Seminar 2

Summary

- The I/O subsystem
- Memory mapped and isolated I/O
- Polling vs interrupts
- Vectored interrupts
- Priority of interrupts
- Latency times
- DMA

Seminar 2

Some references

1. A. Clements, "The principles of computer hardware", Oxford, 2000, cap. 8, pagg. 407-416.
2. J. B. Peatman, "Design with Microcontrollers", McGraw - Hill, 1988, cap. 3, pp. 56-90.

Interrupts

Each microprocessor system is characterized by the same fundamental components, that are:

1. Aritmetic Logic Unit (ALU)
2. Control Unit
3. Memory
4. I/O peripheral units

The management of the I/O subsystem can be operated using different strategies. The more commonly encounterd in mCs and DSPs (for real time control applications) is based on interrupts.

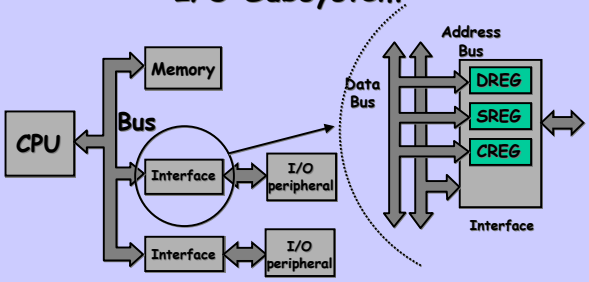
Interrupts

In some simple cases, a program controlled management of the peripheral units is preferable.

This technique (called "polling") is by far less efficient with respect to the use of interrupts.

Besides, some DSPs and top range mCs, allow the use of a dedicated I/O processor, called DMAC (Direct Memory Access Controller). This allows the management of the I/O subsystem with the minimum use of the CPU time.

I/O Subsystem



I/O subsystem organization. The various peripheral units are connected to the bus by a suitable interface circuit.

I/O Subsystem

The interface circuit takes care of logical and, if required, electrical adaptation between the peripheral units and the CPU. Each peripheral unit operates asynchronously with respect to the CPU.

The interface must therefore include different registers (I/O ports) to:

1. allow data exchange with the CPU;
2. allow configuration of the peripheral unit;
3. keep trace of peripheral unit status.

I/O Subsystem

Control register CREG and status register SREG are often made up of a few bits only and so are often part of the same memory location. Also DREG registers have sometimes a different size with respect to the CPU word (for instance in ADCs).

These registers must be read and written. There are two different possible organizations:

1. memory mapped I/O;
2. isolated I/O.

Memory mapped I/O

Memory mapped I/O organizations consider the peripheral unit registers as if they were conventional memory locations, which do not require specific instructions to be read and/or written. This strategy is often used in mCs and DSPs.

Its implementation simply require the connection of peripheral units to some particular address bus lines, that, thanks to decoding circuits (multiplexers), select the different units straightforwardly.

Isolated I/O

The organization with isolated input/output is instead based on specific instructions to program and manage the different peripheral units.

The execution of these instructions require the same address bus management of conventional instructions. Differently from these, they require dedicated control lines to operate on the various peripheral units.

This tecnique is no longer used in modern mCs and DSPs.

Typical peripheral units

mCs and DSPs for embedded control applications are typically characterized by the same peripheral units such as:

1. A/D converters and, sometimes, DACs;
2. timer and counters (PWM modulators);
3. communication modules (serial interfaces, field bus drivers, …);
4. encoder interfaces;
5. display interfaces (only in mCs);
6. external memory drivers (DRAM rarely).

Synchronization

Since the peripheral units operate asynchronously and at a lower speed, with respect to the CPU, the exchange of data between the CPU and the peripherals needs to be synchronized.

There are three fundamental approaches:

1. *polling* or program control;
2. use of an *interrupt system;*
3. use an *I/O processor (DMAC).*

The more commonly adopted is the second one.

Program Control

In program control the CPU periodically polls the peripheral unit status and data registers.

When data are available or when the peripheral unit is ready to receive data, the CPU takes the required steps.

Then it begins polling the peripheral unit again.

This approach is clearly not efficient, because it implies a considerable waste of CPU time.

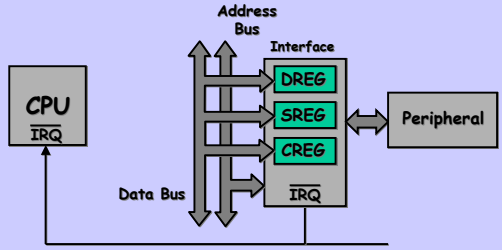
Interrupts

The availability of an interrupt system allows the CPU to take care of other activities while the peripheral units are active. When any peripheral unit requires CPU intervention, e.g. when there are data available for transfer, it sends to the CPU an Interrupt Request (IR).

This signal causes the interruption of the CPU activity and what is called a "context switching". After the acknowledgement of the IR signal the CPU takes care of the peripheral unit: an interrupt service routine, ISR, is started.

Interrupts

Any interrupt system requires the presence of at least a specific control line, that, once asserted by the peripheral units, causes the CPU context switching.



Interrupts

Upon acknowledgement of an interrupt request a typical sequence of events is started within the CPU:

1. the current instruction is completed;
2. context is saved;
3. if possible, an interrupt service routine is started;
4. context is restored and the main program execution is continued.

Point 2 is particularly important: the CPU needs to restore its state, once the interrupt has been served.

Context saving

All interrupt systems require, at least, the program counter (PC) and the CPU status register (SR) to be saved.

Some processors allow to switch to an alternative register bank, so that the original content of the CPU registers is automatically and rapidly preserved from any modification.

In other cases, the programmer must take care of this, pushing the register contents into a stack (normally implemented in memory) at the beginning of the ISR and popping them out at the end.

Interrupt Service Routine

The interrupt service routine is nothing but a subroutine, activated by the CPU upon the acknowledgement of an interrupt request.

The subroutine includes all the instructions required to take care of the peripheral unit's activities, allowing data exchange with the CPU.

At the end of the ISR, a particular instruction (normally named RTI or similarly) makes the CPU restore the original activities.

Interrupts

Since various peripheral units can be simultaneously active in any mC or DSP, several problems require our attention:

1. identification of the peripheral unit that sent the interrupt request signal;
2. simultaneous requests from different units must be suitably managed;
3. interrupt requests occurring within interrupt service routines need to be managed (nested interrupts).

There are several different possible solutions (i.e. hardware organizations).

Program Control

The first solution, used in past architectures, exploits an interrupt status register, where each single bit identifies a peripheral unit. The peripheral unit that sends the interrupt request also asserts its own bit in the status register.

When the CPU acknowledges the IR signal, it sequentially examines the bits of the status register and thus finds out which peripheral sent the IR signal. Then, the CPU calls the appropriate interrupt service subroutine.

Program Control

This solution also implicitly implements a priority mechanism, since the status register bits are read sequentially, in a given order. In the same order the CPU serves possible simultaneous interrupts requests. The management of the status register is left to the program, that has to de-assert the bit of the served interrupt request, so as to avoid dangerous loops.

This strategy is clearly not efficient, since the reading of the interrupt status register normally requires a significant amount of time.

Vectored Interrupts

A more efficient organization uses vectored interrupts. In this case, the processor has several interrupt lines (IRQ1, IRQ2, …), each one is dedicated to a single peripheral (or to a small group of peripherals). It can then automatically activate the proper interrupt sevice routine.

The simplest way to implement this organization is to give the processor a specific instruction, executed upon every IR signal acknowledgement, that operates the fundamental actions: push PC, jump rout_N.

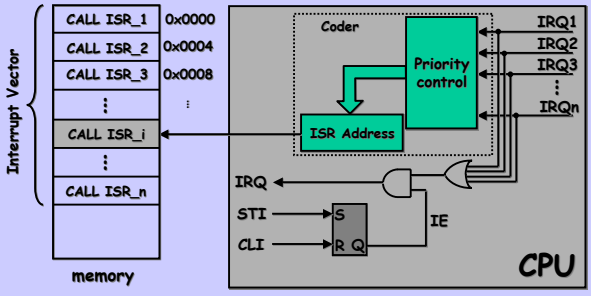
Vectored Interrupts

The management of nested interrupts is also possible, since the coding circuitry can allow the generation of different priority IR signals.

The completion of pending requests must be suitably taken care of, so that all possibly interrupted subroutines are in any case completed.

This requires a quite complex coding circuit design, so that, sometimes, this is not located inside the CPU, but constitutes, by itself another on-chip peripheral unit.

Vectored Interrupts



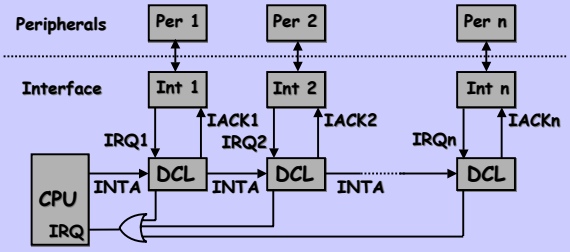
Organization of a vectored interrupt system.

Interrupts with daisy-chain

A different way to manage several interrupt sources with only one IRQ line, is based on the so called daisy-chain concept. It is a particular hardware organization, that allows the CPU to rapidly and automatically identify the peripheral unit that sent the IRQ signal.

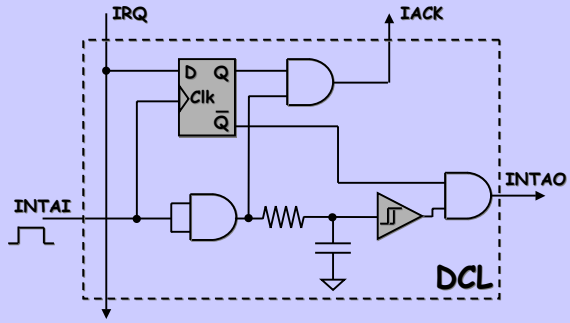
Upon the IRQ acknowledgement, the CPU sends a signal (INTA) down the daisy chain control line DCL. This signal propagates from peripheral to peripheral, until it gets to the one that asserted the IRQ line.

Interrupts with daisy-chain



Organization of an interrupt system with daisy-chain.

Interrupts with daisy-chain



Simplified schematic of DCL node.

Interrupts with daisy-chain

When the peripheral that asserted the IRQ line is reached by the INTA signal, it prevents the signal from propagating further down the DCL and generates the IACK signal. Then, the interface circuit writes a particular code onto the data bus, called interrupt select, that allows the CPU to start to the appropriate ISR.

This procedure also implies a priority coding mechanism, because the peripherals are served in the same order of their location along the DCL.

Interrupts with daisy-chain

The daisy-chain organization is sometimes used together with internal vectorization, to allow the identification of a particular peripheral unit within a small group, associated to a particular processor interrupt line.

In this case the interrupt system is said to be externally vectorized, meaning that the CPU has a smaller number of interrupt request lines with respect to the number of connected peripherals.

Maskerable Interrupts

It is possible to prevent the CPU from being interrupted by peripheral units during certain time intervals, by masking the interrupt sources. In pratice, the IRQ signal is disabled by keeping the IE (Interrupt Enable) signal in a low logic state. Sometimes, it is possible to inhibit any single interrupt source individually. All processors have at least an interrupt source that cannot be masked (NMI). This can be used to manage critical conditions, like, for example, power supply failures.

Interrupts

The fundamental parameters of the interrupt system of any mC or DSP are:

1. number of possible sources;
2. priority management;
3. context switching speed;
4. interrupt masking.

In real time control applications, the context switching speed is often critical. Also crucial is the possibility of protecting critical code areas (also called critical regions) from interrupts.

Interrupts

The more recent mCs and DSPs can count on an internally vectored interrupt system.

The calling of the appropriate interrupt service routine is mainly automatic, as is often the saving of, at least, the fundamental CPU registers.

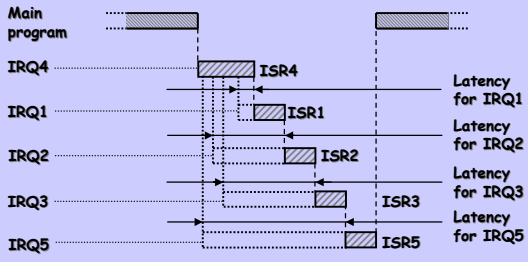
This minimizes the latency time of the interrupt, i.e. the time interval between the interrupt request and the execution of the first instruction of the interrupt service routine.

Interrupts

The number of interrupt sources is normally quite high (>10, but can be as high as 100) and it is often possible to configure any single interrupt source, defining its priority and, in some cases, if it has to be vectored or not.

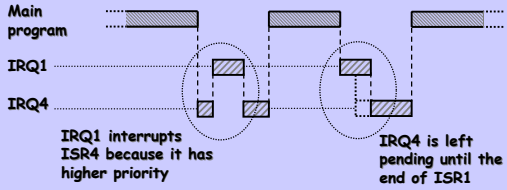
Different strategies can be found, as to the management of priorities: in some processors only simultaneous interrupt requests are taken care of, others, more sophisticated, also allow to implement nested interrupts: higher priority requests can interrupt lower priority ISRs (nesting).

Interrupts



Example of an interrupt system that does not allow interrupt nesting.

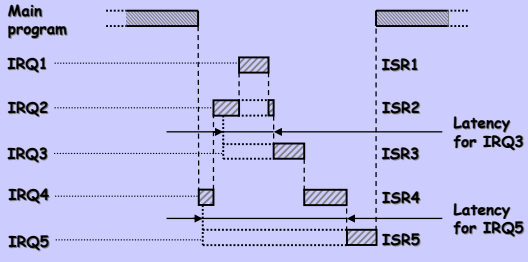
Interrupts



Example of an interrupt system that allows interrupt nesting.

Higher priority interrupts are served immediately, the lower priority ones are left pending.

Interrupts



Again the previous example, now in a system that allows nesting.

Interrupts

The two strategies are different, since with nesting, a higher priority interrupt can be served with minimum latency. Nesting may penalize interrupts with low priority (as, in our example, IRQ4, but not IRQ5).

Only when the different ISRs have all relatively short duration, the strategy that does not allow nesting may offer a good performance level (maximum latency is small).

Indeed, this is the case for almost all DSPs (exceptions: AD21xx DSP by Analog Devices and Motorola 56F8xx).

Latency

It often very important to estimate the maximum delay that can affect an interrupt request, i.e. the interrupt maximum latency.

The interrupt latency always include an intrinsic component (T_{LI}), due to need to complete the current instruction, save (automatically or manually) the context i.e. PC, SR, user registers, ...

However, interrupts with low priority may be affected by latencies much higher than T_{LI}. In general, latency increases as priority decreases.

Latency

In a system allowing nesting, the latency of a *n* priority interrupt, in the worst case, can be expressed by:

$$T_{L\_n} = T_{LI} + \sum_{i=1}^{n-1} T_{ex\_i}$$

where T_{ex_i} represents the total execution time of any ISR whose priority is higher than *n*.

Note that the sum of all execution times is actually independent of nesting being allowed or not.

Latency

In a system that does not allow nesting, the latency of an *n* priority interrupt, in the worst case, is instead given by:

$$T_{L\_n} = T_{LI} + \max_{j>n}(T_{ex\_j}) + \sum_{i=1}^{n-1} T_{ex\_i}$$

that is longer than the previous one. The additional term is equal to the maximum among the execution times of all the ISRs having lower priority with respect to the considered *n* priority one.

Latency

The highest priority interrupt has a latency time that is often higher than T_{LI}, even in a system that allows nesting. This can be due to, at least, two different causes:

1. interrupts have been disabled to protect critical regions in main program (T_{CR});
2. a particularly complex (and time consuming) instruction is being executed.

In the worst case, the total latency time of the highest priority interrupt is equal to the sum of the three terms ($T_{LI} + T_{CR} + T_{INST}$).

Latency

When a program segment has to be executed sequentially, e.g. to guarantee a precise temporization, it is necessary that all interrupts are masked.

On the contrary, the duration of the program critical region is not going to be constant, but varies in an *apparently random* way, depending on the occurrence of other interrupt requests.

This may generate non predictable malfunctions such as jitter or glitch phenomena, very difficult to troubleshoot.

Interrupt Density

When a mC or DSP needs to take care of various interrupt sources, for example N, there may be a problem to guarantee that all of them can be effectively served.
A necessary condition to make this happen is that the following inequality is satisfied by the interrupt system:

$$\sum_{i=1}^{N} \frac{T_{ex\_i}}{T_{p\_i}} < 1$$

where T_{p_i} represents the minimum repetition period of the i-th ISR.

Interrupt Density

The condition is not also sufficient, because it does not take into account that higher priority interrupts may combine in any sequence. A particularly unfortunate one could prevent the processor from serving the i-th interrupt in all repetion periods T_{p_i}.
To make sure this does not happen, for each interrupt source, we need to verify the so called interval condition:

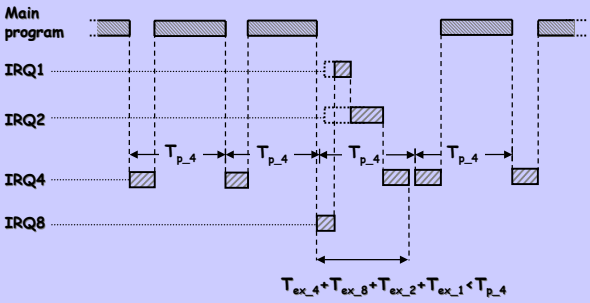
$$T_{ex\_i} + \max_{j>i}(T_{ex\_j}) + \sum_{k=1}^{i-1} N_k \cdot T_{ex\_k} < T_{p\_i}$$

Interrupt Density

The interval condition says that, for the i-th interrupt, the sum of its total execution time, of the maximum duration of any of the lower priority interrupts (in a system where nesting is not allowed) and of all the execution times of all the ISRs relative to higher priority interrupts (multiplied the number of times N_k they may occur in period T_{p_i}) is in any case lower than T_{p_i}. The formal definition for N_k is:

$$N_k = \mathrm{INT}((T_{p\_i} - T_{ex\_i})/T_{p\_k}) + 1$$

Interrupt Density



Interval condition in an interrupt system without nesting.

Problem

In a system with 2 interrupts (vectored with priority 1 and 2, no nesting), the IRs of each source occur every 100 μs. In the main program we created a critical region whose duration is 20 μs.

We want to know what are the possible durations of the ISRs for the two interrupts.

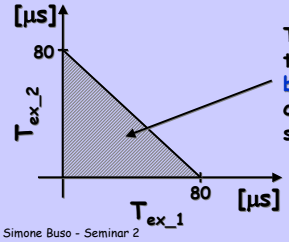
We need to calculate the interval conditions for the two interrupt service routines:

1. $T_{ex\_1} + \max(T_{RC}, T_{ex\_2}) < 100$ [μs]
2. $T_{RC} + T_{ex\_2} + T_{ex\_1} < 100$ [μs]

Problem

We see that the second condition is more restrictive than the first one. Thus, the limit condition on the durations is given by:

$$T_{ex\_2} + T_{ex\_1} < 80 \text{ [μs]}$$



The values falling within the shaded area satisfy both the interval conditions and represent solutions for our problem.

Direct Memory Access

The DMA approach is the most efficient solution to interface the CPU with a given set of I/O peripherals. It basically consists of an additional bus controller, named DMAC, that, once programmed, takes care of data transfers to/from the peripheral units without CPU intervention.

This approach is also the only viable one for those peripherals whose speed is comparable with the CPU's one.

However, it is a fairly expensive solution, only used in high end devices.

Direct Memory Access

The DMAC is extremely useful in all those cases where a peripheral unit produces large amounts of data to move into memory.

The DMAC takes care of all the data moves controlling the data and address buses independently from the CPU, that is left free to execute other operations.

The move operation can be done on single pieces of data (cycle stealing) or on data blocks (burst mode).

When the DMAC is active the CPU cannot access the bus and so operate on memory.

Direct Memory Access

That's why the DMAC is normally used in association with a cache memory. This allows the CPU to continue its computations indipendentely from the DMAC, unless the data stored in the cache are modified by the DMAC in main memory (cache update problem).

The DMAC programming normally consists in the indication of a memory address and of the number of data that need to be moved from that address on. When the data transfer is complete, the DMAC normally generates an interrupt.

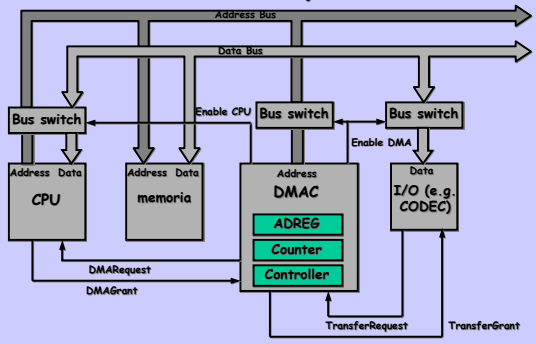
Direct Memory Access

A DMAC can normally take care of more than a single I/O peripheral, since it normally has more parallel channels.

A high performance DMAC for instance is the one built into the TMS320C4x DSP: it is able to complete a data transfer to and from memory in any clock cycle, without interfering with the processor. It is able to manage up to 6 simultaneuos data transfers (6 channels).

Similar devices are on board the 96002 DSP series by Motorola and the ADSP-2106x (and higher) by Analog Devices.

Direct Memory Access



Simplified structure of a DMAC system

Direct Memory Access

The operation protocol of the DMAC is the following:

1. the peripheral asks the DMAC permission to start a data transfer (TransferReq.).
2. the DMAC asks the CPU permission to take control of the buses (DMAReq.).
3. the CPU grants use of the buses: the switches cut the CPU off and connect DMAC and peripheral (DMAGrant) to memory.
4. the DMAC sets the address bus e allows the peripheral to read/write data (TransferGrant) into memory.