**«AZƏRBAYCAN HAVA YOLLARI»**

**CLOSED JOINT-STOCK COMPANY**

**NATİONAL AVİATİON ACADEMY**

**Faculty:** Aerokosmic

**Speciality: 050631-** Computer engineering

**Subject:** System programing and operating systems

**Course Work**

**Topic:** Interrupt processing in assembly language

**Group: 1459**-**i**

**Teacher:**  **Sahmaliyev Mehemmed**

**Student: Aliyeva Ilaha**

**Baku 2022**

# **Contents**

**İntroduction.............................................................................................................3**

1. **TYPES OF INTERRUPT...................................................................................5**
   1. Hardware Interrupts.............................................................................................5
   2. Software Interrupts .............................................................................................7
   3. Interrupt Handling .............................................................................................9
   4. Interrupt Latency ..............................................................................................10
2. **TRIGGERING METHODS ........................................................................... 13**
   1. Level-triggered ................................................................................................14
   2. Edge-triggered .................................................................................................15
3. **FEATURES OF INTERRUPT SERVICE ROUTINES ...............................16**
4. **THE DIFFERENCE BETWEEN SIGNAL() & INTERRUPT( ).................**18
5. **INTERRUPT DISPATCHERS.......................................................................**19
6. **INTERRUPT OVERHEAD**.............................................................................24
7. **INTERRUPT RESTRICTIONS .....................................................................25**

**CONCLUSİON......................................................................................................27**

**LIBRARY...............................................................................................................28**

# **Introduction**

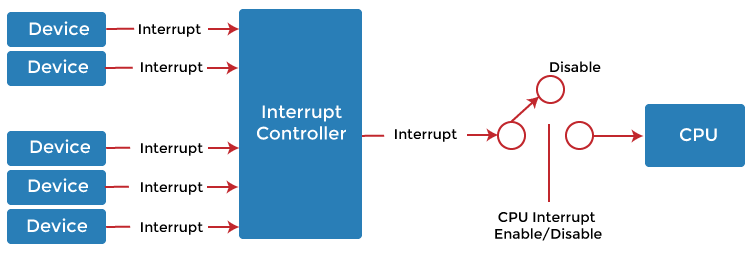
# An embedded system uses its input/output devices to interact with the external world. Input devices allow the computer to gather information, and output devices can display information. Output devices also allow the computer to manipulate its environment. The tight-coupling between the computer and external world distinguishes an embedded system from a regular computer system. The challenge is under most situations the software executes much faster than the hardware. E.g., it might take the software only 1 us to ask the hardware to clear the LCD, but the hardware might take 1 ms to complete the command. During this time, the software could execute tens of thousands of instructions. Therefore, the synchronization between the executing software and its external environment is critical for the success of an embedded system. This chapter presents general concepts about interrupts.. Using interrupts allows the software to  respond quickly to changes in the external environment.

# **What is Interrupt**

An interrupt is a signal emitted by hardware or software when a process or an event needs immediate attention. It alerts the processor to a high-priority process requiring interruption of the current working process. In I/O devices, one of the bus control lines is dedicated for this purpose and is called the **Interrupt Service Routine** (ISR).

When a device raises an interrupt at the process, the processor first completes the execution of an instruction. Then it loads the **Program Counter** (PC) with the address of the first instruction of the ISR. Before loading the program counter with the address, the address of the interrupted instruction is moved to a temporary location. Therefore, after handling the interrupt, the processor can continue with the process.

While the processor is handling the interrupts, it must inform the device that its request has been recognized to stop sending the interrupt request signal. Also, saving the registers so that the interrupted process can be restored in the future increases the delay between the time an interrupt is received and the start of the execution of the ISR. This is called **Interrupt Latency**.

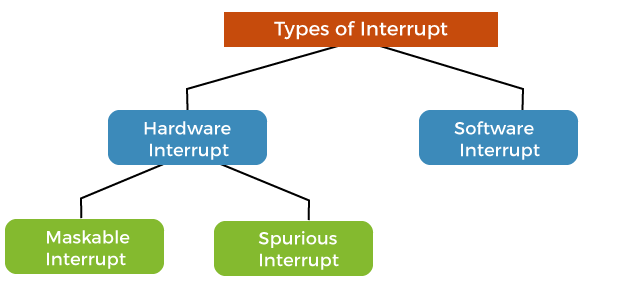


A single computer can perform only one computer instruction at a ti8me. But, because it can be interrupted, it can manage how programs or sets of instructions will be performed. This is known as **multitasking**. It allows the user to do many different things simultaneously, and the computer turns to manage the programs that the user starts. Of course, the computer operates at speeds that make it seem like all user tasks are being performed simultaneously.

An operating system usually has some code that is called an **interrupt handler**. The interrupt handler prioritizes the interrupts and saves them in a queue if more than one is waiting to be handled. The operating system has another little program called a **scheduler** that figures out which program to control next.

### **1. TYPES OF INTERRUPT**

Interrupt signals may be issued in response to hardware or software events. These are classified as **hardware interrupts** or **software interrupts**, respectively.



### **1.1 Hardware Interrupts**

A hardware interrupt is a condition related to the state of the hardware that may be signaled by an external hardware device, e.g., an interrupt request (IRQ) line on a PC, or detected by devices embedded in processor logic to communicate that the device needs attention from the operating system. For example, pressing a keyboard key or moving a mouse triggers hardware interrupts that cause the processor to read the keystroke or mouse position.

Hardware interrupts can arrive asynchronously for the processor clock and at any time during instruction execution. Consequently, all hardware interrupt signals are conditioned by synchronizing them to the processor clock and act only at instruction execution boundaries.

In many systems, each device is associated with a particular IRQ signal. This makes it possible to quickly determine which hardware device is requesting service and expedite servicing of that device.

On some older systems, all interrupts went to the same location, and the OS used specialized instruction to determine the highest priority unmasked interrupt outstanding. On contemporary systems, there is generally a distinct interrupt routine for each type of interrupt or each interrupts source, often implemented as one or more interrupt vector tables. Hardware interrupts are further classified into two types, such as:

* **Maskable Interrupts:**Processors typically have an internal interrupt mask register which allows selective enabling and disabling of hardware interrupts. Each interrupt signal is associated with a bit in the mask register; on some systems, the interrupt is enabled when the bit is set and disabled when the bit is clear, while on others, a set bit disables the interrupt. When the interrupt is disabled, the associated interrupt signal will be ignored by the processor. Signals which are affected by the mask are called **maskable interrupts**.  
  The interrupt mask does not affect some interrupt signals and therefore cannot be disabled; these are called **non-maskable interrupts** (NMI). NMIs indicate high priority events that need to be processed immediately and which cannot be ignored under any circumstances, such as the timeout signal from a watchdog timer.  
  To **mask** an interrupt is to disable it, while to **unmask** an interrupt is to enable it.
* **Spurious-Interrupts:**  
  A spurious interrupt is a hardware interrupt for which no source can be found. The term phantom interrupt or ghost interrupt may also use to describe this phenomenon. Spurious interrupts tend to be a problem with a wired-OR interrupt circuit attached to a level-sensitive processor input. Such interrupts may be difficult to identify when a system misbehaves.  
  In a wired-OR circuit, parasitic capacitance charging/discharging through the interrupt line's bias resistor will cause a small delay before the processor recognizes that the interrupt source has been cleared. If the interrupting device is cleared too late in the interrupt service routine (ISR), there won't be enough time for the interrupt circuit to return to the quiescent state before the current instance of the ISR terminates. The result is the processor will think another interrupt is pending since the voltage at its interrupt request input will be not high or low enough to establish an unambiguous internal logic 1 or logic 0. The apparent interrupt will have no identifiable source, and hence this is called spurious.  
  A spurious interrupt may result in system deadlock or other undefined operation if the ISR doesn't account for the possibility of such an interrupt occurring. As spurious interrupts are mostly a problem with wired-OR interrupt circuits, good programming practice in such systems is for the ISR to check all interrupt sources for activity and take no action if none of the sources is interrupting.

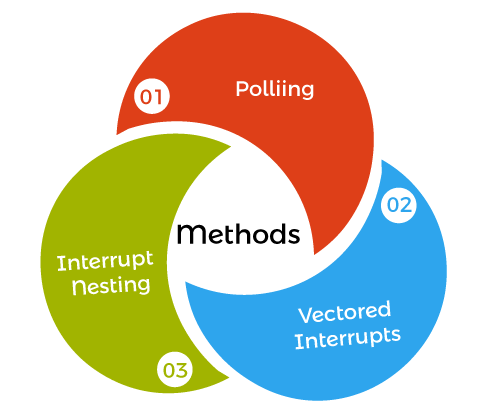
**1.2 Software Interrupts**

The processor requests a software interrupt upon executing particular instructions or when certain conditions are met. Every software interrupt signal is associated with a particular interrupt handler.

A software interrupt may be intentionally caused by executing a special instruction that invokes an interrupt when executed by design. Such instructions function similarly to subroutine calls and are used for various purposes, such as requesting operating system services and interacting with device drivers.

Software interrupts may also be unexpectedly triggered by program execution errors. These interrupts are typically called **traps** or **exceptions**.

### **Handling Multiple Devices**

When more than one device raises an interrupt request signal, additional information is needed to decide which device to consider first. The following methods are used to decide which device to select first,

1. **Polling**  
   In polling, the first device encountered with the IRQ bit set is to be serviced first, and appropriate ISR is called to service the same. It is easy to implement, but a lot of time is wasted by interrogating the IRQ bit of all devices.
3. **Vectored-Interrupts**  
   In vectored interrupts, a device requesting an interrupt identifies itself directly by sending a special code to the processor over the bus. This enables the processor to identify the device that generated the interrupt. The special code can be the starting address of the ISR or where the ISR is located in memory and is called the **interrupt vector**.
4. **Interrupt-Nesting**   
   In this method, the I/O device is organized in a priority structure. Therefore, an interrupt request from a higher priority device is recognized, whereas a lower priority device is not. The processor accepts interrupts only from devices/processes having priority more than it.  
   Processors priority is encoded in a few bits of PS (Process Status register), and it can be changed by program instructions that write into the PS. The processor is in supervised mode only while executing OS routines, and it switches to user mode before executing application programs.

### **1.3 Interrupt Handling**

We know that the instruction cycle consists of fetch, decode, execute and read/write functions. After every instruction cycle, the processor will check for interrupts to be processed. If there is no interrupt in the system, it will go for the next instruction cycle, given by the instruction register. If there is an interrupt present, then it will trigger the interrupt handler. The handler will stop the present instruction that is processing and save its configuration in a register and load the program counter of the interrupt from a location given by the interrupt vector table.

After processing the interrupt by the processor, the interrupt handler will load the instruction and its configuration from the saved register. The process will start its processing where it's left. This saves the old instruction processing configuration, and loading the new interrupt configuration is also called **context switching**. There are different types of interrupt handlers.

1. **First Level Interrupt Handler** (FLIH) is a hard interrupt handler or fast interrupt handler. These interrupt handlers have more jitter while process execution, and they are mainly maskable interrupts.
2. **Second Level Interrupt Handler** (SLIH) is a soft interrupt handler and slow interrupt handler. These interrupt handlers have less jitter.

The interrupt handler is also called an interrupt service routine (ISR). The main features of the ISR are

* Interrupts can occur at any time, and they are asynchronous, and ISR's can call for asynchronous interrupts.
* An interrupt service mechanism can call the ISR's from multiple sources.
* ISR's can handle both maskable and non-maskable interrupts. An instruction in a program can disable or enable an interrupt handler call.
* ISR at the beginning of execution will disable other devices interrupt services. After completion of the ISR execution, it will reinitialize the interrupt services.
* The nested interrupts are allowed in ISR for diversion to other ISR.

### **1.4 Interrupt Latency**

When an interrupt occurs, the service of the interrupt by executing the ISR may not start immediately by context switching. The time interval between the occurrence of interrupt and the start of execution of the ISR is called interrupt latency.

* **Tswitch =** time taken for context switch
* **ΣTexec =** The sum of the time interval for executing the ISR
* **Interrupt Latency** = Tswitch + ΣTexec

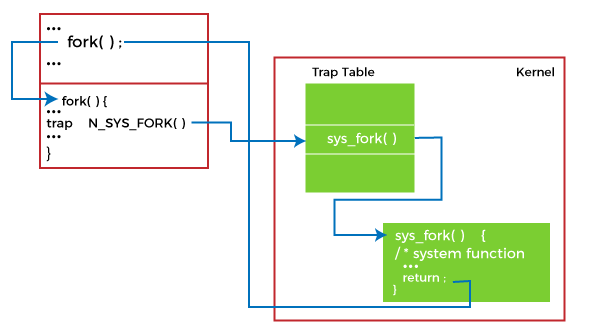
### **How CPU Response to Interrupts**

A key point towards understanding how operating systems work understands what the CPU does when an interrupt occurs. The CPU hardware does the same for each interrupt, allowing operating systems to take control away from the currently running user process. The switching of running processes to execute code from the OS kernel is called a **context switch**.

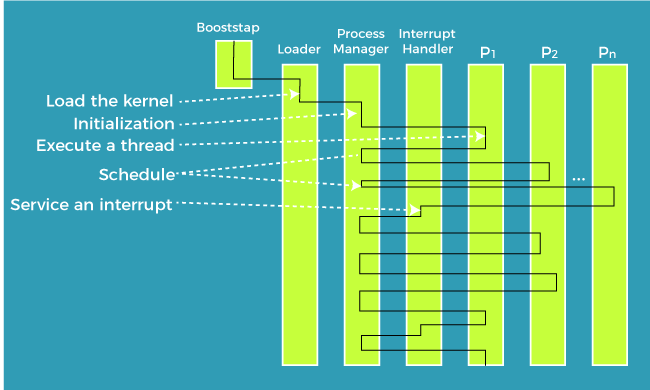
CPUs rely on the data contained in a couple of registers to handle interrupts correctly. One register holds a pointer to the process control block of the currently running process, and this register is set each time a process is loaded into memory. The other register holds a pointer to a table containing pointers to the instructions in the OS kernel for interrupt handlers and system calls. The value in this register and contents of the table are set when the operating

system is initialized at boot time. The CPU performs the following actions in response to an interrupt:

1. Using the pointer to the current process control block, the state and all register values are saved for use when the process is later restarted.
2. The CPU mode bit is switched to supervisory
3. Using the pointer to the interrupt handler table and the interrupt vector, the location of the kernel code to execute is determined. The interrupt vector is the IRQ for hardware interrupts and an argument to the interrupt assembly language instruction for software interrupts.
4. Processing is switched to the appropriate portion of the kernel.



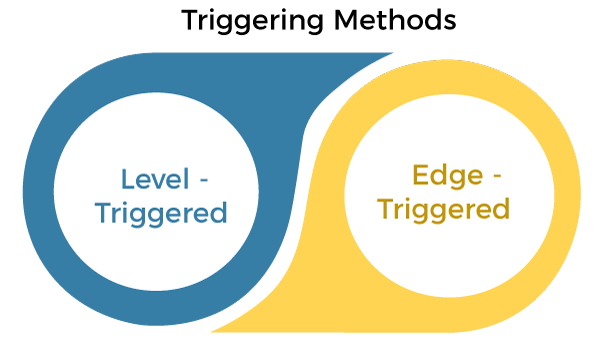
The CPU uses a table and the interrupt vector to find OS the code to execute in response to interrupts. A software interrupt is shown in the below image.



As the computer runs, processing switches between user processes and the operating system as hardware and software interrupts are received.

### **2.TRIGGERING METHODS**

Each interrupts signal input is designed to be triggered by either a logic signal level or a particular signal edge (level transition). Level-sensitive inputs continuously request processor service so long as a particular (high or low) logic level is applied to the input. Edge-sensitive inputs react to signal edges: a particular (rising or falling) edge will cause a service request to be latched. The processor resets the latch when the interrupt handler executes.



### **2.1 Level-triggered**

A level-triggered interrupt is requested by holding the interrupt signal at its particular (high or low) active logic level. A device invokes a level-triggered interrupt by driving the signal to and holding it at the active level. It negates the signal when the processor commands it, typically after the device has been serviced.

The processor samples the interrupt input signal during each instruction cycle. The processor will recognize the interrupt request if the signal is asserted when sampling occurs.

Level-triggered inputs allow multiple devices to share a common interrupt signal via wired-OR connections. The processor polls to determine which devices are requesting service. After servicing a device, the processor may again poll and, if necessary, service other devices before exiting the ISR.

### **2.2. Edge-triggered**

An edge-triggered interrupt is an interrupt signaled by a level transition on the interrupt line, either a falling edge (high to low) or a rising edge (low to high). A device wishing to signal an interrupt drives a pulse onto the line and releases it to its inactive state. If the pulse is too short to be detected by polled I/O, then special hardware may be required to detect it.

# **ANSI C INTERRUPTS**

Besides the hardware interrupts available with the ADSP-21xxx processors, there are 6 interrupts that are required by the ANSI C standard. They are:

|  |  |
| --- | --- |
| SIGABRT | Abort signal |
| SIGFPE | Floating point exception |
| SIGILL | Illegal instruction |
| SIGINT | System interrupt |
| SIGSEGV | Segmentation violation |
| SIGTERM | Software termination signal |

These signals were implemented to support the ANSI standard, they are not supported in hardware on the ADSP-21000 family of processors. The raise() function can invoke them, but they do not occur independently.

You can use these interrupts in an application. They follow the same rules as the hardware interrupts, but since they can only be invoked with a call to the raise() function, it is much more efficient to call the routines directly.

# **3.FEATURES OF INTERRUPT SERVICE ROUTINES**

An interrupt service routine (ISR) is a special routine that is executed outside of the normal program flow. An ISR is invoked in response to a particular interrupt occurring at an undetermined time.

Since an interrupt occurs at an unknown time, it cannot return a value directly to a program. Thus, all ISRs are of return-type void. The void type indicates that no value is returned from the routine. According to the C standard, all ISRs are called with a number indicating the interrupt that invoked them as their parameter. This tells the routine what caused its invocation. The prototype of an interrupt service routine is always:

void handler1(int sig);

This prototype indicates that the function handler1 accepts an integer parameter (the signal name), and has no return value. When you create your ISR, be sure that it has this prototype. If the prototype is incorrect, the compiler warns you with a message similar to this one:

foo.c: In function 'main':

foo.c:10: warning: passing arg 2 of '\_signal060' from

incompatible pointer type

This warning indicates that the handler routine does not have the correct prototype. Although your program might work even with this warning, you should determine and correct the cause.

One way for an ISR to return information to the main program is through the use of global variables. These variables are accessible to both the handler and the main program. When an ISR needs to pass information to the main program, it sets a global variable to that value. When the main program has time to deal with the information, it reads the global variable.

Declare these global variables as volatile . The volatile qualifier tells the compiler that this variable can change in ways and at times unexpected by normal program flow.

The indication that a variable is volatile causes the compiler to treat it specially. Since a volatile variable can change outside of the scope of normal program flow, it is not optimized away, even if it does not appear to be set anywhere.

For example, if your program has a loop that tests the state of a variable that is never modified in the loop, the compiler thinks that it can remove the test from the loop (since the value cannot change within the loop).

void do\_something(int param1);

int global\_variable;

main()

{

global\_variable = 0;

while(1) /\* Stay in this loop \*/

{

if( global\_variable )

do\_something(global\_variable);

}

}

In the section of code shown, the compiler can see that global\_variable is set to zero outside of the loop and not modified within the loop. It assumes that it can execute the test once outside of the loop, and not need to perform it at every iteration.

In general, this is a safe assumption, and can result in a significant performance improvement for your code. If, on the other hand, global\_variable was modified by an interrupt service routine outside of the scope of normal program flow, this assumption is incorrect.

The volatile qualifier tells the compiler that it cannot make any assumptions about the variable, and that it must perform the test every iteration of the loop.

When you write an interrupt service routine, try to spend as little time in the ISR as possible. When an interrupt is serviced, all lower priority interrupts are not serviced until the higher priority interrupt exits.

To reiterate, consider interrupts as exceptional situations. The less time spent in an interrupt service routine, the better.

# **4. THE DIFFERENCE BETWEEN SIGNAL() & INTERRUPT()**

There are two ways to setup a routine as a handler. The first way, which is ANSI compliant, is to use the signal() routine. The signal() routine (as defined by the ANSI standard) setups up an ISR to respond to **one** invocation of the specified interrupt. After the ISR has responded once, the interrupt will be ignored in the future.

Using the signal() function, the only way to respond to an interrupt more than once is to reinitialize the ISR with the signal() function every time the ISR is executed.

Since most systems designed with the ADSP-21000 family of processors need to respond to some interrupts continuously, Analog Devices created an extension to the standard that sets up an ISR to respond continuously to an interrupt. The extension is the interrupt() routines. These routines takes the same parameters as the signal() routine; the only difference is that an ISR that has been setup with interrupt() responds continuously to an interrupt, while one that is setup with signal() only responds once.

# **5.INTERRUPT DISPATCHERS**

The ADSP-21000 Family Runtime C Library provides interrupt dispatcher routines.

The signal.h header file provides function prototypes for the standard ANSI signal.h routines and also for several ADSP-21xxx family extensions such as interrupt() and clear\_interrupt(). It also includes ANSI-standard signal handling functions of the C library.

The signal handling functions process conditions (hardware signals) that can occur during program execution. They determine the way that your C program responds to these signals. The functions are designed to process such signals as external interrupts and timer interrupts.

There are three interrupt dispatchers, *normal*, *fast*, and *super*. The interrupt dispatchers let you disable interrupts and modify the processor’s MODE1 register from an interrupt handler. The *normal interrupt dispatcher* and *fast interrupt dispatcher* preserve MODE1 register writes in an interrupt handler after the interrupt is serviced, but the *super interrupt dispatcher* does not. The interrupt dispatchers are described below.

**Normal Interrupt Dispatcher**—Saves all scratch registers and the loop stack. Do loop and interrupt nesting is allowed because data is pushed onto the stack. Requires approximately 125 cycles for interrupt

overhead. To access this dispatcher, use interrupt() or signal()

.

**Fast Interrupt Dispatcher**—Does not save the loop stack, therefore DO loop handling is restricted to six levels (specified in hardware). If the interrupt service routine (ISR) uses one level of nesting, your code cannot exceed five levels. Interrupt nesting is not restricted (20 levels available). This dispatcher does not send the interrupt number type to the ISR as a parameter. Requires approximately 60 cycles for interrupt overhead. To access this dispatcher, use interruptf() or signalf().

**Super Interrupt Dispatcher**—Does not save the loop stack, therefore do loop handling is restricted to six levels (specified in hardware). Interrupt nesting is disabled. This dispatcher does not send the interrupt number type to the ISR as a parameter. This dispatcher uses the secondary register set. This dispatcher only works with the ADSP-21020 (Rev. 1) and with all ADSP-2106x SHARC processors. Requires approximately 30 cycles for interrupt overhead. To access this dispatcher, use interrupts() or signals() .

# **THE IDLE() FUNCTION**

If your main program waits for an interrupt to occur before doing something, you can use the idle() function supplied in the C Runtime Library. The idle() routine executes the IDLE function of the ADSP-21xxx processor. The IDLE instruction causes the ADSP21xxx to wait at the IDLE instruction until an interrupt occurs. When an interrupt occurs, the ISR is executed, and normal program flow continues after the IDLE instruction.

# **THE CLEAR\_INTERRUPT() FUNCTION**

This function is used to clear a pending interrupt. When an interrupt occurs, the processor latches that interrupt in the IRPTL register. The interrupt remains latched until it is serviced or cleared. If your application is about to setup an ISR for an interrupt, a meaningless occurrence of that interrupt could be pending. If you do not wish to service it, clear the interrupt before calling signal() or interrupt().

The clear\_interrupt() function takes the signal name as its parameter.

# **THE RAISE() FUNCTION**

This function is used to cause an interrupt to be invoked manually. It sets the appropriate bit in the IRPTL register. If the interrupt is not masked, the processor vectors to the ISR (through the dispatcher).

# **A SIMPLE EXAMPLE**

This section provides a very simple example of how to set up an interrupt service routine and how to interface with your main program. It uses the techniques discussed in the preceding sections.

#include <signal.h>

/\* These are function that do useful work. \*/

extern void do\_timer\_things(void);

extern void do\_irq0\_things(void);

/\* Be sure to declare any variables reference by an ISR as volatile \*/

volatile int timer\_expired;

volatile int irq0\_occurred;

/\* Be sure to have the correct prototype for an ISR \*/

void timer\_handler(int signal)

{

timer\_expired = 1;

}

void irq0\_handler(int signal)

{

irq0\_occurred = 1;

}

main()

{

/\* Set timer expired to zero at the beginning \*/

timer\_expired = 0;

irq0\_occurred = 0;

/\* Set up the routines to respond to interrupts \*/

interrupt(SIG\_IRQ0, irq0\_handler);

interrupt(SIG\_TMZ, timer\_handler);

/\* Set up the timer \*/

timer\_set((unsigned int)10000, (unsigned int)10000);

timer\_on();

/\* Loop continuously and respond to interrupts \*/

while(1)

{

idle(); /\* Return from this function after an interrupt \*/

/\* If the timer has expired, clear flag and do something\*/

if( timer\_expired )

{

timer\_expired = 0;

do\_timer\_things();

}

/\* If irq0 has occurred, clear flag and do something\*/

If ( irq0\_occurred )

{

irq0\_occurred = 0;

do\_irq0\_things();

}

}

}

You can use this example as a shell for your programs that use interrupts. There are two interrupts that we use this example. The first is irq0 , which, in this example, is connected to a push button. When the button is depressed, an irq0 interrupts occurs. The global variable irq0\_occurred informs the main program that this has occurred.

The second interrupt we use is the timer. In the main routine we initialize the timer to trigger an interrupt every 10,000 cycles. This means that the timer\_handler routine is executed every 10,000 cycles.

# **6. INTERRUPT OVERHEAD**

This section discuss interrupt overhead and processing. It assumes that you have a knowledge of the internals of the ADSP-21xxx processor. If you encounter terms that you are unfamiliar with, refer to the *ADSP-21020 User’s Manual* or *ADSP-2106x SHARC User’s Manual*.

When an interrupt (that is not masked) occurs on the ADSP-21xxx, the processor vectors to a specific address based on the interrupt that occurred. The runtime header ( 060\_hdr.asm or 020\_hdr.asm ) contains the vectors for each interrupt.

The code in the vector location executes a JUMP to the interrupt dispatcher that is part of the runtime library. The cache update is disabled and the global interrupt enable bit is turned off.

The cache is frozen because otherwise the ISR fills the cache. This is inefficient, since most ISRs are short and do not contain loops. By disabling the cache, it retains its contents. So, it still contains useful information when it returns to the code that was interrupted.

The global interrupt enable bit is shut off temporarily. This is done so that the dispatcher is able to maintain the C runtime stack. There are approximately 9 cycles where interrupts are completely disabled. As soon as the dispatcher performs stack management, interrupts are reenabled.

The interrupt dispatcher stores all scratch registers on the C runtime stack before calling the ISR. The dispatcher needs to save the scratch registers because C routines expect that they can overwrite all scratch registers without saving them. Since the interrupt occurred at an unknown point in program execution, it is likely that these registers have meaningful values in them.

The regular dispatcher then sets up what will happen on the next interrupt, determines the current interrupt, and calls the ISR. This process requires about 100 cycles.

After the ISR returns, the dispatcher restores the saved registers and executes an RTI to return to the interrupted code. This requires about 50 cycles to complete.

In addition to the processing overhead, the regular dispatcher requires a maximum 30 locations on the C runtime stack to store registers.

The interrupt dispatcher may be modified in future releases. Refer to your release note to see if the timing is altered.

**7.INTERRUPT RESTRICTIONS**

There are a few extraordinary situations that might cause problems when using interrupts. When an interrupt occurs and the chip transfers control to the dispatcher, the return address is pushed on the PC stack. The PC stack is a hardware stack on the processor, which is limited to 20 locations.

The C runtime model does not use the PC stack very much, so in most cases it is possible to nest interrupts 20 deep. It is unusual for an application to require this many simultaneous interrupts, but it is possible that the PC stack could overflow in such a case. If your application only uses a few interrupts it is extremely unlikely that PC stack limitations will ever be encountered.

The interrupt dispatcher is designed to support the C runtime model. If your program has assembly routines that change the mode of the processor, it is possible that this could cause trouble with interrupts. If for example, you code enables the bit reverse mode of the chip, and an interrupt occurs while bit reverse is on, and the ISR uses DAG0, this could cause problems. (The dispatcher does not use DAG0, so you could disable bit reversal in the ISR and re-enable it before exiting).

**CONCLUSION**

In conclusion , we are going to talk about disgusting many types of the interrupt and some general interrupt handling practices to help us in structuring ISRs in the right way in this page . As we have mentioned, this is a very broad topic, and it is very important that all functions work accurately.

A solution has been proposed for handling interrupts this proposed solution will handle interrupts like as bidding and helping pattern and solved the problems that occur in real time task dispersed by interrupts and it helps to increase the accuracy and quality of the system. Priority of interrupt matters because if the interrupt has high priority then interrupted process then the interrupt will be responded soon and interrupted process will be paused. It is not important to wait for the execution of interrupt service’s scheduling. Therefore, this answering phenomenon is the new solution of this problem which is created by Interrupts and it should be noted that this solution is widely used nowadays.

LIBRARY

1. [*"Assembly: Review"*](http://web.cse.ohio-state.edu/~sivilotti.1/teaching/3903.recent/lectures/lecture14.pdf) *(PDF)*. Computer Science and Engineering. College of Engineering, [*Ohio State University*](https://en.wikipedia.org/wiki/Ohio_State_University). 2016. [*Archived*](https://web.archive.org/web/20200324153925/http:/web.cse.ohio-state.edu/~sivilotti.1/teaching/3903.recent/lectures/lecture14.pdf) *(PDF)* from the original on 2020-03-24*. Retrieved 2020-03-24*.
2. Archer, Benjamin (November 2016). [*Assembly Language For Students*](https://dl.acm.org/citation.cfm?id=3125846). North Charleston, South Carolina, USA: [*CreateSpace Independent Publishing*](https://en.wikipedia.org/wiki/CreateSpace_Independent_Publishing). [*ISBN*](https://en.wikipedia.org/wiki/ISBN_(identifier)) [*978-1-5403-7071-6*](https://en.wikipedia.org/wiki/Special:BookSources/978-1-5403-7071-6).
3. Ferrari, Adam; Batson, Alan; Lack, Mike; Jones, Anita (2018-11-19) [Spring 2006]. Evans, David (ed.). [*"x86 Assembly Guide"*](http://www.cs.virginia.edu/~evans/cs216/guides/x86.html). Computer Science CS216:
4. [*"The SPARC Architecture Manual, Version 8"*](https://web.archive.org/web/20111210201410/http:/www.sparc.com/standards/V8.pdf) *(PDF)*. [*SPARC International*](https://en.wikipedia.org/wiki/SPARC_International). 1992. Archived from [*the original*](http://www.sparc.com/standards/V8.pdf) *(PDF)* on 2011-12-10*. Retrieved 2011-12-10*.
5. [*Brooks, Frederick P.*](https://en.wikipedia.org/wiki/Fred_Brooks) (1986). "No Silver Bullet—Essence and Accident in Software Engineering". Proceedings of the IFIP Tenth World Computing Conference: 1069–1076.