Chapter 7
Embedded Peripherals

This chapter immerses readers into the array of peripherals typically found in a microcontroller, while also discussing the concepts that allow understanding how to use them in any microprocessor-based system. The chapter begins by discussing how to use interrupts and timers in microcontrollers as support peripherals for other devices. These are the two most important support peripherals in any microcontroller as they form the basis for real-time, reactive operation of embedded applications.

The subjects of flash memory storage and direct memory access are also discussed in this chapter, with special attention to its use as a peripheral supporting for low-power operation in embedded systems. The MSP430 is used as the platform to provide practical insight into the usage of these peripherals.

7.1 Fundamental Interrupt Concepts

Interrupts provide one of the most useful features in microprocessors. As it was explained earlier, in the introductory discussions in Chaps. 3 and 4, interrupts provide an efficient alternative to handle the service requests of peripheral devices and external events in a computer system. When compared to the alternative of polling, interrupt servicing allows a much more efficient use of the CPU. This fact was analyzed in Chap. 3, when a comparison was made of the efficiency of polling versus interrupts (Sect. 3.9).

In addition to the CPU usage efficiency advantage, using interrupt servicing also provides the following advantages:

- Compact and modular code: Interrupt service routines (ISR) induce code modularity and software reusability.
- Reduced energy consumption: As ISRs lead to less CPU cycles, this has a direct impact in the amount of energy consumed by the application, particular when combined with low-power modes.

M. Jiménez et al., *Introduction to Embedded Systems*,
DOI: 10.1007/978-1-4614-3143-5_7,

- Faster response time: When many events and devices need to be served, well designed ISRs provide a quick response to the triggering event. Well coordinated priorities among sources will allow for a quick response with minimal processing overhead.

To have the ability of working with interrupts, the system designer must include both hardware and software considerations. Depending on the level of integration of the processor being used, the hardware components to support interrupts might need to be externally provided, as in the case of microprocessors, or can be found embedded in the chip as happens with most microcontrollers. In either case, once the hardware components are in place, dealing with interrupts requires their correct configuration and programming.

In this section we provide a discussion of fundamental concepts that help understanding how interrupts work in general. Specific treatment of the MSP430 interrupt structure is provided in Sect. 7.2, and guidelines and recommendations on how to design interrupt-based programs are the subject of Sect. 7.3.

7.1.1 Interrupts Triggers

Interrupts are mainly triggered by hardware events. Events such as a push-button depression, a threshold reached, and timer expiration, are few examples of events that might be configured to trigger interrupts. Once a hardware event has been configured to trigger interrupts, their occurrence is asynchronous and unpredictable. They can be triggered at any time. Thus, the software structure supporting the interrupts events, ISRs, variables, configurations, etc. must be written considering this fact.

Interrupt Request Sensitivity

The way an input pin in a processor detects an interrupt request can be either level sensitive or edge sensitive.

A level-sensitive interrupt input has an assertion-level that indicates an interrupt request is active. The assertion level can be either low or high. An active-low assertion level, the most commonly used, specifies an interrupt request by holding at a logic low level the interrupt request input to the processor. Asserted-high interrupt inputs will become activated by a high-level logic input.

Either type of assertion level requires the interrupt pin to remain asserted until an interrupt request is received or until service is rendered. This guarantees the request will be seen by the CPU when interrupt processing conditions are given. Some peripherals are designed to remove the interrupt request signal assertion upon access. For example, it is common for serial communication adapters to drop their receiver ready interrupt request when a read operation is made to their data-in buffer, but this is not a standard feature across all serial adapters. A designer might end-up working with one that needs to be explicitly turned-off from within the ISR. Therefore, the designer must verify in the device data what is the expected behavior of its interrupt request signal.

An edge-triggered interrupt input will respond to the transition detected in the interrupt request pin. The edge detection can be either with a rising or a falling edge of the input signal. To allow this functionality, the request needs to be latched, otherwise the request might go undetected.

Most modern microcontrollers allow for specifying the type of sensitivity desired for their external interrupt inputs. In these MCUs, a configuration register allows to choose between level- and edge-sensitive operation, the assertion level for the former, and the edge type for the latter. In the case of edge sensitivity some devices even allow triggering the interrupt request with an "any edge" option, meaning that either a rising or falling edge would trigger the request. The designer needs to verify the flexibility offered by the device at hand and ensure that the chosen configuration matches the output signal protocol from the requesting device.

Software Interrupts

An ISR may also be called by software, in which case we talk of a *software interrupt*. Unlike hardware triggered interrupts, software interrupts are predictable and become part of the normal program sequence. Being so, the invocation to a software interrupt is equivalent to a function call.

Many embedded systems are designed to perform the solely task of reading and processing data from one or more peripherals whenever there is new data. This is most efficiently done by interrupts. When the MCU is programmed just to respond to interrupts, the system is said to be *interrupt driven*.

7.1.2 Maskable Versus Non-Maskable Interrupts

Earlier, in Chap. 3, two types of interrupt requests were identified: maskable and non-maskable interrupt requests. Let's revisit these types here to see a few additional details.

A maskable interrupt request is one that can be masked by the CPU through the global interrupt enable (GIE) flag to ignore it. This is done by clearing the GIE in the processor status register (PSW). This action makes the CPU ignore all interrupt request from all maskable sources.

The GIE flag is a useful feature that allows the CPU to choose when to accept or reject interrupt requests. A device placing a maskable interrupt request to the CPU knows that its request was served when the CPU executes its designated ISR. Some systems might involve an *Interrupt Acknowledgement Signal*, but that depends on the interrupt management style of the processor architecture.

The second type is a non-maskable interrupt (NMI) request. This type of request cannot be masked by the CPU and therefore always has to be served, even if the GIE flag is clear. This type of request is reserved only for system critical events whose service cannot be postponed under any circumstance. One example of such an event could be a low-battery voltage condition in a portable device. Upon a critical battery level that could be detected by a voltage comparator, an NMI would be triggered

whose ISR would save the CPU state and any critical data held in registers and volatile memory to keep them from being lost and then shut the system down. Many battery operated laptops and cell phones nowadays use such a feature.

The vast majority of the interrupts served by a CPU are maskable interrupts. This is reflected by the fact the maskable interrupts are simply called *interrupts*. When a reference is made to a non-maskable interrupt then the reference is explicit calling it an NMI or non-maskable interrupt.

Interrupt Masking and Polling Support

Disabling or masking an interrupt request is a common and necessary situation in the management of interrupts. There are instances when the CPU might not be ready to manage interrupts, or simply it might not be appropriate or convenient to serve an interrupt. For example, just after a reset, before the system has been configured, the CPU cannot not accept interrupts if they are not configured yet. To this end, whenever the CPU is reset, its GIE is cleared and remains so until the program being executed explicitly enables it. Enabling the interrupt flag shall occur only after the whole interrupt structure has been configured.

Another instance when interrupts are disabled is when the CPU enters an ISR. Before begining the execution of an ISR, the GIE is cleared. This prevents other interrupts from interrupting an ongoing ISR. Upon returning from the ISR the GIE becomes re-enabled. The code within the ISR may include instructions to set the GIE flag, enabling the ISR interruption. However, this requires writing a code with re-entrance and possibly recursive capabilities. Most applications do not need such a prevision, and therefore in most cases it is recommended not to enable the global interrupt flag inside an ISR.

Besides disabling interrupts through the GIE flag, most hardware interfaces with the ability of triggering interrupt requests also include some mechanism for deactivating their interrupt capabilities. Consider for example, an interrupt managed push-button connected to the CPU via a GPIO port. The GPIO will include an interrupt enable flag (IEF) that allows to enable/disable its interrupt generating capability. Thus, for the push-button actually triggering an interrupt to the CPU, both the GIE in the CPU and the GPIO IEF in the push-button interface must be enabled.

Having interrupt enable flags in the interfaces themselves results convenient because they allow for individually enabling or disabling the interrupt generation ability of individual peripherals, as needed, without affecting the interrupt capability of other devices and their interfaces. Remember that when the GIE is disabled, all maskable interrupts become masked. Peripheral IEFs are frequently located within the interface control register(s).

Another feature closely related to an IEFs in device interfaces is the inclusion of status flags indicating when service requests (SRQ) have been placed. These flags are convenient for polled operation of the interface. Polled operation is possible by clearing the local IEF an interrogating the local SRQ flag.

Sometimes it becomes unavoidable to poll devices even when they are being operated by interrupt. This is a common situation arising when multiple events within a peripheral interface trigger a single, shared interrupt request signal. The activation

of the common IRQ signal notifies the CPU about the need for servicing an event in that peripheral interface, but to determine which specific event triggered the request, the first action to be performed within the ISR is to poll the device's SRQ flags to determine what event caused the trigger. A typical scenario where this situation can be found is in general purpose I/O ports with interrupt generation capability. Frequently, the GPIO is designed to give each pin the ability generating service requests, while the entire port has a single request line to the CPU. An ISR serving such a port must first poll the pins SRQ flags to determine which one triggered the request, to then proceed providing the appropriate service.

7.1.3 Interrupt Service Sequence

The sequence of events taking place when an interrupt request is accepted by the CPU were outlined in Sect. 3.9.2. The same sequence is represented here, in a pictorial form in Fig. 7.1.

Steps 1 through 6, graphically denoted in the figure above are explained below. The stack and GIE operations are denoted in subgraphs (a), (b), and (c). A set GIE

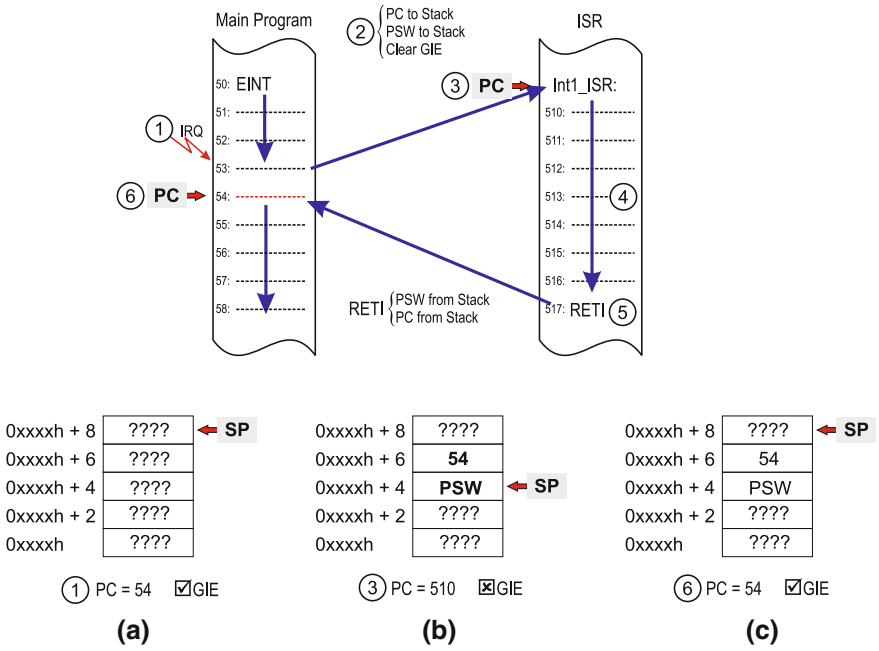


Fig. 7.1 Sequence of events in the CPU upon accepting an interrupt request. **a** Stack before IRQ, **b** Stack after IRQ, **c** Stack after RETI

with is denoted with a checkmark, while a crossed-out GIE denotes it has been cleared.

1. An interrupt request arrives while a main program, with GIE flag set, is executing instruction 53. Instruction 53 completes execution.
2. The processor saves the current PC, pointing to instruction 54, and the status register (PSW), still with the GIE flag set, onto the stack. Next, the GIE flag is cleared.
3. The PC is loaded with the address of the first instruction of interrupt service routine of the device that issued the interrupt request (*How does this happen?*).
4. The ISR is executed, and finishes when a return from interrupt instruction (RETI) is encountered and executed.
5. The execution of the RETI instruction causes the PSW and PC to be restored from the stack.
6. The interrupted program is resumed, continuing from instruction the instruction pointed by the PC, in this case instruction 54.

Note the similitude between the interrupt processing sequence and that of calling a software function, illustrated in Fig. 3.31. This similarity is what makes interrupts to be sometimes regarded as hardware initiated function calls.

One aspect of the interrupt processing sequence that has yet to be explained is: How does the PC gets loaded with the correct ISR address? This question takes particular meaning when considering that there might be multiple interrupt capable devices in an embedded system, with equal number of ISRs. In complex systems there might be dozens of interrupt sources.

Another issue to consider is that having multiple devices issuing interrupt requests at arbitrary times, there is a high probability that two or more devices may simultaneously place service requests to the CPU. Recall that the CPU is a sequential machine and will only serve one device at a time.

Addressing these issues require having in place management mechanisms for (a) identifying the source of an interrupt request among many others that might coexist in the system to activate the correct ISR, and (b) resolving the conflict caused when multiple devices simultaneously place interrupt requests to the CPU. In the next sections, a discussion is made of how these issues are handled.

7.1.4 Interrupt Identification Methods

Throughout the evolution of microprocessors, different methods have been used to identify the source of an interrupt request in systems with multiple interrupt capable interfaces. In general, these methods, although having differences from a processor architecture to another, can be classified into one of three types: non-vectored, auto-vectored, or vectored systems.

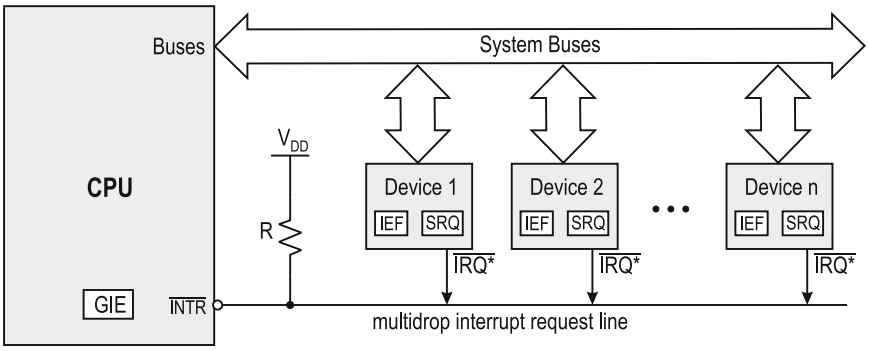


Fig. 7.2 Device interfaces in a system managing non-vectored interrupts

Non-vectored Interrupts

The most elemental way to serve interrupts in a system is by having all requesting devices placing their requests through a single, multidrop interrupt request line. When either of the devices places a request, the INTR line into the CPU becomes asserted, triggering a request. By just sensing the INTR line, the CPU will not be able to identify who placed the request, so within the ISR, the CPU proceeds to check the service request flags of all its peripherals to find out who placed the request. This is called a non-vectored interrupt system. Figure 7.2 illustrates a block diagram of a system managing non-vectored interrupts.

Each device interface has a an open collector or open drain request line tied to the processor $\overline{\text{INTR}}$ input. The activation of a request by a particular device interface will set its SRQ flag. If the interface's interrupt enable flag (IEF) is also set, its $\overline{\text{IRQ}}$* will become asserted, placing a request to the CPU. Assuming the CPU has its GIE flag set, the interrupt will take place by loading the PC with a predefined address where the ISR is stored. The ISR address in non-vectored systems is usually fixed a certain location in program memory where the ISR must be stored in order to be executed. When any of the devices places a request, a single ISR is executed and code within the ISR polls the SRQ flags of each device interface to determine who placed the request. The absence of a hardware mechanism allowing the CPU automatically identifying who placed the service request is what gains this method the name of non-vectored.

A slightly improved mechanism is offered by some CPUs where instead of a fixed address for the ISR, a fixed location is specified where the address of the ISR is stored. Jumping to the ISR is achieved by loading the PC with the value stored at this location. This scheme makes more flexible the process of locating the ISR, but it still does not provide automated identification of the device who placed the request.

If multiple simultaneous requests were placed in this system, the polling order would determine what device would be served first. This establishes a priority service order in the system.

A non-vectored system, although simple, usually results in a delayed response due to the need of checking by software who placed a request at the beginning of the ISR.

Vectored Interrupts

A more efficient scheme to identify the source of an interrupt request is provided in a vectored system. In a vectored system, a hardware mechanism is provided that allows for automatic identification of the interrupt request source without having to poll the service request flags in the interfaces. The most common mechanism uses an interrupt acknowledgment signal (INTA), which, when received at the interrupt requesting interface causes the latter to send an identification code to the CPU. This ID code, which allows the CPU to determine the location of the device's ISR, is named a *vector*. A reserved space in memory holds the addresses associated to each interrupt vector in the system. This memory space is called the system vector table.

Vectored approaches are commonly used in microprocessors, where external interfaces issuing interrupts send their ID codes via the data bus when they receive the INTA signal. In most cases the vector ID is not the actual ISR address. Instead, the ID is a number that can be used to calculate the actual location of the ISR address. Intel processors from the 80x86 and successors use this scheme.

The process triggered in a vectored system by the reception and acceptance by the CPU of an interrupt request is called an *interrupt acknowledgment cycle*. Such a cycle begins with the CPU issuing an INTA signal, followed by the peripheral interface sending its vector ID, and then the CPU determining from the vector, the address that gets loaded into the PC to begin execution of the ISR. The whole interrupt acknowledgment cycle occurs automatically within the CPU control unit, with no instruction involved in this process. Sometimes this scheme is called a *full vectored system*.

Auto-vectored Interrupts

In microcontrollers, where multiple on-chip peripherals like timers, I/O ports, serial ports, data converters, etc., can issue interrupts to the CPU, a simpler mechanism called *auto-vectoring* is used. In an auto-vector system, each device has a fixed vector or a fixed address leading to its ISR. Thus, the CPU does not need to issue an INTA signal and the peripherals do not need to issue a vector. When any of the internal interrupt capable peripherals issues an interrupt request (assuming GIE is set), the MCU loads into the PC the address of the corresponding ISR. MCUs designed with fixed ISR addresses directly load the PC with the predefined ISR address, while MCUs with fixed vectors load the PC with the address stored at the corresponding vector entry from the vector table.

The MSP430 handles all interrupt requests using auto-vectors. This includes all maskable sources, non-maskable, and reset. The top portion of the 64K memory is reserved for storing ISR and reset addresses. Additional details for the MSP430 are discussed in Sect. 7.2.

7.1.5 Priority Handling

When several peripherals with interrupt capability coexist within the same system, there will be instances when two or more of them will simultaneously place service

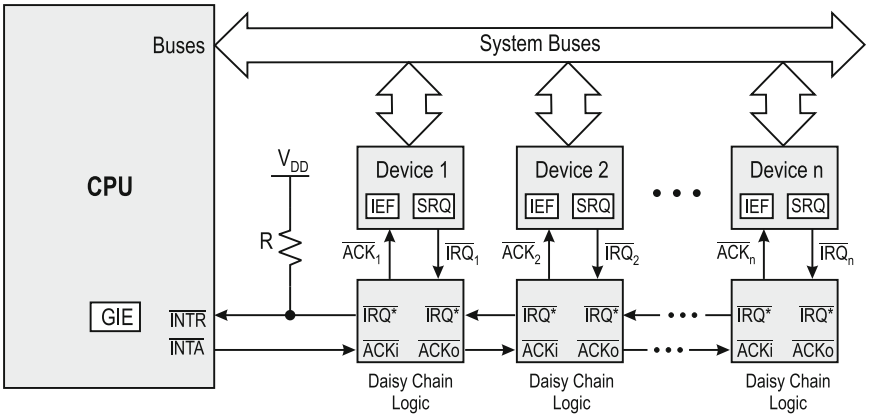


Fig. 7.3 Device interfaces connected in a daisy chain

requests[1] to the CPU. When this situation arises, the CPU needs to have in place a way of prioritizing service. Recall that the CPU can handle only one task at a time.

There are multiple ways to provide for priority management in microprocessor-based systems, either via software or hardware.

When interrupts are served with non-vectored mechanisms, we saw earlier that the order in which service request flags are polled establishes a priority order. In such a system it becomes necessary to identify device priorities prior to deciding the polling order to ensure those with more pressing service needs are served first. The same concept applies when devices are served by polling.

Vectored and auto-vectored systems require hardware support for resolving priorities. Two methods are preferred: using *daisy chain-based arbitration*, or with an *interrupt controller*.

Daisy Chain-based Arbitration

A daisy chain is perhaps the simplest hardware setup to provide priority arbitration in an embedded system. It assumes all devices place service requests through a common multidrop request line, and a common granting or acknowledgment signal is returned indicating the requested service has been accepted. This implies that each peripheral device in this scheme must have the capability of handling both, request and acknowledgment lines. Each peripheral device is connected to a link of the chain.

Each link in daisy chain contains a logic circuit that lets passing any device request to the CPU, but limits the acknowledgment signal to reach only up to the highest priority device interface that has placed a interrupt request to the CPU. Figure 7.3 shows an arrangement of device interfaces sharing a daisy chain scheme.

[1] Since interrupt service requests are served only at the end of each instruction, to consider two request simultaneous, they just need to arrive within the time frame of a single instruction execution.

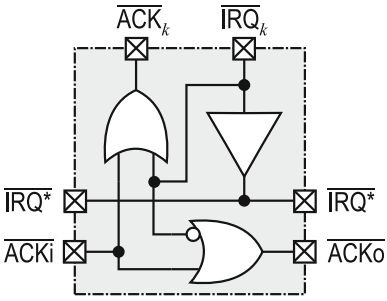


Fig. 7.4 Logic structure of a daisy chain link

Due to the blocking action of link elements in the chain, priorities are hardwired as a function of the distance from the CPU ports: the closer the peripheral to the CPU, the higher the priority. Figure 7.4 shows the structure of a daisy chain link.

If two devices assert their request signals simultaneously, say devices 1 and 2, the interrupt request line into the CPU $\overline{\text{INTR}}$ becomes asserted, causing it to respond with $\overline{\text{INTA}}$ and clearing its GIE. However, the acknowledgment signal will only reach to Device 1. Note how the assertion of $\overline{\text{IRQ}_1}$ blocks the propagation of $\overline{\text{INTA}}$ down the chain. Thus device 1 will issue its vector ID through the systems buses and the CPU will execute ISR1. Upon returning from ISR1, the CPU re-enables the GIE, becoming able to serve another interrupt. Serving device 1 causes it to de-assert $\overline{\text{IRQ}_1}$. Since device 2 still has its $\overline{\text{IRQ}_2}$ asserted, the MCU will detect the interrupt request (from device 2). This time the $\overline{\text{INTA}}$ will reach device 2, allowing the execution of ISR2.

A daisy chain is a simple solution to the priority arbitration problem, but has some limitations. First, priorities are hardwired, making difficult balancing services among peripheral devices. If a device close to the CPU places frequent requests, it could monopolize service, preventing devices down the chain to receive on-time CPU attention. Also, devices down-the-chain have a longer propagation delay to receive the $\overline{\text{INTA}}$ signal, which could delay their servicing. Another limitation is that a daisy chain will only work with level-sensitive interrupt requests. Additional latching hardware would be required to manage edge triggered requests.

If any of the above limitations becomes relevant in an application, the designer would need to resort to a different priority handling mechanism. Next section introduces how dedicated interrupt controllers work.

Interrupt Controller-based Arbitration

An interrupt controller is a special case of a single-purpose processor also called a *priority arbiter*. A priority arbiter is a programmable support peripheral, configured by the CPU at startup to allow for handling requests from multiple devices or events to a common resource. In the case of an interrupt priority arbiter, the common resource being accessed is the attention of the CPU.

Priority interrupt arbiters are the preferred choice for handling interrupts in vectored system as they reduce the hardware requirements of peripheral interfaces to comply with the signal protocol required by the interrupt acknowledgment cycle . An interrupt controller is placed between the CPU and the requesting devices,

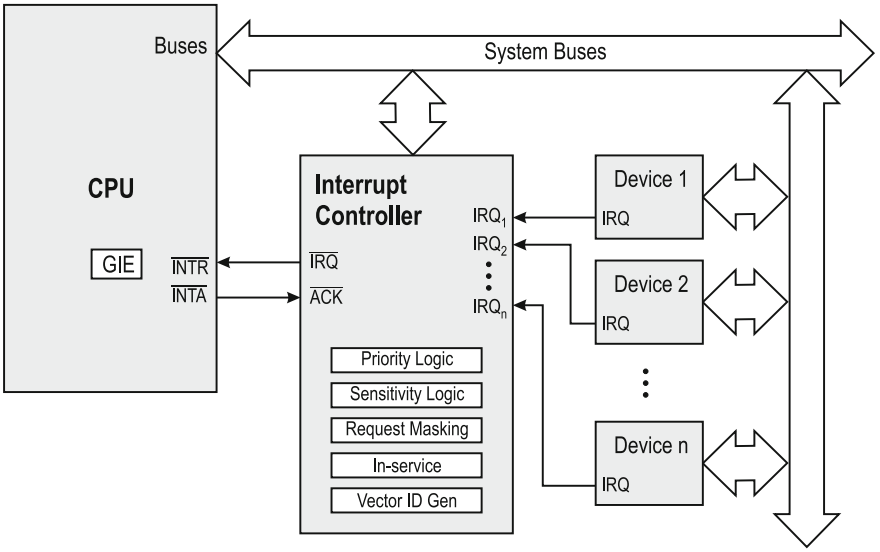


Fig. 7.5 Interrupt management using a programmable interrupt controller

in such a way that individual interrupt requests from peripheral interfaces are received by the arbiter, and the arbiter, based on the configured management scheme, relays them to the CPU. This arrangement is illustrated in Fig. 7.5.

As an arbiter, the interrupt controller not only manages incoming requests from device interfaces allowing to establish a flexible priority management scheme, but also provides other functions that would otherwise require additional logic and software overhead in the ISR to be furnished. Some of the functions provided by an interrupt controller include:

- Signalization for the interrupt acknowledgement cycle: Upon receiving the $\overline{\text{INTA}}$ signal from the CPU, based on the configured priority scheme it selects the device to be served and issues its vector ID.
- Schemes to configure individual IRQ sensitivity: Such a scheme allows for selecting between level- or edge-sensitivity and the type of level or edge in each case. The interrupt controller would also include the latching logic necessary for edge sensitivity.
- A flexible priority management scheme: This adds flexibility in the management of priorities by allowing using either fixed or dynamic priority schemes. Dynamic schemes such as rotating priorities, where a device after being served goes to lowest priority, avoiding service monopolization, are possible.
- Service management registers: These include masking registers allowing disabling individual request lines and in-service indication registers that facilitate automated interrupt nesting.

The specific functions rendered by an interrupt controller change from one architecture to another. In microprocessors, programmable interrupt controllers are specifically designed to comply with host CPU architecture and protocols. Early systems had dedicated chips designed for interrupt control. A representative example of one of such devices is the Intel 8259A, a controller designed for early 80x86 processors, but its flexibility and versatility gained acceptance in a wide number of systems and applications. Modern microprocessor system rely on custom designed cores embedded into the processor or supporting chipsets.

In the arena of microcontrollers, interrupt handling also rely on custom designed hardware embedded into the MCU. The way these designs work internally result transparent to the programmer. The fundamental aspects to be known about a particular embedded interrupt control structure are its handing capabilities and programmer's model. A first step for a system designer successfully utilizing such resources is understanding them both by reading the manufacturer's documentation. The interrupt handling structure of MSP430 devices, discussed in the next section, is a representative example of this trend.

7.2 Interrupt Handling in the MSP430

The MSP430 is a microcontroller designed with a large and versatile interrupt management infrastructure. All its internal peripherals have the capability of being operated by interrupts, resulting in a system with a large number of hardware interrupt sources. As part of its ultra low-power philosophy, all these sources also have the ability of waking up the CPU from its featured low power modes. Among the list of internal devices capable of triggering interrupts in the MSP430 we find: general purpose I/O ports, data converters, serial interfaces, general purpose and watchdog timers, and system exceptions, to name just a few of the most common. A specific list of interrupt sources depends on the device generation, family, and the particular configuration of embedded peripherals in the chosen device.

MSP430 MCUs use a fixed priority management scheme resulting from using a daisy-chain arbitration structure, so peripherals closer to the CPU have a higher priority. As the list of embedded peripherals changes from one MSP430 model to another, knowing the exact priority level of a particular device requires consulting its data sheet.

Interrupt source identification in the MSP430 uses an auto-vector approach with vector locations fixed at the end of the memory map. Early MSP430 families allocated memory addresses from 0xFFE0 to 0xFFFF to form the *interrupt vector table*, including the address for the reset vector at 0xFFFE. Later families have extended the table to start at 0xFFC0, but they maintain backward compatibility with the arrangements provided for earlier generations.

When writing interrupt managed programs, the programmer needs to consult the data sheet of the particular MSP430 model being used in his or her application and

then proceed to configure the vector entries of the interrupt sources used in their programs. The process of allocating vectors is discussed in Chap. 4, Sect. 4.7.3.

In general, an MSP430 microcontroller can handle three types of interrupt sources that include: system resets, non-maskable (NMI), and maskable interrupts. Below we discuss each of them.

7.2.1 System Resets

System resets, although been listed among interrupt sources, behave differently and serve a different purpose than conventional interrupts. The main similarity between resets and interrupts is that both change the normal course of execution of a program by loading into the PC the contents of their corresponding vector. However, a reset is quite different as it does not save the processor status or a return address, does not return, and has as a function initializing the state of the system. Thus the usage of interrupts and resets is not interchangeable.

Chapter 6 provides a detailed discussion of reset events in embedded systems. In particular, Sect. 6.7 discusses in detail the reset structure in MSP430 microcontrollers.

7.2.2 (Non)-Maskable Interrupts

(Non)-maskable interrupts in the MSP430 can be considered as a type of pseudo-NMI, based in our definition in Sect. 7.1.2 of an NMI. The reason is that although MSP430 NMIs cannot be masked by the GIE bit, they *can* be masked by the corresponding individual enable bits in their sources, namely NMIIE, OFIE, and ACCVIE (see sources below).

Whenever a (non)-maskable interrupt is accepted by the MSP430 CPU, *all* NMI enable bits are automatically reset, and program execution begins at the address pointed by the (non)-maskable interrupt vector, stored at address 0FFFCh. This single vector is shared by all NMI sources. (Non)-maskable interrupts in the MSP430 can be generated by any of the following sources:

- **An edge on the $\overline{\text{R}}$ST/NMI pin when configured in NMI mode:** The function of the $\overline{\text{RST}}$/NMI pin after power-up is, by default, in reset mode. All MSP430 generations prior to series x5xx/x6xx, allow this functionality to be changed through the $\overline{\text{RST}}$/NMI bit in the watchdog control register WDTCTL. When in the NMI mode, with the NMI enable bit (NMIIE) is set, a signal edge in the $\overline{\text{RST}}$/NMI input will trigger an NMI interrupt. The triggering edge can be configured to rise or fall through the WDTNMIES bit. Triggering an NMI will also set the $\overline{\text{RST}}$/NMI flag NMIIFG.
- **The detection of an oscillator fault:**] By enabling the oscillator fault flag (OFIE) in the interrupt enable register 1 (IE1), an NMI can be triggered if a malfunction

is detected with the external crystal oscillator. As was explained in Sect. 6.4.1, the absence of signal from an external crystal switches the MCLK to the DCO and sets the OFIFG bit in the interrupt flag register 1 (IFG1). User software must consider this as an expected condition whenever the MSP430 is started. However, if an external crystal were present and after the due crystal startup period no signal were detected, then an enabled oscillator fault NMI can facilitate the implementation of counter measures. User software must check and clear the OIFG upon such an event.

- **An access violation to the flash memory:** To prevent corrupting the contents of the on-chip flash memory, the MSP430 forbids write accesses to the flash memory control register (FCTL1) during flash erase or write operations (WAIT = 0). Failure to observe this rule triggers an exception indicated by the access violation interrupt flag (ACCVIFG) in the flash memory control register FCTL3. If the ACCVIE flag is set, this event also triggers an NMI. User software action is required to clear ACCVIFG after it has been set.

As an NMI can be triggered by multiple sources, the code within the ISR needs to discriminate the activation source and take the appropriate remedial actions. When the NMI is triggered, it automatically clears the enable bits o all its sources, namely NMIIE, OFIE and ACCVIE, preventing interruption of the NMI ISR. Note that these flags are not restored by the RETI instruction and therefore user software needs to re-enable them if their functionality is to be restored. The corresponding interrupt triggering flags, NMIIFG, OFIFG, and ACCVIFG are not. Thus the NMI ISR must also reset the interrupt triggering flags. Figure 7.6 illustrates a recommended flowchart for planning a well-designed NMI handler.

7.2.3 Maskable Interrupts

Most of the interrupt sources in the MSP430 fall in the category of maskable interrupts, so all interrupt sources other than those listed in the NMI section fall in the maskable category.

All maskable interrupt sources in the MSP430 are disabled when the global interrupt enable (GIE) flag in the status register (SR) is cleared. In addition, each individual interrupt source has enable flags in their corresponding interfaces. This means that for any device interrupting the MSP430 CPU, both the GIE flag *and* the particular enable flag associated to the device must be enabled.

Interrupt requests in the MSP430 are checked between instructions, implying that interrupts are only served between instructions. When an interrupt is accepted by the MSP430 CPU, it pushes both the PC and the SR onto the stack in that order. Then, through its internal daisy chain logic it chooses the requesting device with highest priority. When the requesting device has a unique interrupt source, the CPU automatically clears its request; however, in devices featuring multiple sources of interrupt, the disabling must be part of the ISR. Next, the status register (SR) is cleared,

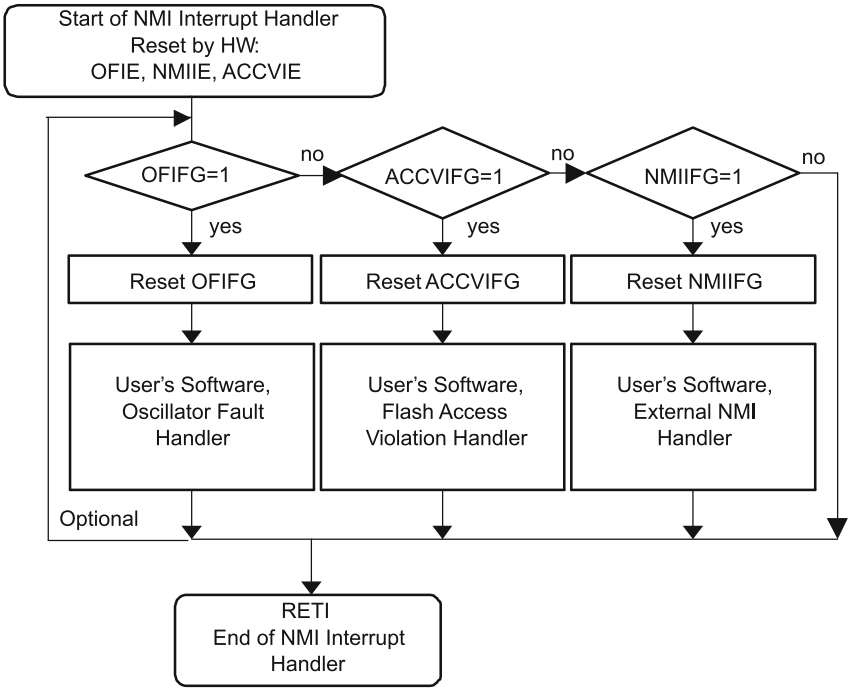


Fig. 7.6 Recommended flowchart for a well-designed NMI handler (*Courtesy of Texas Instruments, Inc.*)

disabling the global interrupt enable flag, which prevents interrupting the ISR. Clearing the SR also cancels any previous low-power mode configured. At this point, the PC is loaded with the vector contents of the chosen interrupt source to begin executing its ISR. Upon executing the return from interrupt instruction (`#RETI#`), the MSP430 restores both SR and PC from the stack, re-enabling the GIE and restoring any low-power mode previous to the interrupt.

The specific list of interrupt sources in a particular MSP430 microcontroller and their priorities, as explained earlier, depends on the particular device chosen for an application. The designer needs to consult the data sheet of the particular device used in an application to have its specific vector and priority assignment. As an illustrative example, the interrupt vector table for an MSP430F2274 model is shown below in Table 7.1.

Note that unlike traditional microcontrollers, the MSP430 does not provide external dedicated interrupt request inputs. Instead, Ports 1 and 2 provide each interrupt capable GPIO inputs. Each pin in these ports can be configured to trigger an interrupt to the CPU, each having individual making and interrupt flags. All pins associated to each port share the same interrupt vector, resulting in up to 16 external interrupt pins with two separate vectors.

Table 7.1 MSP430F2274 interrupt vector table (*Courtesy of Texas Instruments, Inc.*)

Interrupt source	Flag	Type	Address	Priority
Power-up	PORIFG	Reset	0FFFEh	31
External reset	RSTIFG			highest
Watchdog	WDTIFG			
Flash key violation	KEYV(2)			
PC out-of-range(1)				
NMI	NMIIFG	NMI	0FFFCh	30
Oscillator fault	OFIFG			
Flash access violation	ACCVIFG(2)(3)			
Timer_B3	TBCCR0 CCIFG(4)	Maskable	0FFFAh	29
Timer_B3	TBCCR1 TBCCR2 CCIFGs TBIFG(2)(4)	Maskable	0FFF8h	28
			0FFF6h	27
Watchdog Timer	WDTIFG	Maskable	0FFF4h	26
Timer_A3	TACCR0 CCIFG(3)	Maskable	0FFF2h	25
Timer_A3	TACCR1 CCIFG TACCR2 CCIFG TAIFG(2)(4)	Maskable	0FFF0h	24
USCI_A0/USCI_B0 Rx	UCA0RXIFG UCB0RXIFG(2)	Maskable	0FFEEh	23
USCI_A0/USCI_B0 Tx	UCA0TXIFG UCB0TXIFG(2)	Maskable	0FFECh	22
ADC10	ADC10IFG(4)	Maskable	0FFEAh	21
			0FFE8h	20
I/O Port P2 (8 flags)	P2IFG.0 to P2IFG.7(2)(4)	Maskable	0FFE6h	19
I/O Port P1 (8 flags)	P1IFG.0 to P1IFG.7(2)(4)	Maskable	0FFE4h	18
			0FFE2h	17
			0FFE0h	16
(5)			0FFDEh	15
(6)			0FFDCh to 0FFC0h	14 to 0, lowest

(1) A reset is generated if the CPU tries to fetch instructions from within the module register memory address range (0h to 01FFh) or from within unused address range. (2) Multiple source flags (3) (non)-maskable: the individual interrupt-enable bit can disable an interrupt event, but the general interrupt enable cannot. Nonmaskable: neither the individual nor the general interrupt-enable bit will disable an interrupt event. (4) Interrupt flags are located in the module. (5) This location is used as bootstrap loader security key (BSLSKEY). A 0AA55h at this location disables the BSL completely. A zero (0h) disables the erasure of the flash if an invalid password is supplied. (6) The interrupt vectors at addresses 0FFDCh to 0FFC0h are not used in this device and can be used for regular program code if necessary

7.3 Interrupt Software Design

When it comes to develop programs for interrupt-driven embedded systems, the adherence to good programming practices becomes of outmost importance. Interrupt-driven systems induce modular software structures that can be very agile, quickly responding to real-time external events. However, interrupt-driven systems are difficult to debug and tend to grow complex as the number of interrupt generating events increases. For these reasons, careful software planning and orderly code writing are essential.

7.3.1 Fundamental Interrupt Programming Requirements

Before interrupts can work in any application, there are four fundamental requirements that must always be met: stack allocation, vector entry setup, provision of an ISR, and interrupt enable. These requirements have been indicated earlier in Chaps. 3 and 4. Here we provide a more detailed discussion to each of them.

1. Stack Allocation The allocation of a stack area is fundamental for the proper functionality of any interrupt. The stack is used to store the program counter (PC), processor status (SR) and any other register saved upon interrupt acceptance. The stack allocation must be explicitly coded in assembly programs.[2] When allocating a stack, it is necessary to estimate the maximum stack space to be used by ISR invokations, function calls, temporary storage, and parameter passing. If nested functions or reentrant code is being used, nesting depth has to be controlled to avoid stack overflow. Each assembler has its own rules for stack declaration, allocation, and stack pointer management. In the case of MSP430 under the IAR assembler. Example 7.1 provides specific guidelines. Search lines marked with <-(1) to see the format.

2. Vector Entry Setup As discussed earlier, the CPU uses a vector table to determine the addresses of each interrupt service routine to be executed upon each interrupt type. Explicit code must be included in any program to initialize the vector entries of each active interrupt with its corresponding ISR address. Examples 7.1 and 7.2 illustrate how this is done in IAR. Check for lines marked with <–(2) within the comments.

3. Interrupt Service Routine Each active interrupt must have in place an interrupt service routine (ISR). This is the to be executed when the interrupt is triggered. Like all programs, ISRs must observe good programming practices. However, due to the fact that ISRs, once the interrupts are enabled, can be triggered in any point of the code, specific rules must be strictly observed. These include:

- Make ISRs short and quick: Recall that while an ISR is executed, all other interrupts, by default, are disabled. If any other urgent event were triggered while a long ISR is being executed, that other event might experience a long latency that

[2] C-language programs do not require explicit stack allocation as it is performed by the compiler.

will make the system appear slow or even compromise the system integrity. In programming terms, short and quick are not synonymous. Avoid lengthly computations and loops or function calls. If there is a need for lengthly computation, relay them to the main program. Search for lines marked with <–(3) to see example ISRs in Examples 7.1 and 7.2 below.

- Make ISRs register transparent. This rule takes special meaning when programming in assembly language.[3] Programmers must explicitly push any register used within an ISR prior to its first use and pop it before exiting the function.
- No parameter passing or value return: As ISR invocations are unpredictable within a program, there is no way you can safely pass or return parameters via registers or the stack. In most instances, global variables are used. A lot of care is required if code will be reentrant.
- End your ISR with an interrupt return (RETI). Although this might look an obvious requirement, it is common to find novice assembly programmers[4] attempting to exit an ISR with a conventional return. This error will not be detected by the assembler, and will make your code loose track of the stack, status, and prevent further interrupts in the system. Also try to avoid multiple exit points from an ISR as it might complicate debugging.

4. Interrupt Enable For interrupts to take place, all required levels of enabling for an event must be enabled. This requires *at least* two levels of enabling: first at the processor-level by setting the CPU GIE flag and second at the device-level itself by enabling the device to issue interrupts. The interrupt flag in the devices interfaces is sometimes automatically cleared when accessed after they have placed an interrupt request, and some do not, but all need to disable the request signal when served. Make sure their interrupt enable ability is restored before exiting the ISR, or otherwise the device will not issue any further interrupt to the CPU. Search for lines marked with <—(4) in the examples below to see the local and global interrupts enabled.

7.3.2 Examples of Interrupt-Based Programs

The following examples illustrate the usage of interrupts from both, assembly- and C-language standpoints. The comments in the code allow for following through the stages of coding in the interrupt usage.

Example 7.1 (Using Interrupts in Assembly Language) *Consider an MSP430-F2231 with a red LED connected to P1.2 and a hardware debounced push-button connected to P1.3. Write an interrupt-enabled assembly program to toggle the LED every time the push-button is depressed.*

[3] C compilers take care of this detail, transparently to the programmer.

[4] In C-language programs the compiler takes care of that.

Solution: *This problem can be solved by setting P1.3 to trigger an interrupt every time a high-to-low transition is detected in the pin. The ISR just needs to toggle P1.3, which can be done x-oring the pin bit itself in the output port register.*

A program implementing this solution is listed below. Within the comments, we have used numbers in parentheses and arrows to denote where each of the four requirements described in Sect. 7.3.1 are satisfied. This solution also invokes a low-power mode in the CPU after the interfaces have been configured. The occurrence of an interrupt wakes-up the CPU, toggles the I/O pin, and returns to sleep mode. Low-power modes are discussed ahead, in Sect. 7.3.6.

```
;-------------------------------------------------------------------------------
; Code assumes push-button in P1.3 is hardware debounced and wired to
; produce a high-to-low transition when depressed.
;-------------------------------------------------------------------------------
#include  "msp430g2231.h"
;-------------------------------------------------------------------------------
            RSEG CSTACK                   ; Stack declaration <------(1)
            RSEG CODE                     ; Executable code begins
;-------------------------------------------------------------------------------
Init        MOV.W   #SFE(CSTACK),SP       ; Initialize stack pointer <--(1)
            MOV.W   #WDTPW+WDTHOLD,&WDTCTL ; Stop the WDT
;
;---------------------------  Port1 Setup  ---------------------------------
            BIS.B   #0F7h,&P1DIR           ; All P1 pins but P1.3 as output
            BIS.B   #BIT3,&P1REN           ; P1.3 Resistor enabled
            BIS.B   #BIT3,&P1OUT           ; Set P1.3 resistor as pull-up
            BIS.B   #BIT3,&P1IES           ; Edge sensitivity now H->L
            BIC.B   #BIT3,&P1IFG           ; Clears any P1.3 pending IRQ
Port1IE     BIS.B   #BIT3,&P1IE            ; Enable P1.3 interrupt <--(4)

Main        BIS.W   #CPUOFF+GIE,SR         ; CPU off and set GIE <---(4)
            NOP                            ; Debugger breakpoint
;-------------------------------------------------------------------------------
PORT1_ISR  ; Begin ISR <--------------(3)
;-------------------------------------------------------------------------------
            BIC.C   #BIT3,&P1IFG           ; Reset P1.3 Interrupt Flag
            XOR.B   #BIT2,&P1OUT           ; Toggle LED in P1.2
            RETI                           ; Return from interrupt <---(3)
;
;-------------------------------------------------------------------------------
;           Reset and Interrupt Vector Allocation
;-------------------------------------------------------------------------------
            ORG     RESET_VECTOR           ; MSP430 Reset Vector
            DW      Init                   ;
            ORG     PORT1_VECTOR           ; Port.1 Interrupt Vector
            DW      PORT1_ISR              ; <--------(2)
            END
;-------------------------------------------------------------------------------
```

Interrupts are even easier to use from a C-language program, as illustrated in the Example 7.2.

Example 7.2 (Using Interrupts in C-language) *Consider the MSP430 configuration described above in Example 7.1. Provide an interrupt-enabled, C-language solution to the problem.*

Solution: *The code for this solution is quite simple and straightforward. Observe the there is no need to declare a stack or terminate the ISR with IRET, or program the reset vector, as those details are taken care of by the C compiler. The program fundamentally has the main, the ISR vector programming (*`#pragma` *command in line 19), and the ISR.*

```
//===========================================================================
#include <msp430g2231.h>
//---------------------------------------------------------------------------
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;           // Stop watchdog timer
  P1DIR |= 0xF7;                      // All P1 pins as out but P1.3
  P1REN |= 0x08;                      // P1.3 Resistor enabled
  P1OUT |= 0x08;                      // P1.3 Resistor as pull-up
  P1IES |= 0x08;                      // P1.3 Hi->Lo edge selected
  P1IFG &= 0x08;                      // P1.3 Clear any pending P1.3 IRQ
  P1IE  |= 0x08;                      // P1.3 interrupt enabled
//
  _ _bis_SR_register(LPM4_bits + GIE); // Enter LPM4 w/GIE enabled <---(4)
}
//
//---------------------------------------------------------------------------
// Port 1 interrupt service routine
#pragma vector = PORT1_VECTOR         // Port 1 vector configured <---(2)
_ _interrupt void Port_1(void)         // The ISR code <----(3)
{
  P1OUT ^= 0x04;                      // Toggle LED in P1.2
  P1IFG &= ~0x08;                     // Clear P1.3 IFG
}
//===========================================================================
```

7.3.3 Multi-Source Interrupt Handlers

In the interrupt management infrastructure of microprocessors and microcontrollers it is common to find peripherals that generate multiple interrupt events, but have allocated a single vector. These are commonly referred to as multi-source interrupts. The MSP430 is no exception to this case. A quick look at Table 7.1 reveals several multiple source interrupts, like those of I/O ports 1 and 2, each with eight sources, Timers A and B, and the serial interfaces (USC Tx and Rx). When dealing with the ISRs of such sources, there are two rules that must be observed:

A first rule in this case is that unlike single source interrupts, the peripheral flag triggering a multiple source interrupt request is not automatically cleared by the CPU when the peripheral is served. Therefore, this flag must be software cleared within the ISR itself or otherwise there will be an interrupt request for the same event just served upon returning from the ISR. In Example 7.1, this is the purpose of the first instruction in the ISR.

A second rule to observe is that when a multiple sourced interrupt is triggered, the trigger could be from any of its sources. Therefore, the first step that needs to

be performed within the ISR is to identify what was the triggering source. This is similar to the process described in Sect. 7.1.4 for non-vectored interrupts.

The identification can be done via polling, by interrogating each of the interrupt flags associated with the peripheral or by performing a calculated branch.

Identification Via Polling IRQ Flags

When the identification is made by polling, as in the case of a non-vectored interrupt systems, the code just needs to sequentially test the IRQ flags of the shared sources. The polling order will establish the service priorities.

Figure 7.7 shows an example of an ISR design with three events, Event1, Event2, and Event3. Each event marks it interrupt requests by setting its respective flags Flag1, Flag2, or Flag3. Upon entering the shared ISR, the flags are polled to determine which one triggered the interrupt request, and when determined, the request flag is cleared (Rule 1 above) and the corresponding service rendered.

This example also allows to see that the polling order assigns FLAG1 the highest priority, followed by FLAG2, and with the least priority FLAG3. Note that when in the polling sequence, if Flags 1 and 2 are checked and found clear we still check

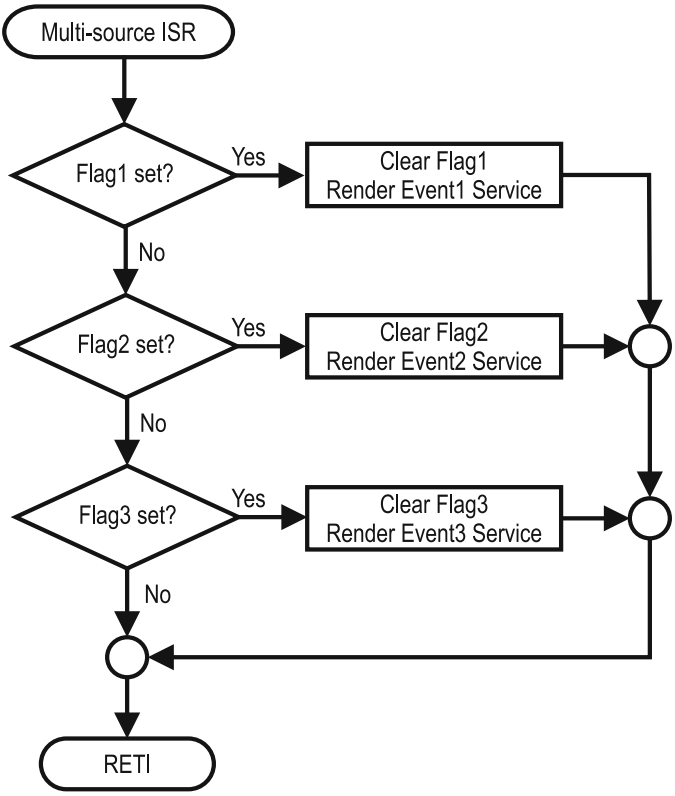


Fig. 7.7 Flowchart of a service routine for a multi-source interrupt vector

for Flag 3. Although a novice programmer might be tempted to go right away and assume Flag3 was the cause of the interrupt, it is actually recommended to check Flag3 as well. Doing so helps to reduce the chances of responding to a false interrupt trigger.

Using Calculated Branching

An even faster technique for identifying the interrupt source in a shared vector is by using *Calculated Branching*. This technique is also known as a *Jump Table* and is commonly used for coding CASE statements.

In this approach, instead of executing a list of successive test and jump instructions, the destination address for the service function is obtained from a branch table (look-up table) holding the addresses of the pieces of code serving each event. The index to enter the look-up table is obtained from the contents of the register holding the service requests of the events shared in the vector. This way, multiple comparisons are avoided, resulting in a much faster and predictable way of branching to the service instructions of each event based on the contents of the shared interrupt request flags. An assembly pseudo-code illustrating the technique for a source shared by three events is listed below:

```
;=============================================================================
#include "headers.h"                    ; Header file for target MCU
;-----------------------------------------------------------------------------
             ...                        ; Preliminary declarations
             RSEG CODE
JumpTable    DW Exit,Serv1,Serv2,Serv2,
                Serv3,Serv3,Serv3,Serv3  ; Branch table with addresses of
                                         ; the 1st line of each service code
Init         ...                        ; First executable instruction
;-----------------------------------------------------------------------------
; Multi source Interrupt Service routine
; Assumes FlagReg contains individual IRQ flags
;-----------------------------------------------------------------------------
MultiSrcISR PUSH R15                ; ISR is R15 transparent
            CLR.W R15               ; Calculation of jump address begins
            MOV.B FlagReg,R15       ; R15 low loaded with IRQ flags
            AND,B #07,R15           ; Only the three lsb assumed active
            ROL R15                 ; Multiply R15 by 2 to get even address
            MOV JumpTable(R15),R15  ; Load address of selected service code
            JMP @R15                ; Jump to selected service code
            ...
Serv1       CLR Flag1              ; Clear Event1 IRQ flag
            Event1 service code    ; Service instructions for Event1
            ...
            JMP Exit
Serv2       CLR Flag2              ; Clear Event2 IRQ flag
            Event2 service code    ; Service instructions for Event2
            ...
            JMP Exit
Serv3       CLR Flag3              ; Clear Event3 IRQ flag
            Event3 service code    ; Service instructions for Event3
            ...
Exit        POP R15                ; Restore R15
            RETI                   ; Exit IRS
;-----------------------------------------------------------------------------
COMMON INTVEC                      ; Interrupt vector area
```

```
;-------------------------------------------------------------------------
            ORG RESET_VECTOR
            DW Init
            ORG MULTI_SOURCE_VECTOR ; Whatever shared vector
            DW MultiSrcISR
;=========================================================================
```

This approach takes the same amount of time to get to any of the functions serving each event, which helps to minimize the interrupt latency. In this particular case, the code is written to assign priorities in decreasing order from the most significant bit of the request register. This priority order is established by the organization of the labels in the branch table.

Note that having three request flags can result in eight different request conditions going from 000 when no request is placed to 111 if all sources simultaneously request service. If for example, events one and three place simultaneous requests (101), Event3 will be served first.

An even more efficient implementation can be achieved when the service request flags are encoded. Several shared sources in MSP430 provide prioritized, encoded flags such as GPIO or serial device. In such cases the destination branch can be determined by just adding to the program counter the contents of the flag request service. The code fragments below illustrate hot to perform such operations in assembly language.

```
;=========================================================================
; Multi source Interrupt Service routine for four events
; Assumes FlagReg contains prioritized, encoded IRQ flags organized as:
; 0000 - No IRQ         0004 - Event2        0008 - Event4
; 0002 - Event 1        0006 - Event3
;-------------------------------------------------------------------------
#include "headers.h"            ; Header file for target MCU
...                             ; Preliminary declarations and code
MultiSrcISR
            ADD &FlagReq,PC     ; Add offset to jump table
            JMP Exit            ; Vector = 0: No interrupt
            JMP Event1          ; Vector = 2: Event1
            JMP Event2          ; Vector = 4: Event2
            JMP Event3          ; Vector = 6: Event3
Event4      Task 4 starts here  ; Vector = 8: Event4
            ...
            JMP Exit            ; Return
Event1      Task 1 starts here  ; Vector 2
            ...                 ; Task starts here
            JMP Exit            ; Return
Event2      Task 2 starts here  ; Vector 4
            ...                 ; Task starts here
            JMP Exit            ; Return
Event2      Task 3 starts here  ; Vector 4
            ...                 ; Task starts here
Exit        JMP Exit            ; Return
```

The code fragment below illustrates a branch table for the multi-source case of four events in C-language.

```
//============================================================================
// Multi source Interrupt Service routine for four events
// Assumes FlagReg contains prioritized, encoded IRQ flags organized as:
// 0000 - No IRQ         0004 - Event2        0008 - Event4
// 0002 - Event 1        0006 - Event3
//----------------------------------------------------------------------------
#include <headers.h>            ; Header file for target MCU
...                             ; Preliminary declarations and code
// Multi-source ISR
#pragma vector = MultiSrc_VECTOR _ _interrupt void MultiSrc_ISR(void) {
    switch(_ _even_in_range(FlagReq,8)) {
        case 0x00: // Vector 0: No interrupts
            break;
        case 0x02: ... // Vector 2: Event1
            break;
        case 0x04: ... // Vector 4: Event2
            break;
        case 0x06: ... // Vector 6: Event3
            break;
        case 0x08: ... // Vector 8: Event4
            break;
        default: break;
    }
}
```

7.3.4 Dealing with False Triggers

In the proceeding discussions we mainly focused in providing vectors and ISRs for active interrupts, meaning by active those that have a hardware or event source to trigger them. Not every system will use every interrupt available in the microcontroller. Those remaining unused are the ones referred to as inactive.

Sometimes, power glitches, electromagnetic interference, electrostatic discharges, or some other form of noise might get coupled into the CPU and corrupt interrupt request lines, registers, or other sensitive system components. As a result, false interrupt triggers might end up occurring in the system. This situation could result in unpredictable consequences for the system integrity, particularly if some of the unused interrupts get to be triggered.

A measure for mitigating this situation can be providing service routines for all sources of interrupts, particularly those that share a common vector. Those that are actually used get the necessary code to be served with provisions to detect false triggering, and those unused can be given a dummy ISRs. A dummy ISR can have just one instruction: RETI. In some instances, as a false trigger might also corrupt other portions of the system, it might be advisable to configure all unused interrupts to point to a common error handler ISR, or a function causing a system reset. For systems working in harsh environments, false triggers might even render interrupt support unusable, resulting more reliable a polled operation [42].

7.3.5 Interrupt Latency and Nesting

Interrupt latency refers to the amount of time a peripheral device or system event has to wait from the instant it requests interrupt service until the first instruction of its ISR is fetched. This time is determinant to the perceived and factual speed of a system, and could become a reliability issue in real-time events and time critical processes. The simple rule that must prime in every interrupt managed system is that every event requesting CPU attention shall be served with low interrupt latency.

Interrupt latency in embedded systems is affected by both hardware and software factors.

In the hardware side, the supporting structure in place for handling the signalization involved in serving an interrupt is the dominant component. Examples of factors associated to the interrupt hardware infrastructure include the propagation delay in the path traversed by the interrupt request signal in its way to the CPU, the mechanism used to identify the requesting source, the way priorities are resolved, and the delay in the interrupt acknowledgement path if there were such a signal involved in the scheme.

Software factors are dominated by the scheduling approach programmed into the application to handle interrupt requests and the ISR coding style. These factors are responsible for the largest latency times in interrupt handling, sometimes hundreds of times longer than any hardware induced latency. A fast hardware infrastructure can be significantly slowed down by a poor priority handling scheme or by the style used to program the interrupt handlers (ISRs), or both.

Interrupt Latency Reduction Methods

Once the interrupt handling structure of a system has been laid, there is little chance to reduce the hardware factors causing interrupt latency. If the designer does have a choice in deciding the hardware components for interrupt management, measures like reducing the number of stages through which request and acknowledgment signals propagate, choosing vectored or auto-vectored approaches over non-vector, and implementing in-service tracking in hardware, are some of the most effective ways to reduce the lag. Most of these provisions are already in place when a programmable interrupt controller is used. Hardware induced latency actually represents the lower bound for the interrupt latency in an embedded system design.

Most opportunities for reducing interrupt latency are tied to good software design practices. Keeping ISRs short and quick, avoiding instructions with long latency, and properly handling interrupt priorities top the list of recommendations. One major source of latency is the disablement of interrupts in the system, that automatically done when an ISR is entered into. In this last aspect, two particular mitigating strategies are commonly used: allowing interrupt nesting and establishing prioritization schemes.

Interrupt Nesting

Interrupt nesting is achieved by re-enabling the GIE after the processor context has been saved to the stack. This can be done in combination with a prioritization

scheme where only interrupts with a higher priority than the currently served are enable. If re-entrancy[5] were desired, equal priority requests had to be enabled as well. These programming practices require careful program control to avoid system havock. Strict software control of the nesting level should be observed to avoid stack overflow. Moreover, if reentrant coding were to be allowed, additional provisions need to be taken in the ISR and the rest of the code to avoid situations that would mess-up the memory system. General recommendations include the following basic rules:

- Avoid Static Variables: A reentrant function shall not hold static (non-constant) data due to the risk of self modification. Although a reentrant function can still work with global data, it is advised to use them with atomic instructions.[6] The best alternative is to have function allocated variables (dynamic allocation), so that every time the function is invoked it allocates its own set of variables. The atomicity observation also applies to the usage of hardware: do not allow a hardware transaction to be interrupted.
- Do not use self-modifying code: A reentrant function cannot contain under any circumstance instructions that modify code memory.
- Do not invoke non-re-entrant functions: All function calls from within a re-entrant function must be reentrant as well.

Additional rules pertinent to ISRs in general also apply, like not passing parameters through the stack or registers and not returning values.

These programming techniques, due to their need of keeping a tight in-service track and centralized management of the interrupt structure, are best suited to be implemented when the hardware infrastructure provided by a programmable interrupt controller. Small microcontrollers usually do not provide such facilities. In such instances, the user software needs to take care of all the details. When this is done, the software usually becomes complex, and in most cases ends-up worsening the interrupt latency itself.

7.3.6 Interrupts and Low-Power Modes

Many embedded applications, particularly those in real-time reactive systems have the MCU most of the time waiting for an event to occur. This results in a CPU that is used only sporadically, with long idle periods. Upon this situation we could write the application software in one of two ways:

The first alternative would initialize the peripherals and system resources to then send the CPU into an infinite loop implemented through either polling the expected event or waiting for an interrupt to trigger the CPU response. This is probably the most intuitive way of doing it, but not the most power efficient. By continuously

[5] A re-entrant ISR is one that safely allows being interrupted by itself.

[6] Instructions able to perform read-modify-write operations without interruption.

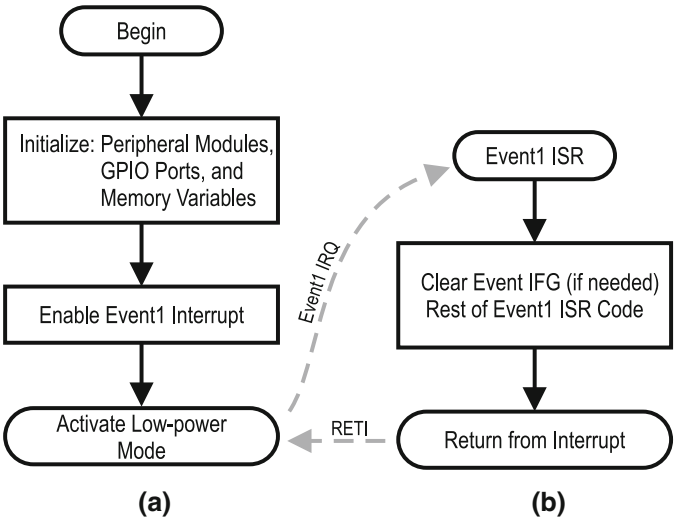


Fig. 7.8 Flowchart of a main program using a low-power mode and a single event ISR. **a** Main program, **b** Interrupt service routine

polling the event flag or leaving the CPU active in an infinite loop to be broken by an interrupt would waste lots of CPU cycles and energy.

A more energy efficient alternative is to initialize the peripherals, system components, enable the interrupts of the expected events and then send the CPU into a sleep mode. Every time an enabled interrupt is triggered, it will wake-up the CPU to serve the event and go back to sleep. This second alternative will be orders of magnitude more energy efficient as the CPU will consume energy only when waken-up to serve the enabled peripherals and the rest of the time will be in a low-power mode consuming minimal or no energy. Figure 7.8 illustrates a flowchart for a main program and one ISR (Event1 ISR) designed to operate with the CPU in low-power mode. This plan assumes that all the tasks needed in the system can be performed inside the ISR.

If there were additional events to be served in this system, the only additional requirement would be enabling the interrupts of the additional events and providing their corresponding ISRs. Assuming the vectors of all active events are configured, then they all can be served the same way. If you look back at the structure of the programs in Examples 7.1 and 7.2, you will notice that both have the structure illustrated in Fig. 7.8.

Low-power modes are of outmost importance in battery powered applications due to the dramatic reduction in power consumption they induce. But not only battery-powered designs benefit from a low-power consumption. Energy efficiency, as discussed in Chap. 1, must be a primary objective in every embedded design, as it has implications improving system reliability, reducing power supply requirements, size, weight, and overall cost of applications.

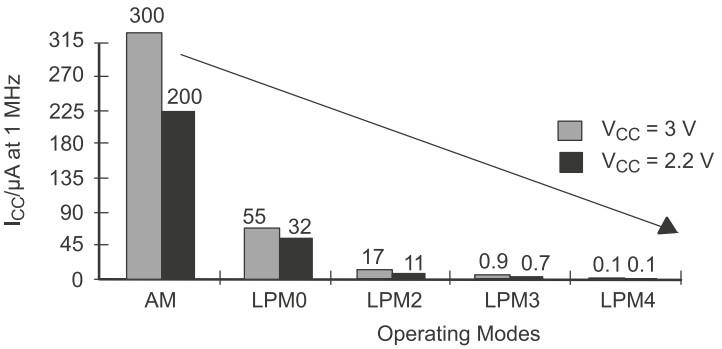


Fig. 7.9 Current consumption of MSP430F21x1 devices in their different operating modes (*Courtesy of Texas Instruments, Inc.*)

To have an idea of how dramatic can be the power reduction, consider the case of MSP430F21x1 devices. The activation of their low-power modes can reduce the MCU supply current well below 1 % of the current consumption in active mode. Figure 7.9 shows the current consumption of this MCU family in their different low-power modes.

Enabling Low-power Modes

A low power mode in an MCU is activated by writing the processors operating mode bits, typically located in the processor status register, with the bit pattern of the desired operating mode. Most modern microcontrollers feature one or more low-power modes, where the power reduction is achieved by either reducing the clock rate or completely shutting-off the clock generator, and/or selectively powering down unused peripherals, including the CPU itself.

When a microcontroller is sent into a low-power mode, the way to bring it back into activity is through an interrupt. Recall that in the interrupt cycle, one of the steps prior to executing the ISR clearing the status register, deactivating any low-power mode in effect. This highlights the importance of enabling interrupts prior to sending an MCU to a low-power mode. The designer, when planning an application, needs to determine which events will wake-up the CPU, to include in the program the necessary support to have their interrupts functional and enabled prior to sending the CPU into a low-power mode.

At the end of the ISR, if no changes were made to the saved SR, its retrieval by the interrupt return instruction restores the low-power mode in effect prior to the interrupt. That is why the flowchart in Fig. 7.8 shows the program flow returning to the low-power mode in the main upon exiting the ISR.

Coping with Complex Service Routines

More than often, the operations to be performed as a result of serving a peripheral module can be complex enough for not being feasible to perform all tasks completely inside an ISR. It should not be forgotten that one of the strongest recommendations in the design of ISRs is to keep them short and quick.

To cope with the requirements of a complex ISRs the solution would be relaying part or all the processing to the main program, while still triggering the code execution through interrupts. This can be easily and efficiently done with the aid of a low-power mode and software defined global flags as outlined below:

1. Declare in the Main program one global variable as flag per each ISR to be partially or completely executed in the Main. Initialize all declared flags cleared. These flags will be used to indicate which interrupt was activated. Also initialize the rest of the system resources as usual.
2. Still in the main program, enable interrupts as usual (GIE and individual events IF).
3. Activate the appropriate low-power mode (if more than one were available). This is where the execution of the main will stop to until an interrupt is triggered.
4. Code in the main, beginning right after the instruction activating the low-power mode, a loop to check the software flags. Each flag detected set will be the indication for executing the code of the corresponding ISR event. Don't forget to clear the flag. After rendering the corresponding service, keep the program in the loop checking additional software flags that might be set until all flags are clear. At this point go back in the main to the instruction where the low-power mode was originally activated.
5. In the ISR of each event to be served, set the corresponding software flag and cancel the low-power mode in the saved status register. This will cause the CPU to restore an Active Mode when the IRET instruction is executed, allowing the main to go executing the loop where the software flags are checked.

At first sight, this approach might seem complex, but it is actually very simple. Figure 7.10 shows a flowchart illustrating this process with a program serving three interrupt events. An advantage of this approach is that it allows reducing the length of time the GIE flag is cleared, since the actual ISR is really short.

Example 7.3 illustrates how simple is to implement this approach in assembly language for an MSP430.

7.3.7 Working with MSP430 Low-Power Modes

One of the greatest features of MSP430 microcontrollers is their low-power consumption and low-power modes, where the already low power consumption of the MCU can be reduced to nanoampere range currents, as indicated in Fig. 7.9.

MSP430 devices prior to series x5xx/x6xx feature four low-power modes, denoted LPM0, LPM1, LP2, and LPM3. The default operating mode of the CPU, the Active Mode with the CPU and all clocks active is how the chip operates just after a reset or when entering an interrupt service routine. Changing from the active mode to any of the low-power modes under software control can be achieved by configuring the status register (SR) bits CPUOFF, SCG1, SCG0 and OSCOFF, as indicated below, in Table 7.2.

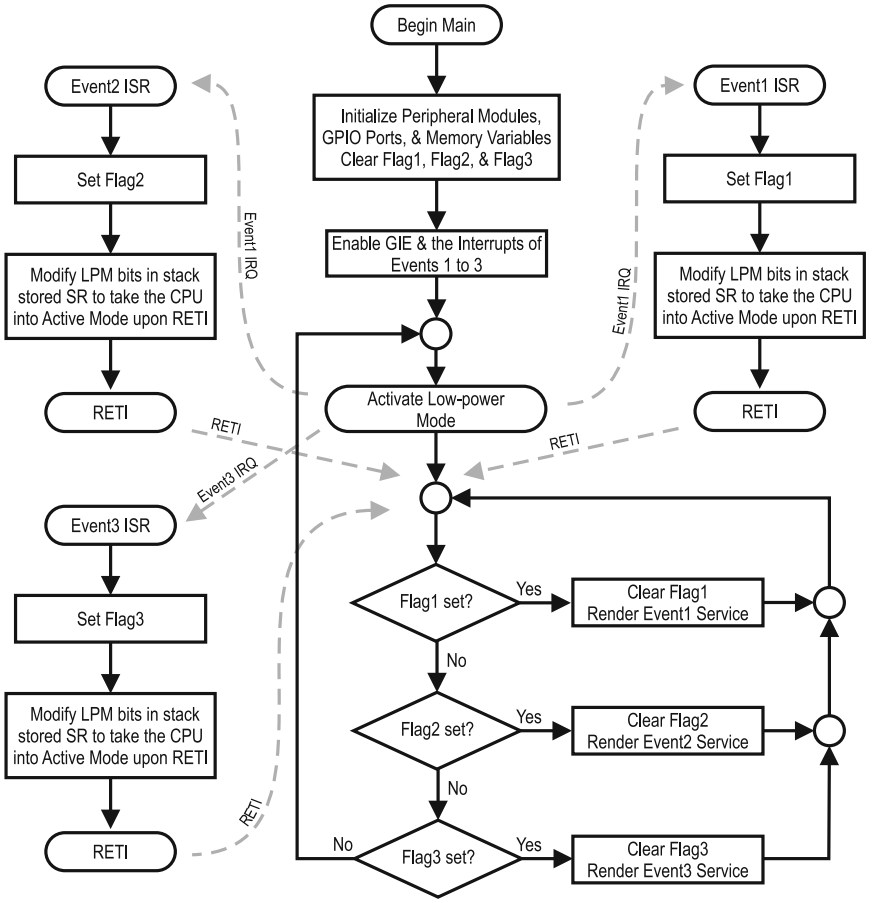


Fig. 7.10 Software strategy to execute ISR code in the main

Series x5xx/x6xx feature two additional operating modes denominated LPM3.5 and LPM4.5, which, in addition to disabling the modules listed for LPM3 and LPM4 in the table above, also turn off power to the RAM and register areas of the MCU, enabling an even lower power consumption. These modes result useful in applications where loss data retention capabilities in the MCU registers and RAM can be tolerated. Additional details of these modes can be found in the corresponding device data sheet.

Each operating mode, either Active or LPM*n*, achieves different power consumption rates which vary depending on the particular device being configured.

The selection of a particular low-power mode over another is done based on the modules that must stay active in the chip during the sleep mode to allow its interrupt to wake the CPU. For example if a particular application has a timer triggering an interrupt to wake the CPU, and that timer is fed by ACLK, then the chip can be sent into LPM3, which keeps ACLK on.

Table 7.2 MSP430 Operating Modes (*Courtesy of Texas Instruments, Inc.*)

SCG1	SCG0	OSC-OFF	CPU-OFF	Mode	CPU & Clocks status
0	0	0	0	Active	CPU and all enabled clocks are active Default mode upon power-up
0	0	0	1	LPM0	CPU & MCLK disabled, SMCLK & ACLK active
0	1	0	1	LPM1	CPU & MCLK disabled. DCO & DC generator off if DCO is not used for SMCLK. ACLK active
1	0	0	1	LPM2	CPU, MCLK, SMCLK, & DCO disabled DC generator enabled & ACLK active
1	1	0	1	LPM3	CPU, MCLK, SMCLK, & DCO disabled DC generator disabled, ACLK active
1	1	1	1	LPM4	CPU and all clocks disabled

The example below illustrates a case of low-power mode operation with the servicing code executed in the main program.

Example 7.3 (Relying service code to the Main) *Re-write the push-button/LED program of Example 7.1 to have the servicing instructions executed in the main.*

Solution: *In this particular case, there is only one event to be served, thus, there is no need to use global software flags or to poll after resuming the main. See how simple is to modify the status register while still in the stack. Recall that just after entering into the ISR, the top-of-stack (TOS) is at the last push, which corresponds to the SR.*

```
#include  "msp430g2231.h"
;-------------------------------------------------------------------------
            RSEG CSTACK                     ; Stack Segment
            RSEG CODE                       ; Executable code begins
;-------------------------------------------------------------------------
Init        MOV.W   #SFE(CSTACK),SP         ; Initialize stack pointer
            MOV.W   #WDTPW+WDTHOLD,&WDTCTL ; Stop the WDT

;           Port1 Setup
            BIS.B   #0F7h,&P1DIR             ; All P1 pins but P1.3 as output
            BIS.B   #BIT3,&P1REN             ; P1.3 Resistor enabled
            BIS.B   #BIT3,&P1OUT             ; Set P1.3 resistor as a pull-up
            BIC.B   #BIT3,&P1IFG             ; Clears any P1.3 pending IRQ
            BIS.B   #BIT3,&P1IE              ; Enable P1.3 interrupt

Main        BIS.W   #CPUOFF+GIE,SR           ; CPU off and set GIE
            NOP                              ; Assembler Breakpoint
            XOR.B   #0x04,&P1OUT             ; Toggle LED
            JMP     Main                     ; Go back to reactivate LPM
;-------------------------------------------------------------------------
PORT1_ISR  ; Begin ISR
;-------------------------------------------------------------------------
            BIC.C   #BIT3,&P1IF              ; Reset Interrupt Flag
```

```
            bic     #LPM4+GIE,0(SP)         ; CPU in active mode
            reti                            ; Return from interrupt

;-------------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------------
            ORG     RESET_VECTOR            ; MSP430 RESET Vector
            DW      Init
            ORG     PORT1_VECTOR            ; Port1 Interrupt Vector
            DW      PORT1_ISR
            END
;=========================================================================
```

7.4 Timers and Event Counters

This section discusses one of the most useful resources of a microcontroller: timer units. Timers are important because they are used to implement several signature applications. Among them, the implementation of watchdog timers, interval timers, event counters, real-time clocks, pulse width modulation, and baud rate generation. We will explain some of these in Sect. 7.4.4.

Microcontrollers may include one or more timers among their peripheral units. Timers are generally configurable to greatly enhance their basic functionality. They can generate a square signal that can be accommodated to our needs, by adjusting its frequency, duty cycle, or both. Timers also have interrupt capabilities, or that of capturing the specific time at which some external event occurs. By controlling these and other timer features an embedded system designer may relieve the CPU from a long list of time related tasks in many applications.

7.4.1 Base Timer Structure

In its most basic form, a timer is a counter driven by a periodic clock signal. Its operation can be summarized as follows: Starting from an arbitrary value, typically zero, the counter is incremented every time the clock signal makes a transition. Each clock transition is called a "clock event". The counter keeps track of the number of clock events. Notice that only a single type of transition, either low-to-high or high-to-low (not both), will create an event. If the clock were a periodic signal of known frequency f, then the number of events k in the counter would indicate the time kT elapsed between event 0 and the current event , being $T = 1/f$ the period of the clock signal. The name "timer" stems from this characteristic.

To enhance the capabilities of a timer, further blocks are typically added to the basic counter. The conglomerate of the counter plus the additional blocks enhancing the structure form the architecture of a particular timer.

Figure 7.11 illustrates the basic structure of a timer unit. Its fundamental component include the following:

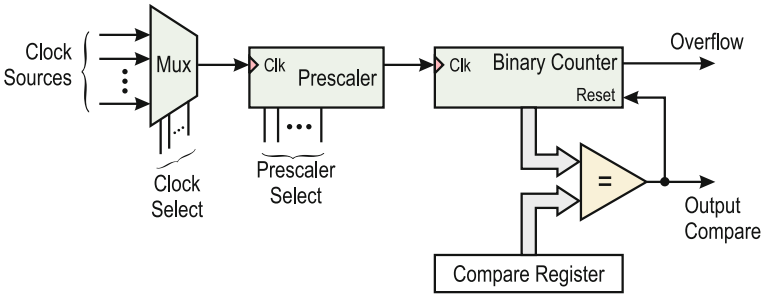


Fig. 7.11 Components of a base timer

- Some form of a clock selector (Mux) allowing for choosing one from multiple clock sources.
- A prescaler, that allows for pre-dividing the input clock frequency reaching the counter.
- An n-bit binary counter, providing the basic counting ability.
- An n-bit compare register, that allows for limiting the maximum value the counter can reach.
- A hardware comparator that allows for knowing when the binary counter reaches the value stored in the compare register. Note how a match of these values resets the binary counter.

Below we provide additional information about these components.

Clocks and Pre-scaler (Divider)

In general, the clock signal for a timer may come from one or several different sources that include the external or internal clock sources, or even asynchronous events.

The accuracy of the clock signal is particularly important in time sensitive applications. Clocks and their selection criteria were discussed in Chap. 6. In particular Sect. 6.3.2 provides a discussion of factors that need to be considered when choosing a clock source.

In most MCUs and many microprocessor-based systems the clock signal for timers is derived from the system clock. In many applications such a frequency value might result too high for practical purposes. This situation calls for a means of reducing the frequency of the clock signal reaching the binary counter. This can be done with the aid of a timer pre-scaler or input divider. The scaling factor dividing the input frequency is generally software selectable. The output of the divider drives the binary counter.

Counter and Compare units

The counter register is probably the most important component of a timer. It is usually of the same size as the rest of the registers in the microcontroller. In most systems, it is also a read/write register.

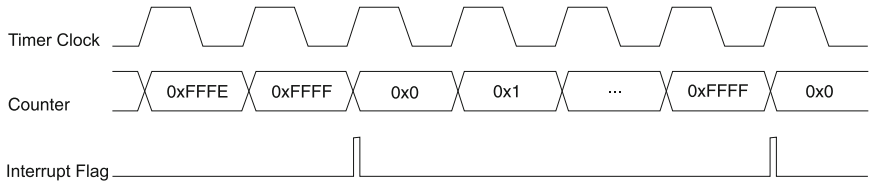


Fig. 7.12 Example of signals commonly obtained from a timer. The interrupt signal can be obtained from the compare output

The counter changes its state with each pulse of the driving signal that comes from the pre-scaler. In this way, it "counts" the number of pulses that have elapsed.

With n bits, the counter will count from 0 to $2^n - 1$ and then go back again to 0. When it reaches the highest value, it generates an *overflow signal*. The overflow can be optionally configured to trigger an interrupt to the CPU. Figure 7.12 illustrates the behavior of an overflow signal in a 16-bit counter. The illustration also includes the behavior of an optional interrupt signal triggered by the overflow condition. The unit will count from 0x0000 to 0xFFFF, or 65535, firing the overflow signal when the counting sequence passes from 0xFFFF to 0x0000.

In many applications it results convenient limiting the maximum value that can be reached by the binary counter. In such cases, the availably of a compare register and a hardware comparator comes in handy. The combination of these two components allows for triggering an *output compare signal* when the binary counter reaches the value stored in the compare register. The process is similar to that illustrated by Fig. 7.12 with the appropriate changes in the signal names and the firing instant. For example, assuming the interrupt signal is configured to trigger upon an output compare condition, we might make it fire every 1000 cycles (instead of 65,533) by writing 0x03F8 to the compare register.

Example 7.4 *Assume that we want to generate a signal at an output port that divides the counter's input clock by* 6.

This would mean that one period of our output signal has to cover six periods of the clock signal driving the timer. This in turn would require toggling the I/O pin once every three clock periods. Hence, we load the compare register with the value 3 *and use the compare output signal to toggle the output port. This is illustrated in Fig. 7.13.*

Notice that, in general, to divide the frequency by n, we load the compare register with a value equal to n/2.

7.4.2 Fundamental Operation: Interval Timer vs. Event Counter

Most applications of timers involve one of two basic interpretations to its operation as an event counter, or as an interval timer. As an event counter, a timer counts the

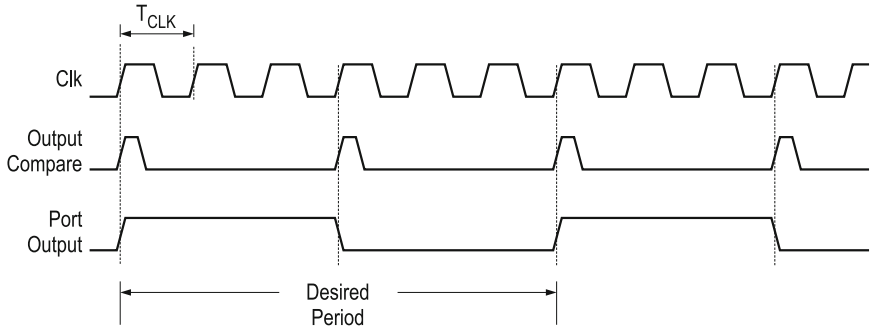


Fig. 7.13 Signal obtained when loading the compare register with a value of three (3)

occurrence of external events. As an interval timer, it measures the time elapsed after k clock cycles. In the case of event counter operation, the clock signal is derived from the event being counted. For timer operation, the clock input is driven by a square signal of known frequency.

Event Counter Operation

When operated as an event counter, a timer simply counts the number of events it detects in its clock input. Note that in this case the clock signal needs to be derived from the event being counted. This would yield a clock signal does not necessarily have a periodic behavior, as illustrated in Fig. 7.14b.

Let's illustrate the operation of an event counter with a practical example.

Example 7.5 *Consider an application where we need to count the number of people passing through a door. Describe how a timer configured as event counter could be used for providing a solution.*

Solution: *In this case we could place in the door frame a digital detector (like an optocoupler) to generate a pulse every time someone passes through. The optocoupler output would then be connected to the timer clock input. With this arrangement, once the timer is enabled, every time someone crosses the door the timer would increment by one.*

Note that in the door example above, the prescaler is assumed to be configured in the divide by one mode. This is an important detail, because otherwise the timer would increase by one after p events have occurred, where p is the prescaler factor.

In the event that we expect to count a number of events larger than the timer capacity, it is easy to extend the timer range. A simple solution would be via a software variable that counts the occurrence of overflows or the number of compare register pulses. Hardware solutions could include configuring the prescaler to a value other than divide by one, or by cascading multiple timers as shown in Fig. 7.15.

In the case of resorting to the prescaler, note that the timer will no longer count single events. Instead it will count sets of p events, where p is the value configured in the prescaler. Example 7.6 sheds some light into this situation.

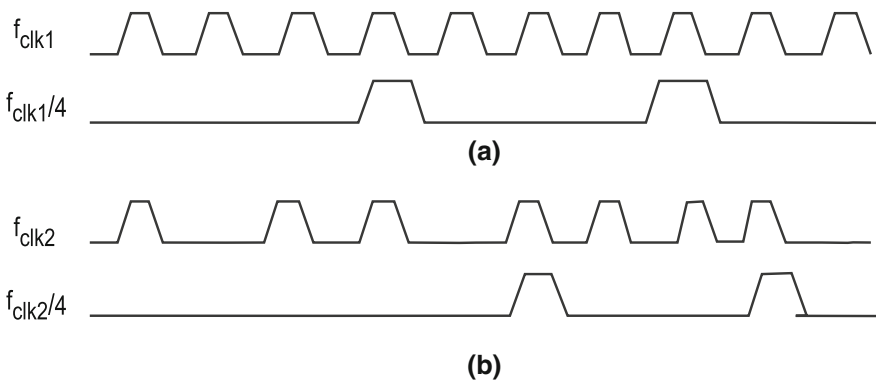


Fig. 7.14 Dividing by 4 in pre-scaler: **a** periodic signal; **b** non periodic signal

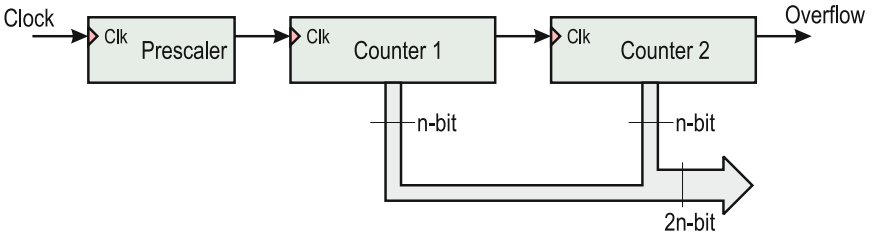


Fig. 7.15 Cascading of counters

Example 7.6 *The pre-scaler in a timer has been set for a factor of* 4*. The counter size is* 16 *bit. (a) What is the maximum number of events that can be counted? (b) What number should be loaded in the compare register to count* 75000 *events and fire the compare output signal? (c) Is it possible to count* 83253 *events? Justify your answer.*

Solution: *(a) Since each counter step covers four events, the maximum number of events that can be registered is*

$$(65,535) \times 4 = 262,140 \textit{ events}$$

(b) The counter must reset when it reaches the value representing 75000 *events. Since the count is scaled by a factor of* 4*, the value to be written on the compare register is*

$$75000 \div 4 = 18750$$

(c) This event counter cannot register 83253 *because the counter steps represent a group of* 4 *input events, and the desired value is not a multiple of* 4*.*

Interval Timer Operation

When the input clock signal is periodic of frequency f, the timer can be used to measure time intervals between two events. If the frequency is f Hz, then the period will be

$$\text{Period } T = \frac{1}{f} \text{ seconds}$$

Therefore, when the counter shows k pulses, it has registered a duration of $kT = k/f$ seconds. **An interval timer**

Example 7.7 *A* 38 KHz *crystal oscillator is used as a clock source, and passed through the pre-scaler with a factor of* 16*. (a) If the timer's counter is reset at a certain moment and at the occurrence of a particular event the counter is reflecting* 0x8A39*, what is the time interval length between the reset instant and the event occurrence? (b) How can we have the counter trigger a signal every 50 ms? What are the absolute and relative errors in the triggering time?*

Solution: *The frequency actually driving the counter is* $38\,\text{KHz}/16 = 2.375\,\text{KHz}$ *or a period of* $(1/2.375)\,\text{ms} = 421\,\mu\text{s}$*. This is the value which we should use in our problem.*

(a) Since the number of periods that the counter has registered is 0x8A39 = 35385*, we compute the elapsed time as* $35385 \times 421\mu\text{s} = 14.9\text{s}$*.*

(b) Since we need to fire the flag every 50 ms*, we must count* $(50\,\text{ms}) \times (2.375\,\text{kHz}) = 118.75$ *periods. Rounding to the nearest integer, the compare register is loaded with* 119*. The actual interval, disregarding any delay introduced by hardware, would be* $119 \times 421\mu\text{s} = 50.105\,\text{ms}$*. This yields an absolute error in excess of* $105\,\mu\text{s}$*, or* 0.21 %*. Based on the application, the designer will determine if this error is acceptable or not.*

Example 7.8 *Consider the problem of counting the number of people passing through a door in Example 7.5. Assume that we'd like to add to this application the capability of measuring the average rate at which people is passing through the door. Outline a possible solution.*

Solution: *A simple solution would be providing a running average of the time between individual passes. Let's say we want to take the average of the last eight passes and still be able to keep track of the number of people passing through the door.*

One possibility would be using a second timer driven by the system clock or other clock source of known frequency. Using the optocoupler output to also trigger an interrupt that reads the value of the second timer in every consecutive optocoupler event. By subtracting each new timer lecture from the previous yields the number of clock cycles between passes. Storing the difference in a circular buffer of eight positions and computing the buffer average upon every interrupt would do it.

An even simpler solution using a single tilmer could be enabled if the timer unit had input capture capability. An input capture uses a free-running counter, which is read upon each external event, in this case the optocoupler output. The input capture ISR holds a count variable for the number of events detected and the input

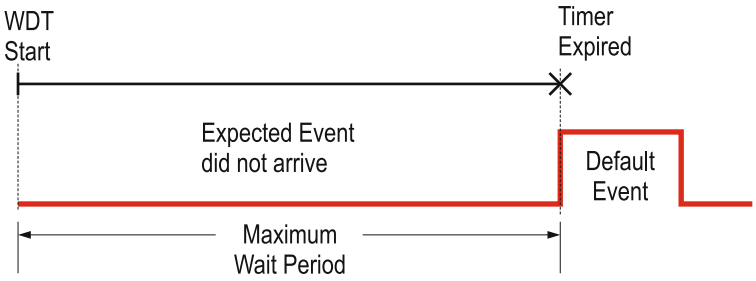


Fig. 7.16 Watchdog timer expiration

capture register automatically holds the timer count between reset and the current event or between two consecutive events. This value can be fed to the running average buffer described above.

7.4.3 Signature Timer Applications

7.4.3.1 Watchdog Timers

In this section we discuss a selection of the most common uses of timers in embedded systems.

A rather common application of timers in embedded systems is that of a watchdog timer (WDT). This is a special case of an interval timer used to perform a certain default event or action, like issuing an interrupt or a reset, if within a predetermined period of time an expected event does not occur. Figure 7.16 illustrates how this works. A maximum allowable period of time is preset by software; a default event will be executed when the preset time period expires unless a certain expected event occurs first.

To prevent a watchdog timer from triggering its default event, the interrupt service routine associated to the expected event must cancel the timer before the preset time. Figure 7.17 illustrates this action.

To illustrate the operation of a watchdog timer, consider the case of a security door operated with an electronic card. To enter through the door a user must swipe a card through a reader. This action unlocks the door to allow the user to open it. But if after a certain time, say 30 seconds, the door has not been pushed open, the lock is activated again. Here, the WDT start occurs when the card is swiped. The default event is locking back the door. The expected event is opening the door.

A watchdog is basically a safety device. Under a normal conditions, it is assumed that the timer will be serviced (WDT cancelled) before its period elapses. Otherwise, something must be wrong and the system must execute the established default action. In embedded systems, watchdog timers can be used for diverse purposes, such as

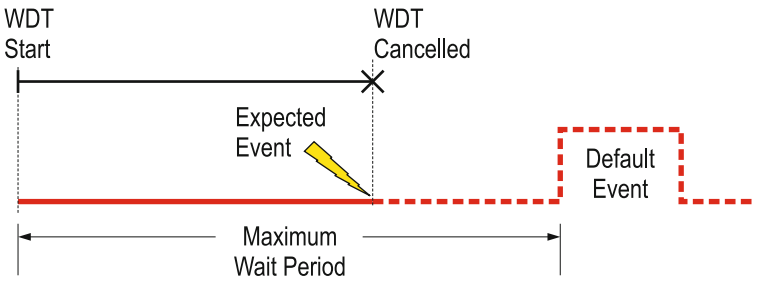


Fig. 7.17 Watchdog timer cancellation

for cancelling runaway programs, exiting infinite loops, or detecting anomalous bus transactions, among others.

All MSP430 devices activate a WDT upon reset, configured to prevent runaway programs. The wait period in this case is 32,768 clock cycles, the default event is a system reset, and the expected event is the user software explicitly cancelling the WDT. This is the reason why all MSP430 programs begin by cancelling the WDT. Section 7.4.4 provides a detailed explanation of the MSP430 WDT.

Real-time Clocks

One of the most common applications for timers in embedded systems is that of a real-time clock. A real-time clock is simply a timer configured to measure seconds, minutes, hours, etc. Often, the resolution of real-time clocks might be needed to go down to the level of fractions of a second, becoming chronometers. Other times, functionality can be taken to resolve days of the week, months, and years, becoming a real-time clock calendar (RTCC). In either case, the fundamental requirement to make a timer a real-time clock is the ability to resolve a period of a second with minimum or no error.

Some microcontrollers have specialized timers that can handle all the functions of a real-time clock. In such cases, the timer includes registers for handling seconds, minutes, hours, days of the week, months, and years. Some can even handle leap year calculations. Although the functionality of an RTCC can be fully implemented in software, using a dedicated timer for such purposes saves a lot of CPU cycles as the timer, once configured and enabled, operates with minimal or no necessity of CPU intervention.

Many MSP430 devices feature real-time clocks among its timers. Figure 7.18 shows a simplified block diagram of the real- time clock in MSP430 4xx series devices.

The accuracy of a real-time clock will greatly depend on the frequency and accuracy of the crystal used.

A typical clock source in RTC applications is a 32.768 kHz crystal. Since this base frequency can be successively divided by two until reaching exactly 1Hz ($32{,}768 = 2^{15}$, it is a preferred value for RTC applications. Care must be taken in providing the appropriate load capacitance when such a crystal is used. The resonant

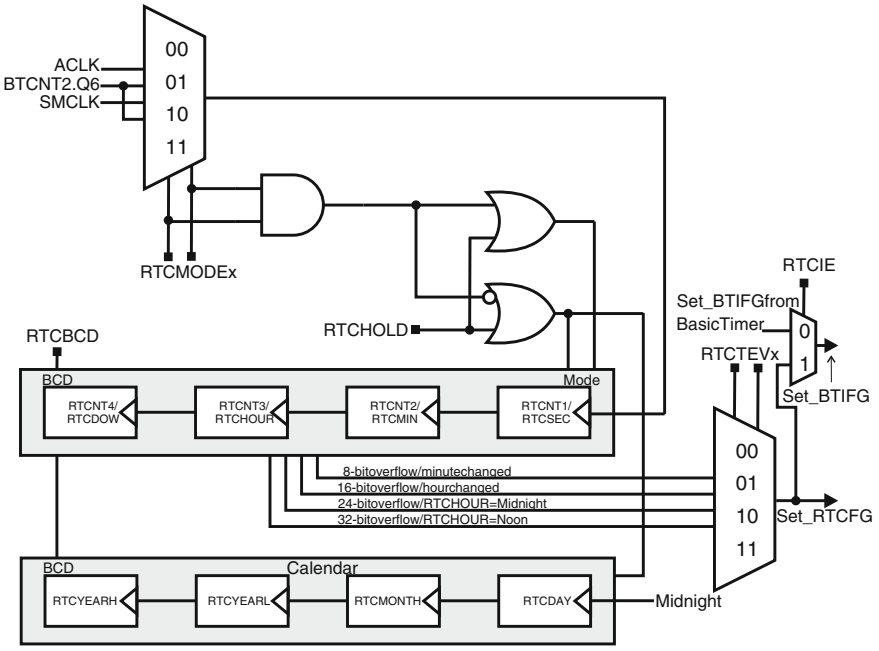


Fig. 7.18 MSP430x4xx real time clock simplified diagrams

frequency of a crystal can be affected by the total capacitance loading the oscillator. Section 6.3 provides a detailed discussion of factors affecting crystal oscillators.

Baud Rate Generation

One of the most common ways to produce the periodic signal required by serial interfaces to produce a desired data transfer rate is through a dedicated timer. In such applications, the timer's clock frequency is divided by the desired baud rate to determine the number of counts between successive bits or *bit time*. The resulting value can then be used as the timer's counter value.

Although baud rate generation using this method is possible, it deserves to mention that most contemporary microcontrollers include serial communication modules with dedicated baud rate generators. A detailed explanation of how to implement baud rate generation is presented in Sect. 9.5.

Pulse Width Modulation

Pulse-width modulation (PWM) is another widely used timer application in embedded systems. A PWM module produces a periodic signal whose duty cycle is controlled by the MCU. Actually, since the signal frequency is also controlled by the MCU, we can say the both, signal period and duty-cycle, can be controlled in this application.

Figure 7.19a illustrates the structure of a PMW module. It fundamentally contains an n-bit timer (with clock selector and prescaler) whose count is compared in

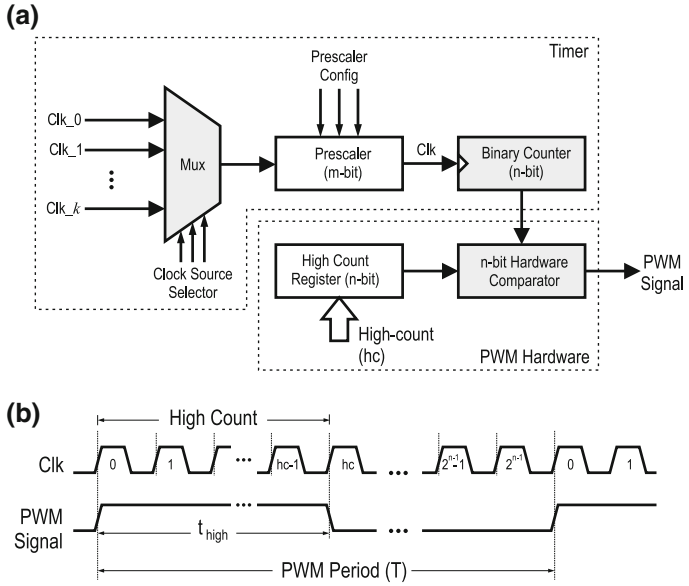


Fig. 7.19 Pulse-widt modulation module: (**a**) hardware structure, (**b**) signal output

hardware to the contents of a "high-count register" (*hc*). While the timer has a value less than *hc* the PWM signal is high. Otherwise the PWM signal is low.

The signal relations can be observed in Fig. 7.19b. Note that the the number of bits (n) in the timer determine the PWM resolution, while the value stored in *hc* controls the signal pulse width (duty cycle = t_{high}/T. The signal frequency can be controlled through the prescaler and/or the selected input frequency in the multiplexer (Mux).

Pulse width modulation is widely used in energy control systems as the amount of energy in the PWM is a function of the signal duty cycle. Applications like DC motor control speed, heater's temperature, light intensity in LEDs, and even musical tones can be implemented with PWM. This widespread of applications make PWM modules a useful addition to the list of MCU peripherals.

Example 7.9 *Consider an application where we wish to control the brightness of an illumination LED using PWM. In LEDs, brightness is a function of the average current intensity passing through the LED junction.*

The LED in this application, an SR-05-WC120, produces a maximum brightness of 240 lumens (lm) when applied its maximum current of 720mA. Assume we want to make an MCU controlled dimmer for this LED able to produce 8 different intensity levels: 0lm, 30lm, 60lm, 90lm, 120lm, 150lm, 180lm, and 210lm. Assuming a 3.3V compatible MCU and a 5V power supply for the LED, the interface would require a buffer to drive the LED from an I/O port. The MCU is assumed to provide a maximum of 20mA per pin (See section 8.8 for analysis). Figure 7.20 shows the recommended interface and required duty cycle values.

Note that each brightness level corresponds to a 1/8 increase in duty cycle. As we are specifying only eight levels, including 0% (LED off), the maximum specifiable brightness value would be 7/8 (why?).

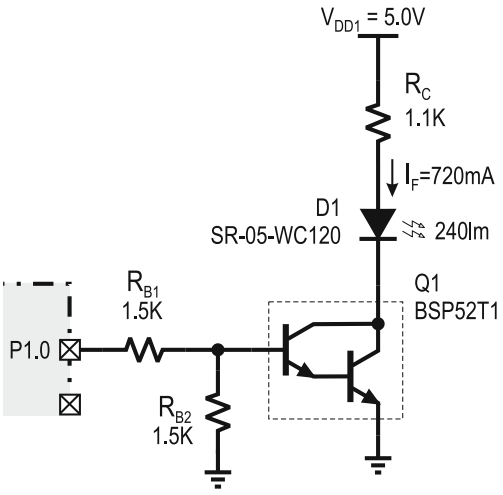
Assuming an 8-bit timer and a selected clock frequency of 32.768KHz, the values to be loaded in the "count-high" register for the corresponding levels of brightness would be in increments of 1/8 of 2^8, this is: 0, 32, 64, 96, 128, 160, 192, or 224. Note that the value required for maximum LED brightness, 256, would not be possible to be specified under this scheme.

A few observations:

- *The default frequency of 32.768KHz was used, meaning the prescaler was ser to divide-by-one.*
- *The LED blinking rate for the above frequency rating would be 32768Hz/256 = 128Hz. A rate fast enough to avoid visible flickering. (What would be the possible maximum prescaler value if the blinking rate were to be reduced the closest to the barely minimum 0f 24Hz to avoid visible flickering?).*
- *Although the number of brightness levels in this example was specified at eight, it would be possible to have up to 256 levels. This is the maximum resolution that can be obtained with an 8-bit counter.*
- *We leave as an exercise to the reader writing a program to implement this solution in the microcontroller of your preference.*

7.4.4 MSP430 Timer Support

All MSP430 microcontrollers include a Watchdog Timer and at least a timer called Timer_A. For this timer, some models have just a basic configuration while others



Brightness (lm)	Duty Cycle
0	0%
30	12.5%
60	25.0%
90	37.5%
120	50.0%
150	62.5%
180	75.0%
210	87.5%
240	100.0%

Fig. 7.20 LED connection diagram and required duty cycle values

include a second Timer_B, and the most recent models include yet a third timer called Timer_D. Depending on the model and generation, some models include from one to three Real Time Clocks. In the sections below we discuss the basic features in the timer support offered by MSP430 microcontrollers. For fine details about each type of timer unit, the reader is referred to the corresponding device User's Manuals and data sheets.

MSP430 Watchdog Timer

The main purpose of the Watchdog Timer (WDT), as explained in Sect. 7.4.3, is basically to reset the system when a software problem happens. The WDT may be put in watchdog mode, regular interval timer mode, or stopped. Since the watchdog mode in the WDT is automatically configured after a PUC, the user must setup, and/or take care to stop the WDT prior to the expiration of the reset interval.

Configuration of the WDT is performed via a 16-bit read/write watchdog timer control register WDTCTL illustrated in Fig. 7.21. This register is password protected. When read, WDTCTL will read 069h in the upper byte. When written, the write password 05Ah must be included in the upper byte, otherwise a security key violation will occur and a power-up clear (PUC) reset will be triggered. Bits 7 to 0 work as follows:

WDTHOLD (Bit 7) Watchdog timer hold. If 0, WDT is not stopped; if 1, WDT is stopped and conserves power.

WDTNMIES (Bit 6) Watchdog timer NMI edge select. When WDTNMI = 1 it has the following effect:

- 0: NMI happens on rising edge
- 1: NMI happens on falling edge

WDTNMI (Bit 5) This bit selects the function for the $\overline{\text{RST}}$/NMI pin: When 0, it has a Reset function; when 1, the pin has an NMI function

WDTTMSEL (Bit 4) Watchdog timer mode select: 0 for Watchdog mode; 1 for Interval timer mode

WDTCNTCL (Bit 3) Watchdog timer counter clear. Setting WDTCNTCL = 1 – by software – clears the count value of the counter, WDTCNT, to 0000 h. This bit is automatically reset.

WDTSSEL (Bit 2) Watchdog timer clock source select: 0 for the SMCLK; 1 for the ACLK

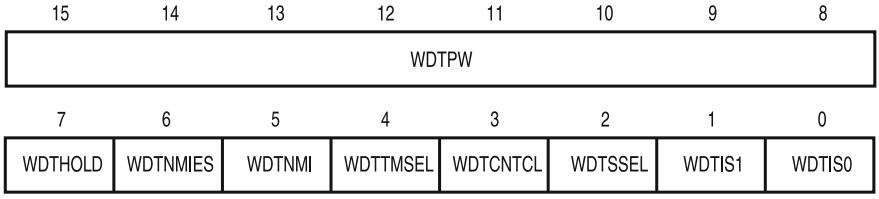


Fig. 7.21 MSP430 Watchdog Timer Control Register (WDTCTL)

WDTISx (Bits 1-0) Watchdog timer interval select. These bits select the watchdog timer interval to set the WDTIFG flag and/or generate a PUC. The alternatives are

- WDTIS0 for 00: Watchdog clock source / 32768 (Default)
- WDTIS1 for 01: Watchdog clock source / 8192
- WDTIS2 for 10: Watchdog clock source / 512
- WDTIS3 for 11: Watchdog clock source / 64

Some useful examples of instructions associated to the WDT configuration are the following.

```
  mov #WDTPW+WDTHOLD,&WDTCTL ; To stop WDT

  mov #WDTPW+WDTCNTCL,&WDTCTL ;Reset WDT

  mov #WDTPW+WDTCNTCL+WDTSSEL,&WDTCTL ; select ACLK clock

;WDT interval timer mode with ACLK, and interval clock/512
  mov #WDTPW+WDTCNTCL+WDTMSEL+WDTIS2,&WDTCTL
```

The watchdog timer includes a 16-bit watchdog timer+ (WDT+) counter WDTCNT not accessible to the user, whose operation and source are controlled through the WDTCTL register. As illustrated in the above code lines, WDTCNTCL should be included in any instruction affecting the counter to avoid unwanted results such as an accidental PUC.

Table 7.3 shows different features of the implementation of the WDT across the different MSP430 families. Notice that the number of available software selectable time intervals varies with the MSP430 version from four in the x4xx, x1xx, and x2xx versions to eight in the x3xx and x5xx/x6xx versions. Observe also that the RST/NMI pin function control is absent in the x5xx/x6xx versions but available in all the other versions.

Table 7.3 Watchdog timer implementation features

WATCHDOG TIMER	x3xx	x4xx	x1xx	x2xx	x5xx/x6xx
Software selectable time intervals	8	4	4	4	8
Watchdog mode	Yes	Yes	Yes	Yes	Yes
Interval mode	Yes	Yes	Yes	Yes	Yes
Password protected write	Yes	Yes	Yes	Yes	Yes
Selectable clock source	Yes	Yes	Yes	Yes	Yes
Power conservation	Yes	Yes	Yes	Yes	Yes
Clock fail-safe feature	No	Yes	Yes	Yes	Yes
RST/NMI pin function control	Yes	Yes	Yes	Yes	No

The clock failsafe protection identified as WDT+ was added starting with the x4xx version of MSP430 microcontrollers. Failsafe means that the clock to the WDT+ cannot be disabled if the watchdog function is enabled. This is important because, if for example ACLK is the clock source for the WDT+, then the WDT+ will prevent ACLK from being disabled and thus LPM4 will not be available. If SMCLK is the clock source for the WDT+ and the user chooses LPM3 the WDT+ will not allow SMCLK to be disabled, increasing power consumption in LPM3.

If either ACLK or SMCLK fail while watchdog mode is enabled, the clock source is automatically switched to MCLK. If this happens with MCLK using a crystal as its source and the crystal fails, then the DCO will be activated by the failsafe to source MCLK. There is no failsafe protection for when the WDT is in interval timer mode. This mode is selected with bit 4, WDTTMSEL, in the WDTCTL.

Two interrupts are associated to the WDT operation: the NMI interrupt and the WDT interrupt. They are signaled by flags MMIIF and WDTIFG respectively, which may be cleared or set by software too. These flags are located in the special function register IF1. Enabling and disabling are handled with bits NMIIE and WDTIE in the special function register IE1 using the instructions

```
bis.b #NMIIE,&IE1; enable NMI interrupt
bis.b #WDTIE,&IE1; enable WDT interrupt
bic.b #NMIIE,&IE1; disable NMI interrupt
bic.b #WDTIE,&IE1; disable WDT interrupt
```

WDTIFG is set at the expiration of the selected time interval. In WDT mode, it generates a PUC. In this case, the WDTIFG flag can be used by the reset vector interrupt service routine to determine the cause of the reset. If WDTIFG is set, then the reset condition was initiated by either the expiration of the time interval or by a security key violation. On the other hand, if WDTIFG is cleared then the reset was caused by some other source.

In interval timer mode, the WDTIFG flag is set after the time interval expires but will trigger an interrupt only if both the WDTIE and GIE bits are set. The WDTIFG is reset automatically in interval mode when the interrupt is serviced. It is important to note that the interval timer interrupt vector is different from the reset vector used in watchdog mode.

Let us close this discussion with an example. The following code was written by D. Dang from Texas Instruments [80].[7] The standard constant WDT_MDLY_32 is defined in the header by the declaration:

#define WDT_MDLY_32 (WDTPW+WDTTMSEL+ WDTCNTCL).

Example 7.10 *Toggle P1.0 using software timed by the WDT ISR. Toggle rate is approximately 30 ms based on default DCO/SMCLK clock source used in this example for the WDT.* $ACLK = n/a$, $MCLK = SMCLK = default\ DCO$

```
;-------------------------------------------------------------------------
;MSP430G2xx1 Demo - WDT, Toggle P1.0, Interval Overflow ISR, DCO SMCLK
;By D. Dang - October 2010
```

[7] See Appendix E.1 for terms of use.

```
;Copyright (c) Texas Instruments, Inc.
;-------------------------------------------------------------------------------
#include <msp430.h>
;-------------------------------------------------------------------------------
            ORG     0F800h                  ; Program Reset
;-------------------------------------------------------------------------------
RESET       mov.w   #0280h,SP               ; Initialize stackpointer
SetupWDT    mov.w   #WDT_MDLY_32,&WDTCTL    ; WDT ~30ms interval timer
            bis.b   #WDTIE,&IE1             ; Enable WDT interrupt
SetupP1     bis.b   #001h,&P1DIR            ; P1.0 output
                                            ;
Mainloop    bis.w   #CPUOFF+GIE,SR          ; CPU off, enable interrupts
            nop                             ; Required only for debugger
                                            ;
;-------------------------------------------------------------------------------
WDT_ISR;    Toggle P1.0
;-------------------------------------------------------------------------------
            xor.b   #001h,&P1OUT            ; Toggle P1.0
            reti                            ;
                                            ;
;-------------------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------------------
            ORG     0FFFEh                  ; MSP430 RESET Vector
            DW      RESET                   ;
            ORG     0FFF4h                  ; WDT Vector
            DW      WDT_ISR                 ;
            END
```

MSP430 Timer_A: Structure and Counter Characteristics

The MSP430 family supports three timers. Timer_A, present in all models; Timer_B included in all but the legacy 3xx series; and Timer_D, appearing in the 5xx/6xx series. Any timer can be used for applications such as real-time clock, pulse width modulation, or baud rate generation, among others. We focus our explanation on Timer_A in this section since it is the timer included in all MSP430 chips. Timer_B and Timer_D features are only summarized later on. The reader can consult device User's Guide for more details on details of a particular device timer.

A simplified block diagram for Timer_A is depicted in Fig. 7.22. Timer_A is a 16-bit timer/counter (TAR), with at least two capture/compare registers TACCR0 and TACCR1, configurable PWM outputs, and interval timing. The timer also has ample interrupt capabilities with interrupts that may be generated on overflow conditions and from the capture/compare registers. These interrupts are decoded using an interrupt vector register TAIV that encompasses all Timer_A interrupts.

Table 7.4 summarizes the implementation features of Timer_A. Except for the now legacy x3xx series which does not include PWM capability, asynchronous input and output latching, or an interrupt vector register, the implementation is quite uniform across the different series.

The timer's operation is software configurable with the Timer_A control register TACTL, shown in Fig. 7.23. A summarized description of the control bits in TACTL is as follows:

- Bit TAIFG is the counter's interrupt flag, and bit TAIE enables the timer interrupt.

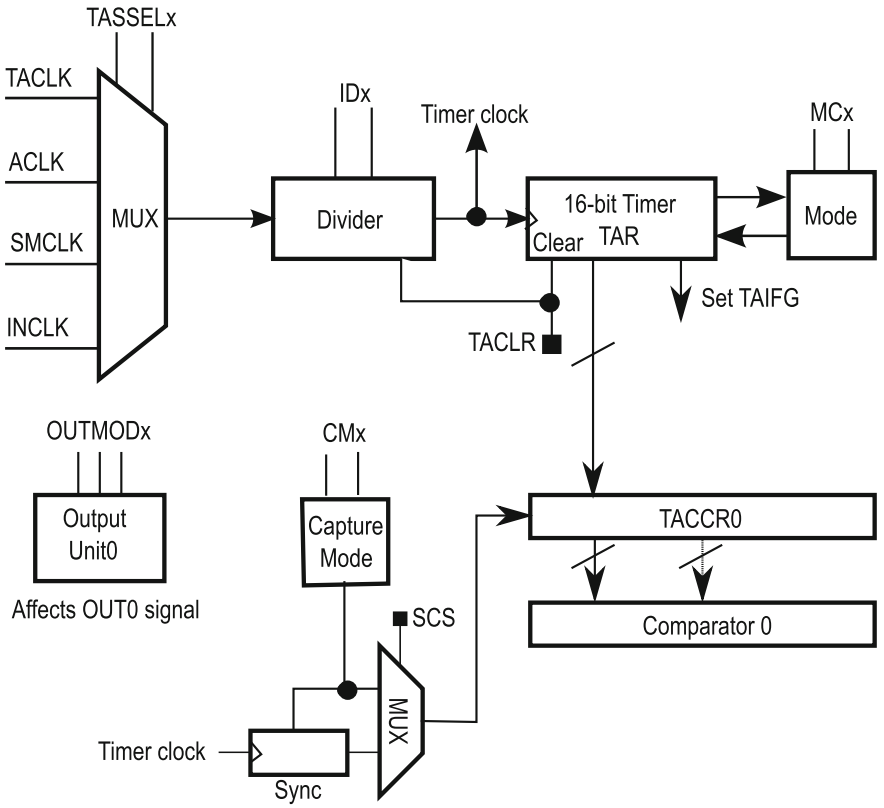


Fig. 7.22 Simplified MSP430 Timer_A block diagram

Table 7.4 Timer_A implementation features

TIMER_A	x3xx	x4xx	x1xx	x2xx	x5xx/x6xx
16-bit timer/counter w/4 operating modes	Yes	Yes	Yes	Yes	Yes
Asynchronous	No	Yes	Yes	Yes	Yes
Selectable and configurable clock source	Yes	Yes	Yes	Yes	Yes
Independently configurable CCRs	5	3 or 5	3	2 or 3	up to 7
PWM output capability	No	Yes	Yes	Yes	Yes
Asynchronous I/O latching	No	Yes	Yes	Yes	Yes
Interrupt vector register	No	Yes	Yes	Yes	Yes
Second Timer_A	No	Yes	No	No	No

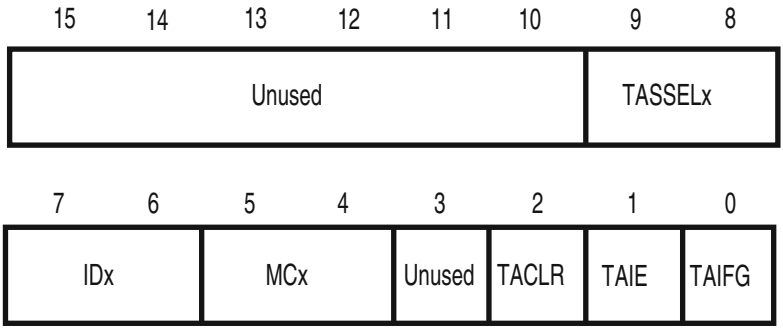


Fig. 7.23 MSP430 Timer_A Control Register (TACTL)

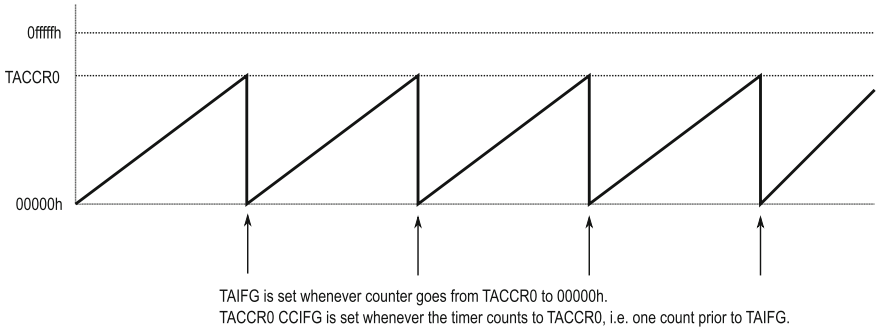


Fig. 7.24 Timer operating in up mode

- Setting TACLR clears the counter, the clock divider and the count direction when the timer is in the up/down mode.
- TASSELx bits select the clock source: TACLK (00), ACLK (01), SMCLK (10), or INCLK (11)
- IDx bits determine the frequency division factor in the prescaler: 1 (00), 2 (01), 4 (10), and 8 (11)
- MCx bits set the operation mode: Halt (00), up mode (01), continuous mode (10), and up/down mode (11).

Except when modifying the interrupt enable TAIE and interrupt flag TAIFG, the timer should always be stopped before modifying its settings. Any modification will take effect immediately. Also, to avoid unpredictable results, stop the timer before reading the counter.

The three non-stop modes are illustrated in Figs. 7.24, 7.25, and Fig. 7.26, respectively.

In the up and up/down modes, the timer can be stopped writing a 0 to TACCR0 and started from zero by writing a nonzero value to TACCR0. This register sets the upper count limit, which must be different from 0FFFFh. In both cases the TACCR0

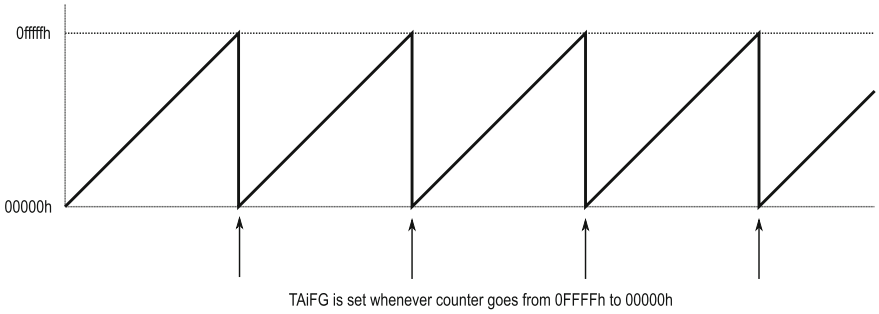


Fig. 7.25 Timer operating in continuous mode

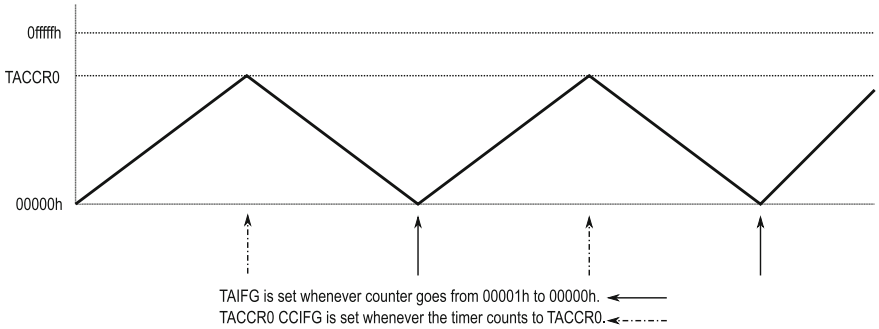


Fig. 7.26 Timer operating in up/down mode

CCIFG is set by the (TACCR0 - 1) to TACCR0 transition. Interrupt generation by TAIFG differs depending on the mode:

- In the up mode the TAIFG flag is set by the TACCR0-to-zero transition, and
- in the up/down mode the TAIFG is set when count goes from 1h to 0h.

In the continuous mode the counter always counts up to 0FFFFh and sets the TAIFG interrupt flag in the 0FFFFh to 0h transition. This mode is particularly useful for generating independent time intervals controlled by hardware and output frequencies with an interrupt generated each time a time interval is completed without impact from interrupt latency. Each of the output/compare registers can be used independently to generate different time intervals or output frequencies.

If the contents of TACCR0 were changed while the timer is counting up, and the new period were greater than the current count, it would continue up to the new value. Otherwise, the new period would not take place until the count reaches 0.

We end this discussion on timer_A with an example. The following code was written by D. Dang from Texas Instruments [80].[8]

[8] See Appendix E.1 for terms of use.

Example 7.11 *Toggle P1.0 using software and TA_0 ISR. Toggles every 50000 SMCLK cycles. SMCLK provides clock source for TACLK During the TA_0 ISR, P1.0 is toggled and 50000 clock cycles are added to CCR0. TA_0 ISR is triggered every 50000 cycles. CPU is normally off and used only during TA_ISR.* $ACLK = n/a, MCLK = SMCLK = TACLK = default\ DCO$

```
;-------------------------------------------------------------------------
;MSP430G2xx1 Demo - Timer_A, Toggle P1.0, CCR0 Cont. Mode ISR, DCO SMCLK
;By D. Dang - October 2010
;Copyright (c) Texas Instruments, Inc.
;-------------------------------------------------------------------------
#include <msp430.h>
;-------------------------------------------------------------------------
            ORG     0F800h                  ; Program Reset
;-------------------------------------------------------------------------
RESET       mov.w   #0280h,SP               ; Initialize stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop WDT
SetupP1     bis.b   #001h,&P1DIR            ; P1.0 output
SetupC0     mov.w   #CCIE,&CCTL0            ; CCR0 interrupt enabled
            mov.w   #50000,&CCR0            ;
SetupTA     mov.w   #TASSEL_2+MC_2,&TACTL   ; SMCLK, contmode
                                            ;
Mainloop    bis.w   #CPUOFF+GIE,SR          ; CPU off, interrupts enabled
            nop                             ; Required only for debugger
                                            ;
;-------------------------------------------------------------------------
TA0_ISR;    Toggle P1.0
;-------------------------------------------------------------------------
            xor.b   #001h,&P1OUT            ; Toggle P1.0
            add.w   #50000,&CCR0            ; Add Offset to CCR0
            reti                            ;
                                            ;
;-------------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------------
            ORG     0FFFEh                  ; MSP430 RESET Vector
            DW      RESET                   ;
            ORG     0FFF2h                  ; Timer_A0 Vector
            DW      TA0_ISR                 ;
            END
```

MSP430 Timer_A Capture/Compare Units

A brief introduction to the capture/compare registers is provided here. More details can be found in the user guides or data sheets.

We can operate the capture/compare blocks in either the capture mode or the compare mode. The first mode is useful to capture timer data, whereas the compare mode is used to generate PWM output signals or interrupts at specific time intervals. We will explain the capture/compare features of the MSP430 timers using Timer_A, although these blocks are available in other timers as well.

The Timer_A capture/compare registers TACCRx, $x = 0, 1, 2$, are configured with the capture/compare control register TACCTLx, illustrated in Fig. 7.27. The bits and bit groups of this register will be explained as needed.

Capture mode To operate in capture mode we must have CAP $= 1$.

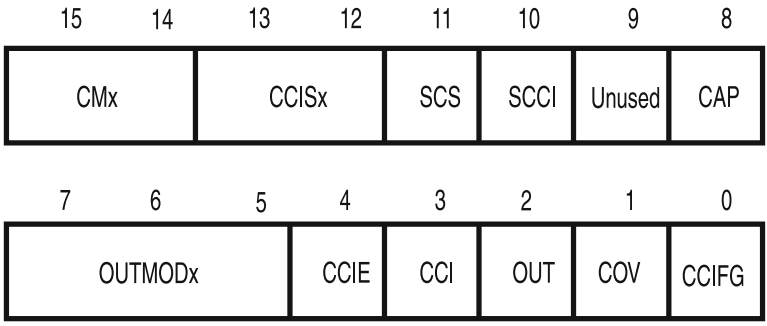


Fig. 7.27 Timer_A Capture/Compare Control Register TACCTLx

When a capture occurs, the TAR timer data is copied into the corresponding TACCRx register and the TACCRx CCIFG interrupt flag is set. In this way the user can record time computations and use them, for example, to measure time, calculate the speed of an event, or any other applications. Capture is done using either external or internal signals connected to capture inputs which are selected with the CCISx bits. It is also possible to initiate capture by software.

Complementing the capture input selection, the CMx bits allow us to select the capture input signal edge: rising, falling, or both. If a second capture were performed before the previous one was read, then the capture overflow bit or COV, would be set to indicate this condition.

A race condition may occur when the capture input signal is asynchronous to the clock. In order to avoid this situation, it is recommended to synchronize both signals by setting the synchronize capture source bit, SCS.

Compare mode To operate in the compare mode we need CAP $= 0$. As mentioned earlier, this mode is used to generate PWM output signals or interrupts at specific time intervals.

When in capture mode, the capture/compare interrupt flag will be set, i.e. TACCRx CCIFG $= 1$ when counter TAR counts to the value stored in register TACCRx. This also sets an internal signal EQUx, i.e. EQUx $= 1$, which triggers an output according to a selected output mode. Furthermore, the capture compare input signal CCI will be latched into the synchronized capture/compare input SCCI.

The output modes available for the capture/compare blocks are discussed next.

Timer_A Output Units

As it can be seen in the Timer_A block diagram, each capture/compare block has an output unit to generate output signals such as pulse width modulation or periodic signals. Bits 5, 6, and 7, i.e. the OUTMODx bits, in register TACCTLx define the output mode. These modes are described in Table 7.5 [32].

Let us now close this discussion on Timer_A with two examples.

Example 7.12 *The following piece of code configures Timer_A to select the ACLK clock source and the up mode. After Low Power Mode 0 is entered, the CPU is only*

Table 7.5 MSP430 output modes. *Courtesy of Texas Instruments, Inc.*

MODx	Mode	Description
000	Output	The output signal OUTx is defined by the OUTx bit. The OUTx signal updates immediately when OUTx is updated
001	Set	The output is set when the timer counts to the TACCRx value. It remains set until a reset of the timer, or until another output mode is selected and affects the output
010	Toggle/Reset	The output is toggled when the timer counts to the TACCRx value. It is reset when the timer counts to the TACCR0 value
011	Set/Reset	The output is set when the timer counts to the TACCRx value. It is reset when the timer counts to the TACCR0 value
100	Toggle	The output is toggled when the timer counts to the TACCRx value. The output period is double the timer period
101	Reset	The output is reset when the timer counts to the TACCRx value. It remains reset until another output mode is selected and affects the output
110	Toggle/Set	The output is toggled when the timer counts to the TACCRx value. It is set when the timer counts to the TACCR0 value
111	Reset/Set	The output is reset when the timer counts to the TACCRx value. It is set when the timer counts to the TACCR0 value

awaken to handle the ISR after an interrupt caused by the capture/compare interrupt flag CCIFG. A LED at pin P1.0 is then toggled.

```
#include    <msp430.h>
;-------------------------------------------------------------------------
            ORG     0F800h  ; Program Start
;-------------------------------------------------------------------------
RESET   mov   #0280h,SP                  ; initialize SP
            mov   #WDTPW+WDTHOLD,&WDTCTL  ; stop WDT
            bis.b   #BIT0,&P1DIR          ; output pin P1.0
            bic.b   #BIT0,&P1OUT          ; red LED off
            mov   #CCIE,&TACCTL0          ; CCR0 interrupt
            mov   #10000,&TACCR0          ; Load upper bound
            mov   #MC_1+TASSEL_1,&TACTL   ; up mode, ACLK
            bis   #LPM0+GIE,SR            ; low power mode 0, interrupts
            nop                           ; only to sync debugger
;-------------------------------------------------------------------------
;   TACCR0 ISR    ; Interrupt Service Routine
;-------------------------------------------------------------------------
TACCR0_ISR   xor.b  #BIT0,&P1OUT          ; toggle LED
                                          ; CCIFG automatically reset
```

```
                                        ; when TACCR0 ISR is serviced
            reti                        ; return from ISR
;-------------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------------
            ORG     0FFFEh                ; Reset vector address
            DW      RESET                 ; RESET label address
            ORG     0FFF2h                ; TACCR0 vector address
            DW      TACCR0_ISR            ; TACCR0_ISR address
            END
```

The following code illustrates the use of Timer_A to generate PWM. It was written by D. Dang from Texas Instruments [80].[9]

Example 7.13 *This program generates one PWM output on P1.2 using Timer_A configured for up mode. The value in CCR0, 512-1, defines the PWM period and the value in CCR1 the PWM duty cycles. A 75% duty cycle is on P1.2.* $ACLK = n/a$, $SMCLK = MCLK = TACLK = default\ DCO$

```
;=========================================================================
;MSP430G2xx1 Demo - Timer_A, PWM TA1, Up Mode, DCO SMCLK
;By D. Dang - October 2010
;Copyright (c) Texas Instruments, Inc.
;-------------------------------------------------------------------------
#include <msp430.h>
;-------------------------------------------------------------------------
            ORG     0F800h      ; Program Reset
;-------------------------------------------------------------------------
RESET       mov.w   #0280h,SP   ; Initialize stackpointer
StopWDT     mov.w   #WDTPW+WDTHOLD,&WDTCTL    ; Stop WDT
SetupP1     bis.b   #00Ch,&P1DIR              ; P1.2 and P1.3 output
            bis.b   #00Ch,&P1SEL              ; P1.2 and P1.3 TA1/2 otions
SetupC0     mov.w   #512-1,&CCR0              ; PWM Period
SetupC1     mov.w   #OUTMOD_7,&CCTL1          ; CCR1 reset/set
            mov.w   #384,&CCR1                ; CCR1 PWM Duty Cycle
SetupTA     mov.w   #TASSEL_2+MC_1,&TACTL     ; SMCLK, upmode
                                              ;
Mainloop    bis.w   #CPUOFF,SR                ; CPU off
            nop                               ; Required only for debugger
                                              ;
;-------------------------------------------------------------------------
;           Interrupt Vectors
;-------------------------------------------------------------------------
            ORG     0FFFEh                    ; MSP430 RESET Vector
            DW      RESET                     ;
            END
```

Timer_B and Timer_D

These two timers are briefly discussed below. A more complete treatment of this topic can be found in the user guides.

Timer_B While Timer_B is practically identical to Timer_A, there are a few notable differences which are summarized in Table 7.6. In Timer_B, TBCLx rather than TACCRx, is used to determine interrupts, and TBCL0 takes on the role of TACCR0

[9] See Appendix E.1 for terms of use.

in count modes. Also, the capture/compare registers can be grouped so that multiple TBCCRs can be loaded together into TBCL. Notice that there is no synchronized capture/compare input SCCI bit function.

Timer_D introduces several important features with respect to Timer_A. The most important include:

- Providing for internal and external clear signals.
- Allowing for routing internal signals between Timer_D instances, and external clear signals.
- Introducing interrupt vector generation of external fault and clear signals.
- Generating feedback signals to the Timer capture/compare channels to affect the timer outputs.
- Supporting high resolution mode.
- Allowing to combine two adjacent TDCCRx registers in one capture/compare channel.
- Supporting dual capture event mode.
- Allowing for synchronizing with a second timer.

Basic Timer and Real-Time Clock

The Basic Timer (BT) was included only in the first two generations of MSP430 devices, i.e. the x3xx and the x4xx. Figure 7.28 shows a simplified block diagram, in which we can appreciate the signals used to control its operation. Probably one of the most important contribution of this timer was the LCD control signal generator which provides the timing for common and segment lines.

The basic features of the Basic Timer are summarized next.

- Two independent 8-bit timers/counters, BTCNT1 and BTCNT2, that can be cascaded to produce a 16-bit counter.
- Selectable clock source for BTCNT2 among ACLK, ACLK/256, and SMCLK. BTCNT1 is driven with the ACLK.
- Interrupt capability.

Table 7.6 Timer_B distinctive features

Feature	x4xx, x1xx, and x2xx	x5xx/x6xx
Programmable length (8, 10, 12, or 16 bits)	Yes	Yes
Double-buffered compare latches w/synchronized loading	Yes	Yes
Three-state Output	Yes	Yes
SCCI	No	No
Double-buffered TBCCRx registers	Yes	Yes
Configurable CCRs[1]	3 or 7	up to 7

Note 1: Timer_A has up to 3 CCRs

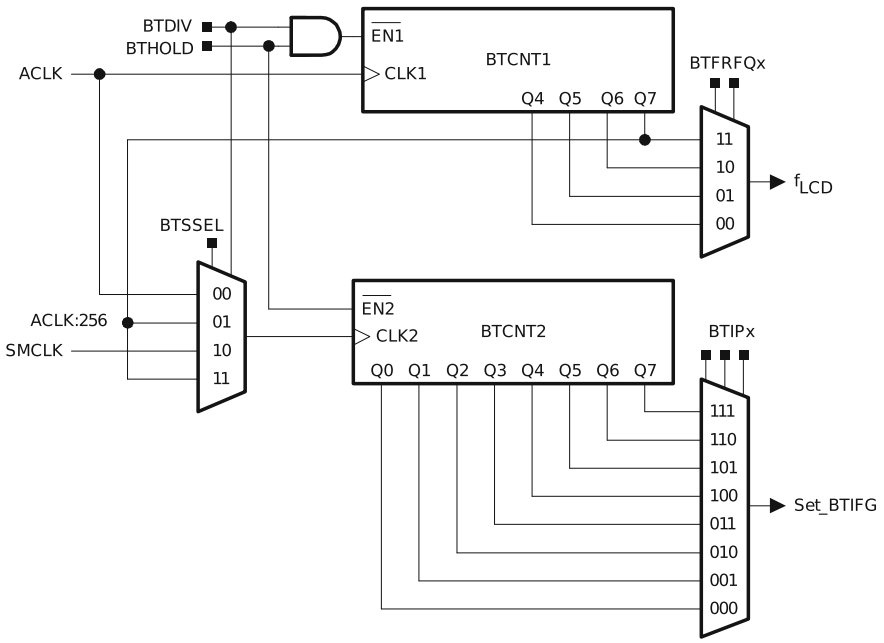


Fig. 7.28 MSP430 Basic Timer_1 block diagram

As with any timer, the BT could be used as an interval timer or an event counter, or to implementing a real-time clock. Although RTC is not present in series x1xx or x2xx, the x5xx/x6xx series include three separate real time clocks: RTC_A, RTC_B, and RTC_C.

The Real Time Clock in the x4xx series has the following features:

- 32-bit counter module with calendar function.
- Calendar and clock mode.
- Automatic counting of seconds, minutes, hours, day of week, day of month, month, and year in calendar mode.
- Selectable clock source.
- Interrupt capability.
- Selectable BCD format.

Table 7.7 summarizes the features included of the three different real time clocks implemented in the x5xx/x6xx MSP430 devices.

Final Remarks on Timer_A Interrupts

Timer_A interrupts are maskable, which means that the GIE bit should be set in the Status Register to call an ISR. The module has two interrupt vectors associated with it: the TACCR0 interrupt vector for TACCR0 CCIFG, and the TAIV interrupt vector associated with all other CCIFG flags and with TAIFG.

Table 7.7 MSP430x5xx/x6xx RTC_A, B, and C implementations

x5xx/x6xx REAL TIME CLOCK	RTC_A	RTC_B	RTC_C
Configurable for calendar function or counter mode	Yes	No	Yes
Provides second, minutes, hours, day of week, day of month, month, and year in calendar mode	Yes	Yes	Yes
Leap year correction	No	Yes	Yes
Interrupt capability	Yes	Yes	Yes
Selectable BCD or binary format	Yes	Yes	Yes
Calibration logic for offset correction in RTC mode	Yes	Yes	See below
Crystal offset error and temperature drift	No	No	Yes
Programmable Alarms	No	Yes	Yes
Operation in LPMx5	No	Yes	LPM3.5
Protection for RTC registers	No	No	Yes
Operation from a separate voltage supply with programmable charger (device-dependent)	No	No	Yes

By hardware, the CCIFG flag is set in capture mode when a value is captured in register TACCRx or in compare mode when TAR counts to the value in TACCRx. The CCIFG action is enabled with the CCIE bit.

When an interrupt request is serviced, i.e. when RETI is executed, the CCIFG flag will be reset. On the other hand, we could also set or clear the CCIFG flag using software. Whenever the corresponding CCIE and GIE bits are set, the CCIFG flags request an interrupt. Timer_A interrupt flag, TAIFG, is bit 0 of the Timer_A Control Register, TACTL whereas CCIE and CCIFG correspond to bit 4 and bit 0 from the Capture/Compare Control Register (TACCTLx).

The TACCR0 CCIFG flag has the highest priority from among the Timer_A interrupt sources. This means that, if the interrupt flags corresponding to TACCR0 and any of the other TACCRx, say TACCR1 are set, then the TACCR0 interrupt will be serviced first, as long as there is not a higher priority interrupt pending. The rest of Timer_A interrupt sources share a common interrupt vector. This means that TACCR1 CCIFG, TACCR2 CCIFG, and TAIFG are sourced by a single interrupt vector. In this case, register TAIV is used to determine which flag has requested an interrupt, and also as an arbiter for multiple Timer_A interrupts. Figure 7.29 illustrates the TAIV register.

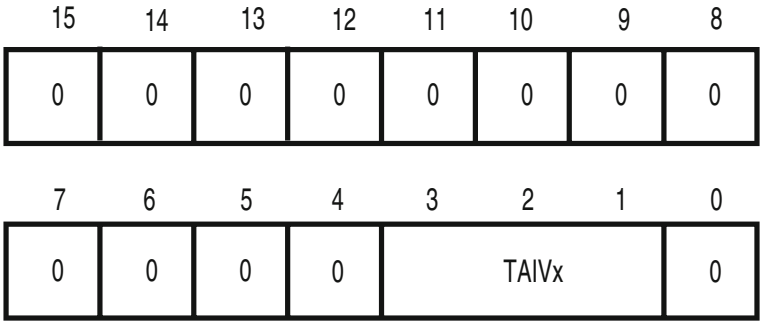


Fig. 7.29 Timer_A Interrupt Vector Register TAIV

Only bits 3, 2, or 1 in TAIV can have a nonzero value. This means that the only values available are even from 00h to 0Eh. From this set, the following cases are highlighted:

- 00h: No interrupt pending
- 02h: Interrupt source is capture/compare 1 (TACCR1 CCIFG), with highest priority.
- 04h: Interrupt source is capture/compare 2 (TACCR2 CCIFG)
- 0Ah: Interrupt source is Timer Overflow (TAIF), with lowest priority.

All other values are reserved. It should be noted that reading from or writing to the TAIV register will automatically reset the highest pending interrupt flag.

The following example from [32] shows an skeleton code used as an illustration on how to handle the different Timer_A interrupt sources.

Example 7.14 *This example from [32][10] is used to show the recommended use of TAIV. Note how the TAIV value is added to the PC register in order to reference the appropriate piece of code from the interrupt handler or interrupt service routine. Courtesy of Texas Instruments, Inc.*

```
;  Interrupt handler for TACCR0 CCIFG
CCIFG_0_HND
;   ... ; Start of handler
reti
;  Interrupt handler for TAIFG, TACCR2, and TACCR1
TA_HND
...
add &TAIV,PC; Add offset to Jump table
reti    ; Vector 0: No interrupt
jmp CCIFG_1_HND ; Vector 2: TACCR1
jmp CCIFG_2_HND ; Vector 4: TACCR2
reti    ;  Vector 6: Reserved
reti    ;  Vector 8: Reserved
TAIFG_HND   ;  Vector 10: TAIFG Flag
```

[10] See Appendix E.1 for terms of use.

```
... ;  Task starts here
reti
CCIFG_2_HND     ;  Vector 4: TACCR2
... ;  Task starts here
reti  ;  Back to main program
CCIFG_1_HND     ;  Vector 2: TACCR1
... ;  Task starts here
reti    ;  Back to main program
```

7.5 Embedded Memory Technologies

Memory technologies in embedded systems provide the fundamental functions of storing programs and data, like in any other computing system. In embedded applications, however, the requirements of storage have traditionally being different in the sense of size and speed. Most small and distributed embedded applications require small amounts of data memory to hold process variables. These values typically require frequent modification so loosing them when the system is shut-down has no lasting effect. Most applications use volatile data memory and rarely require more than a few hundred bytes of space.

Program memory requirements are also moderate. As embedded applications are single functioned, in most cases, a few kilobytes of program memory results sufficient to accommodate the programs needed to complete an embedded application. Unlike data memory, program memory is required to hold its contents when the system is powered-down. This way the embedded system can regain its functionality when powered-up anew. Therefore program memory is provided in the form of non-volatile memory, and it is quite common to find that most embedded systems require less than 64KB of storage for programs.

Of course, if we move into the arena of high-performance systems, the memory requirements might dramatically increase. Depending on the number of tasks to be performed by the system and the amount of data it needs to process to perform its function, will be the requirements of program and data memory. However, when considering the universe of embedded applications, these cases requiring large amounts of memory are the least frequent.

7.5.1 A Classification of Memory Technologies

Embedded memory can be classified according to two main criteria: *storage permanence* and *write ability*. The first criterion refers to the ability of a programmed memory to hold its contents, while write ability, refers to how easily the memory contents can be modified.

Looking at all the memory technologies that have reached mainstream (ROM, PROM, EPROM, EEPROM, FLASH, and RAM), masked, read-only memory (ROM) earns the mark as the least writable and most permanent of all types. The

zeros and ones defining a ROM contents are hardcoded in the chip through the metallization layer in their fabrication process. This characteristic also makes the memory non-volatile. Recall that volatility in memory refers to the loss of contents when the chip is de-energized.

Programmable ROM (PROM) closely follows ROM in terms of write ability and permanence. PROM contents is written by selectively blowing internal metal fuses in each cell via a dedicated programming circuit. This writing method makes cells non-volatile, but only allows writing their contents once as blown fuses cannot be brought back to an un-blown state. PROMs are also called One-time Programmable non-volatile memories or OTP NVM. They are considered more writable than ROM because un-programmed, blank chips can be programmed in the field.

Erasable PROM (EPROM for short), marked a breakthrough in memory technology as it provided a non-volatile cell that could be erased and reprogrammed many times. This was enabled by the introduction of floating-gate transistors, a MOSFET arrangement able to trap charges in its gate by magnifying the effect of "hot carriers". Charges trapped in the double gate structure increase the transistor threshold voltage. Thus the minimum voltage needed to turn on a non-charged transistor will not turn on a charged one, making possible to use the non-stored versus stored charge states as ones and zeros, respectively, in the memory cells. Moreover, the charge retention in the floating gate is retained even when the cell is de-energized, making it non-volatile. Conveniently, bombarding the gate with high energy particles, like UV light photons, allows for removing the charge accumulation, thus erasing the cell.

Erasing an UV EPROM required extended exposure (up to 15 min) to an UV light source. EPROMS were fabricated with a convenient transparent window facilitating the erasure procedure. Writing the cell required application of relatively high voltages (12–18 V) provided by an external circuit for relatively extended periods (milliseconds) compared to the typical read access time of the cell (nanoseconds). Due to these limitations, although EPROMs could be re-written, they needed to be physically removed from the application to be erased and then re-programmed to modify their contents. Nevertheless, with its ability to reprogram its contents, EPROMs moved up in the write-ability scale with respect to PROM. However, their susceptibility to loose contents due to radiation and gradual charge loss gave it a lower mark in storage permanence than ROM and PROM technologies. Under normal conditions an EPROM can retain its contents for over a decade and tolerate erase-write cycles up to thousands of times (the double gate charge trapping ability is lost with time).

The form of modifying EPROM cells brought-up the issue of how the write-ability was provided in a memory chip. In this respect, it is common to distinguish between off- and in-system write ability. The "in-system" denomination is given to the type of memory whose contents can be modified without physically removing it from the system or board where it is installed. Off-system writable memories require moving the chip to specialized hardware to erase them and then re-writing new contents in order to be modified.

EPROM technology further evolved to give floating transistors the ability to release charge by the application of an electric field. This improvement led to the

technology of Electrically Erasable-programmable Read-only Memory (EEPROM). This new cell technology shared the characteristics of EPROMs in terms of non-volatility and storage permanence, but without the necessity of UV to erase the cells. Moreover, the provision of on-chip circuitry to erase and reprogram the cells, enabled this technology to eventually become in-system re-writable, moving up its score in terms of write ability. Further development in floating-gate-based cells and their interface lead to the FLASH technology, discussed in more detail in Sect. 7.5.2.

While non-volatile memory technologies evolved, volatile memory has also seen improvement in different aspects. Non-volatile technologies moved from magnetic core-based to semiconductor-based technologies. Two types of volatile memory are predominant now-a-day: static and dynamic cells.

A static cell uses a flip-flop as storage element. The flip-flop retains its contents as long as the cell is energized, reason for the static denomination. A dynamic cell uses a capacitor as storage element. Due to unavoidable charge leak, the capacitor contents needs to be periodically refreshed to avoid information loss. This refresh requirement is what gives the dynamic denomination to this type of memory. The capacitor contents is also lost when the cell is de-energized, making the technology volatile.

Semiconductor-based volatile memories score the highest mark in terms of write ability due to easiness of being quickly read and written. At the same time, its volatility makes it score the lowest mark in terms of storage permanence. Because of historical reasons, volatile, read-write memory has been called RAM.[11] Two important parameters associated to RAM technologies are their reading and writing speeds, quantified as the memory access time. Static RAM (SRAM) is the fastest, but dynamic RAM (DRAM) is preferred for large memory arrays because of its density. Newer technologies like Ferroelectric RAM (FRAM) are begining to show up in the market. This technology uses ferroelectric materials to implement a micro-magnetic capacitor as storage element, inheriting the non-volatility of early core memory cells with their inherent read-write ability. Current FRAMs are not as dense as the other types of semiconductor memories, but because of its non-volatility advantages result an attractive alternative for embedded applications.

Contemporary embedded systems use fundamentally static RAM for data memory. The relatively small requirements of RAM can be easily accommodated on-chip. Moreover, the modest program memory requirements, coupled with their non-volatility characteristic are accommodated in most contemporary systems using FLASH. The storage permanence, density, and in-system programmability characteristics of embedded FLASH in contemporary microcontrollers make necessary to delve a bit deeper into this technology, as discussed in the next section.

[11] In its origin, Random Access Memory (RAM) designated the type of memory whose contents could be randomly addressed, unlike Sequential Access Memory (SAM) where contents had to be sequentially accessed.

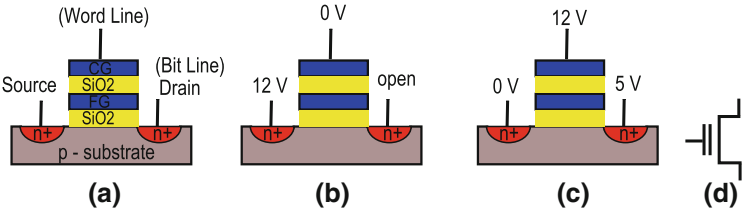


Fig. 7.30 Floating Gate Mosfet (FGMOS) (**a**) Basic structure; (**b**) erasing process; (**c**) Programming; (**d**) Symbol

7.5.2 Flash Memory: General Principles

Flash is a type of EEPROM differentiated by its own name because it may be written and read in blocks. Other EEPROM's need to be completely erased byte by byte, or by very small blocks in some cases, making the process slow.[12]

Memory chips based on floating gate transistors were introduced in the early 1970s. The first flash memory was built by Toshiba in 1984, but it was not until late 1980s that it became reliable enough for mass production. Nowadays, flash memory is the dominant memory type wherever a significant amount of non-volatile, solid state storage is needed.

When flash memory operates in *read mode*, it works like any ROM. Flash memory offers fast read access times, as fast as dynamic RAM, although not as fast as static RAM. The other modes of operation are *erase* and *program*. In the first case, the cells are taken to an "erased state", which may be a logic 0 or logic 1, depending on the hardware configuration. In the programming mode, the cells are taken to (or left in) a logic 1 or logic 0, depending on the programming needs.

Another important and distinctive feature of flash is its capability to write in one block while reading another one, as far as the two blocks are not the same one. It is not possible to program and read simultaneously in the same block.

The basic element used in flash memory is a floating gate MOS transistor (FGMOS), whose structure is illustrated in Fig. 7.30a. This is a MOSFET transistor with two gates, one floating gate (FG) electrically isolated from the rest of the structure, and a control gate (CG) connected to the word line (address). The drain is connected to the bit line, that is, the data. Charge put on floating gate affects the threshold voltage. Since the FG is surrounded by an insulator, the trapped charge remains practically forever, unless modified by the user. When enough electrons are present on this gate, no current flows through the transistor. When electrons are removed from the floating gate, the transistor starts conducting. If a pull down p-mosfet is used, in the first case we have a logic 0, and in the second case a logic 1. The opposite happens if there is no pull-up transistor.

[12] The term "flash memory" was coined because of the speed, like a flash.

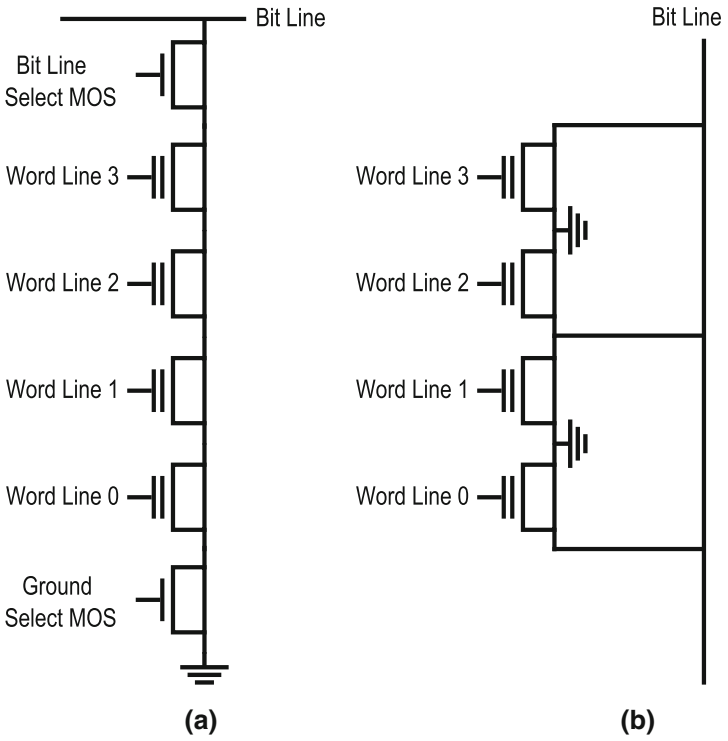


Fig. 7.31 Basic flash architectures: **a** NAND Flash, and **b** NOR Flash

Charging and discharging the FG is done with a relatively high voltage, say 12 V, applied to the CG. Ground or a lower voltage is applied to either the source or drain, as illustrated in Fig. 7.30b, c.

There are two main types of flash memory, NAND and NOR logic gate, so named because of their structures, illustrated in Fig. 7.31. NAND type flash memory may be written and read in blocks (or pages) which are generally much smaller than the entire device. Compared to the NOR type, they are faster to write but slower to read. The NOR type has better endurance and it allows a single machine word (byte) to be written or read independently. Both types are used in several applications such as personal computers, digital audio players, digital cameras, and so on. However, NOR type is preferred for code storage and NAND type for data storage. The reader may consult for more detailed hardware descriptions and working principles elsewhere.

Flash memory is not forever, of course. It can withstand only a limited number of program-erase cycles. The NOR flash endures more cycles than the NAND type. However, this depends not only on the architecture, but also on how the cycles are executed. If a particular memory block is programmed and erased repeatedly without writing to any other blocks, it will wear out before all the other blocks and the whole

storage device will become useless. Also, different cells can be or become worn out or not functioned correctly, causing malfunction of the memory.

To take care of these issues, and to manage the program-erase cycles in flash memory devices, or in systems with this type of memory, engineers include a flash memory controller.

Flash Memory Controller

A flash memory controller, or flash controller, manages the data stored on flash memory and communicates with the CPU, computer or whatever electronic device the memory is serving.

Initially, at fabrication, the flash controller starts by formatting the flash memory and to ensure that the device is operating properly. It maps out bad flash memory cells, and allocates spare cells to be substituted for future failed cells. Part of the spare cells is also used to hold the firmware which operates the controller and other special features for the particular storage device. Finally, the controller creates a directory structure to allow conversion of requests for logical sectors into the physical locations on the actual flash memory chips.

The system communicates with the controller whenever it needs to read data from or write data to the memory. The controller uses a technique called *wear leveling* to distribute writes as evenly as possible across the flash blocks to prevent unfair wearing of blocks. Ideally, wear leveling technique enables every block to be written to its maximum life.

7.5.3 MSP430 Flash Memory and Controller

Except for the original 'x3xx series, now obsolete, all other series have models with flash memory. MSP430 flash memory is partitioned as follows:

- Information segments A, B, C and D.
- Bootstrap loader (BSL) segments A, B, C and D
- Main flash memory, partitioned in banks
  - Banks are partitioned into segments 0, 1, 2, ….

Only series '5xx/'6xx have BSL segments. Small models and some series may contain only one or two information segments. Also, the main memory may consist of only one bank, depending on the size. Flash memory sizes may be as small as 0.5 kB (model 'G2001) or up to 512 kB as found in some models of the 'x6xx series.

All flash-based MSP430 devices incorporate a flash controller that allows for the execution of code from the same flash module in which the software is concurrently modifying the data or re-programming code segments. The flash module of MSP430 consists of three blocks:

- Control logic: State machine and the timing-generator control of flash erase/program

Table 7.8 Flash control registers in MSP430 models

Series	FCCTL1	FCCTL2	FCCTL3	FCCTL4
x1xx	Yes	Yes	Yes	No
x2xx	Yes	Yes	Yes	Yes1
x4x	Yes	Yes	Yes	Yes1, 2
x5/xx	Yes	No	Yes	Yes
x6/xx	Yes	No	Yes	Yes

1: Not available in all devices. Must consult the specific device data sheet 2: The higher byte is 00h on reset for this series

- Flash protection logic: Protection against inadvertent erase/program operations
- Programming voltage generator: An integrated charge pump that provides all voltages required for flash erase/program

The module controller consists of four 16-bit Flash Controller Control registers, FCCTLx, $x = 1, \ldots 4$, not all present in all models, as shown in Table 7.8. All registers are password protected. The high byte is 96h at reset or after a writing execution, and must be A5h when writing.[13] The low bytes are illustrated in Fig. 7.32.

In addition to these registers, the Interrupt Enable 1 register, which contains information for several peripheral modules, houses as bit 5 the *Access Violation Interrupt Enable* bit, ACCVIE, which is used to enable Flash Memory Access Violation Interrupts (ACCVIFG).

Bits FSSELx from control register FCCTL2 select the clock source from ACLK, SMCLK, or MCLK for the flash timing generator. The frequency of the generator must comply with some specifications and often require the clock source frequency to be divided. This is accomplished with the FNx bits.

Memory size writing is controlled by the bits BLKWRT and WRT; erasing is controlled with MERAS and ERASE. The size of writing/erasing is as shown in Tables 7.9 and 7.10, respectively.

Information segment A is locked separately from all other segments with the LOCKA bit. When LOCKA = 1, Segment A cannot be written or erased, and all information memory is protected from erasure during a mass erase or production programming. When LOCKA = 0, it is treated as any other flash memory segment.

Below is a code fragment, courtesy of Texas Instruments, Inc. for writing a block of data to the flash memory of an MSP430 of the x4xx family. User guides provide an extensive detailed treatment of flash memory and also several code examples as well as recommended management strategies.

```
; Write one block starting at 0F000h.
; Must be executed from RAM, Assumes Flash is already erased.
; 514 kHz < SMCLK < 952 kHz
; Assumes ACCVIE = NMIIE = OFIE = 0.
```

[13] This is a case where standard constant naming is not as standard as one should expect. In series 5/6, the name for the reading and writing passwords are, respectively, “FRPW” and “FWPW”. In all other cases, the constants are named “FRKEY” and “FWKEY”, respectively. The user should consult the guides or the header files if necessary.

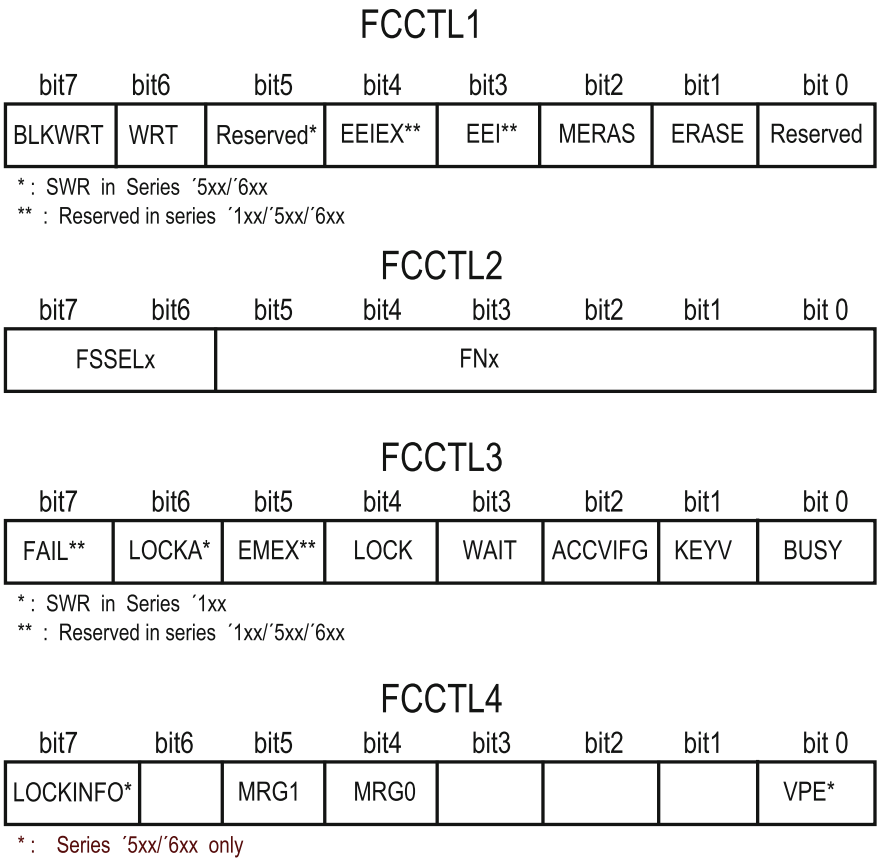


Fig. 7.32 MSP430 low bytes of flash memory control registers

Table 7.9 Flash memory writing mode

BLKWRT	WRT	Action
0	0	No write
0	1	byte/word write
1	0	long word (32 bits) write[1]
1	1	Mass erase (

1: Series ’5xx/’6xx only

```
        MOV #32,R5                  ; Use as write counter
        MOV #0F000h,R6              ; Write pointer
        MOV #WDTPW+WDTHOLD,&WDTCTL  ; Disable WDT
L1      BIT #BUSY,&FCTL3            ; Test BUSY
        JNZ L1                      ; Loop while busy
        MOV #FWKEY+FSSEL1+FN0,&FCTL2 ; SMCLK/2
        MOV #FWKEY,&FCTL3           ; Clear LOCK
        MOV #FWKEY+BLKWRT+WRT,&FCTL1 ; Enable block write
```

Table 7.10 Flash memory erasing mode[1]

MERAS	ERASE	'x1xx, 'x2xx, and 'x4xx	'x5xx/'x6xx
0	0	No erase	No erase
0	1	Individual segment	Individual Segment
1	0	Main memory segments	One bank erase
1	1	Erase all flash memory[2]	Mass erase all banks

1: MSP430FG461x have another bit, GMERAS, for more options. 2: Information Segment A is not erased when locked with LOCKA = 1.

```
L2    MOV Write_Value,0(R6)        ; Write location
L3    BIT #WAIT,&FCTL3             ; Test WAIT
JZ    L3                           ; Loop while WAIT = 0
      INCD R6                      ; Point to next word
      DEC R5                       ; Decrement write counter
      JNZ L2                       ; End of block?
      MOV #FWKEY,&FCTL1            ; Clear WRT,BLKWRT
L4    BIT #BUSY,&FCTL3             ; Test BUSY
      JNZ L4                       ; Loop while busy
      MOV #FWKEY+LOCK,&FCTL3       ; Set LOCK
      ...                          ; Re-enable WDT if needed
```

7.6 Bus Arbitration and DMA Transfers

Direct Memory Access (DMA) techniques offer a convenient way to accelerate data transfers from peripherals or memory to memory or viceversa. In embedded systems with such a capability a DMA also offers enhanced opportunities for implementing low-power solutions for data transfer intensive applications.

DMA controllers are however a special kind of peripherals with bus master capabilities, as they are able to substitute the CPU in controlling the bus activity. As such, in order for a DMA to gain control of the buses, a bus arbitration procedure has to mediate. In this section we introduce bus arbitration as a preamble to the discussion of DMA controllers. The next section is taylored to provide a better understanding of the fundamental concepts on the subject of bus arbitration.

7.6.1 Fundamental Concepts in Bus Arbitration

Bus arbitration is the process through which devices with bus mastering capabilities can request to a default bus master the control of a set of shared buses (address, data, and control) and become temporary bus masters.

In a computer system, a bus master capable device is one that has the resources to command the system buses. This is, a device capable of issuing addresses, managing control signals, and regulating the activity on the data bus.

In single processor systems and small embedded applications, the CPU is typically the only device with bus master capabilities, and thus the bus master by default. In such a system, every transaction occurring in the system buses is regulated by the CPU. An example of such an structure is the classical computer system organization depicted back in Chap. 3, when the general architecture of a microcomputer system was introduced and explained with Fig. 3.1.

When a system contains devices with bus mastering capabilities other than the CPU, such as math co-processors, graphical processing units, direct memory access (DMA) controllers, or even other CPUs like occurs in multi-core or multi-processor systems, the access to a shared system bus becomes a source of contention. At any given moment, multiple devices might need to use the buses, but only one master at a time can have control of them. This situation creates the necessity of establishing a mechanism in which each potential bus master can request control of the buses and, in an orderly fashion, control can be granted to requesting devices without creating conflicts. This is the scenario where bus arbitration becomes a necessity.

When a system has multiple bus master capable devices, the device that is most likely to request the buses or the one that controls the buses when no other potential master is requesting them is designated as the default bus master. Usually, the default bus master is given the task of booting the system upon a power-up or soft reset condition. Therefore, bus arbitration transactions place requests to the default bus master.

Basic Bus Arbitration Protocol

Consider a simple scenario where a CPU and one additional bus master capable device coexist, as illustrated in Fig. 7.33. For a bus arbitration transaction taking place between these two devices, both of them must have been designed to support bus arbitration. This requires an ability in the involved devices for accepting requests from external potential masters ($\overline{\text{BRQ}}$ CPU input), some way for the CPU notifying

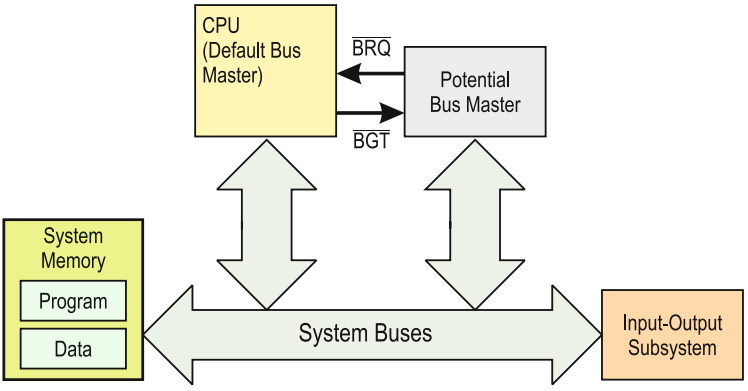


Fig. 7.33 Simple scenario for a bus arbitration transaction

the potential master when a request has been granted ($\overline{\text{BGT}}$ CPU output), and some mechanism for the CPU learning that buses are free anew.

Assuming the CPU is the default master, the bus arbitration transaction begins when the potential master asserts its bus request signal $\overline{\text{BRQ}}$ to the default master. This situation could result either because the potential master has its own tasks to perform, or because the CPU delegated some task to it. If the CPU were in the middle of an instruction or in some other state where it could not honor the bus request, it would just complete the current instruction or continue its current task until reaching a state where the buses could be granted.

When the CPU becomes able to accept the bus request (assuming $\overline{\text{BRQ}}$ is still asserted) it would proceed to disable its internal bus drivers, isolating itself from the system buses, and then would assert the bus grant signal $\overline{\text{BGT}}$. At this point if the CPU had internal or local resources that allow for it to continue processing without accessing the buses, it would continue to do so. Otherwise it would enter a suspend mode waiting for the release of the buses.

The assertion of the $\overline{\text{BGT}}$ line would be the indication to the requesting master that it can take control over the buses. Thus, it would now enable its own internal bus drivers, taking control of the buses to perform whatever task for which it requested to be master.

When the potential master completes its task, it would disable its internal bus drivers, isolating itself from the bus and then would unassert the $\overline{\text{BRQ}}$ line. This would be the indication to the CPU that the buses are free. The CPU would then proceed by unasserting the $\overline{\text{BGT}}$ signal and enabling its bus drivers, becoming again the bus master. Figure 7.34 shows the timing in signals $\overline{\text{BGT}}$ and $\overline{\text{BGT}}$ during this process.

There are different ways in which the request-grant-release process can be indicated. Intel x86 processors use a single bidirectional line that is first pulsed by the potential master to make a request, then pulsed by the default master to grant the buses and last pulsed by the potential master to release the buses. Other processors like the ARM use a 3-line scheme where a request and grant signals operate as described above but granting the bus for a single clock cycle. A third line, BLOK (Bus Lock) prevents the potential master from releasing the buses from one cycle to another. In general, handshaking protocols change from one processor to another, but the transaction is remains the same.

Bus Arbitration Schemes

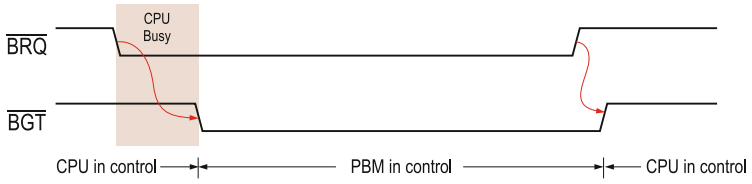


Fig. 7.34 Timing of a bus request and granting process

When a system has multiple potential masters, the arbitration solution requires a few additional considerations than those illustrated in the previous section as it would become necessary to introduce prioritization schemes to handle simultaneous request from multiple potential masters.

In a sense, this problem is not different to that of interrupt priority discussed earlier in Sect. 7.1.5. The solutions to the priority arbitration problem use the same approaches as for interrupt prioritization: serial daisy chains or central arbitration.

In a daisy-chain solution, simple hardware links like those for the interrupt approach are used, with the same simplicity advantages and hard-wiredness disadvantages that were discusses for the interrupt case.

A central arbiter is usually a more flexible solution, as like in the case of interrupts it provides rotative priorities, programmability, and masking levels that centralize the solution.

7.6.2 Direct Memory Access Controllers

Conventional data transfers from peripheral devices to or from memory or from memory to memory in microprocessor-based systems can occur either via polling or via interrupts. In either case, the CPU plays a central role in two aspects.

First, every transfer is the result of the executing of some data transfer instruction that needs to be fetched and decoded before it can be executed. The second aspect is that as transfers require specifying source and destination addresses, these two actions cannot occur in a single bus cycle. So, the CPU needs to perform them with two separate instructions, the first reading the source data into a CPU register and the second transferring the character from the CPU register into the destination address. Figure 7.35 graphically illustrates this process for an I/O to memory transfer.

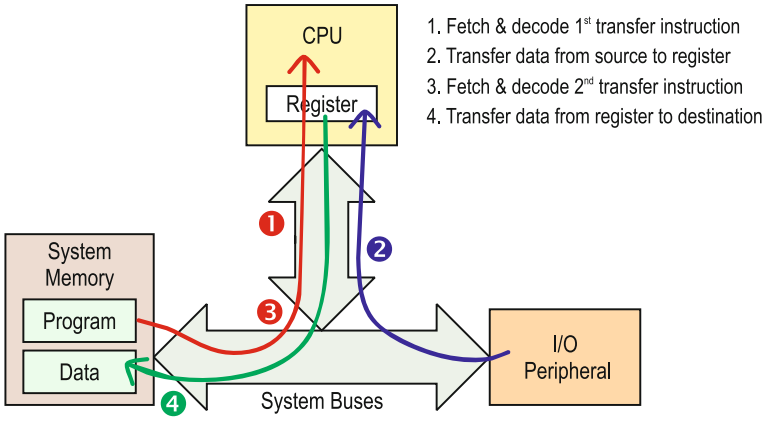


Fig. 7.35 Sequence of events in conventional data transfer

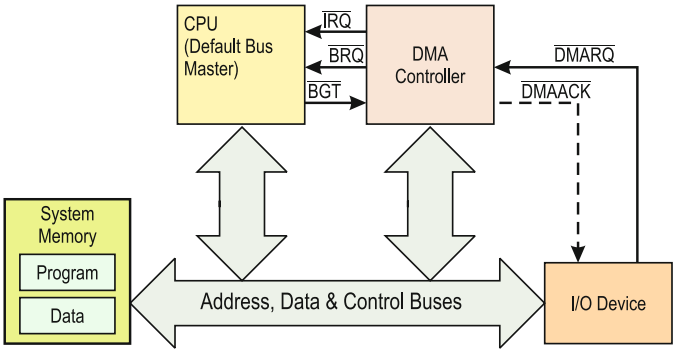


Fig. 7.36 Connecting a DMA controller to an MPU system and I/O device

By considering also the additional overhead introduced by either the polling or interrupt-based scheme (checking a flag or branching to the corresponding ISR and returning), it is possible to realize the considerable number of clock cycles it takes to make a transfer under the traditional scheme.

Although many embedded applications will not be affected by this overhead, high-speed systems might face severe limitations in their throughput by this scheme. Consider for example the embedded application retrieving bits and bytes from a terabyte hard drive spinning at 7200RPM through a magnetic pick-up head, or the embedded controller sending streaming video data to a digital TV screen. None of these applications would tolerate the overhead of a conventional transfer. A solution for such high speed applications would be using a DMA controller (DMAC).

DMA Operation

A DMA controller can operate as an I/O device when is configured by the CPU or as a bus master device. When operating as a bus master, the DMA replaces the CPU in regulating data transfers in the system buses in such a way that data can be directly accessed to and from memory without executing CPU instructions or passing the data through the CPU. This is what gives this controller the "Direct Memory Access" denomination. Figure 7.36 illustrates the way a DMA controller is inserted in a microprocessor-based system to support memory-to-memory or memory to/from I/O transfers.

A DMAC receives transfer requests from its associated I/O devices and can optionally respond with an acknowledgment signal. As a result of a DMA request, the DMA initiates a bus arbitration transaction with the CPU to gain control of the system buses. When the buses are granted, the DMA, as bus master, can generate the address signals to point to memory locations and/or select the I/O device such that data transfers can directly occur between memory and the device without the intervention of the CPU. The DMA also generates the required control signals to establish the transfer direction.

When operated as an I/O device, a DMAC needs to be configured by the CPU to specify the initial addresses where transfers will take place, the number of words that

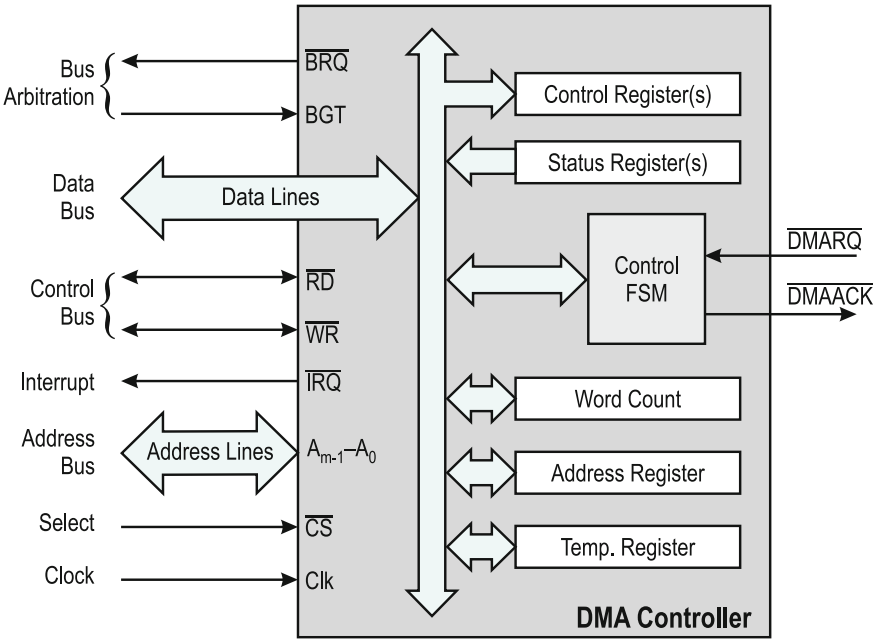


Fig. 7.37 Minimal structure of a one-channel DMA controller

it needs to transfer, and the modality and desired type of transfer to perform. After configuration, then DMAC needs to be enabled, allowing it to initiate bus arbitration transactions when its associated peripheral needs to perform a transfer. Transfer requests can also be initiated by the CPU. In order to support such operations, the minimal structure of a DMA controller is expected to be similar to that illustrated in Fig. 7.37.

An address register is needed to specify the initial address of a transfer. A second address register and a temporary data register are also required if memory-to-memory transfers are to be supported (discussed below as two-cycle transfers).

A word count register is required to specify how many words the DMAC needs to transfer in a session.

Finally, one or several control and status registers will be required for configuring and indicating the modes and status conditions developed during operation.

The CPU side interface of the DMAC includes bidirectional connections to address, control, and data buses to allow the dual operation of the DMA as I/O peripheral when accessed by the CPU or as bus master when coordinating transfers. In the device side, the minimum requirement calls for a DMA request input to accept transfer requests from a peripheral I/O device. Optionally a DMA acknowledgement signal might be also provided.

Most DMA controllers are able to support more than one I/O peripheral by featuring multiple *DMA Channels*. One DMA channel is just an arrangement of Word

Count, Address, and Temp. Register per each supported I/O device. Each supported I/O device has a dedicated DMARQ line, and internally the DMAC takes care of prioritizing requests received from the supported peripherals.

Burst Mode Versus Cycle Stealing

A DMA controller can request the buses to transfer data in one of three modalities: *burst*, *cycle stealing*, or *transparent*.

In a burst mode transfer, the DMAC retains command of the buses until performing the transfer of an entire block of data. This mode is preferred when performing memory-to-memory transfers, transfers from fast peripherals, or when devices feature a multi-word buffers that need to be retrieved or filled in a single session.

In cycle stealing mode, the DMA controller performs a bus arbitration transaction for every word transferred. This mode is convenient for slower devices whose data throughput could generate idle time in the bus while the DMA waits for new data or when interleaved operation with the CPU is desired. After each word transfer, the buses are released and requested anew when the peripheral device places the next DMA transfer request. This operation can also be controlled via a timer-counter for establishing uniform sampling rates in data acquisition applications.

The transparent mode requires additional signalization between the DMAC and the CPU for determining periods where the CPU is not using the buses. This way the DMA can become bus master without even performing a bus arbitration transaction, operating transparently from the CPU point of view. As this mode requires a DMAC capable of sensing the processor bus activity, it is supported in only a reduced number of controllers.

Regardless of the transfer modality, DMA controllers are typically configured to perform transfers of blocks of data containing hundreds or thousands of words. As this process might take some time to complete, an interrupt signal can be optionally generated to the CPU to indicate the end of a transfer, allowing the CPU to react to the event.

One-cycle Versus Two-cycle DMA Transfer

Depending on the path through which data traverses and the number of cycles necessary to complete a transfer, a DMAC can perform one- or two-cycle transfers.

Two-cycle Transfers: In two-cycle transfers the DMA controller uses two bus cycles to complete a transfer. In the first cycle, the DMAC performs a read access from the source location and stores the datum into a temporary register inside the DMAC. In the second cycle the DMAC performs a write access to the destination location transferring there the contents of the temporary register. This process is illustrated in Fig. 7.38 with an a memory to I/O transfer.

The sequence of steps taking place for performing the transfer illustrated in the figure can be outlined as follows:

Step 1. A $\overline{\text{DMARQ}}$ signal is issued by the I/O device. This could be initiated by the CPU through the I/O device interface to send data to its associated peripheral or by the peripheral itself.

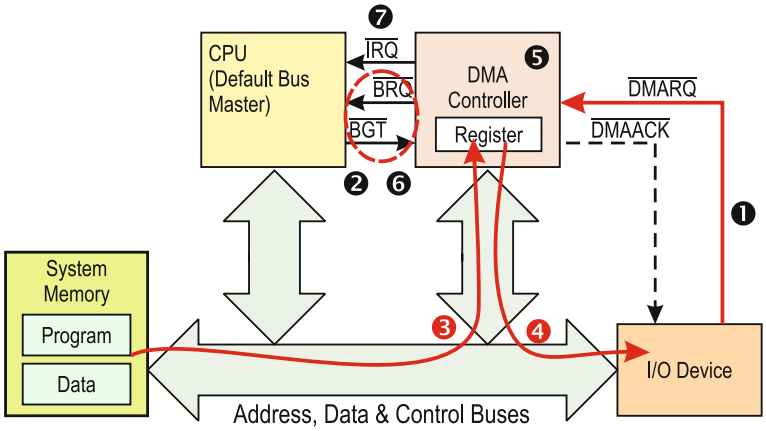


Fig. 7.38 Sequence of events in a two-cycle DMA transfer

Step 2. The DMA controller initiates a bus arbitration transaction to gain control of the buses. The CPU becomes isolated from the buses and grants the buses to the DMA controller. At this point the DMA becomes the bus master.

Step 3. This is the first cycle of DMA transfer. The DMAC sends the memory address of the source data, activates the $\overline{\text{RD}}$ signal, and the datum is transferred from the source location into the DMA temporary register via the data bus.

Step 4. This is the second cycle in the DMA transfer. The DMA controller places in the address bus the destination address, which corresponds to the I/O device interface, activates the $\overline{\text{WR}}$ signal, and the datum is transferred from the temporary register into the I/O device interface.

Step 5. Internally, de DMAC updates the source address and transfer counter. If the transfer were in burst mode, go back to Step 3 until the entire block of data is transferred (transfer count $= 0$). In cycle stealing mode, continue to Step 6.

Step 6. The DMAC releases the buses and the CPU becomes bus master.

Step 7. If the transfer counter is zero, optionally, the DMAC, as an I/O peripheral, issues an interrupt to the CPU.

Observe that as two different bus cycles are used, one for reading the source and other for writing the destination, this modality allows performing memory-to-memory transfers since two different addresses can be issued, one in each cycle. This is the most commonly used transfer mode in DMA controllers.

One-cycle Transfers: In one-cycle transfers the DMA controller performs a complete a transfer within a single bus cycle. Two conditions are necessary to achieve such a task. First, a DMAACK signal from the DMAC to the I/O device interface becomes necessary, and second the CPU must be able to issue separate signals to strobe memory and I/O transfers.

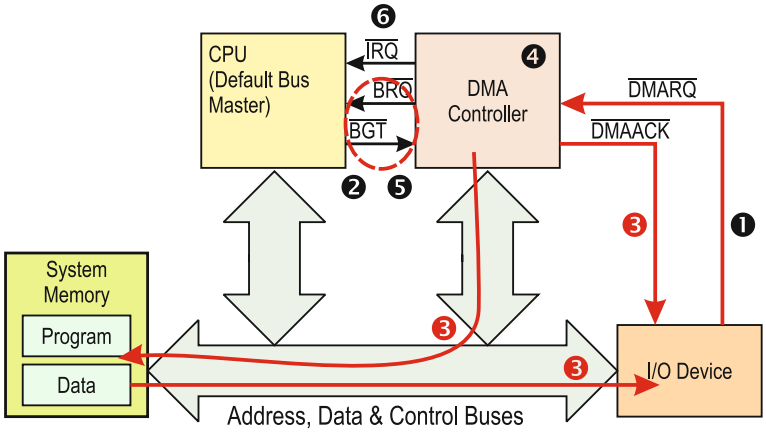


Fig. 7.39 Sequence of events in a one-cycle DMA transfer

The first condition allows using the DMAACK signal as a select strobe for the I/O device peripheral such that it can be selected without having to issue its explicit address. The second condition implies having separate read and write signals for memory and I/O locations, an strategy called *separate memory and I/O spaces*.[14]

Assuming both conditions are satisfied, the way the transfer is completed is simple. When the DMAC becomes master, within a single bus cycle it issues the address of the memory location involved in the transfer along with the memory access strobe ($\overline{\text{MemRD}}$ or $\overline{\text{MemWR}}$) AND asserts the $\overline{\text{DMAACK}}$ with the corresponding I/O strobe ($\overline{\text{IOWR}}$ or $\overline{\text{IORD}}$) allowing data to flow directly from source to destination within a single bus cycle. This process is illustrated in Fig. 7.39 with a memory to I/O transfer.

The sequence of steps taking place for performing the transfer can be outlined as follows:

Step 1. A $\overline{\text{DMARQ}}$ signal is issued by the I/O device.
Step 2. The DMA controller initiates a bus arbitration transaction, becoming the bus master.
Step 3. The DMAC sends the memory address of the source data, activates the $\overline{\text{MemRD}}$ signal selecting the source memory location, and simultaneously activates $\overline{\text{DMAACK}}$ signal selecting the I/O device. While still in the same cycle, the DMAC also activates the $\overline{\text{IOWR}}$ strobe making possible for the word coming out from memory in the data bus to be strobed into the I/O device. This way the whole transfer is completed in just once bus cycle.
Step 4. Internally, de DMAC updates the source address and transfer counter. If the transfer were in burst mode, go back to Step 3 until the entire block of data is transferred (transfer count = 0). In cycle stealing mode, continue to Step 5.
Step 5. The DMAC releases the buses and the CPU becomes bus master.

[14] This modality uses independent address spaces for memory and I/O devices.

Step 6. If the transfer counter is zero, optionally, the DMAC, as an I/O peripheral, issues an interrupt to the CPU.

One-cycle DMA transfers can achieve higher transfer rates than two-cycle transfers, but the conditions imposed on the system hardware to make them possible make this a restrictive transfer type. First, additional decoding logic is needed in the I/O device interface to accommodate the DMAACK signal as a second device select line. Second, separate memory and I/O spaces is rarely supported in contemporary systems. Most microprocessors now-a-day use memory-mapped I/O[15] For these reasons only a limited number of systems are able to support this type of transaction.

7.6.2.1 Programming Fundamentals

In a direct memory access process using a data transfer controller, the CPU must provide information to the controller about the process. Although each data transfer controller has its own features that may distinguish it from others, there are some general guidelines to be considered.

First, the user must decide what device(s) will trigger the DMA. Alternatively, transfers might be initiated by a software trigger.

Second, for each DMA channel active the user must provide starting addresses for source and destination, as well as the number of words to be transferred. If the controller has the option, the increasing/decreasing direction of addresses in transfers involving multiple words. Note that this is independent of wether the controller would work in burst mode or cycle stealing.

Third, the user must take the application into consideration when planning the transfer mode. If all data is needed before proceeding further, or the system bus will not be used while transfer is taking place, then the burst or block transfer mode is the natural choice. For example, to take the average of a data set, we must have all data available first. If this is the system's task, to acquire data and find average, there is nothing to do while collecting data, so the CPU may be put into a low power mode.

If, on the other hand, buses are needed for other tasks, cycle-stealing mode is preferred.

Most controllers work on a two-cycle method, mainly because most contemporary architectures use memory-mapped I/O. If one-cycle transfers are required, the user should look for an appropriate controller, preferably with the two options and ensure the used processor will be able to support it.

[15] Memory-mapped I/O places memory and I/O peripherals in the same address space.

7.6.3 MSP430 DMA Support

Apart from the data transfer controller that may be included in the ADC10, several MSP430 models have a DMA controller peripheral. There are several channels available in the different models. Its features include:

- Several independent transfer channels, depending on MSP model
- Configurable DMA channel priorities
- Supports only two-cycle transfers, taking two MCLK clock periods per transfer
- Ability to perform byte-to-byte, word-to-word, and mixed byte/word transfers
- The maximum block sizes are up to 65,535 bytes or words
- Configurable transfer trigger selections
- The ability to select edge or level-triggered DMA transfer requests
- Support for single, block, or burst-block transfer modes

The DMA is in general configured with several control registers, and each channel has four associated registers for operation. The entire DMA controller module features a single, multi-sourced interrupt vector shared by all supported channels. This makes necessary to use either flag poll or calculated branching in the DMA ISR.

The main differences among the different MSP430 models supporting DMACs are summarized in Table 7.11

Below we describe the registers used the MSP430x2xx family as an example. The reader is referred to the specific User's Guide and dataasheet of the device used for more information.

7.6.3.1 MSP430 DMA Registers for 'x2xx Series

The DMA has three general registers and in addition each channel has four associated registers. The general registers include:

DMA control 0 (DMACTL0) This register defines what event triggers each channel. The event may be a software request or an interrupt request from one peripheral. The selection is device dependent, so the user must consult the respective data sheet.

Table 7.11 DMA Registers and channels in MSP430 families

Series	Number of Control Reg.	Number of Channels	Interrupt vector Register
'x1xx	2	3	No
'x2xx	2	3	Yes
'x4xx	2	3	Device specific
'x5xx&'x6xx	4	Up to 8	Yes

DMA control 1 (DMACTL1) It works with three bits:

- **DMAONFETCH** (Bit 2): 0 if the transfer occurs when triggered; 1 if transfer occurs after next instruction;
- **ROUNDROBIN** (Bit 1): 0 for DMA channel priority DMA0 - DMA1 - DMA2; 1 for DMA channel priority changes with each transfer; and
- **ENNMI** (Bit 0): 0 if transfer cannot be interrupted by a non maskable interrupt; 1 if it can. In this case, the DMA finishes current transfer and then stops.

DMA interrupt vector (DMAIV)

Each channel has the following registers (x=0, 1, or 2):

DMA channel x control (DMAxCTL): This register determines

- the transfer mode,
- independent up/down directions for destination and source addresses in block transfers,
- independent source and destination byte/word sizes,
- triggering of the process.
- turns off or on the DMA channel,
- includes the interrupt flag and interrupt enabling, and
- includes a flag to signal interrupted transfer.

DMA channel x source address (DMAxSA) It contains the source address for single transfers or the first source address for block transfers. The register remains unchanged during block and burst-block transfers.

DMA channel x destination address (DMAzDA) It contains the destination address for single transfers or the first destination address for block transfers. The register remains unchanged during block and burst-block transfers.

DMA channel x transfer size (DMA0SZ) it defines the number of byte/word data per block transfer, up to 65,535. DMAxSZ register decrements with each word or byte transfer. When DMAxSZ decrements to 0, it is immediately and automatically reloaded with its previously initialized value.

7.6.3.2 DMA Transfer Modes

DMA transfers occur when they are triggered by an event, either requested by software of by the peripheral. The transfer mode is defined by the DMADTx bits 14-12 of the DMAxCTL register, and activated with the DMAEN (DMA enable) bit. The transfer modes are:

Single line transfer: Each transfer requires a trigger, and DMAEN is disabled after transfer

Block transfer: A complete block transfer occurs when triggered, and DMAEN is disabled after the complete transfer is achieved.

Burst-block transfer: Is a block transfer with interleaved CPU activity. DMAEN is disabled at the end of the block transfer.

Repeated single line transfer: Each transfer requires a trigger, but DMAEN remains enabled.

Repeated block transfer: Each block transfer requires a separate trigger, but DMAEN remains enabled.

Repeated burst-block transfer: Each burst-block transfer requires a separate trigger, but DMAEN remains enabled.

7.6.3.3 Triggering and Stopping DMA

DMA channels are independently configured for the trigger source with the DMAxTSELx bits with the DMAEN bit = 0. The operation may be started by software of by peripherals such as timers, ADC, DMA channel, etc. The set of peripherals and the selection is device specific.

The trigger can be edge-sensitive or level sensitive. When DMALEVEL = 0, the DMA is triggered when the signal goes from low to high; when DMALEVEL = 1, the DMA operates while the triggering signal is high. Pay attention to the fact that the only level-sensitive trigger is the external DMA signal.

Maskable interrupts do not affect DMA operation. On the other hand, DMA can stop an interrupt service execution. It is therefore necessary to disable the DMA prior to ISR execution if interruption is unacceptable.

Any type of transfer may be interrupted by an NMI only if the ENNMI is set. Block and burst-block transfers can also be halted by clearing DMAEN.

Finally, the DMAONFETCH bit controls when the CPU is halted for a DMA transfer. The CPU may halted immediately when a trigger is received and the transfer begins when DMAONFETCH = 0 . When DMAONFETCH = 1, the CPU finishes the currently executing instruction before the DMA controller halts the CPU and the transfer begins. The DMAONFETCH bit must be set when the DMA writes to flash memory to prevent unpredictable results.

7.6.3.4 Application Examples

To configure the DMA operation, consider at least the following actions:

1. Configure trigger source with appropriate control register (DMACTL0 for families '1xx, '2xx and 4'xx)
2. Define source and destination addresses, as well as the number of transfers
3. Define the addressing mode for transfer
4. Define the direction (unchanged, up or down) for source and destination addresses – default is unchanged.
5. Enable DMA and turn on the DMAONFETCH bit if necessary.

Let us work some configuration examples for the DMA in '2xx series:

Example 1 We want to reverse the upward storage sequence of N bytes in a RAM segment starting at address `SourceAddr`.

Solution: For that reason, we store it in a reverse order at a RAM segment whose highest address is `Dest_up_Addr`. Transfer is done in one block transfer by software request. The following sequence of instructions do the work.

(A) Assembly instructions:

```
Setup_DMA:  mov.w  #SourceAddr,&DMA0SA       ; Source Block Address
            mov.w  #Dest_up_Addr,&DMA0DA     ; Destination Block Address
            mov.w  #N,&DMA                   ; Block size
            mov.w  #DMADT_2+DMADSTINCR_2+DMASRCINCR_3,&DMA0CTL
                                             ; Block Trans., decrement
                                             ; Dest. Addr, and Inc.Src. addr.
            bis.w  #DMASRCBYTE|DMADSTBYTE,&DMA0CTL    ; Byte transfers
            bis.w  #DAEN,&DMA0CTL            ; Enable DMA
                                             ;
                                             ;
Init_trans  bis.w  #DMAREQ,&DMA0CTL          ; Software trigger for DMA
```

(B) C instructions:

```
DMA0SA = SourceAddr;                       // Source block address
DMA0DA = Dest_up_Addr;                     // Destination single address
DMA0SZ = N;                                // Block size
DMA0CTL = DMADT_2+DMADSTINCR_2+DMASRCINCR_3; // Block Trans., decrement
                                           // Dest. Addr, and Inc.Src. addr.
DMA0CTL |= DMASRCBYTE|DMADSTBYTE    ;      //Byte transfers
DMA0CTL |= DAEN,&DMA0CTL            ;      //Enable DMA
                                           //
                                           //
DMA0CTL |= DMAREQ;                         //  trigger transfer
```

Example 2 (*Courtesy of Texas Instruments*: MSP430x26x Demo - DMA0, single transfer Mode UART1 9600, ACLK, by B. Nisarga, Texas Instruments, 2007.)[16]

The string “Hello World” is transferred as a block to U1TXBUF using DMA0. UTXIFG1 WILL trigger DMA0. The rates is 9600 baud on UART1. The Watchdog atimer triggers block transfer every 1000ms. Level sensitive trigger used for UTXIFG1 to prevent loss of initial edge sensitive triggers—UTXIFG1 which is set at POR.

ACLK = UCLK 32768Hz, MCLK = SMCLK = default DCO 1048576Hz

Baud rate divider with 32768hz XTAL @9600 = 32768Hz/9600 = 3.41 (0003h)

(A) Assembly code

```
#include "msp430x26x.h"
;-------------------------------------------------------------------------
LF          EQU     0ah                    ; ASCII Line Feed
CR          EQU     0dh                    ; ASCII Carriage Return
;-------------------------------------------------------------------------
```

[16] See Appendix E.1 for terms of use.

```
            RSEG    CSTACK                  ; Define stack segment
;-------------------------------------------------------------------------
            RSEG    CODE                    ; Assemble to Flash memory
;-------------------------------------------------------------------------
RESET       mov.w   #SFE(CSTACK),SP         ; Initialize stackpointer
StopWDT     mov.w   #WDT_ADLY_1000,&WDTCTL  ; WDT 1000ms, ACLK,
                                            ; interval timer
            bis.b   #WDTIE,&IE1             ; Enable WDT interrupt
SetupP3     bis.b   #BIT6+BIT7,&P3SEL       ; P3.6,7 = USART1 TXD/RXD
SetupUSCI1  bis.b   #UCSSEL_1,&UCA1CTL1     ; ACLK
            mov.b   #3,&UCA1BR0             ; 32768Hz 9600 32k/9600=3.41
            mov.b   #0,&UCA1BR1             ; 32768Hz 9600
            mov.b   #UCBRS_3,&UCA1MCTL      ; Modulation UCBRSx = 3
            bic.b   #UCSWRST,&UCA1CTL1      ; **Initialize
                                            ; USCI state machine**
SetupDMA0   mov.w   #DMA0TSEL_10,&DMACTL0   ; UTXIFG1 trigger
            movx.a  #String1,&DMA0SA        ; Source block address
            movx.a  #UCA1TXBUF,&DMA0DA      ; Destination single address
            mov.w   #0013,&DMA0SZ           ; Block size
            mov.w   #DMASRCINCR_3+DMASBDB+DMALEVEL,&DMA0CTL; Repeat,
                                            ; inc src
Mainloop    bis.w   #LPM3+GIE,SR            ; Enter LPM3 w/ interrupts
            nop                             ; Required only for debugger
                                            ;
;-------------------------------------------------------------------
WDT_ISR;    Trigger DMA block transfer
;-------------------------------------------------------------------
            bis.w   #DMAEN,&DMA0CTL         ; Enable
            reti                            ;
;-------------------------------------------------------------------
String1     DB      CR,LF, 'Hello World'
;-------------------------------------------------------------------
            COMMON  INTVEC                  ; Interrupt Vectors
;-------------------------------------------------------------------
            ORG     WDT_VECTOR              ; Watchdog Timer
            DW      WDT_ISR
            ORG     RESET_VECTOR            ; POR, ext. Reset
            DW      RESET
            END
```

B) C Code:

```
#include "msp430x26x.h"

const char String1[13] = "\nHello World";

void main(void)
{
  WDTCTL = WDT_ADLY_1000;             // WDT 1000ms, ACLK, interval timer
  IE1 |= WDTIE;                       // Enable WDT interrupt
  P3SEL |= BIT6 + BIT7;               // P3.6,7 = USART1 TXD/RXD
  //Configure USCIA1, UART
  UCA1CTL1 = UCSSEL_1;                // ACLK
  /* baud rate = 9600 */
  UCA1BR0 = 03;                       // 32768Hz 9600 32k/9600=3.41
  UCA1BR1 = 0x0;
  UCA1MCTL = UCBRS_3;                 // Modulation UCBRSx = 3
```

```
  /* Initialize USCI state machine */
  UCA1CTL1 &= ~UCSWRST;

  // Configure DMA0
  DMACTL0 = DMA0TSEL_10;              // UTXIFG1 trigger
  DMA0SA = (int)String1;              // Source block address
  DMA0DA = (int)&UCA1TXBUF;           // Destination single address
  DMA0SZ = sizeof String1-1;          // Block size
  DMA0CTL = DMASRCINCR_3 + DMASBDB + DMALEVEL;
                                      // Repeat, inc src, byte transfer
  __bis_SR_register(LPM3_bits + GIE); // Enter LPM3 w/ interrupts
  _NOP();                             // Reqd  for debugger
}

#pragma vector = WDT_VECTOR           // Trigger DMA block transfer
__interrupt void WDT_ISR(void)
{
  DMA0CTL |= DMAEN;                   // Enable
}
```

7.7 Chapter Summary

Embedded peripherals give MCUs the versatility that allow for configuring them into single-chip computers. Essential to embedded peripherals is an interrupt system that allows for timely servicing devices needing CPU attention. This chapter focused in the support structures allowing MCUs to use interrupts, timers, flash, and DMA.

An interrupt is a signal or event that changes the normal course of execution a program to transfer control to a special code called an interrupt service routine or ISR. When the special instruction RETI (return from interrupt) is executed, program control is transferred back to the main program.

For interrupts to work in a system, four fundamental requirements must be met:

- There must be a properly allocated stack.
- The interrupt vectors must be configured.
- An ISR must be in place.
- Interrupts must be enabled at both, the GIE flag level and at the local peripheral level.

There are maskable, and non-maskable interrupts, identifying whether the interrupt can or cannot be disabled by the GIE flag.

The discussion of timers revealed that a timer is just a binary counter fitted with an input frequency divider (prescaler) and one or more output compare registers that facilitate either counting time or external events. Key applications of timers were discussed including watchdog timers, real-time clocks, and pulse-width modulation.

The section on memory provided an overview of memory technologies considering their ability to be modified and the permanence of its contents. Special attention was placed on FLASH memories, as this is the mainstream non-volatile technology in contemporary embedded controllers.

The last section of the chapter was devoted to bus arbitration and direct memory access controllers. This particular subject brought up the discussion of how to attain high transfer throughput and enabling further opportunities for low-power design in embedded systems.

7.8 Problems

7.1 Can you explain why it is necessary for the MSP430 to finish the instruction it is currently executing before attempting to service an interrupt?

7.2 Consider an embedded application of TV remote control. The microcontroller function in this application is to react to a key press, debounce the key, assign a numeric code to the depressed key, and wrap it according to the communication protocol defined between remote and TV, and transmit the encoded value via the infrared (IR) transmitter.
Assume the remote control user depresses, in average, one key every minute. This would allow for a reasonable key operation rate considering an average usage cycle where the user is either flipping channels, watching a show with sporadic volume adjustments, or not using the control at all (TV off periods). In all of them the TV remote is on and, with the processor running at 1 MHz, it takes 300 μs to process each keystroke. Let's also assume the MCU consumes 1 mA in active mode and 100 μA in sleep mode, and runs on batteries with a capacity of 1,200 mAh. Assume all active chores are completed with the MCU active power budget, except the IR transmitter, that takes 50 of the 300 μs to send each key code and consumes 20 mA.

(a) Devise an algorithm to operate the remote control via polling. Provide a flowchart for a modular plan depicting how to structure a main program and all necessary functions to operate the remote control.
(b) For the data provided, estimate how long would the battery last under the polling scheme devised in part (a).
(c) Repeat part (a) but instead of using polling, provide an interrupt-based approach, where the main program initializes all required peripherals, including interrupts, and then sends the CPU into sleep mode.
(d) Repeat part (b), now based on the system behavior induced by the software plan in part (c).
(e) Estimate the improvement in energy usage induced by the low-power algorithm devised in part (c) with respect to the polling approach used in part (a).

7.3 List three advantages of using the low power modes in the MSP430. Briefly justify each.

7.4 Is there any advantage in having a flag to deactivate the (Non) maskable interrupts in the MSP430? Would you use them in a critical application or a real-time application? Justify your answers.

7.5 Does it make any difference recognizing an interrupt during a rising or falling edge of the trigger difference? Briefly explain your answer.

7.6 Show the sequence of instructions in assembly language to

(a) Enable Timer_A Capture Compare Register 1 interrupt.
(b) Load Capture Compare Register 1 with 32768.
(c) Select the SMCLK clock.
(d) Configure the timer for continuous mode of operation.

7.7 An Engineering student needs to generate a square waveform to drive a device. The student decides to use the MSP430's Timer_A with a duty cycle of 60 % at 32.768 khz. Show the sequence of instructions in assembly language that the student might have used to configure Timer_A to accomplish the requirements.

7.8 What is the race condition that could develop if the capture input signal is asynchronous to the clock signal in an MSP430 capture/compare unit?

7.9 Setup the MSP430 LaunchPad to use Timer_A to toggle the red LED in Port 1 each time the timer counts to 50,000. You should be able to do this using:

(a) A monolithic assembly language program.
(b) An assembly language program that calls a subroutine to configure Timer_A to toggle the LED every 50,000 cycles.
(c) An assembly language program that sets up an interrupt from Timer_A every 50,000 cycles.
(d) Repeat the previous parts (a) through (c) using C language instead of assembly.

7.10 A periodic signal with a period equals to 10 times the MCLK clock in the MSP430 is needed. A designer wants to use one of the compare units in Timer_A to accomplish this. What are the values that need to be written in the Timer_A Control register TACTL and in the Timer_A Capture Compare register (TACCRx) in order to accomplish this? Show the corresponding instructions in both assembly language and C.

7.11 It was stated in this chapter that the watchdog timer could be configured as an interval counter. Configure the MSP430 Watchdog Timer to toggle the red LED in the MSP430 LaunchPad every three (3) seconds. To conserve energy the MSP430 needs to enter a low power mode after your configuration is complete.

7.12 An embedded application requires a signal with a duty cycle of 30 % to drive four LEDs. It is desired to obtain such signal using an MSP430 device. How would you configure the output unit to obtain the necessary signal assuming everything else is configured correctly? What changes would you have to make to the output unit if a different clock source is chosen?

7.13 Write a small program to test the WDT+ failsafe feature of the MSP430. Have the MSP430 perform some function like toggling one of the LaunchPad LEDs and then attempt to disable all clocks.

7.14 The frequency at which an event is occurring needs to be determined. Show how to configure the MSP430 watchdog timer, with the SMCLK clock source,

as an interval timer in order to accomplish this task. What formula would you use to determine the frequency of the even?

7.15 On page 673 (need a marker on the page) a piece of assembly code for writing a block of data to the flash memory of the MSP430x4xx device was shown. Write the corresponding code in the C language. As explained in the comments, the code shown must be executed from RAM, what is the reason for this? Could we have executed the program from Flash rather than from RAM?

7.16 Explain the need for wear leveling. What would be the effect of not using it?

7.17 It was explained hat the IE1 register houses the ACCVIE bit used to enable the Flash memory access violation interrupts. How could this bit be used while the system is executing programs from Flash?

7.18 Answer the following questions:

(a) Explain what is a "bus arbitration" transaction.
(b) What methods can be used for dealing with multiple simultaneous bus requests in a bus arbitration transaction.
(c) Explain the concept of "Direct Memory Access"
(d) Describe the three basic modes for supporting direct memory access transfers.
(e) Identify two limitations for implementing one-cycle DMA transfers in an MSP430.

7.19 Describe the difference between one-cycle and two-cycle DMA transfers. Emphasize the weakness and strength of each method with respect to the other.

7.20 Describe important registers to be found in almost any DMA controller. Why are these registers so important?

7.21 What are the basic setup steps that should be considered when programming DMA transfers. Provide pieces of code for an MSP430 device where those steps are illustrated.

7.22 How do you configure a DMA in the MSP430 family for a repeated burst-block transfer of 4 KB from Loc_A to Loc_B?

7.23 A set of three devices DEV0, DEV1 and DEV2 are to be connected in a daisy chain configuration with priority scheme DEV0-DEV1-DEV2. Each device can generate a bus-request signal, and starts transmission upon a bus-acknowledge signal received. Once a device has been granted bus control, it generates a busy signal that prevents the other devices to request bus control. The device deactivates the busy signal after transfer is completed. Design a daisy chain logic system for these devices.