Difference between ISR and Function Call?

I want to understand difference between ISR (Interrupt Service Routine) and Function call.

I feel both the function call and ISR are the same from the hardware perspective. Please Correct me if I am wrong. All I could found about ISR and Function call is as follows:

ISR:

- Asynchronous event that can occur any time during the execution of the program
- Saves the PC, Flags and registers on the stack and disables all the interrupts and loads the address of the ISR
- ISR cannot have arguments that can be passed to it
- Cannot return values
- Enables the interrupts
- Generally small as they are taking the time of some other process
- Some of ISR have have their own stack

Function:

- Occurs when ever there is a function call
- Saves the PC and registers on the stack
- Can have arguments
- Can return values
- No restriction on the size and duration of execution

Is there any more difference other than this ? Please let me know. I have also read about having a function call from ISR how does that take place. Please highlight on it.

operating-system embedded computer-architecture function-calls isr

edited Jul 21 '13 at 0:48

contradictioned
1,006 1 11 23

asked Jul 21 '13 at 0:35

Nilesh Agrawal
1,250 7 14 47

5 Answers

They are not necessarily the same as you statet in the first point on ISRs: Interrupts are asynchronous and therefore have to somehow 'interrupt' the work of the main processor(s).

For example, lets look at this MIPS code decorated with addresses, which does not make anything useful:

```
4000.       add $1, $2, $3
4004.       sw $ra, 0($sp)
4008.       jal subr   # function call, sets $ra to 4012 and jumps to 4024
4012.       lw $ra, 0($sp)
4016.       jr $ra
4020.
```

```
4024. subr: sub $2, $1, $3
4028.       jr $ra
```

This code can be handeled from the main processor: the arithmetic operations (lines 1, 7) are done by the arithmetic unit, memory access (lines 2, 4) by the memory controller, and the jumps (lines 3, 5, 8) are done by the main cpu as well. (The actual address of `jal` is set during binding of the object file.)

This is for function calls. At any time it is determined, where the code is right now and which code is executed at the next point in time (i.e. when the programm counter is incremented: PC+=4).

Now there comes the point, when your functions do something complicated but you still want the software to react on a key stroke. Then a so called coprocessor comes into play. This coprocessor waits until some event (like a key stroke on your keyboard) occurs, and then calls the interrupt handler. This is a block of code located on a certain address in the memory.

Think, the processor is in the computation above, but in the meantime you want to store the number of key strokes on address `keys`. Then you write a programm starting at address `0x80000180` (this is defined as the exeption handler address in MIPS):

```
lw $at, keys
addi $at, $at, 1
sw $at, keys
eret
```

Now what happens at a keystroke?

1. The coprocessor gets aware of the keystroke
2. The current PC of the main procesor is saved
3. The PC of the main processor is set to 0x80000180, the interrupt code is executed
4. On `eret` the PC is set to the PC of the main processor before the interrupt occured
5. The execution of the main progamm continues there.

Here there is a switch from normal execution to interrupt handling between steps 2 and 3 and back again from 4 to 5.

Note: I have simplified this a lot, but it should be clear, how interrupts are different from function calls, and how the hardware has to have additionall capabilities for interrupt handling.

answered Jul 21 '13 at 1:24

contradictioned
1,006 1 11 23

Unfortunately I can't offer online resources on that, since this relies on a written script for computer systems ;) – contradictioned Jul 21 '13 at 1:24

So does it means that for microprocessor like 8051 or on microcontroller 8091 which doesnot have co-processor will have interrupts and function calls as same ? Please help me, I am really confused – Nilesh Agrawal Jul 21 '13 at 2:12

1 I'm no expert on microcontrollers, but on this site is a block diagram of the 8051: aninditadhikary.wordpress.com/tag/intel-8051 where you can see the 'Interrupt Control', which is placed next to the cpu, similar to the mips coprocessor. – contradictioned Jul 21 '13 at 10:19

And here is an tutorial for 8051 which explains interrupts: 8052.com/tutint.phtml . Essence: The CPU checks after every line of "normal" code, wether there is an exception and if so, it jumps to the exception handler. – contradictioned Jul 21 '13 at 10:23

The main difference is that interrupt handlers are, (usually), invoked by peripheral hardware - an actual hardware signal is generated by the peripheral and hardware in the processor transfers control to the appropriate handler without any action by the code that was running before the interrupt. Unlike functions, there is no call - execution is ripped away from the interrupted code by the processor hardware.

On OS that support multithreading/processes, function calls take place witin the same process/thread context as the caller. An interrupt, OTOH, has no thread or process context - a network interrupt resulting from a background BitTorrent download may occur while you are editing a Word document, and so the handler is very restricted in what it can do. It can load data to/from pre-allocated buffers belonging to the process/thread that it is bound to, it can signal a semaphore, it may be able to set OS event flags. That's about it.

Often, an interrupt-handler performs an interrupt-return directly, so allowing execution of the interrupted code to proceed wtihout any further interference. On simpler controllers, like yopur 8051, that often run embedded code without a compex OS, this is the only course available. With a preemptive multithreaded OS, an interrupt-handler has the additional option of performing its interrupt-return via OS code and so causing a scheduler run. This allows interrupt-handlers to make threads that were waiting for the interrupt ready, and possibly running, (and so maybe preempting the thread that was originally interrupted). This allows such systems to have good I/O performance without any polling.

The hardware interrupt sources my be peripherals embedded in the processor chip - network controllers, disk controllers, display controllers, DMA controllers, USB controllers, intercore-comms controllers, (on processors with multiple cores), timers etc. or interrupt-request pin/s on the package can be used to generate an interrupt from an external hardware source, (maybe a pushbutton, keyboard, keypad or touchscreen hardware).

answered Jul 21 '13 at 9:16

Martin James
21k 3 24 46

So having asserted that they are the same, you go on to list the ways in which they are different - which perhaps rather answers your question.

Your first four points about ISRs are broadly and generally true. The points about enabling interrupts is not necessarily the case and is an implementation decision by the programmer, and may be determined by the architecture, and being small is a guideline not a requirement - and "small" is entirely subjective".

The differences are not so much in respect of how they are coded (though ISR's typically impose a number of restrictions and may also have privileges that normal functions do not), but rather in how they are invoked and the behaviour of the processor.

A function (or procedure or sub-routine more generally) must be explicitly called and is part of the same context and thread of execution as its caller. A hardware ISR is not explicitly called but rather invoked by some external event (external to the processor core that is - on-chip peripherals may generate interrupts). When an interrupt is called the context of the current thread is automatically preserved before switching context to the ISR. On return, the reverse context switch occurs restoring the state of the processor before the interrupt so that execution continues from the point of interruption.

The mechanism can be complicated by the presence of a multi-threaded operating system or scheduler whereby the ISR itself may cause a thread-context switch so that on return from an ISR a different thread of execution or context is switched in. Such mechanisms are managed by the operating system in this case.

There is another kind of ISR supported on some processors - that of a *software interrupt*. A software interrupt is used like a function call in the sense that it is explicitly invoked by an instruction rather than a single event, but offers an indirection mechanism whereby the caller does not need to know the address of the ISR and indeed that address may change. In that sense it is little different than calling a function through a pointer, but because it is an ISR it runs in the interrupt context, not the caller's context, so may have restrictions and privileges that a normal function does not.

Fundamentally an interrupt is able to respond directly and deterministically to events where otherwise you might poll or test for an event then handle it, but could only handle it at the time you choose to test for it rather than on its actual occurrence, which may be variable and unacceptably long.

answered Jul 21 '13 at 10:04

Clifford
50.5k 7 48 103

The above answers are pretty much complete...special note to the software interrupts by Clifford.

The only addition I would make is this. The register context stored on a function call is defined by the Procedure Calling Convention for the CPU Architecture. This usually means that the caller saves somethings on stack and the callee saves some things and is pretty much a static set. Exception: IA64 which has a dynamic window of register saves/restores.

On ISR, the only register context stored is what is going to be used in the ISR. If one register is used, only that register is saved/restored.

On most cpus, the register set stored/restored in a function call is much bigger than the ones stored/restored in an ISR due to the static nature of procedure calling conventions.

answered Aug 13 '13 at 20:40

lsk
472 4 5

A function call is usually program generated.

An interrupt is usually generated by hardware.

for example: If you program in C, You call the function by making the function. There is the Main function and the Sub-functions. The sub functions are programmed by the programmer.

The interrupts however, are hardware specific. for example: In a computer, The CPU is processing a program and running it. If you press a key on your keyboard, the keyboard then sets the keyboard interrupt to the cpu. Now the CPU can act on keypressed action.

The best way to sum up? 1. functions are created by programmers. 2. interrupts are physically designed into the hardware and utilized by other hardware
https://www.youtube.com/SaqibJaved

edited Oct 25 at 6:05

answered Oct 25 at 5:47

Muhammad Saqib Javed
11 2
