpose parallel I/O

Device pins that are not connected to a specific peripheral function are controlled by the GPIO registers. Pins may be dynamically configured as inputs or outputs. Separate registers allow setting or clearing any number of outputs simultaneously. The value of the output register may be read back as well as the current state of the port pins.

LPC17xx use accelerated GPIO functions:

- GPIO registers are accessed through the AHB multilayer bus so that the fastest possible I/O timing can be achieved.
- Mask registers allow treating sets of port bits as a group, leaving other bits unchanged.
- All GPIO registers are byte and half-word addressable.
- Entire port value can be written in one instruction.
- Support for Cortex-M3 bit banding.
- Support for use with the GPDMA controller.

Additionally, any pin on Port 0 and Port 2 (total of 42 pins) providing a digital function can be programmed to generate an interrupt on a rising edge, a falling edge, or both. The edge detection is asynchronous, so it may operate when clocks are not present such as during Power-down mode. Each enabled interrupt can be used to wake up the chip from Power-down mode.

Features

- ❖ Bit level set and clear registers allow a single instruction to set or clear any number of bits in one port.
- ❖ Direction control of individual bits.
- ❖ All I/O default to inputs after reset.

Pull-up/pull-down resistor configuration and open-drain configuration can be programmed through the pin connect block for each GPIO pin

Ethernet

Remark: The Ethernet controller is available on parts LPC1769/68/67/66/64. The Ethernet block supports bus clock rates of up to 100 MHz (LPC1768/67/66/64) or 120 MHz (LPC1769).

The Ethernet block contains a full featured 10 Mbit/s or 100 Mbit/s Ethernet MAC designed to

provide optimized performance through the use of DMA hardware acceleration. Features include a generous suite of control registers, half or full duplex operation, flow control, control frames, hardware acceleration for transmit retry, receive packet filtering and wake-up on LAN activity. Automatic frame transmission and reception with scatter-gather DMA off-loads many operations from the CPU.

The Ethernet block and the CPU share the Arm Cortex-M3 D-code and system bus through the AHB-multilayer matrix to access the various on-chip SRAM blocks for Ethernet data, control, and status information.

The Ethernet block interfaces between an off-chip Ethernet PHY using the Reduced MII (RMII) protocol and the on-chip Media Independent Interface Management (MIIM) serial bus.

Features

- Ethernet standards support:
- Supports 10 Mbit/s or 100 Mbit/s PHY devices including 10 Base-T, 100 Base-TX, 100 Base-FX, and 100 Base-T4.
- Fully compliant with *IEEE standard 802.3*.
- Fully compliant with 802.3x full duplex flow control and half duplex back pressure.
- Flexible transmit and receive frame options.
- Virtual Local Area Network (VLAN) frame support.
- Memory management:
- Independent transmit and receive buffers memory mapped to shared SRAM.
- DMA managers with scatter/gather DMA and arrays of frame descriptors.
- Memory traffic optimized by buffering and pre-fetching.
- Enhanced Ethernet features:
- Receive filtering.
- Multicast and broadcast frame support for both transmit and receive.
- Optional automatic Frame Check Sequence (FCS) insertion with Cyclic Redundancy Check (CRC) for transmit.
- Selectable automatic transmit frame padding.
- Over-length frame support for both transmit and receive allows any length frames.
- Promiscuous receive mode.
- Automatic collision back-off and frame retransmission.
- Includes power management by clock switching.
- Wake-on-LAN power management support allows system wake-up: using the receive filters or a magic frame detection filter.
- Physical interface:
- Attachment of external PHY chip through standard RMII interface.
- PHY register access is available via the MIIM interface.

USB host controller

The host controller enables full- and low-speed data exchange with USB devices attached to the

bus. It consists of a register interface, a serial interface engine, and a DMA controller. The register interface complies with the OHCI specification.

Features

- OHCI compliant.
- One downstream port.
- Supports port power switching.

USB OTG controller

USB OTG is a supplement to the USB 2.0 specification that augments the capability of existing mobile devices and USB peripherals by adding host functionality for connection to USB peripherals.

The OTG Controller integrates the host controller, device controller, and a master-only I^2C-bus interface to implement OTG dual-role device functionality. The dedicated I2C-bus interface controls an external OTG transceiver.

Features

- ✓ Fully compliant with *On-The-Go supplement to the USB 2.0 Specification, Revision 1.0a*.
- ✓ Hardware support for Host Negotiation Protocol (HNP).
- ✓ Includes a programmable timer required for HNP and Session Request Protocol (SRP).
- ✓ Supports any OTG transceiver compliant with the *OTG Transceiver Specification (CEA-2011), Rev. 1.0*.

CAN controller and acceptance filters

Remark: The CAN controllers are available on parts LPC1769/68/66/65/64.

The Controller Area Network (CAN) is a serial communications protocol which efficiently supports distributed real-time control with a very high level of security. Its domain of application ranges from high-speed networks to low cost multiplex wiring.

The CAN block is intended to support multiple CAN buses simultaneously, allowing the device to be used as a gateway, switch, or router among a number of CAN buses in industrial or automotive applications.

Features

- ✓ Two CAN controllers and buses.
- ✓ Data rates to 1 Mbit/s on each bus.
- ✓ 32-bit register and RAM access.
- ✓ Compatible with *CAN specification 2.0B, ISO 11898-1*.
- ✓ Global Acceptance Filter recognizes standard (11-bit) and extended-frame (29-bit) receive identifiers for all CAN buses.
- ✓ Acceptance Filter can provide FullCAN-style automatic reception for selected Standard Identifiers.
- ✓ FullCAN messages can generate interrupts.

GPIOs

GPIO in Cortex-M3 LPC1768 Microcontroller is the most basic peripheral. GPIO, General Purpose Input Output is what let's your microcontroller be something more than a weak auxiliary processor. With it you can interact with physical world, connecting up other devices and turning your microcontroller into something useful. GPIO has two fundamental operating modes, input and output. Input let's you read the voltage on a pin, to see whether it's held low(0V) or high(3V) and deal with that information programmatically. Output let's you set the voltage on a pin, again either high or low.

Every pin on LPC1768 can be used as GPIO pin and can be independently set to act as input or output. In next tutorial we'll get you into how to achieve this goal. I mean reading the status of switch and making LED blink. But for now we only have to look at basics, which is very important to understand before we go and build application. Depending on LPC17xx version the pin out maybe different. Here we'll focus on 100 pin LPC1768 as an example. Please keep LPC1768 User Manual with you [Chapter: 9, Page No:129]. pins on LPC1768 are divided into 5 groups (PORTs) starting from 0 to 4. Pin naming convention: P0.0 (group 0, pin 0) or (port 0, pin 0). Each pin has 4 operating modes: GPIO(default), 1st alternate function, 2nd alternate function, 3rd alternate function. Almost all GPIO pins are powered automatically so we don't need to turn them on always. Let's have a look at details about configuration of these GPIO port pins.

Pin Function Setting

The LPC_PINCON register controls operating mode of these pin.
LPC_PINCON –> PINSEL0 [1:0] control PIN 0.0 operating mode.
[Page No: 102, Table: 74].
…….
LPC_PINCON –> PINSEL0 [31:30] control PIN 0.15 operating mode.
LPC_PINCON –> PINSEL1 [1:0] control PIN 0.16 operating mode.
……..
LPC_PINCON –> PINSEL1 [29:28] control PIN 0.30 operating mode.
LPC_PINCON –> PINSEL2 [1:0] control PIN 1.0 operating mode.
……..
LPC_PINCON –> PINSEL2 [31:30] control PIN 1.15 operating mode
LPC_PINCON –> PINSEL3 [1:0] control PIN 1.16 operating mode
……
LPC_PINCON –> PINSEL3 [31:30] control PIN 1.31 operating mode
……..
LPC_PINCON –> PINSEL9 [25:24] control PIN 4.28 operating mode
LPC_PINCON –> PINSEL9 [27:26] control PIN 4.29 operating mode

NOTE: some register bits are reserved and are not used to control a pin for example,
LPC_PINCON –> PINSEL9 [23:0] are reserved.
LPC_PINCON –> PINSEL9 [31:28] are reserved.

Bit Value	Function
00	GPIO Function
01	1st alternate function
10	2nd alternate function
11	3rd alternate function

Example:
To set pin 0.3 as GPIO (set corresponding bit to 00)
LPC_PINCON –> PINSEL0 &= ~ ((1<<7) | (1<<6));

To set pin 0.3 as ADC channel 0.6 (2nd alternate function, set corresponding bit to 10)
LPC_PINCON –> PINSEL0 &= ((1<<7) | (0<<6)); // you may omit (0<<6)

Pin Direction Setting

Register LPC_GPIOn –> FIODIR [31:0] control the pin input/output, where 'n' stands for pin group (0-4). To set a pin as output, set the corresponding bit to '1'. To set a pin as input, set the corresponding bit to '0', by default, all pins are set as input (all bits are 0).
Example:
To set 0.3 as output
LPC_GPIO –> FIODIR |= (1<<3);

Pin is Set as Output

A pin digital high/low setting
LPC_GPIOn –> FIOSET is used to turn a pin to HIGH. Register LPC_GPIOn –> FIOCLR is used to turn a pin to low. To turn a pin to digital '1' (high), set the corresponding bit of LPC_GPIOn –> FIOSET to 1. To turn a pin to digital '0' (low), set the corresponding bit of LPC_GPIOn –> FIOCLR to 1.
Example
Turn Pin 0.3 to high
LPC_GPIO0 –> FIOSET |= (1<<3);
If we set LPC_GPIOn –> FIOSET bit to '0' there is no effect.
Turn Pin 0.3 to low
LPC_GPIO0 –> FIOCLR |= (1<<3);
If we set LPC_GPIOn –> FIOCLR bit to '0' there is no effect.

Pin is Set to Input

Read a Pin Value
Register LPC_GPIOn –> FIOPIN stores the current pin state. The corresponding bit is '1' indicates that the pin is driven high.
Example
To read current state of Pin 0.3
Value = ((LPC_GPIO0 –> FIOPIN & (1<<3)) >> 3);
Note: write 1/0 to corresponding bit in LPC_GPIOn –> FIOPIN can change the output of the pin to 1/0 but it is not recommended. We should use LPC_GPIOn –> FIOSET and GPIOn –> FIOCLR instead.
Pin Internal Pull up Setting

Register LPC_PINCON –> PINMODEn is used to set up a pin internal pull-up.
LPC_PINCON –> PINMODE0 [1:0] control P0.0 internal pull-up
…..
LPC_PINCON –> PINMODE0 [31:30] control P0.15 internal pull-up,

Bit Value	Pin Mode
00	On chip pull-up resistor enabled
01	Repeater Mode
10	Tri-State mode, (neither pull-up nor pull-down)
11	On chip pull-down resistor enabled

Timers

Using delays in a software code is usual to embedded programmers. A normal delay function might be used to create a period of no operation through a for loop iterating for a few 1000 cycles. But these types of delays need not be accurate and fundamentally, it is not a good programming practice. So, TIMER/COUNTER is software designed to count the time interval between events. It counts the cycle of the peripheral clock or an externally-supplied clock.

Features

The LPC 1768 has 4 general purpose timers. (TIMER 0, TIMER 1, TIMER 2, TIMER 3)
32-bit Timer/Counter
32-bit prescaler
Four 32 bit match registers
Four external outputs corresponding to match registers

Configuration

As usual, the first configuration setting is enabling the power supply of the peripheral, in this case, PCTIMx where x is 0,1,2, or 3.
Next, the peripheral clock must be set in the PCLKSEL0 register to set the appropriate bits of the respective timer.
Timer pins have to be selected through the PINSEL register…
Finally, the necessary match registers, values, and the eventual event must be configured based on the necessary results.

POWER

We will be using Timer 0 in our tutorial. Since Timer 0/1 have reset value as 1, the peripheral is already enabled. But if you are using Timer 2/3, this step is important as they have 0 as the reset value.

```
Power control
1 Power_Control_Reg |= (1<<Timer0_Power);
2 Match_Control_Reg = (1<<MatchReg0_Int) | (1<<MatchReg0_Rst);
```

PERIPHERAL CLOCK & PRESCALER

The PCLKSEL0 register contains the peripheral clock selection for timers 0 and 1 and PCLKSEL1 register contains the peripheral clock selection for timers 2 and 3. As in most case, the reset value is 00.The PCLK is necessary as it gives the number of clock cycles required to increment the timer counter by 1.
And the required value is entered in the prescaler register.

```
Prescalar
1 PreScale = pclk/ReqCntsPerSec;
2 Prescalar_Reg = PreScale - 1;
```

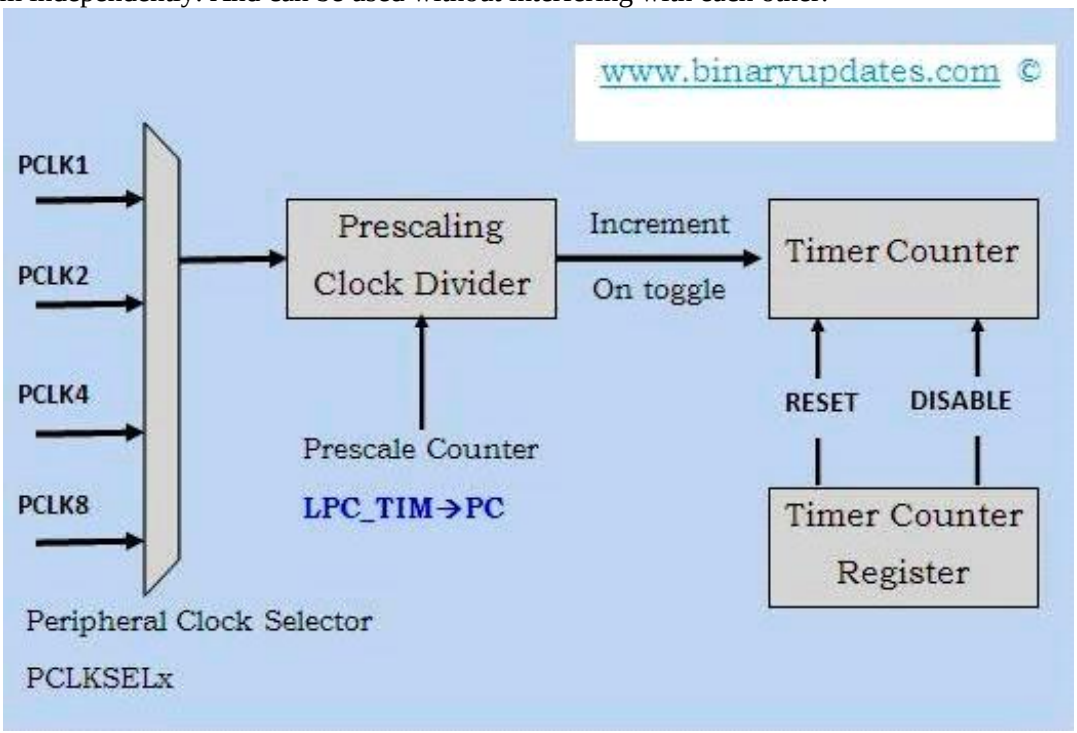
PCLK_TIMERx bits	Function
<0,0>	PCLK_TIMERx = CCLK/4
<0,1>	PCLK_TIMERx = CCLK
<1,0>	PCLK_TIMERx = CCLK/2
<1,1>	PCLK_TIMERx = CCLK/8

MATCH CONTROL REGISTER:

The match register will be configured in such a way that it resets on every match with the timer counter as well as an interrupt occurs on the match. We are going to generate a 1-second delay, hence the required value has been entered in the MR0 register.

Timer is internal peripheral in LPC1768. They use CPU clock to keep track of time and count. Timer enhances use of microcontroller in number of ways. Timers send periodic events and make precise measurements. It makes time available for your microcontroller project. This means you can start using temporal information in your program, without having to use unwieldy spin loops. In this tutorial our goal is to set up a timer and then with the help of interrupts we'll blink the LED.

In LPC1768 Microcontroller there are four timers Timer 0, Timer1, Timer2 and Timer3. These are 32-bit timer/counter with programmable 32-bit prescaler. All are identical but can have options to set them independently. And can be used without interfering with each other.



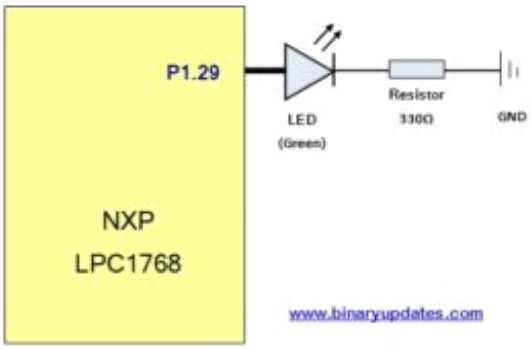
All timer's are built around an internal Timer Counter (TC), which is a 32-bit register and incremented periodically. The rate of change is derived from the current speed of the CPU clock you've connected up and what the Prescale Counter is set to. There is nothing more to it than that. The Prescale Counter is clock divider. As mentioned earlier, Timer Counter is a 32-bit register and it counts in range from 0x00000000 to 0xFFFFFFFF.

Before we get into programming fundamentals, it's important to understand powering procedure for LPC1768 microcontroller. Powering device is necessary before choosing the peripheral clock and

setting the prescale register. We need to turn on timer. Upon RESET most of the LPC's peripherals are OFF and aren't being supplied power by microcontroller.

This can save a lot of power, but few core peripherals are turned on when the LPC starts, among these GPIO, TIMER0 and TIMER1. This means that TIMER2 and TIMER3 start off and if you need them in your application then you'll have to turn them on. Additionally, if you don't need TIMER0 or TIMER1, you can even turn them OFF to save some power.

Usually, power control is considered a system feature and is controlled by register LPC_SC–>PCONP.



Circuit Schematic: LED with LPC1768

ADC IN LPC 17XX MICROCONTROLLER

Introduction

The data that we use for programming a microcontroller normally deals with digital signals. But there are situations where a microcontroller has to deal with inputs from external devices that gives an analog output. In such cases, we can interface the microcontroller with an external device such as an ADC0808 to convert the analog signal to a digital signal.

But in more advanced and powerful microcontrollers such as LPC1768, these processes are handled internally rather than externally, using an internal ADC module. In this tutorial, we will learn the basic functionality of the built-in ADC module of the LPC1768 ARM controller. A potentiometer will be used to take the input and the measured value will be processed by the built-in ADC module and the output will be displayed in the serial terminal via UART. More info on UART, refer UART Communication.

Features

The ADC module in the LPC1768 uses the technique of successive approximation to convert signals from analog to digital values. The internal SAR (Successive Approximation Register) is designed to take the input from a voltage comparator of the internal module to give a 12-bit output resulting in a high precision result.

- ✓ The 12-bit conversion rate is clocked at 200 kHz.
- ✓ This speed is achieved with 8 multiplexed channels.
- ✓ The measurement range is set between Vrefn and Vrefp which is usually 3v and should not exceed the V_{dd} voltage level.
- ✓ The module supports burst conversion mode as well.

The clock required for analog to digital converter is provided by the APB clock and is scaled to the clock required for the successive approximation process using a programmable divider that is included in each converter.

Configuration

The power to the ADC module must be given initially. The reset value for PCADC or the power control for ADC bit in the PCONP register is 0. Therefore, it is mandatory that this step must not be skipped in the software. The clock to the ADC module must be set. The corresponding pins in the

PINSEL register must be selected to function as ADC pins.

Control the ADC operation using the ADC control register, ADCR. The basic operational flow is to start the conversion process, read the result and stop after conversion and reading is completed.

LPC1768 ADC Block

LPC1768 has an inbuilt 12 bit Successive Approximation ADC which is multiplexed among 8 input pins.The ADC reference voltages is measured across VREFN to VREFP, meaning it can do the conversion within this range. Usually the VREFP is connected to VDD and VREFN is connected to GND.

- ✓ As LPC1768 works on 3.3 volts, this will be the ADC reference voltage.
- ✓ Now the $$resolution of ADC = 3.3/(2^{12}) = 3.3/4096 =0.000805 = 0.8mV$$
- ✓ The below block diagram shows the ADC input pins multiplexed with other GPIO pins.
- ✓ The ADC pin can be enabled by configuring the corresponding PINSEL register to select ADC function.
- ✓ When the ADC function is selected for that pin in the Pin Select register, other Digital signals are disconnected from the ADC input pins.

Adc Channel	Port Pin	Pin Functions	Associated PINSEL Register
AD0	P0.23	0-GPIO, 1-**AD0[0]**, 2-I2SRX_CLK, 3-CAP3[0]	14,15 bits of PINSEL1
AD1	P0.24	0-GPIO, 1-**AD0[1]**, 2-I2SRX_WS, 3-CAP3[1]	16,17 bits of PINSEL1
AD2	P0.25	0-GPIO, 1-**AD0[2]**, 2-I2SRX_SDA, 3-TXD3	18,19 bits of PINSEL1
AD3	P0.26	0-GPIO, 1-**AD0[3]**, 2-AOUT, 3-RXD3	20,21 bits of PINSEL1
AD4	P1.30	0-GPIO, 1-VBUS, 2- , 3-**AD0[4]**	28,29 bits of PINSEL3
AD5	P1.31	0-GPIO, 1-SCK1, 2- , 3-**AD0[5]**	30,31 bits of PINSEL3
AD6	P0.3	0-GPIO, 1-RXD0, 2-**AD0[6]**, 3-	6,7 bits of PINSEL0
AD7	P0.2	0-GPIO, 1-TXD0, 2-**AD0[7]**, 3-	4,5 bits of PINSEL0

ADC Registers

The below table shows the registers associated with LPC1768 ADC.

We are going to focus only on ADCR and ADGDR as these are sufficient for simple A/D conversion. However once you are familiar with LPC1768 ADC, you can explore the other features and the associated registers.

Register	Description
ADCR	A/D COntrol Register: Used for Configuring the ADC
ADGDR	A/D Global Data Register: This register contains the ADC's DONE bit and the result of the most recent A/D conversion
ADINTEN	A/D Interrupt Enable Register
ADDR0 - ADDR7	A/D Channel Data Register: Contains the recent ADC value for respective channel
ADSTAT	A/D Status Register: Contains DONE & OVERRUN flag for all the ADC channels

Steps for Configuring ADC

Below are the steps for configuring the LPC1768 ADC.

Configure the GPIO pin for ADC function using PINSEL register.

Enable the CLock to ADC module.

Deselect all the channels and Power on the internal ADC module by setting ADCR.PDN bit.
Select the Particular channel for A/D conversion by setting the corresponding bits in ADCR.SEL
Set the ADCR.START bit for starting the A/D conversion for selected channel.
Wait for the conversion to complete, ADGR.DONE bit will be set once conversion is over.
Read the 12-bit A/D value from ADGR.RESULT.
Use it for further processing or just display on LCD.

12-bit ADC

The LPC17xx contain a single 12-bit successive approximation ADC with eight channels and DMA support.

Features

- 12-bit successive approximation ADC.
- Input multiplexing among 8 pins.
- Power-down mode.
- Measurement range VREFN to VREFP.
- 12-bit conversion rate: 200 kHz.
- Individual channels can be selected for conversion.
- Burst conversion mode for single or multiple inputs.
- Optional conversion on transition of input pin or Timer Match signal.
- Individual result registers for each ADC channel to reduce interrupt overhead.
- DMA support.

10-bit DAC

The DAC allows generating a variable analog output. The maximum output value of the DAC is VREFP.

Features

- 10-bit DAC
- Resistor string architecture
- Buffered output
- Power-down mode
- Selectable output drive
- Dedicated conversion timer
- DMA support

UARTs

The LPC17xx each contain four UARTs. In addition to standard transmit and receive data lines, UART1 also provides a full modem control handshake interface and support for RS-485/9-bit mode allowing both software address detection and automatic address detection using 9-bit mode.

The UARTs include a fractional baud rate generator. Standard baud rates such as 115200 Bd can be achieved with any crystal frequency above 2 MHz.

Features

- Maximum UART data bit rate of 6.25 Mbit/s.
- 16 B Receive and Transmit FIFOs.
- Register locations conform to 16C550 industry standard.
- Receiver FIFO trigger points at 1 B, 4 B, 8 B, and 14 B.
- Built-in fractional baud rate generator covering wide range of baud rates without a need for external crystals of particular values.
- Auto baud capabilities and FIFO control mechanism that enables software flow control implementation.
- UART1 equipped with standard modem interface signals. This module also provides full support for hardware flow control (auto-CTS/RTS).
- Support for RS-485/9-bit/EIA-485 mode (UART1).
- UART3 includes an IrDA mode to support infrared communication.
- All UARTs have DMA support.

SPI serial I/O controller

The LPC17xx contain one SPI controller. SPI is a full duplex serial interface designed to handle multiple masters and slaves connected to a given bus. Only a single master and a single slave can communicate on the interface during a given data transfer. During a data transfer the master always sends 8 bits to 16 bits of data to the slave, and the slave always sends 8 bits to 16 bits of data to the master.

Features

- Maximum SPI data bit rate of 12.5 Mbit/s
- Compliant with SPI specification
- Synchronous, serial, full duplex communication
- Combined SPI master and slave
- Maximum data bit rate of one eighth of the input clock rate
- 8 bits to 16 bits per transfer

SSP serial I/O controller

The LPC17xx contain two SSP controllers. The SSP controller is capable of operation on a SPI, 4-wire SSI, or Microwire bus. It can interact with multiple masters and slaves on the bus. Only a single master and a single slave can communicate on the bus during a given data transfer. The SSP supports full duplex transfers, with frames of 4 bits to 16 bits of data flowing from the master to the slave and from the slave to the master. In practice, often only one of these data flows carries meaningful data.

Features

- Maximum SSP speed of 33 Mbit/s (master) or 8 Mbit/s (slave)
- Compatible with Motorola SPI, 4-wire Texas Instruments SSI, and National Semiconductor Microwire buses
- Synchronous serial communication
- Master or slave operation
- 8-frame FIFOs for both transmit and receive
- 4-bit to 16-bit frame
- DMA transfers supported by GPDMA

I2C-bus serial I/O controllers

The LPC17xx each contain three I2C-bus controllers.

The I2C-bus is bidirectional for inter-IC control using only two wires: a Serial Clock line (SCL) and a Serial DAta line (SDA). Each device is recognized by a unique address and can operate as either a receiver-only device (e.g., an LCD driver) or a transmitter with the capability to both receive and send information (such as memory). Transmitters and/or receivers can operate in either master or slave mode, depending on whether the chip has to initiate a data transfer or is only addressed. The I2C is a multi-master bus and can be controlled by more than one bus master connected to it.

Features

- I2C0 is a standard I2C compliant bus interface with open-drain pins. I2C0 also supports Fast mode plus with bit rates up to 1 Mbit/s.
- I2C1 and I2C2 use standard I/O pins with bit rates of up to 400 kbit/s (Fast I2C-bus).
- Easy to configure as master, slave, or master/slave.
- Programmable clocks allow versatile rate control.
- Bidirectional data transfer between masters and slaves.
- Multi-master bus (no central master).
- Arbitration between simultaneously transmitting masters without corruption of serial data on the bus.
- Serial clock synchronization allows devices with different bit rates to communicate via one serial bus.
- Serial clock synchronization can be used as a handshake mechanism to suspend and resume serial transfer.
- The I2C-bus can be used for test and diagnostic purposes.
- All I2C-bus controllers support multiple address recognition and a bus monitor mode.

I2S-bus serial I/O controllers

Remark: The I2S-bus interface is available on parts LPC1769/68/67/66/65/63.

The I2S-bus provides a standard communication interface for digital audio applications.

The *I2S-bus specification* defines a 3-wire serial bus using one data line, one clock line, and one word select signal. The basic I2S-bus connection has one master, which is always the master, and one slave. The I2S-bus interface provides a separate transmit and receive channel, each of which can operate as either a master or a slave.

Features

- The interface has separate input/output channels each of which can operate in master or slave mode.
- Capable of handling 8-bit, 16-bit, and 32-bit word sizes.
- Mono and stereo audio data supported.
- The sampling frequency can range from 16 kHz to 96 kHz (16, 22.05, 32, 44.1, 48, 96) kHz.

- Support for an audio master clock.
- Configurable word select period in master mode (separately for I2S-bus input and output).
- Two 8-word FIFO data buffers are provided, one for transmit and one for receive.
- Generates interrupt requests when buffer levels cross a programmable boundary.
- Two DMA requests, controlled by programmable buffer levels. These are connected to the GPDMA block.
- Controls include reset, stop and mute options separately for I2S-bus input and I2S-bus output.

General purpose 32-bit timers/external event counters

The LPC17xx include four 32-bit timer/counters. The timer/counter is designed to count cycles of the system derived clock or an externally-supplied clock. It can optionally generate interrupts, generate timed DMA requests, or perform other actions at specified timer values, based on four match registers. Each timer/counter also includes two capture inputs to trap the timer value when an input signal transitions, optionally generating an interrupt.

Features

- A 32-bit timer/counter with a programmable 32-bit prescaler.
- Counter or timer operation.
- Two 32-bit capture channels per timer, that can take a snapshot of the timer value when an input signal transitions. A capture event may also generate an interrupt.
- Four 32-bit match registers that allow:
- Continuous operation with optional interrupt generation on match.
- Stop timer on match with optional interrupt generation.
- Reset timer on match with optional interrupt generation.
- Up to four external outputs corresponding to match registers, with the following capabilities:
- Set LOW on match.
- Set HIGH on match.
- Toggle on match.
- Do nothing on match.
- Up to two match registers can be used to generate timed DMA requests.

Pulse width modulator

The PWM is based on the standard Timer block and inherits all of its features, although only the PWM function is pinned out on the LPC17xx. The Timer is designed to count cycles of the system derived clock and optionally switch pins, generate interrupts or perform other actions when specified timer values occur, based on seven match registers. The PWM function is in addition to these features, and is based on match register events.

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

Two match registers can be used to provide a single edge controlled PWM output. One match register (PWMMR0) controls the PWM cycle rate, by resetting the count upon match. The other match register controls the PWM edge position. Additional single edge controlled PWM outputs require only

one match register each, since the repetition rate is the same for all PWM outputs. Multiple single edge controlled PWM outputs will all have a rising edge at the beginning of each PWM cycle, when an PWMMR0 match occurs.

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

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

Features

- One PWM block with Counter or Timer operation (may use the peripheral clock or one of the capture inputs as the clock source).
- Seven match registers allow up to 6 single edge controlled or 3 double edge controlled PWM outputs, or a mix of both types. The match registers also allow:
- Continuous operation with optional interrupt generation on match.
- Stop timer on match with optional interrupt generation.
- Reset timer on match with optional interrupt generation
- Supports single edge controlled and/or double edge controlled PWM outputs. Single edge controlled PWM outputs all go high at the beginning of each cycle unless the output is a constant low. Double edge controlled PWM outputs can have either edge occur at any position within a cycle. This allows for both positive going and negative going pulses.
- Pulse period and width can be any number of timer counts. This allows complete flexibility in the trade-off between resolution and repetition rate. All PWM outputs will occur at the same repetition rate.
- Double edge controlled PWM outputs can be programmed to be either positive going or negative going pulses.
- Match register updates are synchronized with pulse outputs to prevent generation of erroneous pulses. Software must ‘release’ new match values before they can become effective.
- May be used as a standard 32-bit timer/counter with a programmable 32-bit prescaler if the PWM mode is not enabled.

Motor control PWM

The motor control PWM is a specialized PWM supporting 3-phase motors and other combinations. Feedback inputs are provided to automatically sense rotor position and use that information to ramp speed up or down. An abort input is also provided that causes the PWM to immediately release all motor drive outputs. At the same time, the motor control PWM is highly configurable for other generalized timing, counting, capture, and compare applications.

Quadrature Encoder Interface (QEI)

A quadrature encoder, also known as a 2-channel incremental encoder, converts angular displacement into two pulse signals. By monitoring both the number of pulses and the relative phase of the two signals, the user can track the position, direction of rotation, and velocity. In addition, a third channel, or index signal, can be used to reset the position counter. The quadrature encoder interface decodes the digital pulses from a quadrature encoder wheel to integrate position over time and determine direction of rotation. In addition, the QEI can capture the velocity of the encoder wheel.

Features

- Tracks encoder position.
- Increments/decrements depending on direction.
- Programmable for 2× or 4× position counting.
- Velocity capture using built-in timer.
- Velocity compare function with "less than" interrupt.
- Uses 32-bit registers for position and velocity.
- Three position compare registers with interrupts.
- Index counter for revolution counting.
- Index compare register with interrupts.
- Can combine index and position interrupts to produce an interrupt for whole and partial revolution displacement.
- Digital filter with programmable delays for encoder input signals.
- Can accept decoded signal inputs (clk and direction).
- Connected to APB.

Repetitive Interrupt (RI) timer

The repetitive interrupt timer provides a free-running 32-bit counter which is compared to a selectable value, generating an interrupt when a match occurs. Any bits of the timer/compare can be masked such that they do not contribute to the match detection. The repetitive interrupt timer can be used to create an interrupt that repeats at predetermined intervals.

Features

- 32-bit counter running from PCLK. Counter can be free-running or be reset by a generated interrupt.
- 32-bit compare value.
- 32-bit compare mask. An interrupt is generated when the counter value equals the compare value, after masking. This allows for combinations not possible with a simple compare.

Arm Cortex-M3 system tick timer

The Arm Cortex-M3 includes a system tick timer (SYSTICK) that is intended to generate a dedicated SYSTICK exception at a 10 ms interval. In the LPC17xx, this timer can be clocked from the internal AHB clock or from a device pin.

Watchdog timer

The purpose of the watchdog is to reset the microcontroller within a reasonable amount of time if it

enters an erroneous state. When enabled, the watchdog will generate a system reset if the user program fails to 'feed' (or reload) the watchdog within a predetermined amount of time.

Features

- Internally resets chip if not periodically reloaded.
- Debug mode.
- Enabled by software but requires a hardware reset or a watchdog reset/interrupt to be disabled.
- Incorrect/Incomplete feed sequence causes reset/interrupt if enabled.
- Flag to indicate watchdog reset.
- Programmable 32-bit timer with internal prescaler.
- Selectable time period from ($T_{cy(WDCLK)} \times 256 \times 4$) to ($T_{cy(WDCLK)} \times 2^{32} \times 4$) in multiples of $T_{cy(WDCLK)} \times 4$.
- The Watchdog Clock (WDCLK) source can be selected from the Internal RC (IRC) oscillator, the RTC oscillator, or the APB peripheral clock. This gives a wide range of potential timing choices of Watchdog operation under different power reduction conditions.
- It also provides the ability to run the WDT from an entirely internal source that is not dependent on an external crystal and its associated components and wiring for increased reliability.
- Includes lock/safe feature.

RTC and backup registers

The RTC is a set of counters for measuring time when system power is on, and optionally when it is off. The RTC on the LPC17xx is designed to have extremely low power consumption, i.e. less than 1 mA. The RTC will typically run from the main chip power supply, conserving battery power while the rest of the device is powered up. When operating from a battery, the RTC will continue working down to 2.1 V. Battery power can be provided from a standard 3 V Lithium button cell.

An ultra-low power 32 kHz oscillator will provide a 1 Hz clock to the time counting portion of the RTC, moving most of the power consumption out of the time counting function.

The RTC includes a calibration mechanism to allow fine-tuning the count rate in a way that will provide less than 1 second per day error when operated at a constant voltage and temperature. A clock output function makes measuring the oscillator rate easy and accurate.

The RTC contains a small set of backup registers (20 bytes) for holding data while the main part of the LPC17xx is powered off.

The RTC includes an alarm function that can wake up the LPC17xx from all reduced power modes with a time resolution of 1 s.

Features

- Measures the passage of time to maintain a calendar and clock.
- Ultra low power design to support battery powered systems.
- Provides Seconds, Minutes, Hours, Day of Month, Month, Year, Day of Week, and Day of Year.
- Dedicated power supply pin can be connected to a battery or to the main 3.3 V.
- Periodic interrupts can be generated from increments of any field of the time registers.
- Backup registers (20 bytes) powered by VBAT.
- RTC power supply is isolated from the rest of the chip.

Clocking and power control

Crystal oscillators

The LPC17xx include three independent oscillators. These are the main oscillator, the IRC oscillator, and the RTC oscillator. Each oscillator can be used for more than one purpose as required in a particular application. Any of the three clock sources can be chosen by software to drive the main PLL and ultimately the CPU.

Following reset, the LPC17xx will operate from the Internal RC oscillator until switched by software. This allows systems to operate without any external crystal and the bootloader code to operate at a known frequency.

Internal RC oscillator

The IRC may be used as the clock source for the WDT, and/or as the clock that drives the PLL and subsequently the CPU. The nominal IRC frequency is 4 MHz. The IRC is trimmed to 1 % accuracy over the entire voltage and temperature range.

Upon power-up or any chip reset, the LPC17xx use the IRC as the clock source. Software may later switch to one of the other available clock sources.

Main oscillator

The main oscillator can be used as the clock source for the CPU, with or without using the PLL. The main oscillator also provides the clock source for the dedicated USB PLL.

The main oscillator operates at frequencies of 1 MHz to 25 MHz; this frequency can be boosted to a higher frequency, up to the maximum CPU operating frequency, by the main PLL. The clock selected as the PLL input is PLLCLKIN. The Arm processor clock frequency is referred to as CCLK elsewhere in this document. The frequencies of PLLCLKIN and CCLK are the same value unless the PLL is active and connected. The clock frequency for each peripheral can be selected individually and is referred to as CLK.

RTC oscillator

The RTC oscillator can be used as the clock source for the RTC block, the main PLL, and/or the CPU.

Main PLL (PLL0)

The PLL0 accepts an input clock frequency in the range of 32 kHz to 25 MHz. The input frequency is multiplied up to a high frequency, then divided down to provide the actual clock used by the CPU and/or the USB block.

The PLL0 input, in the range of 32 kHz to 25 MHz, may initially be divided down by a value 'N', which may be in the range of 1 to 256. This input division provides a wide range of output frequencies from the same input frequency.

Following the PLL0 input divider is the PLL0 multiplier. This can multiply the input divider output through the use of a Current Controlled Oscillator (CCO) by a value 'M', in the range of 1 through 32768. The resulting frequency must be in the range of 275 MHz to 550 MHz. The multiplier works by dividing the CCO output by the value of M, then using a phase-frequency detector to compare the divided CCO output to the multiplier input. The error value is used to adjust the CCO frequency.

The PLL0 is turned off and bypassed following a chip Reset and by entering Power-down mode. PLL0 is enabled by software only. The program must configure and activate the PLL0, wait for the PLL0 to lock, and then connect to the PLL0 as a clock source.

USB PLL (PLL1)

The LPC17xx contain a second, dedicated USB PLL1 to provide clocking for the USB interface.

The PLL1 receives its clock input from the main oscillator only and provides a fixed 48 MHz clock to the USB block only. The PLL1 is disabled and powered off on reset. If the PLL1 is left disabled, the USB clock will be supplied by the 48 MHz clock from the main PLL0.

The PLL1 accepts an input clock frequency in the range of 10 MHz to 25 MHz only. The input frequency is multiplied up the range of 48 MHz for the USB clock using a Current Controlled Oscillators (CCO). It is insured that the PLL1 output has a 50 % duty cycle.

RTC clock output

The LPC17xx feature a clock output function intended for synchronizing with external devices and for use during system development to allow checking the internal clocks CCLK, IRC clock, main crystal, RTC clock, and USB clock in the outside world. The RTC clock output allows tuning the RTC frequency without probing the pin, which would distort the results.

Wake-up timer

The LPC17xx begin operation at power-up and when awakened from Power-down mode by using the 4 MHz IRC oscillator as the clock source. This allows chip operation to resume quickly. If the main oscillator or the PLL is needed by the application, software will need to enable these features and wait for them to stabilize before they are used as a clock source. When the main oscillator is initially activated, the wake-up timer allows software to ensure that the main oscillator is fully functional before the processor uses it as a clock source and starts to execute instructions. This is important at power on, all types of Reset, and whenever any of the aforementioned functions are turned off for any reason. Since the oscillator and other functions are turned off during Power-down mode, any wake-up of the processor from Power-down mode makes use of the wake-up timer.

The Wake-up Timer monitors the crystal oscillator to check whether it is safe to begin code execution. When power is applied to the chip, or when some event caused the chip to exit Power-down mode, some time is required for the oscillator to produce a signal of sufficient amplitude to drive the clock logic. The amount of time depends on many factors, including the rate of $V_{DD(3V3)}$ ramp (in the case of power on), the type of crystal and its electrical characteristics (if a quartz crystal is used), as well as any other external circuitry (e.g., capacitors), and the characteristics of the oscillator itself under the existing ambient conditions.

Power control

The LPC17xx support a variety of power control features. There are four special modes of processor power reduction: Sleep mode, Deep-sleep mode, Power-down mode, and Deep power-down mode. The CPU clock rate may also be controlled as needed by changing clock sources, reconfiguring PLL values, and/or altering the CPU clock divider value. This allows a trade-off of power versus processing speed based on application requirements.

In addition, Peripheral Power Control allows shutting down the clocks to individual on-chip peripherals, allowing fine tuning of power consumption by eliminating all dynamic power use in any peripherals that are not required for the application. Each of the peripherals has its own clock divider which provides even better power control.

Integrated PMU (Power Management Unit) automatically adjust internal regulators to minimize power consumption during Sleep, Deep sleep, Power-down, and Deep power-down modes.

The LPC17xx also implement a separate power domain to allow turning off power to the bulk of the device while maintaining operation of the RTC and a small set of registers for storing data during any of the power-down modes.

Sleep mode

When Sleep mode is entered, the clock to the core is stopped. Resumption from the Sleep mode does not need any special sequence but re-enabling the clock to the Arm core. In Sleep mode, execution of instructions is suspended until either a Reset or interrupt occurs. Peripheral functions continue operation during Sleep mode and may generate interrupts to cause the processor to resume execution. Sleep mode eliminates dynamic power used by the processor itself, memory systems and related controllers, and internal buses.

Deep-sleep mode

In Deep-sleep mode, the oscillator is shut down and the chip receives no internal clocks. The processor state and registers, peripheral registers, and internal SRAM values are preserved throughout Deep-sleep mode and the logic levels of chip pins remain static. The output of the IRC is disabled but the IRC is not powered down for a fast wake-up later. The RTC oscillator is not stopped because the RTC interrupts may be used as the wake-up source. The PLL is automatically turned off and disconnected. The CCLK and USB clock dividers automatically get reset to zero.

The Deep-sleep mode can be terminated and normal operation resumed by either a Reset or certain specific interrupts that are able to function without clocks. Since all dynamic operation of the chip is suspended, Deep-sleep mode reduces chip power consumption to a very low value. Power to the flash memory is left on in Deep-sleep mode, allowing a very quick wake-up.

On wake-up from Deep-sleep mode, the code execution and peripherals activities will resume after 4 cycles expire if the IRC was used before entering Deep-sleep mode. If the main external oscillator was used, the code execution will resume when 4096 cycles expire. PLL and clock dividers need to be reconfigured accordingly.

Power-down mode

Power-down mode does everything that Deep-sleep mode does, but also turns off the power to the IRC oscillator and the flash memory. This saves more power but requires waiting for resumption of flash operation before execution of code or data access in the flash memory can be accomplished.

On the wake-up of Power-down mode, if the IRC was used before entering Power-down mode, it will take IRC 60 ms to start-up. After this 4 IRC cycles will expire before the code execution can then be resumed if the code was running from SRAM. In the meantime, the flash wake-up timer then counts 4 MHz IRC clock cycles to make the 100 ms flash start-up time. When it times out, access to the flash will be allowed. Users need to reconfigure the PLL and clock dividers accordingly.

Deep power-down mode

The Deep power-down mode can only be entered from the RTC block. In Deep power-down mode, power is shut off to the entire chip with the exception of the RTC module and the RESET pin. The LPC17xx can wake up from Deep power-down mode via the RESET pin or an alarm match event of the RTC.

Wake-up interrupt controller

The Wake-up Interrupt Controller (WIC) allows the CPU to automatically wake up from any enabled priority interrupt that can occur while the clocks are stopped in Deep sleep, Power-down, and Deep power-down modes.

The WIC works in connection with the Nested Vectored Interrupt Controller (NVIC). When the CPU enters Deep sleep, Power-down, or Deep power-down mode, the NVIC sends a mask of the current interrupt situation to the WIC.This mask includes all of the interrupts that are both enabled and of sufficient priority to be serviced immediately. With this information, the WIC simply notices when one of the interrupts has occurred and then it wakes up the CPU.The WIC eliminates the need to periodically wake up the CPU and poll the interrupts resulting in additional power savings.

Peripheral power control

A Power Control for Peripherals feature allows individual peripherals to be turned off if they are not needed in the application, resulting in additional power savings.

UNIT-V
VLIW ARCHITECTURE

The Necessity of Digital Signal Processors

In the 1960s it was predicted that artificial intelligence would revolutionize the way humans interact with computers and other machines. It was believed that by the end of the century we would have robots cleaning our houses, computers driving our cars, and voice interfaces controlling the storage and retrieval of information. This hasn't happened; these abstract tasks are far more complicated than expected, and very difficult to carry out with the step-by-step logic provided by digital computers.

However, the last forty years have shown that computers are extremely capable in two broad areas, (1) **data manipulation**, such as word processing and database management, and (2) **mathematical calculation**, used in science, engineering, and Digital Signal Processing. All microprocessors can perform both tasks; however, it is difficult (expensive) to make a device that is optimized for both. There are technical tradeoffs in the hardware design, such as the size of the instruction set and how interrupts are handled. Even more important, there are marketing issues involved: development and manufacturing cost, competitive position, product lifetime, and so on. As a broad generalization, these factors have made traditional microprocessors, such as the Pentium® which is primarily directed at data manipulation. Similarly, DSPs are designed to perform the mathematical calculations needed in Digital Signal Processing.

Figure 1 lists the most important differences between these two categories. Data manipulation involves storing and sorting information. For instance, consider a word processing program. The basic task is to store the information (typed in by the operator), organize the information (cut and paste, spell checking, page layout, etc.), and then retrieving the information (for example, printing a document with a laser printer). These tasks are accomplished by *moving* data from one location to another, and *testing* for inequalities (A=B, A<B, etc.).

	Data Manipulation	Math Calculation
Typical Applications	Word processing, database management, spread sheets, operating systems, etc.	Digital Signal Processing, motion control, scientific and engineering simulations, etc.
Main Operations	data movement (A --> B) value testing (If A== B then ...)	addition (A+B=C) multiplication (A×B=C)

Figure 1: Data Manipulation and Math Calculation

In comparison, most DSPs are used in applications where the processing is continuous, not having a defined start or end. For instance, consider an engineer designing a DSP system for an audio signal, such as a hearing aid. If the digital signal is being received at 20,000 samples per second, the DSP must be able to maintain a sustained throughput of 20,000 samples per second. However, there are important reasons not to make it any faster than necessary. As the speed increases, so does the *cost*, the *power consumption*, the *design difficulty*, and so on. This makes an accurate knowledge of the execution time critical for selecting the proper device, as well as the algorithms that can be applied.

History of Digital Signal Processors

In 1978, Intel released the 2920 as an "analog signal processor". It had an on- chip ADC/DAC with an internal signal processor, but it didn't have a hardware multiplier and was not successful in the market. In 1979, AMI released the S2811. It was designed as a microprocessor peripheral, and it had to be initialized by the host. The S2811 was likewise not successful in the market.

The First Generation (1979 to 1987)

In 1979, Bell Labs introduced the first single chip DSP, the Mac 4 Microprocessor. Then, in 1980 the first stand-alone, complete DSPs -- the NEC

µPD7720 and AT&T DSP1 -- were presented at the IEEE International Solid-State Circuits Conference '80. Both processors were inspired by the research in PSTN telecommunications.

The first DSP produced by Texas Instruments (TI), the TMS32010 presented in 1983, proved to be an even bigger success. It was based on the Harvard architecture, and so had separate instruction and data memory. It already had a special instruction set, with instructions like load-and-accumulate or multiply-and- accumulate. It could work on 16-bit numbers and needed 390ns for a multiply-add operation. TI is now the market leader in general purpose DSPs. Another successful design was the Motorola 56000.

The Second Generation (1988 to 1995)

About five years later, the second generation of DSPs began to spread. They had 3 memories for storing two operands simultaneously and included hardware to accelerate tight loops; they also had an addressing unit capable of loop-addressing. Some of them operated on 24-bit variables and a typical model only required about

21ns for a MAC (multiply-accumulate). Members of this generation were for example the AT&T DSP16A or the Motorola DSP56001.

The Third Generation (1996 to 2002)

The main improvement in the third generation was the appearance of application-specific units and instructions in the data path, or sometimes as coprocessors. These units allowed direct hardware acceleration of very specific but complex mathematical problems, like the Fourier-transform or matrix operations. Some chips, like the Motorola MC68356, even included more than one processor cores to work in parallel. Other DSPs from 1995 are the TI TMS320C541 or the TMS 320C80.

The Fourth Generation (2002 onwards)

The fourth generation is best characterized by the changes in the instruction set and the instruction encoding/decoding. SIMD and MMX extensions were added; VLIW and the superscalar architecture appeared. As always, the clock-speeds have increased, a 3ns MAC became now possible.

DSPs in 2007

Today's signal processors yield much greater performance. This is due in part to both technological and architectural advancements like lower design rules, fast- access two-level cache, (E) DMA circuit and a wider bus system. Of course, not all DSPs provide the same speed and many-many kind of signal processors exist, each one of them being better suited for a specific task, ranging in price from about 1.50 to 300 dollars. A Texas Instruments C6000 series DSP clocks at 1Ghz and implements separate instruction and data caches as well as a 8Mbyte 2nd level cache, and its I/O speed is rapid thanks to its 64 EDMA channels. The top models are capable of even 8000 MIPS (million instructions per second), use VLIW encoding, perform eight operations per clock-cycle and are compatible with a broad range of external peripherals and various buses (PCI/serial/etc).

Another big signal processor manufacturer today is Analog Devices. The company provides a broad range of DSPs, but its main portfolio is multimedia processors, such as codecs, filters and digital-analog converters. Its SHARC-based processors range in performance from 66Mhz/198MFLOPS (million floating-point operations per second) to 400Mhz/2400MFLOPS. Some models even support multiple multipliers and ALUs, SIMD instructions and audio processing-specific components and peripherals. Another product of the company is the Blackfin family of embedded digital signal processors, with models like the ADSP-BF531 to ADSP- BF536. These processors combine the features of a DSP with those of a general use processor. As a result, these processors can run simple operating systems like μCLinux, velOSity and Nucleus while operating relatively efficiently on real-time data.

Most DSPs use fixed-point arithmetic, because in real world signal processing, the additional range provided by floating point is not needed, and there is a large speed benefit; however, floating point DSPs are common for scientific and other applications where additional range or precision may be required. General purpose CPU's have ideas and influences from digital signal processors with extensions such as the MMX extensions in the Intel IA-32 architecture instruction set.

Generally, DSPs are dedicated integrated circuits; however DSP functionality can also be realized using Field Programmable Gate Array chips.

1. The Chip-Makers

The DSP market is very large and growing rapidly. As shown in Figure 2, it will be about 8-10 billion dollars/year at the turn of the century, and growing at a rate of 30-40% each year. This is being fueled by the incessant demand for better and cheaper consumer products, such as: cellular telephones, multimedia computers, and high-fidelity music reproduction. These high-revenue applications are shaping the field, while less profitable areas, such as scientific instrumentation, are just riding the wave of technology.

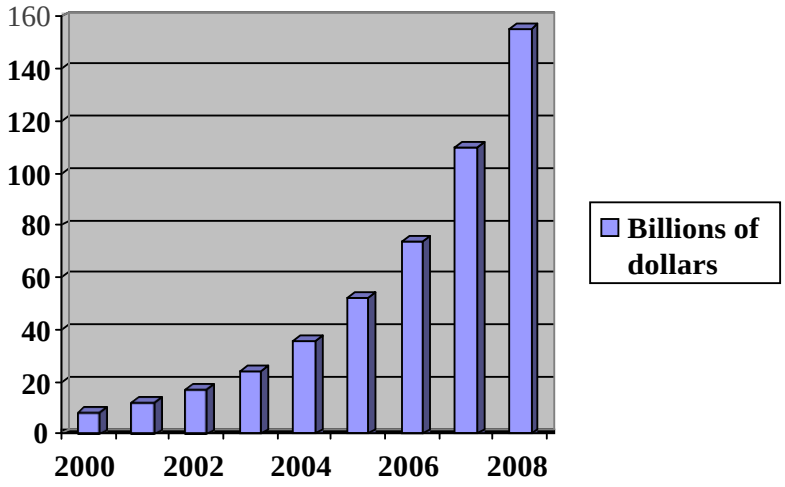


Figure 2: Revenue from the DSP Industry

DSPs can be purchased in three forms, as a **core**, as a **processor**, and as a **board level** product. In DSP, the term "core" refers to the section of the processor where the key tasks are carried out, including the data registers, multiplier, ALU, address generator, and program sequencer. A complete **processor** requires combining the core with memory and interfaces to the outside world. While the core and these peripheral sections are designed separately, they will be fabricated on the

same piece of silicon, making the processor a single integrated circuit. Lastly, there are several dozen companies that will sell you DSPs already mounted on a printed circuit board. These have such features as extra memory, A/D and D/A converters, EPROM sockets, multiple processors on the same board, and so on.

The present day Digital Signal Processor market (2007) is dominated by four companies. Here is a list, and the general scheme they use for numbering their products:

- **Analog Devices** (www.analog.com/dsp) ADSP-
  21xx 16 bit, fixed point
  ADSP-21xxx 32 bit, floating and fixed point

- **Lucent Technologies** (www.lucent.com)
  DSP16xxx 16 bit fixed point
  DSP32xx 32 bit floating point

- **Motorola** (www.mot.com)
  DSP561xx 16 bit fixed point
  DSP560xx 24 bit, fixed point
  DSP96002 32 bit, floating point

- **Texas Instruments** (www.ti.com)
  TMS320Cxx 16 bit fixed point
  TMS320Cxx 32 bit floating point

2. The TMS320C6000 Family

In 1982, Texas Instruments (TI) introduced the TMS32010 — the first fixed- point DSP in the TMS320 family. Before the end of the year, Electronic Products magazine awarded the TMS32010 the title "Product of the Year". Today, the TMS320 family consists of many generations:

- C1x, C2x, C2xx, C5x, and C54x fixed-point DSPs
- C3x and C4x floating-point DSPs, and
- C8x multiprocessor DSPs.

The TMS320C6000™ digital signal processor (DSP) platform is part of the TMS320™ DSP family. Refer Figure 3. The TMS320C62x™ DSP generation and the TMS320C64x™ DSP generation comprise fixed-point devices in the C6000™ DSP platform, and the TMS320C67x™ DSP generation comprises floating-point devices in the C6000 DSP platform. The TMS320C62x and TMS320C64x™ DSPs are code- compatible. The TMS320C62x and TMS320C67x DSPs are code-compatible. All three DSPs use the VelociTI™ architecture, a high-performance, advanced VLIW (very long instruction word) architecture, making these DSPs excellent choices for multi-channel and multifunction applications.

Now there is a new generation of DSPs, the TMS320C6x™ generation, with performance and features that are reflective of Texas Instruments commitment to lead the world in DSP solutions.

- *4.1 General Features of the C6000*

With a performance of up to 6000 million instructions per second (MIPS) and an efficient C compiler, the TMS320C6x DSPs give system architects unlimited possibilities to differentiate their products. High performance, ease of use and

affordable pricing make the TMS320C6x generation the ideal solution for multi- channel, multifunction applications, such as:

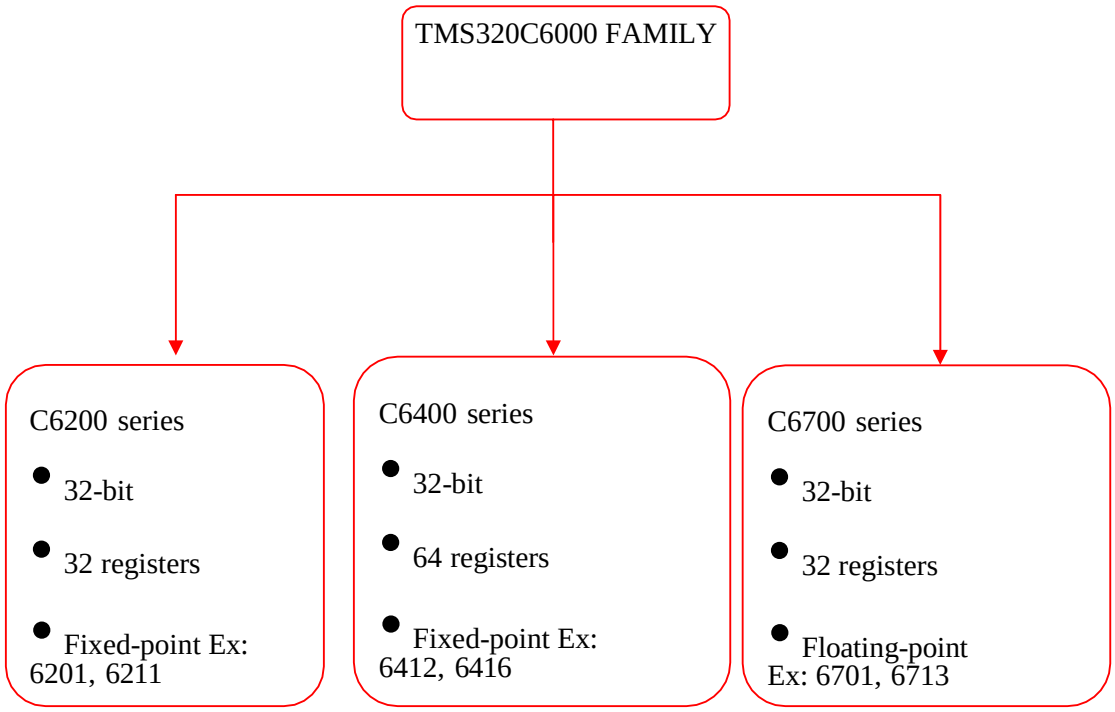


Figure 3: The TMS320C6000 Family

- Pooled modems
- Wireless local loop base stations
- Remote access servers (RAS)
- Digital subscriber loop (DSL) systems
- Cable modems
- Multi-channel telephony systems

The C6000 devices execute up to eight 32-bit instructions per cycle. The C62x/C67x device's core CPU consists of 32 general-purpose registers of 32-bit word length and eight functional units. The C64x core CPU consists of 64 general- purpose 32-bit registers and eight functional units. These eight functional units contain:

- Two multipliers
- Six ALUs

Features of the C6000 devices include:

- Advanced VLIW CPU with eight functional units, including two multipliers and six arithmetic units
- Executes up to eight instructions per cycle for up to ten times the performance of typical DSPs
- Allows designers to develop highly effective RISC-like code for fast development time
- Instruction packing
- Gives code size equivalence for eight instructions executed serially or in parallel
- Reduces code size, program fetches, and power consumption
- Conditional execution of all instructions
- Reduces costly branching
- Increases parallelism for higher sustained performance
- Efficient code execution on independent functional units
- Industry's most efficient C compiler on DSP benchmark suite
- Industry's first assembly optimizer for fast development and improved parallelization
- 8/16/32-bit data support, providing efficient memory support for a variety of applications
- 40-bit arithmetic options add extra precision for vocoders and other computationally intensive applications
- Saturation and normalization provide support for key arithmetic operations
- Field manipulation and instruction extract, set, clear, and bit counting support common operation found in control and data manipulation applications.

4.2 Features Unique to the C6700

The C67x has these additional features:

- Hardware support for single-precision (32-bit) and double-precision (64-bit) IEEE floating-point operations
- 32 x 32-bit integers multiply with 32- or 64-bitresult.

- The TMS320C67x™ floating-point DSP uses all of the instructions available to the TMS320C62x™, but it also uses other instructions that are specific to theC67x™.
- These specific instructions are for 32-bit integer multiply, doubleword load, and floating-point operations, including addition, subtraction, and multiplication.

4.3 Features Unique to the C6400

The C64x has these additional features:

- Each multiplier can perform two 16 x 16-bit or four 8 x 8 bit multiplies every clock cycle.
- Quad 8-bit and dual 16-bit instruction set extensions with data flow support
- Support for non-aligned 32-bit (word) and 64-bit (double word) memory accesses
- Special communication-specific instructions have been added to address common operations in error-correcting codes.
- Bit count and rotate hardware extends support for bit-level algorithms.

3. Processor Architectures

One of the biggest bottlenecks in executing DSP algorithms is transferring information to and from memory. This includes *data*, such as samples from the input signal and the filter coefficients, as well as *program instructions*, the binary codes that go into the program sequencer. For example, suppose we need to multiply two numbers that reside somewhere in memory. To do this, we must fetch three binary values from memory, the numbers to be multiplied, plus the program instruction describing what to do.

5.1 Von Neumann Architecture

The Figure 4 shows how this seemingly simple task is done in a traditional microprocessor. This is often called **Von Neumann architecture**, after the brilliant American mathematician John Von Neumann (1903-1957). Von Neumann guided the mathematics of many important discoveries of the early twentieth century. His many achievements include: developing the concept of a stored program computer, formalizing the mathematics of quantum mechanics, and work on the atomic bomb. If it was new and exciting, Von Neumann was there!

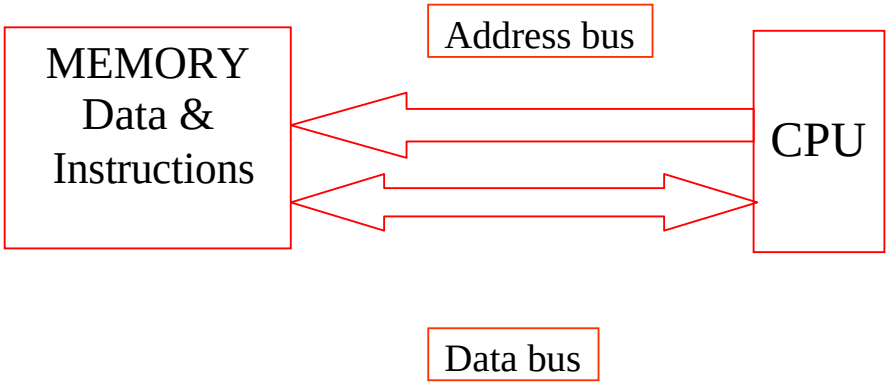


Figure 4: Von Neumann Architecture

As shown in Figure 4, Von Neumann architecture contains a single memory and a single bus for transferring data into and out of the central processing unit (CPU). Multiplying two numbers requires at least three clock cycles, one to transfer each of the three numbers over the bus from the memory to the CPU. We don't count the time to transfer the result back to memory, because we assume that it remains in the CPU for additional manipulation (such as the sum of products in an FIR filter).

The Von Neumann design is quite satisfactory when you are content to execute all of the required tasks in serial. In fact, most computers today are of the Von Neumann design. We only need other architectures when very fast processing is required, and we are willing to pay the price of increased complexity.

5.2 Harvard Architecture

This leads us to the **Harvard** architecture, shown in Figure 5. This is named for the work done at Harvard University in the 1940s under the leadership of Howard Aiken (1900-1973). As shown in this illustration, Aiken insisted on separate memories for data and program instructions, with separate buses for each. Since the buses operate independently, program instructions and data can be fetched at the same time, improving the speed over the single bus design. Most present day DSPs use this quadruple bus architecture.

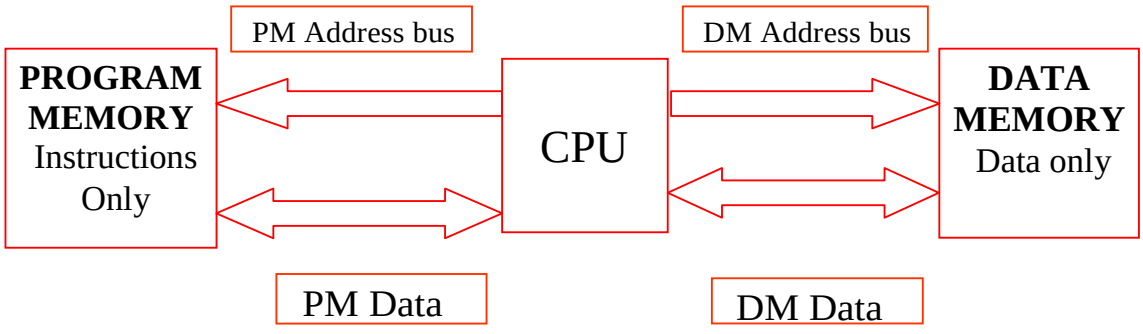


Figure 5: Harvard Architecture

PM: Program Memory

DM: Data Memory

A handicap of the basic Harvard design is that the data memory bus is busier than the program memory bus. When two numbers are multiplied, two binary values (the numbers) must be passed over the data memory bus, while only one binary value (the program instruction) is passed over the program memory bus.

- *5.3 VLIW Architecture*

VLIW stands for Very Long Instruction Word. This architecture was introduced by Texas Instruments. As the name itself denotes, we fetch a "long instruction". Meaning, in the C6000, **eight** instructions (this is definitely long!) are always fetched in every clock cycle. This constitutes a *fetch packet*.

A traditional VLIW architecture consists of multiple execution units running in parallel, performing multiple instructions during a single clock cycle. Refer Figure 6. Here there are three ALUs (i.e. execution units) that share the same program and data memory. Each ALU has its own address and data bus which is independent of all other buses.

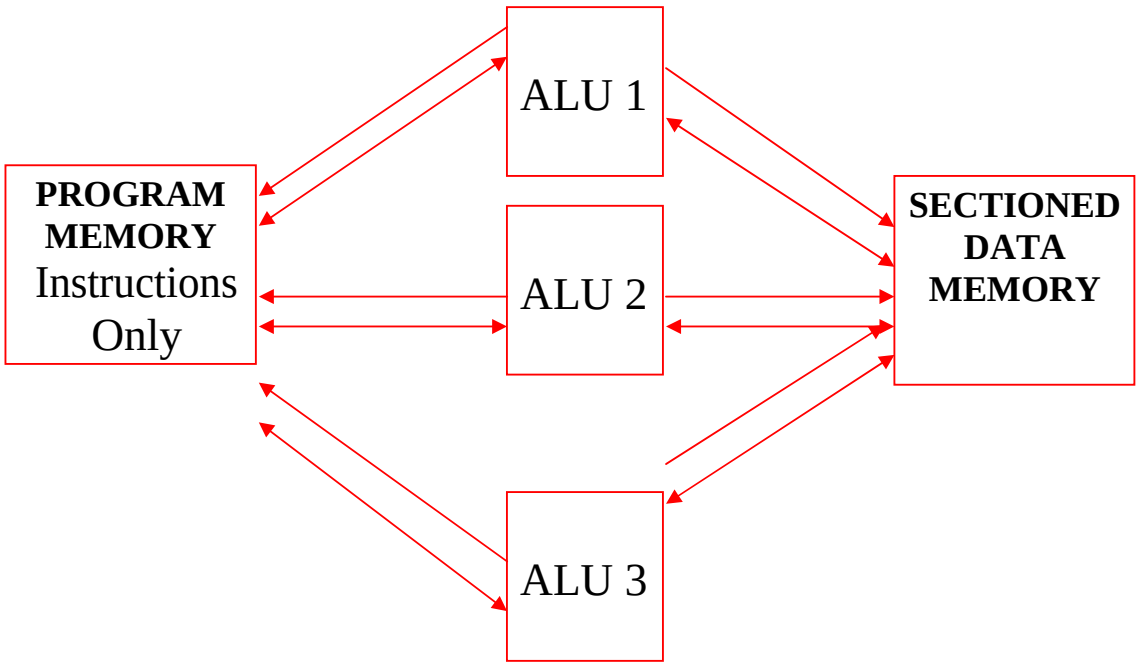


Figure 6: VLIW Architecture

A major difference between VLIW and the previous architectures is the presence of *sectioned* data memory. The memory is divided into different sections such as stack, heap, const, text, var (variables), args (arguments), cio (I/O in C language), etc. The user can explicitly decide the beginning and end of each memory section, which is not possible in previous architectures.

5.4 The VelociTI™ Architecture

The VelociTI architecture of the C6000 platform of devices makes them the first off-the-shelf DSPs to use advanced VLIW to achieve high performance through increased instruction-level parallelism.

VelociTI is a highly deterministic architecture, having few restrictions on how or when instructions are fetched, executed, or stored. VelociTI's advanced features include:

- Instruction packing: reduced code size
- All instructions can operate conditionally: flexibility of code
- Variable-width instructions: flexibility of data types
- Fully pipelined branches: zero-overhead branching.

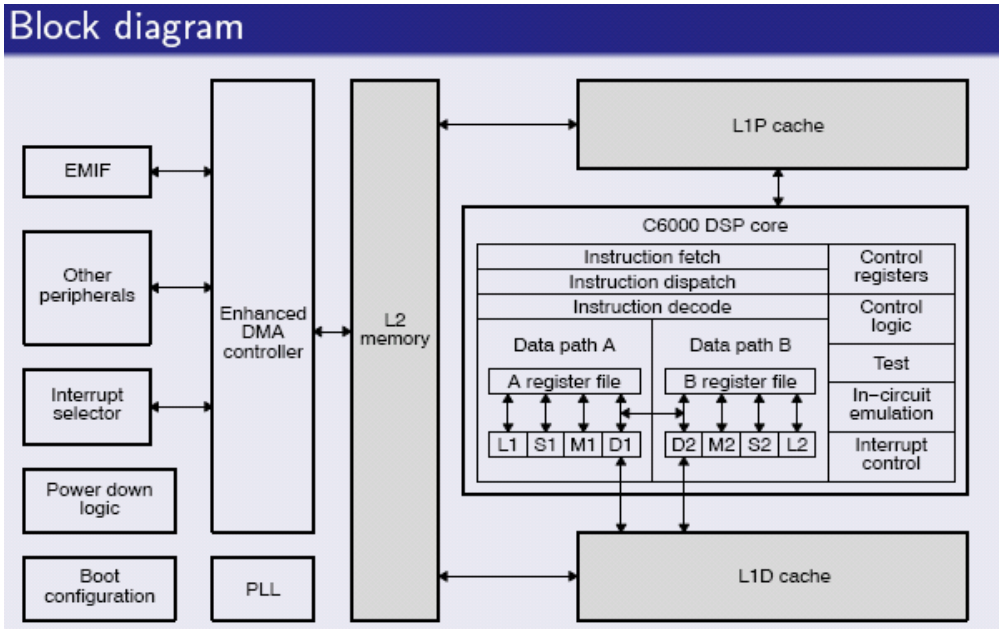


Figure 7: Block Diagram of the C6000

Other on-board peripherals include:

(a) Enhanced Direct Memory Access (EDMA) controller

(b) External Memory Interface (EMIF)

(c) Interrupt selector

(d) Power-down logic

(e) Phase-Locked Loop (PLL) controller

4. The C6000 DSP Core

The DSP core consists of registers and the functional units which can be programmed by the user. The functions of the various components which make-up the C6000 core (Figure 8) are explained next.

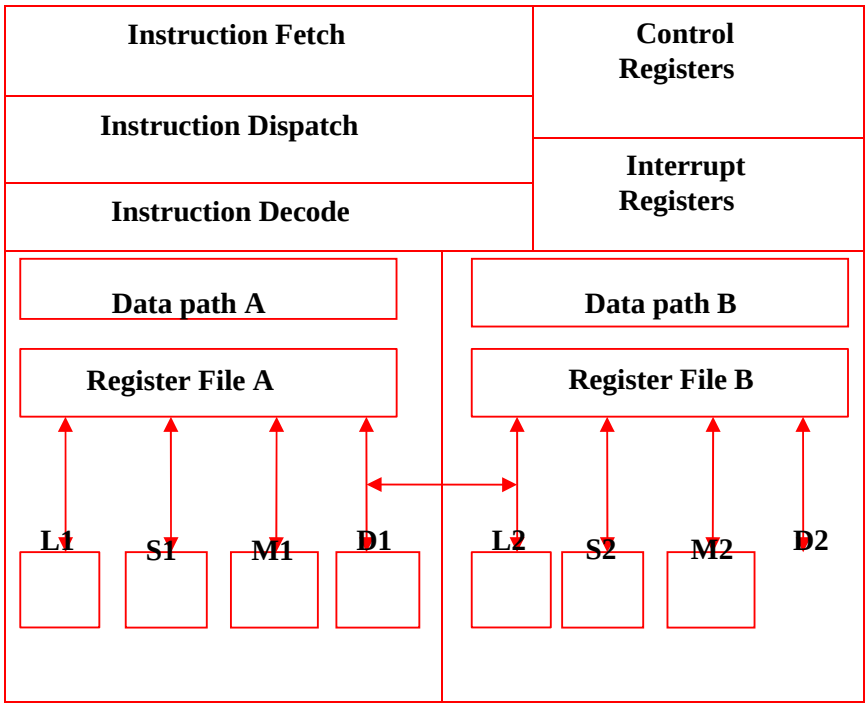


Figure 8: The C6000 Core

The C6000 CPU, shown in Figure 8, is common to all the C62x/C64x/C67x devices. The CPU contains:

- Program fetch unit
- Instruction dispatch unit, advanced instruction packing (C64 only)
- Instruction decode unit

- Two data paths, each with four functional units

- 32 32-bit registers, 64 32-bit registers (C64 only)
- Control registers
- Control logic
- Test, emulation, and interrupt logic

The program fetch, instruction dispatch, and instruction decode units can deliver up to eight 32-bit instructions to the functional units every CPU clock cycle. The processing of instructions occurs in each of the two data paths (A and B), each of which contains four functional units (.L, .S, .M, and .D) and 16 32-bit general- purpose registers for the C62x/C67x and 32 32-bit general purpose registers for the C64x. The data paths are described in more detail in section 6.1.A control register file provides the means to configure and control various processor operations. To understand how instructions are fetched, dispatched, decoded, and executed in the data path, see Section 8, "Pipelining and Parallelism".

Internal Memory

The C62x, C64x & C67x have a 32-bit, byte-addressable address space. Internal (on-chip) memory is organized in separate data and program spaces. When off-chip memory is used, these spaces are unified on most devices to a single memory space via the external memory interface (EMIF). The C62x and C67x have two 32-bit internal ports to access internal data memory. The C64x has two 64-bit internal ports to access internal data memory. The C62x, C64x & C67x have a single internal port to access internal program memory, with an instruction-fetch width of 256 bits.

- *6.1 Data path and Control*

Register Files

There are two general-purpose register files (A and B) in the C6000™ data paths. For the C62x/C67x DSPs, each of these files contains 16 32-bit registers (A0– A15 for file A and B0–B15 for file B). The general-purpose registers can be used for data; data address pointers, or condition registers. The C64x DSP register file

doubles the number of general-purpose registers that are in the C62x/C67x cores, with 32 32-bit registers (A0–A31 for file A and B0–B31 for file B).

The C62x/C67x general-purpose register files support data ranging in size from packed 16-bit data through 40-bit fixed-point and 64-bit floating point data. Values larger than 32 bits, such as 40-bit long and 64-bit float quantities are stored in register pairs (Figure 9). In these the 32 LSBs of data are placed in an even- numbered register and the remaining 8 or 32 MSBs in the next upper register (which is always an odd-numbered register). The C64x register file extends this by additionally supporting packed 8-bit types and 64-bit fixed-point data types. (The C64x does not directly support floating-point data types.) Packed data types store either four 8-bit values or two 16-bit values in a single 32-bit register, or four 16-bit values in a 64-bit register pair.

Figure 10 illustrates the register storage scheme for 40-bit long data. Operations requiring a long input ignore the 24 MSBs of the odd-numbered register. Operations producing a long result fill the 24 MSBs of the odd-numbered register with zeros. The even-numbered register is encoded in the opcode.

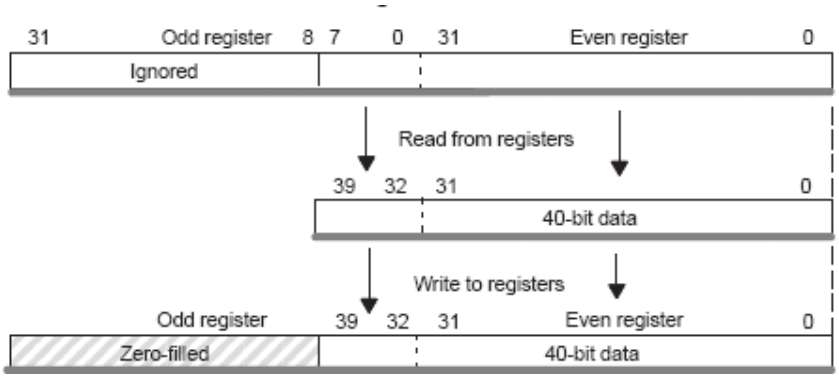


Figure 10: 40-bit long integer

Register Files		Applicable devices
A	**B**	
A1:A0	B1:B0	C62x, C64x and C67x
A3:A2	B3:B2	
A5:A4	B5:B4	
A7:A6	B7:B6	
A9:A8	B9:B8	
A11:A10	B11:B10	
A13:A12	B13:B12	
A15:A14	B15:B14	
A17:A16	**B17:B16**	**C67x only**
A19:A18	**B19:B18**	
A21:A20	**B21:B20**	
A23:A22	**B23:B22**	
A25:A24	**B25:B24**	
A27:A26	**B27:B26**	
A29:A28	**B29:B28**	
A31:A30	**B31:B30**	

Figure 9: Register Pair Structure

Functional Units

The eight functional units in the C6000 data paths can be divided into two groups of four; each functional unit in one data path is almost identical to the corresponding unit in the other data path. The functional units are described in Figure 11.

Besides being able to perform all the C62x instructions, the C64x also contains many 8-bit to 16-bit extensions to the instruction set. For example, the **MPYU4** instruction performs four 8x8 unsigned multiplies with a single instruction on an .M unit. The **ADD4** instruction performs four 8-bit additions with a single

instruction on an .L unit. The additional C64x operations are shown in boldface in Figure 11.

Functional Unit	**Fixed-point Operations**	**Floating-point Operations**
.L unit (.L1 and .L2)	32/40-bit arithmetic and compare operations 32-bit logical operations Leftmost 1 or 0 counting for 32 bits Normalization count for 32 and 40 bits **Byte shifts** **Data packing/unpacking 5-bit constant generation Dual 16-bit arithmetic Quad 8-bit arithmetic Dual 16-bit min/max** **Quad 8-bit min/max**	Arithmetic operations DP→SP, INT→DP, INT→SP conversion operations
.S unit (.S1 and .S2)	32-bit arithmetic operations 32/40-bit shifts and 32-bit bit-field operations 32-bit logical operations Branches Constant generation Register transfers to/from control register file (.S2 only) **Byte shifts** **Data packing/unpacking** **Dual 16-bit compare Quad 8-bit compare Dual 16-bit shift** **Dual 16-bit saturated arithmetic** **Quad 8-bit saturated arithmetic**	Compare Reciprocal and reciprocal square-root Operations Absolute value operations SP→DP conversion operations

.M unit (.M1 and .M2)	16 x 16 multiply operations **16 x 32 multiply operations** **Quad 8 x 8 multiply operations** **Dual 16 x 16 multiply operations** **Dual 16 x 16 multiply with add/subtract operations Quad 8 x 8 multiply with add Bit expansion** **Bit interleaving/de-interleaving** **Variable shift operations Rotation** **Galois Field Multiply**	32 x 32-bit fixed-point multiply operations Floating-point multiply operations
.D unit (.D1 and .D2)	32-bit add, subtract, linear and circular address calculation Loads and stores with 5-bit offset Loads and stores with 15-bit offset **Load and store double words with 5-bit constant** **Load and store non-aligned words and double words** **5-bit constant generation** **32-bit logical operations**	Load doubleword with 5-bit constant offset

Figure 11: Functional units and their operations

Control Register File

One unit (.S2) can read from and write to the control register file, as shown in this section. Figure 12 lists the control registers contained in the control register file and describes each. If more information is available on a control register, the table lists where to look for that information. Each control register is accessed by the **MVC** instruction.

Additionally, some of the control register bits are specially accessed in other ways. For example, arrival of a maskable interrupt on an external interrupt pin, INT *m*, triggers the setting of flag bit IFR*m*. Subsequently, when that interrupt is processed, this triggers the clearing of IFR*m* and the clearing of the global interrupt enable bit, GIE. Finally, when that interrupt processing is complete, the B IRP instruction in the interrupt service routine restores the pre-interrupt value of the GIE. Similarly, saturating instructions like SADD set the SAT (saturation) bit in the CSR (Control Status Register).

Abbr.	**Register Name**	**Description**
AMR	Addressing mode register	Specifies whether to use linear or circular addressing for each of eight registers; also contains sizes for circular addressing
CSR	Control status register	Contains the global interrupt enable bit, cache control bits, and other miscellaneous control and status bits
IFR	Interrupt flag register	Displays status of interrupts
ISR	Interrupt set register	Allows manually setting pending interrupts
ICR	Interrupt clear register	Allows manually clearing pending interrupts
IER	Interrupt enable register	Allows enabling/disabling of individual interrupts
ISTP	Interrupt service table pointer	Points to the beginning of the interrupt service table
IRP	Interrupt return pointer	Contains the address to be used to return from a maskable interrupt
NRP	Nonmaskable interrupt return pointer	Contains the address to be used to return from a nonmaskable interrupt
PCE1	Program counter, E1 phase	Contains the address of the fetch packet that is in the E1 pipeline stage

Figure 12: Control Registers

- *6.2 Instruction Set Mapping*

Figure 13 shows the mapping between instruction set and the functional units.

.L Unit	.M Unit	.S Unit		.D unit
ABS	MPY	ADD	SET	ADD
ADD	MPYU	ADDK	SHL	ADDAB
ADDU	MPYUS	ADD2	SHR	ADDAH
AND	MPYSU	AND	SHRU	LDB
CMPEQ	MPYH	B disp	SSHL	LDBU
CMPGT	MPYHU	B IRP	SUB	LDH
CMPGTU	MPYHUS	B NRP	SUBU	LDHU
CMPLT	MPYHSU	B reg	SUB2	LDW
CMPLTU	MPYHL	CLR	XOR	MV
LMBD	MPYHLU	EXT	ZERO	STB
MV	MPYHULS	EXTU		STH
NEG	MPYHSLU	MV		STW
NORM	MYPLH	MVC		SUB
NOT	MPYLHU	MVK		SUBAB
OR	MPYLUHS	MVKH		SUBAH
SADD	MPYLSHU	MVKLH		SUBAW
SAT	SMPY	NEG		ZERO
SSUB	SMPYHL	NOT		
SUB	SMPYLH	OR		
SUBU	SMPYH			
SUBC				
XOR				
ZERO				

Figure 13: Instruction Set Mapping

The fact that each functional unit can execute only a particular instruction is extremely important when writing assembly programs. This information must be entered in every assembly instruction.

- *6.3 Register Usage*

All instructions are conditional instructions. If no condition is specified, it is *always* executed. Conditional instructions are represented in code by using square brackets, [ ], surrounding the condition register name. The following execute packet contains two ADD instructions in parallel. The first ADD is conditional on B0 being nonzero. The second ADD is conditional on B0 being zero. The character "!" (exclamation mark) indicates the inverse of the condition.

```
[  B0  ] ADD.L1 A1,A2,A3        ;executes if B0 is nonzero
|| [ ! B0 ] ADD .L2 B1,B2,B3     ;executes if B0 = 00000000h
```

The above instructions are mutually exclusive. This means that only one will execute. Only A1, A2, B0, B1 and B2 registers can be used as conditional registers. No two instructions within the same execute packet can use the same resources. Also, no two instructions can write to the same register during the same cycle.

The registers A4, A5, A6, A7, B4, B5, B6 and B7 can be used for circular addressing. No other registers can be used for this purpose. Also some registers are assigned some special purpose:

- B15 is the Stack pointer (SP)
- A15 is the Frame Pointer (FP)
- A14 is the Data Page Pointer (DPP)

The programmer must avoid using A14, A15 and B15 as much as possible.

5. The Addressing Modes

Addressing Modes are methods used to specify the address of an operand in assembly instructions.

- *7.1 Types of Addressing Modes*

The addressing modes on the C62x, C64x, and C67x are

- Linear mode
- Circular mode using BK0
- Circular mode using BK1

The mode is specified by the addressing mode register, or AMR (Figure 12 and section 7.3). All registers can perform linear addressing. Only eight registers can perform circular addressing: A4–A7 are used by the .D1 unit and B4–B7 are used by the .D2 unit. No other units can perform circular addressing. LDB(U)/LDH(U)/LDW, STB/STH/STW, ADDAB/ADDAH/ADDAW/ADDAD, and SUBAB/SUBAH/SUBAW

instructions all use the AMR to determine what type of address calculations are performed for these registers.

If no addressing mode is explicitly written into the AMR then linear addressing mode is used by default. The different ways of addressing modes are:

Register Addressing Mode: The operand is the contents of a processor register; the name of the register is given in the instruction.

```
Ex:     ADD .L1 A1, A2, A3      ; A1 + A2 = A3
        SUB  .L2 B1, B2, B6     ; B1 – B2 = B6
```

Note that the functional unit is a must in writing assembly instructions.

Immediate Addressing Mode: The operand is a numeric constant which is directly specified in the instruction.

```
Ex:     ADD .L1 A1, 20, A3      ; A1 + 20 = A3
        SUB  .L2 B1, 15, B6     ; B1 – 15 = B6
```

Indirect Addressing Mode: The effective address of the operand is the contents of a register that appears in the instruction. An asterisk (*) is used as an indirection operator. Also increment (++) and decrement (- -) operators are supported.

```
Ex:     LDW .D2 *B0, B1
        STW .D1A1,*A2++
```

- *7.2 Circular Addressing Mode*

A circular buffer is a fixed number of memory locations which is circular i.e. cyclic in nature. If you increment the address of last memory location in the buffer it points to the first location. Note that in linear mode, incrementing the last memory location in the memory map causes an overflow error.

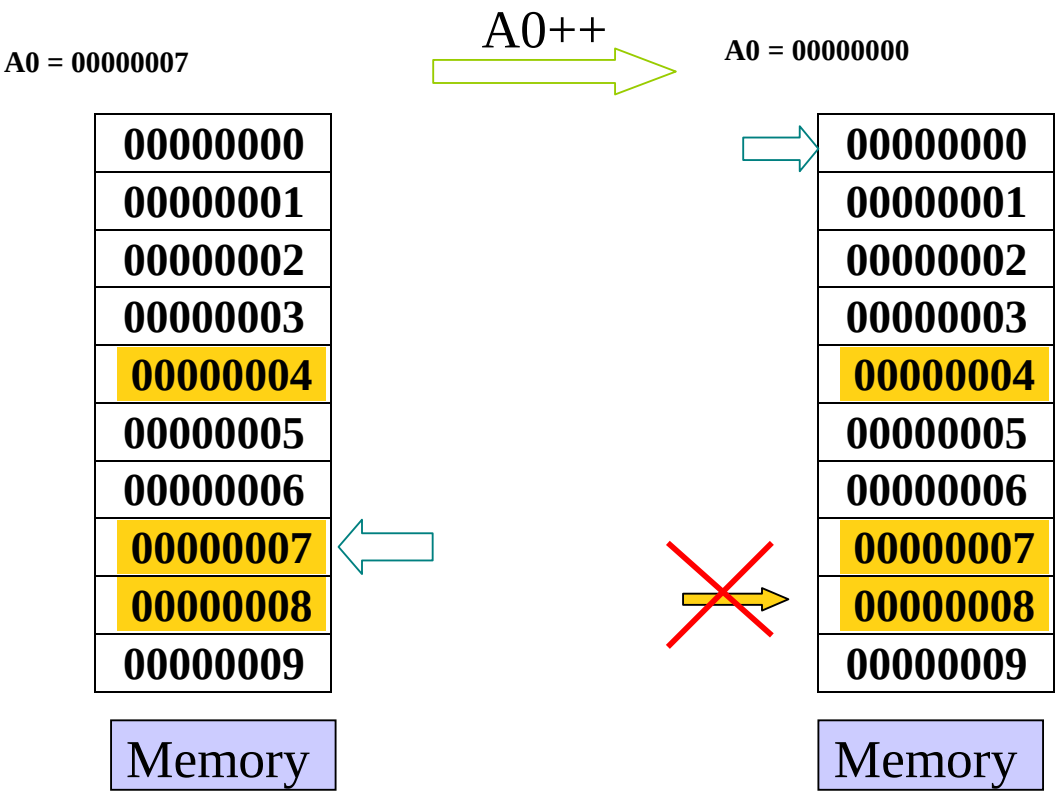


Figure 14: Circular Addressing

Consider the example shown in Figure 14. Let the block size of the circular buffer be N = 8. Also let A0 = 00000007h initially. Assuming that A0 is set for circular mode in the AMR, we increment A0 by 1 (A0++). If it were linear addressing A0 would contain 00000008h. But the block size is set to 8 itself (in circular mode) and since modulo (8, N) = 0, the A0 now points to zeroth memory location.

- *7.3 Addressing Mode Register*

For each of the eight registers (A4–A7, B4–B7) that can perform linear or circular addressing, the AMR specifies the addressing mode. A 2-bit field for each register selects the address modification mode: linear (the default) or circular mode. With circular addressing, the field also specifies which BK (block size) field to use for a circular buffer. In addition, the buffer must be aligned on a byte boundary equal to the block size. The mode select fields and block size fields are shown in Figure 15 and the mode select field encoding is shown in Figure 16 (next page).

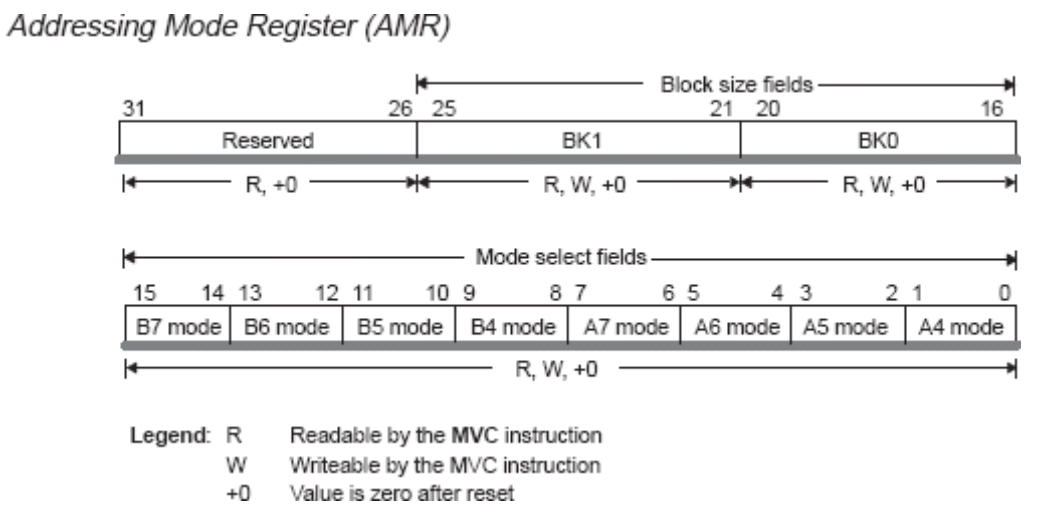


Figure 15: Addressing Mode Register

The reserved portion of AMR is always 0. The AMR is initialized to 0 at reset. The block size fields, BK0 and BK1, contain 5-bit values used in calculating block sizes for circular addressing.

$$\textit{Block size (in bytes)} = 2^{(N+1)}$$

where N is the 5-bit value in BK0 or BK1

Mode	Description
00	Linear mode (default at reset)
01	Circular Addressing using BK0 field
10	Circular Addressing using BK1 field
11	Reserved

Figure 16: Mode Select Field Encoding

Pipelining and Parallelism

Pipelining is a technique of executing several instructions concurrently. The highlights of the C6000 pipeline are:

- The pipeline can dispatch eight parallel instructions every cycle.
- Parallel instructions proceed simultaneously through each pipeline phase.
- Serial instructions proceed through the pipeline with a fixed relative phase difference between instructions.

All instructions require the same number of pipeline phases for fetch and decode, but require a varying number of execute phases.

8.1 The C6000 Pipeline Phases

The pipeline phases are divided into three stages:

- Fetch
- Decode
- Execute

All instructions in the C62x/C64x instruction set flow through the fetch, decode, and execute stages of the pipeline. The fetch stage of the pipeline has four phases for all instructions, and the decode stage has two phases for all instructions. The execute stage of the pipeline requires a varying number of phases, depending on the type of instruction. The stages of the C6000 fixed-point pipeline are shown in Figure 17.

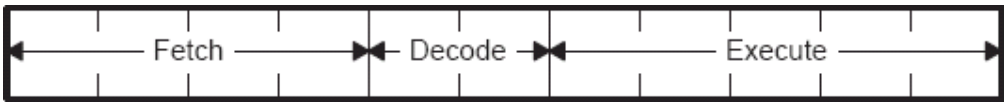


Figure 17: Fixed-point Pipeline

The different phases in each stage of the pipelining are shown in Figure 18.

Stage	Phase	Symbol
Program Fetch	Program Address Generation	PG
	Program Address Send	PS
	Program Access Wait	PW
	Program Fetch Packet Receive	PR
Program Decode	Instruction Dispatch	DP
	Instruction Decode	DC
Program Execute	Execute Packet 1	E1
	:	:
	Execute Packet 5	E5

Figure 18: The Phases of Pipeline

During the **PG** phase, the program address is generated in the CPU. In the **PS** phase, the program address is sent to memory. In the **PW** phase, a memory read occurs. Finally, in the **PR** phase, the fetch packet is received at the CPU. In the **DP** phase of the pipeline, the fetch packets are split into execute packets. In the **DC** phase, the source registers, destination registers, and associated paths are decoded for the execution of the instructions in the functional units. The execute portion of the fixed-point pipeline is subdivided into five phases (**E1–E5**). Different types of instructions require different numbers of these phases to complete their execution.

Figure 19 shows an example of the pipeline flow of consecutive fetch packets that contain eight parallel instructions. In this case, where the pipeline is full, all instructions in a fetch packet are in parallel and split into one execute packet per fetch packet. The fetch packets flow in lockstep fashion through each phase of the pipeline. For example, examine cycle 7 in Figure 19. When the instructions from FP n reach E1, the instructions in the execute packet from FP (n +1) are being decoded. FP (n + 2) is in dispatch while FPs (n + 3), (n + 4), (n + 5), and (n + 6) are each in one of four phases of program fetch.

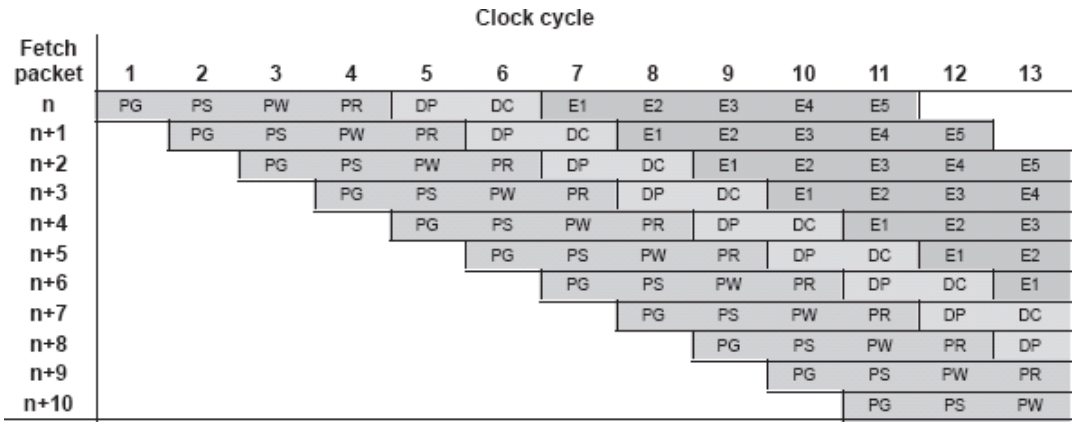


Figure 19: Pipeline Operation: One Execute Packet per Fetch Packet

- *8.2 Vector Summation: An Example*

Consider a simple C function to add two *N*-element arrays **a** and **b,** element- by-element.

$$N \square 1$$

$$c[n] \square \underset{n \square 0}{\square} (a[n] \square b[n])$$

```
void sum (int *a, int *b, int *c, int N)
{
        int n;
        for(n=0;n<N;n++)            c[n]=a[n]+b[n] ;
 }
```

The inner loop can be implemented using the following assembly program:

```
                mvk   .S1 10, A1          ; N = 10
LOOP :          ldw   .D1 *a++, A0        ;A0 = a[n]
                ldw   .D2 *b++, B0        ; B0 = b[n]
                add     .L2 A0,B0,B1      ;
                stw   .D2 B1, *c++        ;c[n] = A0+B0 , n++
                sub     .L1 A1, 1 , A1    ;N = N-1
     [ A1 ]      b      LOOP              ;branch to LOOP if A1 != 0
```

We shall see next how the program operates with and without pipelining.

- *8.3 Vector Summation without Pipelining*

When pipelining is not applied, the resources (i.e. functional units) are utilized as shown below (Figure 20):

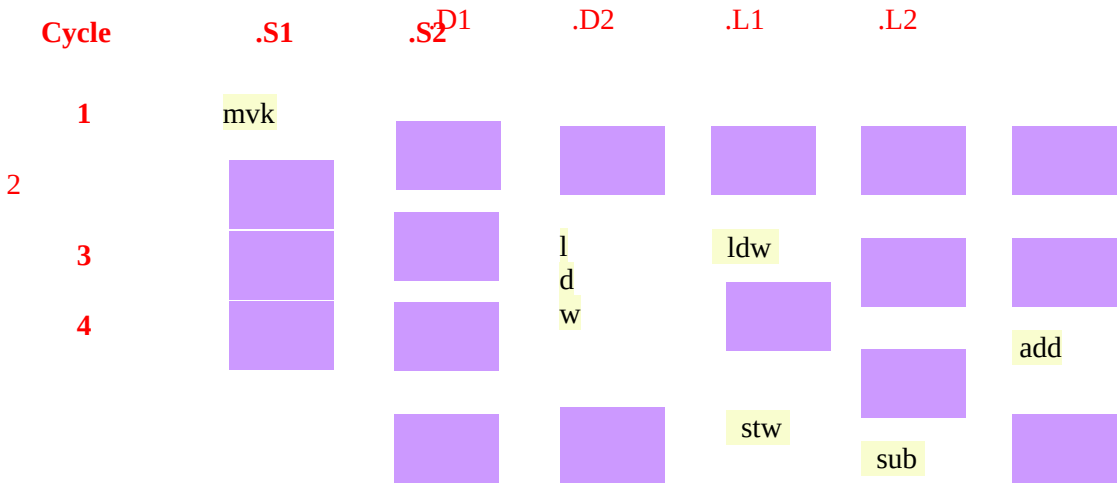


Figure 20: Resource Utilization

The first clock cycle simple loads the constant *N*=10. This is executed only once. The next three clock cycles keep repeating *N* times. In the second clock cycle only S1 and S2 are used. The remaining units are idle. Similarly in the third and fourth clock cycles, not all units are used. This leads to ineffective utilization of the functional units. Hence pipelining is needed. The entire program thus takes 32 clock cycles.

- *8.4 Vector Summation with Pipelining*

Software pipelining is a technique used to schedule instructions from a loop so that multiple iterations of the loop execute in parallel. Pipelining can be enabled in CCStudio™ using the option –o1, –o2 or –o3 (recommended) while running the compiler cl6x. Figure 21 illustrates a software pipelined loop. The stages of the loop are represented by A, B, C, D, and E. In this figure, a maximum of five iterations of the loop can execute at one time. The shaded area represents the *loop kernel*. In the

loop kernel, all five stages execute in parallel. The area above the kernel is known as the *pipelined loop prolog*, and the area below the kernel is known as *the pipelined loop epilog*.

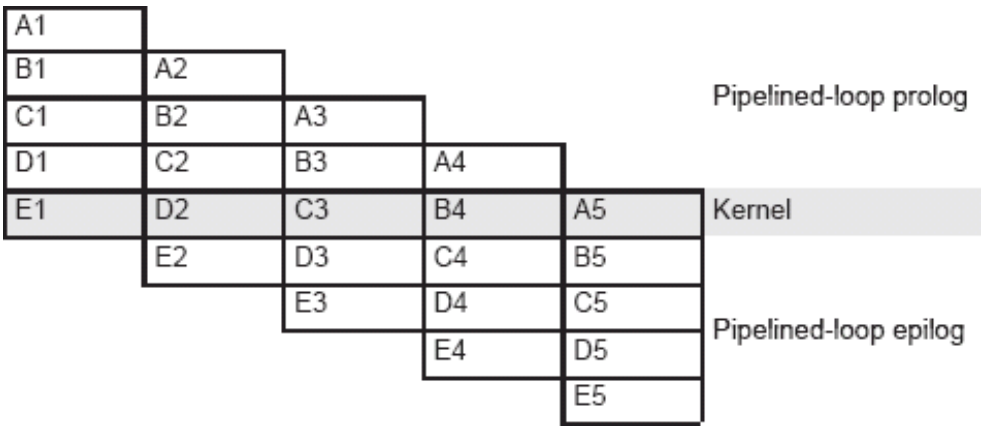


Figure 21: Software-pipelined loop

When pipelining is enabled, instead of performing one summation in one execution of the loop, the compiler *unrolls* the loop so that two summations are performed in one execution of the loop. The equivalent C function as seen by the compiler is:

```
void sum (int *a, int *b, int *c, int N)
{
        int n;
  for(n=0;n<N;n+=2)
        { c[n]     =  a[n]    + b[n] ;
          c[n+1] = a[n+1] + b[n+1];
        }
}
```

The pipelined loop kernel is shown below (as generated by the cl6x compiler).

```
LOOP:
; <PIPED LOOP KERNEL> ADD .L1X B7,
                A3, A3
||   [ B0]      B       .S1      L2
||              LDH   .D1T1    *++A4(4),A3
||              LDH   .D2T2    *++B4(4),B7
     [!A1]      STH   .D1T1     A3,*++A0(4)
||              ADD   .L2X      B6,A5,B6
||              LDH   .D2T2    *+B4(2),B6
     [ A1]      SUB   .L1       A1,1,A1
```

```
||   [!A1]      STH .D2T2 B6,*++B5(4)
||   [ B0]      SUB   .L2       B0,1,B0
||              LDH   .D1T1    *+A4(2),A5
```

Instructions marked with double bars (||) execute simultaneously in a single clock cycle. As a result, the entire program takes only 14 clock cycles now. Previously it was 32 cycles.

6. The Memory Map of C6000

The memory of any C6000 device in general consists of the following divisions (Figure 22):

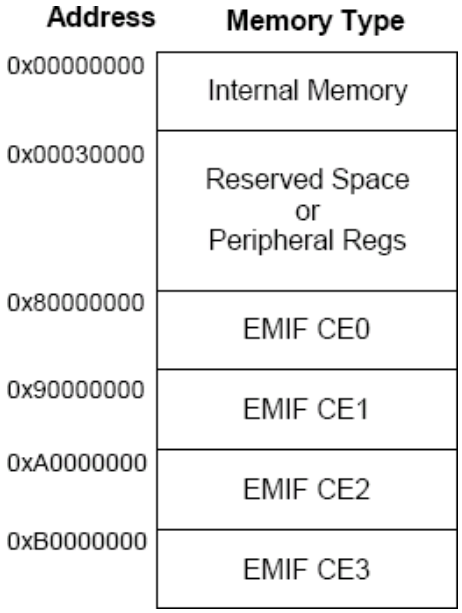


Figure 22: Memory Map

Its features can be summarized as follows:

- Internal Memory of up to 2 MB with 1 MB being commonly found.
- External Memory Interface (EMIF) can support up to 24 MB
- External interface can be SDRAM, synchronous burst RAM
- Two-level (L1 and L2) cache up to 512 KB
- Separate cache for PMEM (program memory) and DMEM (data memory) The

actual memory map depends from device to device and cannot be generalized.

7. The Peripherals of C6000

The functions of various peripherals supported by the C6000 are:

(a) DMA (Direct Memory Access) Controller transfers data between address ranges in the memory map without intervention by the CPU. The DMA controller has four programmable channels and a fifth auxiliary channel.

(b) EDMA (Enhanced DMA) Controller performs the same functions as the DMA controller. The EDMA has 16 programmable channels, as well as a RAM space to hold multiple configurations for future transfers.

(c) HPI (Host Port Interface) is a parallel port through which a host processor can directly access the CPU's memory space. The host device has ease of access because it is the master of the interface. The host and the CPU can exchange information via internal or external memory. In addition, the host has direct access to memory-mapped peripherals.

(d) Expansion bus is a replacement for the HPI, as well as an expansion of the EMIF. The expansion provides two distinct areas of functionality (host port and I/O port) which can co-exist in a system. The host port of the expansion bus can operate in either asynchronous slave mode, similar to the HPI, or in synchronous master/slave mode. This allows the device to interface to a variety of host bus protocols. Synchronous FIFOs and asynchronous peripheral I/O devices may interface to the expansion bus.

(e) McBSP (Multi-channel Buffered Serial Port) is based on the standard serial port interface found on the TMS320C2000 and C5000 platform devices. In addition, the port can buffer serial samples in memory automatically with the aid of the

DMA/EDMA controller. It also has multi-channel capability compatible with the T1, E1, SCSA, and MVIP networking standards.

(f) Timers in the C6000 devices are two 32-bit general-purpose timers used for these functions:

- Time events
- Count events
- Generate pulses
- Interrupt the CPU
- Send synchronization events to the DMA/EDMA controller.

(g) Power-down logic allows reduced clocking to reduce power consumption. Most of the operating power of CMOS logic dissipates during circuit switching from one logic state to another. By preventing some or all of the chip's logic from switching, you can realize significant power savings without losing any data or operational context.

8. The DaVinci™ Technology

The Texas Instruments DaVinci™ Technology combines TI's offering of DSP and tools for developing a broad spectrum of optimized digital video end equipments.

- *11.1 The Origin of the DaVinci ™ Effect*

A typical multimedia system such as a digital video recorder or digital camera can be split roughly into two pieces: control and media. The *control* portion handles tasks such as memory card or hard disk access, user interface, and networking, while the *media* portion covers tasks such as encoding and decoding of audio and video. A general-purpose processor performs well in control tasks, but all but the fastest of these processors are not sufficiently powerful for intensive media-related tasks such as real-time, high-quality video encoding. A DSP, on the other hand, is superb at the repetitive, easily parallelizable media-related tasks, but usually performs poorly in control-related jobs.

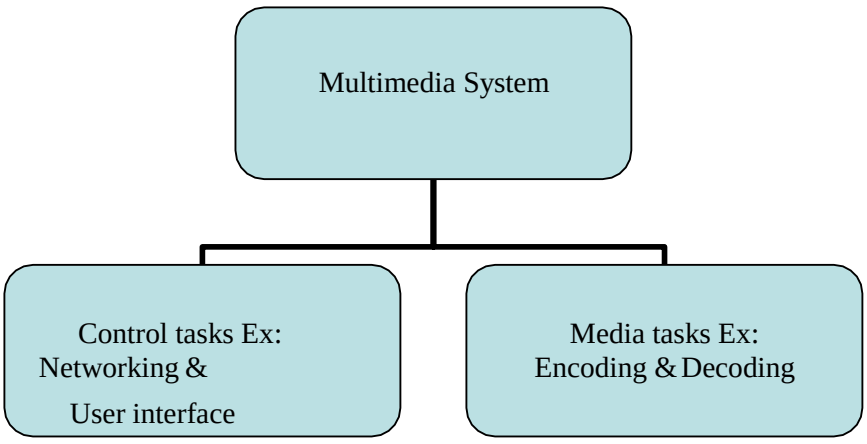


Figure 23: Multimedia System

The idea behind DaVinci is that by using both a general-purpose processor and a DSP, the control and media portions can both be executed by processors that excel at their respective tasks. The integration of these two components into one chip simplifies the system design and allows for more efficient communication between the two components.

11.2 Existing Systems

DaVinci Technology has grown to support DSP only processors including TMS320DM6431, TMS320DM6433, TMS320DM6435 and TMS320DM6437.

Models

DM6443 = ARM9 + Texas Instruments TMS320C64x+ DSP + DaVinci Video (Decode) Video Accelerator and Networking for display

DM6446 = ARM9 + Texas Instruments TMS320C64x+ DSP + DaVinci Video (Encode and Decode) Video Accelerator and Networking for capture and display

11.3 Peripherals and Operating System

The DaVinci includes a number of on-chip peripherals. These include:

- Support for memory cards such as CompactFlash, SD Card and MMC
- ATA interface
- CCD Controller for digital camera/camcorder applications
- Connectivity, including USB 2.0 Host and Client modes, VLYNQ (interface for FPGA, Wireless LAN, PCI), EMAC (Ethernet MAC) with MDIO
- Enhanced DMA
- Interrupt controller
- Digital LCD controller
- Serial interfaces, including SPI, I^2C, and I^2S, UART
- Histogram, autofocus, autoexposure, and auto-white-balance (H3A) acceleration
- Image resize acceleration
- A/D and D/A converters for analog video input and output

The DSP in the DaVinci generally runs TI's **DSP/BIOS** RTOS. DSP/BIOS Link drivers run on both the ARM processor and the DSP to provide communication between the two. A number of operating systems support DaVinci and the DSP/BIOS Link drivers:

- Green Hills Software INTEGRITY RTOS
- Montavista Linux
- QNX Neutrino
- Windows CE
- DaVinci Linux OS is currently (2007) under development.
- **Blackhawk XDS560™** is a real-time debugger released on March 19, 2007. It captures even the toughest real-time hardware bugs.

11.4 Applications

High-end applications include:

- Stream live-video on to a portable, handheld device
- An on-board intelligence system in your car can record obstructions in front of the windscreen.
- Surveillance videos directly captured by a TV and transferred to a computer/controller

Other everyday applications which have a potential to exploit DaVinci are:

- 1200- to 56Kbps modems
- Digital Subscriber Loop
- X.25 packet switching
- Image Compression
- Homomorphic Processing
- Robotic Vision
- Pattern Recognition

9. Conclusions

- "Necessity is the mother of all inventions". Hence the origin and four generations of DSPs were first considered.
- The failure of Von Neumann architecture led to the Harvard architecture, which was further developed into VLIW architecture.
- The data path and control registers are a dominating feature of the C6000's architecture. Each instruction is mapped uniquely to every functional unit.
- The addressing modes, especially circular addressing, is extremely useful in implementing FIR filters and circular convolution.
- Pipelining is the watchword when it comes to parallel execution. Loop unrolling nearly doubles the execution speed.
- The DaVinci™ is a state-of-the-art technology that is designed with video and multimedia applications in mind.

References

Technical Documentation:

- *TMS320C6000 Programmer's Guide* (literature number 198G), Texas Instruments, August 2003. Describes ways to optimize C and assembly code for the TMS320C6000 DSPs and includes application program examples.

- *TMS320C6000 CPU and Instruction Set Reference Guide* (literature number SPRU189F), Texas Instruments, October 2004. Describes the C6000 CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

- *TMS320C6000 Optimizing C Compiler* (literature number SPRU187L), Texas Instruments, May 2004. Describes the C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI C source code and produces assembly language source code. The assembly optimizer helps to optimize assembly code.

- Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1999. A non-mathematical textbook that teaches DSP intuitively.

Newsletters:

- XML-RSS DaVinci™ feed from Texas Instruments.

Web Resources:

- http://en.wikipedia.org/wiki/Texas_Instruments_DaVinci: Describes the DaVinci™ technology (Chapter 11).
- http://en.wikipedia.org/wiki/Digital_signal_processor : History of DSPs (Chapter 2)
- http://www.thedavincieffect.com : Official website of DaVinci™ technology.