Seminar 1

Contents

- Definition of microcontroller (mC)
- Definition of Digital Signal Processor (DSP)
- Criteria for performance comparison of mCs e DSPs
- Performance measurements

Microcontrollers (mCs)

A microcontroller is a processor specifically designed and optimized to perform control, timing, supervising tasks on various target devices. It is characterized by the availability of relatively large amounts of "on chip" memory (ROM, EEPROM, Flash ...) and of several peripheral units, for different functions (I/O, A/D conversion, timer, counters, PWM, …). It is normally characterized by reduced complexity and low cost.

Microcontrollers (mCs)

Peripheral units in mCs:

- A/D converters (number of bit, conversion speed, linearity vary a lot among different devices)
- Timer and counters
- PWM modulators
- External memories (ROM, EEPROM, FLASH)
- Communicaion ports (serial, I2C, field bus e.g. CAN)

Microcontrollers (mCs)

The use of mCs is very common for the implementation of:

- portable measurement instruments;
- PC periferals;
- fax/photocopiers;
- home appliances;
- cell phones;
- industrial applications, in particular in the automotive and electrical drives fields.

Digital Signal Processors (DSPs)

DSPs are microprocessors specifically designed and optimized to efficiently perform real time signal processing tasks. They are characterized by high computational power and relatively low cost (if compared to general purpose processors). Particular care is taken in minimizing the power consumption (e.g. in embedded portable applications).

Digital Signal Processors (DSPs)

Several different DSP families are available on the market. They all exhibit some common features:

- availability of a built-in multiplier circuit (MAC instruction);
- capability to operate multiple memory accesses in a single clock cycle;
- specific addressing modes for circular registers and stacks;
- sophisticated program flow control instructions;
- availability of DMA circuitry (top level).

Digital Signal Processors (DSPs)

The major application areas for DSPs are related to:

- coding/decoding of speech, hi-fi audio segnals, video signal processing;
- compression/decompression of data;
- encryption/decryption of data;
- mixing of audio and video signals;
- sound synthesis.

DSPs vs mCs

Traditionally, mCs were used in the implementation of control functions, thanks to the wide range of peripheral units available on-chip. The computational power was limited (CPUs had 8 bits or less, no hardware multiplier).

DSPs were used, instead, almost only for signal-processing applications, where the key parameter is computational power.

Currently, the differences in the application fields of mCs and DSPs are a lot fuzzier.

DSPs vs mCs

More recent DSPs include peripheral units traditionally typical of mCs. On the other hand, mCs present, at least in top range models, hardware organizations and computational powers closer and closer to those typical of DSPs. Costs and performance may be very close and, for particular applications, the choice of the device may be quite difficult.

We definitely need criteria to compare different devices.

DSPs vs mCs

The fundamental parameters for the comparison are, of course, cost and performance.

To minimize the cost parameter, for given specifications, it is normally required to take into account not only the device cost, but also the estimated development time, the so called time to market.

DSPs vs mCs

The cost of device is largely dependent on the expected production volume.

Time and resources required by the development of the application are a function of several factors, like:

- availability of high quality and high reliability development tools;
- effective technical support from the device manufacturer.

DSPs vs mCs

The application specifications determine the performance level required for the selected microprocessor in terms of:

- required peripheral units and their basic parameters (e.g. A/D converter with 8, 10 or 12 bits);
- operating conditions (e.g. maximum allowable power consumption, temperature range);
- required computational power (real time control, signal processing …).

Performance measurement

The perfomance level of any processor can be measured only in terms of time required to excute a given program.

In the case of mCs or DSPs this is the same time the processor effectively spends on the program instructions (computation time), unless an operating system coordinating several tasks in time sharing is running on the device.

Estimation of computation time

The computation time of a program is a key parameter in real time applications (both in control and signal processing). This can be estimated based on three parameters:

- processor clock period;
- number of clock cycles required by the instructions in the program;
- number of di instructions required by the program.

Estimation of computation time

The clock period and the number of clock cycles required by the various program instructions can be read on the processor datasheet/user manual.

The number of instructions required by a given algorithm is a function of the processor architecture.

By architecture we mean the set of resources that are available to the programmer for the implementation of the algorithm (instruction set).

Estimation of computation time

Any given architecture can be implemented in different ways at the hardware level.

We therefore make a distinction between processor organization and architecture: the former is the particular hardware implementation of the latter.

The architecture has a direct effect on the number of instructions required by a given program. The organization determines the clock period and the number of clock cycles required by any instruction.

Estimation of computation time

The computation time of a program can be estimated by using the following formula:

$$T_{cal} = T_{clk} \cdot \sum_{i=1}^{N_{cl}} N_i \cdot NC_i \qquad (1)$$

where T_{clk} is the processor clock period, N_i is the number of class *i* instructions in the program, NC_i is the average number of clock cycles required by class *i instructions*, N_{cl} is the number of considered instruction classes.

Stimation of computation time

Relation (1) assumes that the program is not interrupted by other processes and neglects the delays due to memory accesses.

To increase the speed of a processor, we therefore need to:

- reduce the clock cycle (T_{clk});
- reduce the number of cycles required by the more commonly used instructions (NC);
- reduce the number of instructions required by a given algorithm.

Speed limits!

Reducing the clock cycle duration always implies the increase of power consumption for the processor.

This can be limited by reducing also the supply voltage.

Which tells us why there is a strong need for lower and lower power supply voltages (<1V) in computer applications.

The limitations are basically technological (we need new processes/materials).

Speed limits!

The reduction of the number of clock cycles required by an instruction calls for a more sophisticated hardware organization of the processor, e.g. wired control instead of micro-programmed control, higher degree of parallelism (achievable in several different ways: VLIW, SIMD, etc.) or the use of pipelines.

This trend leads to complex processors, with high cost. The limitation in this case is basically "economical".

Speed limits!

The number of instructions required by a given algorithm is a function of the processor architecture, i.e. of its instruction set, as seen by the programmer/compiler.

The reduction of this parameter leads to complex instruction set computers (CISC), instead of reduced (and simple) instruction set computers (RISC). This again affects the processor organization and its cost. That´s why RISC processors are a lot more used than CISC processors.

Maximizing performance

The processor performance is a function of both its architecture and of its organization, at the hardware level.

The maximization of performance calls for a co-ordinated design of hardware and software.

The problem is further complicated by the action of several design constraints such as:

- cost;
- electric power consumption.

Performance measurements

A typical way to measure a general purpose processor performance is to use benchmarking, i.e. the execution of suitably designed test programs. This method helps to evaluate the overall processor performance, including memory management.

Recently the same strategy is being applied also to DSPs. The considered test programs are typical signal processing algorithms (FFT, FIR, IIR filters etc.).

Performance measurements

The usual benchmarking method for general purpose processors uses complete applications (e.g. SPEC method).

This approach cannot be used with DSPs, because the results would be strongly dependent on the quality of the adopted compiler.

Besides, any code optimization becomes very difficult and comparison of different devices almost impossible. Kernel benchmarking is the adopted solution.

DSP Benchmarking

Benchmarking programs must have, at least, the following basic features:

1) relevance with respect to the typical DSP applications;

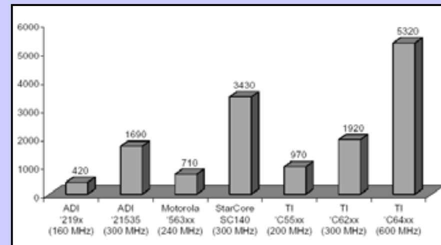
2) clear and precise definition (e.g. what type of DFT algorithm is considered);

3) simplicity;

4) optimization: they must be easy to optimize for any given DSP architecture.

DSP Benchmarking

Results of benchmarking tests are available on the market as reports edited by several specialized companies (e.g. BDTi). They are extremely expensive (>2000 Euros).



Performance indexes

The following list include the more commonly encountered perfomance indexes:

- MACS: MAC operations per second
- MIPS: millions of instructions per second
- MOPS: millions of operations per second
- FLOPS: millions of floating point operations per second

All the indexes give little information on the actual processor speed. They do not allow any comparison between different devices.

Performance indexes

The MACS index shows the maximum number of multiplies (with sum on the accumulator) a CPU is able to operate in a second (peak value).

However, in any DSP program, a lot of different operations are used (e.g. sums, memory read/write ...), whose impact on speed is not described by the index.

The MACS index does not give any serious measure of the actual processor speed.

Performance indexes

The MIPS index shows the maximum number of instructions a CPU is able to execute in a second (peak value).

But, the number of instructions required by any algorithm depends on the CPU architecture (a single instruction can perform different amounts of operations in different architectures) and also on the compiler quality.

It is, at most, only possible to compare devices sharing the same basic architecture.

Performance indexes

The MOPS index shows the maximum number of operations a CPU is able to execute in a second (peak value).

Its definition is ambiguous itself, because it does not clearly define what exactly is an operation, or at least what is the reference set of operations, if any.

The meaning of the index is therefore not very clear and this should not be considered for comparison purposes.

Performance indexes

The FLOPS index shows the maximum number of floating point operations a CPU is able to execute in a second (peak value).

The validity of this index is similar to that of the MOPS index, with the further limitation that it can be applied only to floating point architectures.

None of the presented indexes takes into account other key issues for CPU performance measurement, like, for example, memory management and organization.

Example: MIPS vs Computation Time

Instruction classes	Average number of clock cycles (NC)
A	1
B	2
C	3

A processor has 3 different instruction classes. Each class requires a different number of clock cycles. Any given program will use a certain amount of instructions of each class (compiler dependent).

Example: MIPS vs Computation Time

Compiler / Programmer	Number of instructions per class (hundreds)		
	A	B	C
1	5	1	1
2	10	1	1

Two different compilers/programmers produce two different programs for the same algorithm, using a different number of instructions and a different distribution among the three classes.

Example: MIPS vs Computation Time

Applying (1) we can now evaluate the computation time of the two programs. We find:

$$T_{cal1} = T_{clk} \cdot 100 \cdot (5 \cdot 1 + 1 \cdot 2 + 1 \cdot 3) = 1000 \cdot T_{clk}$$

$$T_{cal2} = T_{clk} \cdot 100 \cdot (10 \cdot 1 + 1 \cdot 2 + 1 \cdot 3) = 1500 \cdot T_{clk}$$

The second program has a computation time longer than the first by 50%. It is then much slower the the first.

Example: MIPS vs Computation Time

If we compute the MIPS index for the two programs we find:

$$MIPS_1 = 10^{-6} \cdot 100 \cdot (5+1+1)/(1000 \cdot T_{clk}) =$$

$$0.7 \cdot F_{clk} \cdot 10^{-6}$$

$$MIPS_2 = 10^{-6} \cdot 100 \cdot (10+1+1)/(1500 \cdot T_{clk}) =$$

$$0.8 \cdot F_{clk} \cdot 10^{-6}$$

The second program has a higher MIPS rate compared to the first. According to this index the second program should be faster!

Example: Data-sheet

The manufacturer claims a 30 MIPS (F_{clk} = 40 MHz) operating speed.

MICROCHIP dsPIC30F

dsPIC30F Enhanced FLASH 16-bit Digital Signal Controllers Motor Control and Power Conversion Family

High Performance Modified RISC CPU:

- Modified Harvard architecture
- C compiler optimized instruction set architecture
- 88 base instructions
- 24-bit wide instructions, 16-bit wide data path
- Linear program memory addressing up to 4M Instruction Words
- Linear data memory addressing up to 64 Kbytes
- Up to 144 Kbytes on-chip FLASH program space
  - Up to 48K Instruction Words
- Up to 8 Kbytes of on-chip data RAM
- Up to 4 Kbytes of non-volatile data EEPROM
- 16 x 16-bit working register array
- Three Address Generation Units that enable:
  - Dual data fetch
  - Accumulator write back for DSP operations
- Flexible Addressing modes supporting:
  - Indirect, Modulo and Bit-Reversed modes
- Two, 40-bit wide accumulators with optional saturation logic
- 17-bit x 17-bit single cycle hardware fractional/integer multiplier
- Single cycle Multiply-Accumulate (MAC) operation
- [illegible] Barrel Shifter
- Up to 30 MIPs operation:
  - DC to 40 MHz external clock input
  - 4 MHz - 10 MHz oscillator input with PLL active (4x, 8x, 16x)

Peripheral Features (Continued):

- Addressable UART modules supporting:
  - Interrupt on address bit
  - Wake-up on START bit
  - 4 characters deep TX and RX FIFO buffers
- CAN bus modules

Motor Control PWM Module Features:

- Up to 8 PWM output channels
  - Complementary or Independent Output modes
  - Edge and Center Aligned modes
- 4 duty cycle generators
- Dedicated time-base with 4 modes
- Programmable output polarity
- Dead-time control for Complementary mode
- Manual output control
- Trigger for A/D conversions

Quadrature Encoder Interface Module Features:

- Phase A, Phase B and Index Pulse input
- 16-bit up/down position counter
- Count direction status
- Position Measurement (x2 and x4) mode
- Programmable digital noise filters on inputs
- Alternate 16-bit Timer/Counter mode
- Interrupt on position counter rollover/underflow

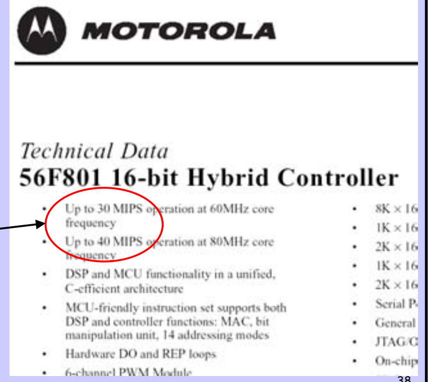
Example: Data-sheet

The manufacturer claims a maximum speed of 8000 MIPS! But here F_{clk} = 1000 MHz!

TMS320C6414T, TMS32
FIXED-POINT DIGITAI

- Highest-Performance Fixed-Point Digital Signal Processors (DSPs)
  - 1.67-, 1.39-, 1-ns Instruction Cycle Time
  - 600-, 720-MHz, 1-GHz Clock Rate
  - Eight 32-Bit Instructions/Cycle
  - ~~Twenty-Eight Operations/Cycle~~
  - 4800, 5760, 8000 MIPS
  - ~~Fully Software-Compatible With C62x™~~
  - C6414/15/16 Devices Pin-Compatible
- VelociTI.2™ Extensions to VelociTI™ Advanced Very-Long-Instruction-Word (VLIW) TMS320C64x™ DSP Core
  - Eight Highly Independent Functional Units With VelociTI.2™ Extensions:
    - Six ALUs (32-/40-Bit), Each Supports Single 32-Bit, Dual 16-Bit, or Quad 8-Bit Arithmetic per Clock Cycle
    - Two Multipliers Support Four 16 x 16-Bit Multiplies (32-Bit Results) per Clock Cycle or Eight 8 x 8-Bit Multiplies
- Two External Mer
  - One 64-Bit (EM
  - Glueless Interf Memories (SRA Synchronous M SBSRAM, ZBT
  - 1280M-Byte To Memory Space
- Enhanced Direct- Controller (64 Ind
- Host-Port Interfac
  - User-Configura
- 32-Bit/33-MHz, Interface Conforr [C6415T/C6416T]
  - Three PCI Bus Prefetchable Non-Prefetcl
  - Four-Wire Seria
  - PCI Interrupt R Program Contr

Example: Data-sheet


MOTOROLA

Technical Data

56F801 16-bit Hybrid Controller

- Up to 30 MIPS operation at 60MHz core frequency
- Up to 40 MIPS operation at 80MHz core frequency
- DSP and MCU functionality in a unified, C-efficient architecture
- MCU-friendly instruction set supports both DSP and controller functions: MAC, bit manipulation unit, 14 addressing modes
- Hardware DO and REP loops
- 6-channel PWM Module
- 8K × 16
- 1K × 16
- 2K × 16
- 1K × 16
- 2K × 16
- Serial P
- General
- JTAG/C
- On-chip

The manufacturer claims a maximum speed of 40 MIPS (F_{clk} = 80 MHz).

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.

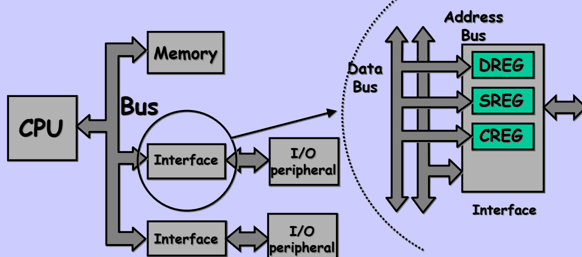
Interrupts

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.

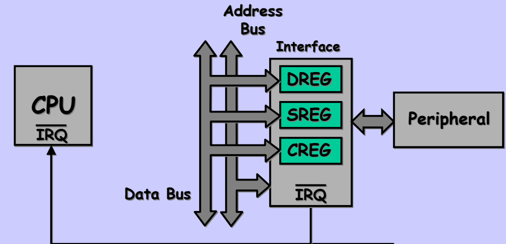
Interrupts

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.

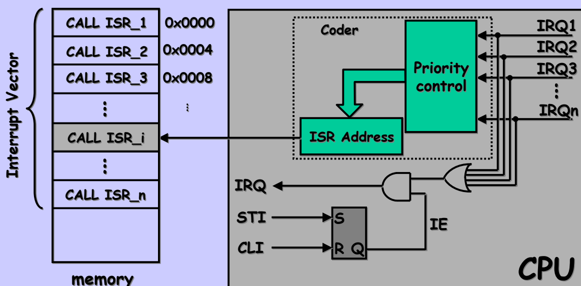
Vectored Interrupts

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



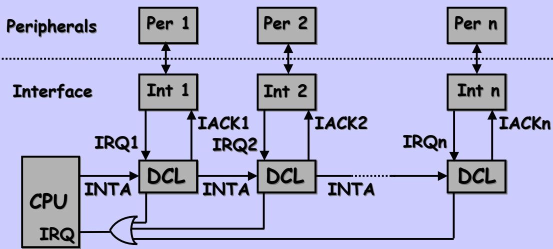
Organization of a vectored interrupt system.

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.

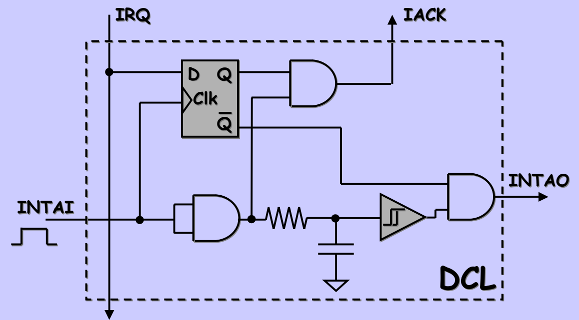
Upon the IRQ acknowledgement, the CPU sends a signal (INTA) down the daisy chain control line DCL. This signal propagates from peripheral to peripheral, until it gets to the one that asserted the IRQ line.

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.

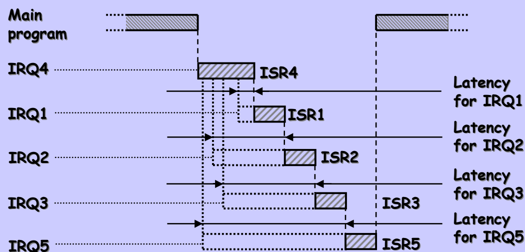
This minimizes the latency time of the interrupt, i.e. the time interval between the interrupt request and the execution of the first instruction of the interrupt service routine.

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.

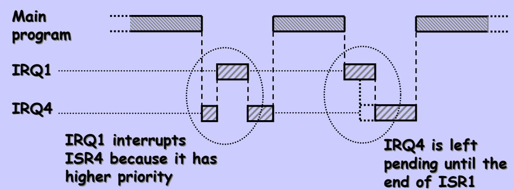
Different strategies can be found, as to the management of priorities: in some processors only simultaneous interrupt requests are taken care of, others, more sophisticated, also allow to implement nested interrupts: higher priority requests can interrupt lower priority ISRs (nesting).

Interrupts



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

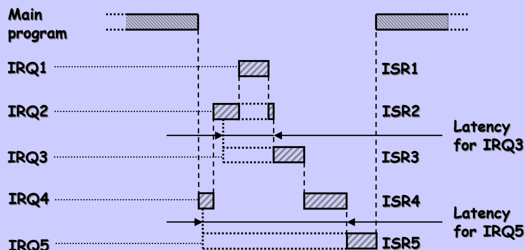
Interrupts



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:

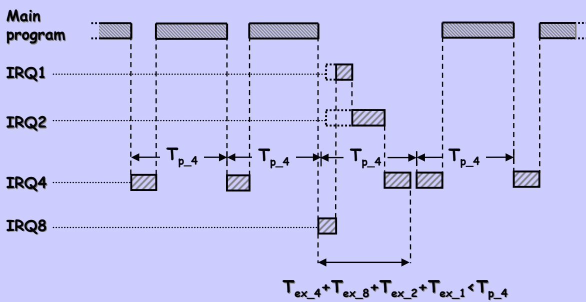
$$T_{ex_i} + \max_{j>i}(T_{ex_j}) + \sum_{k=1}^{i-1} N_k \cdot T_{ex_k} < T_{p_i}$$

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:

$$N_k = INT((T_{p_i} - T_{ex_i})/T_{p_k}) + 1$$

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.

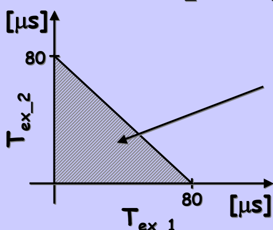
We need to calculate the interval conditions for the two interrupt service routines:

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.

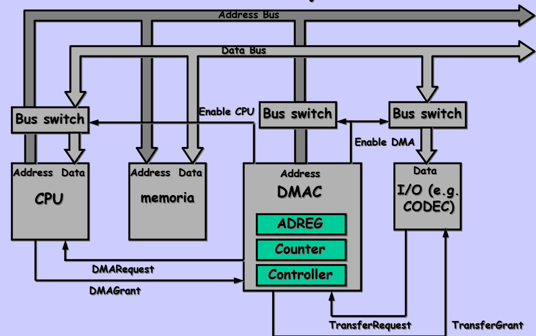
Direct Memory Access

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.

Seminar 3

Summary

- Use of mCs and DSPs in signal processing and control applications
- FIR and IIR filters
- PI and PID regulators
- Predictive regulators (basics)

Seminar 3

Basic references

1. D. Glover, J.R. Deller, "Digital Signal Processing and the Microcontroller", Prentice Hall, 1999.
2. A.V. Oppenheim, R.W. Schafer, J.R. Buck, "Discrete Time Signal Processing", Second Edition, Prentice Hall.
3. K. Ogata, "Discrete Time Control Systems", Prentice Hall, 1987.
4. M. Morari, E. Zafirou, "Robust Process Control", Prentice Hall, 1989.

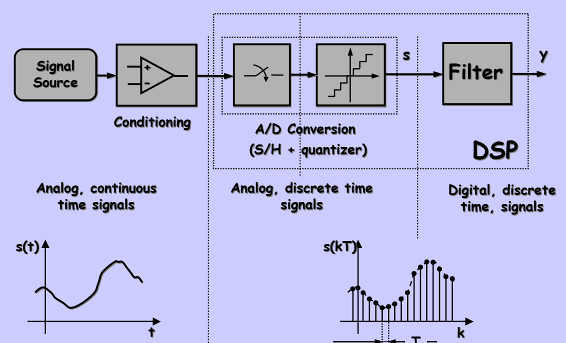
Digital Signal Processing

One of the more frequent uses of mCs and DSPs is in digital signal processing applications and/or real-time control of processes and systems.

The fundamental difference between the two is represented by feedback, not present in the first case, fundamental in the second one.

The problems encountered in these applications are related to discrete time operation of processors, to the finite precision of their arithmetic unit and to the quantization of data and coefficients.

Digital Filters



Digital Filters

In a digital filter, a signal is acquired by the mC or DSP through a A/D converter.

This process implies two effects: sampling and quantization.

Sampling changes the signal from continuous time s(t) to discrete time s(kT).

Quantization changes the signal from analog to digital.

The elaboration takes places on a sequence of quantized samples and generates a new sequence (y).

Digital Filters

As with analog filters, digital filters may have different characteristics:

1. low pass;
2. high pass;
3. band pass or notch;

depending on their frequency response behavior.

Any analog filter can be turned into a digital equivalent within a given precision. Vice-versa is not true: some digital filters do not have analog equivalents (FIR filters).

Digital Filters

Any digital filter can be written as a n-th order difference equation such as:

$$y(k) = b_0x(k) + b_1x(k-1) + \ldots + b_nx(k-n) +$$
$$+ a_1y(k-1) + a_2y(k-2) + \ldots + a_my(k-m)$$

When one, at least, of the a_i coefficients is <> 0 the filter is called IIR (infinite impulse response).

A FIR filter (finite impulse response) is instead characterized by having all the a_i coefficients equal to zero.

IIR digital low pass filter

A very simple example of a IIR digital low pass filter is given by:

$$y(k) = b \cdot x(k) + (1-b) \cdot y(k-1)$$

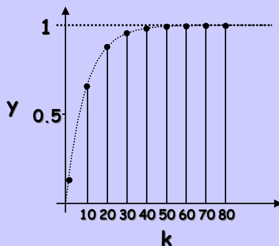
It is easy to analyze the filter's step response, i.e. its response to the input sequence g = {1, 1, 1, 1, 1, … }.

Choosing, for instance, b = 0.1 we get:

y = {0.1, 0.19, 0.27, 0.34, 0.41, … }.

The sequence y goes to 1, but with a infinite duration transient. This is due to the term y(k-1). We also call this a recursive filter.

IIR digital low pass filter



Our IIR filter responds to a step input as if it was the sampled version of a first order low-pass analog filter.

Its speed of response, with respect to the sampling period, depends on our choice of *b*. The bigger *b*, but < 1 (!), the faster the speed of response.

IIR digital low pass filter

Varying parameter b between 0 and 1 we can approximate any first order low pass analog filter. The x(k) and y(k-1) coefficients could be chosen freely, but:

1. if the sum of the coefficients is equal to 1 then the dc gain of the filter is unity;
2. if the y(k-1) coefficient has magnitude < 1 then the filter is also stable.

As an example, the filter:

$$y(k) = 2.1 \cdot x(k) - 1.1 \cdot y(k-1)$$

is unstable!

IIR digital low pass filter

When we need to get a very fast speed of response, we may want to choose values for b very close to 1.

In this case, however, the finite precision of the processor we are using limits our capability to represent the coefficients!

For instance, in an 8 bit processor, if we choose b > 0.992 we are no longer in a condition to correctly represent 1-b.

Indeed, the minimum number we will be able to represent will be $2^{-7} \cong 0.008$. Lower numbers are all "seen" as 0.

FIR digital low pass filter

We can get a similar low pass filter also without using recursion. For example, a filter like the following:

$$y(k) = \frac{1}{N} \cdot \sum_{i=0}^{N-1} x(k-i)$$

is called N-th order moving average filter. It is basically a low pass filter, but its response gets to the steady state after N sampling periods. As always with FIR filters, there is no stability problem, even in case of a wrong coefficient choice.

FIR digital low pass filter

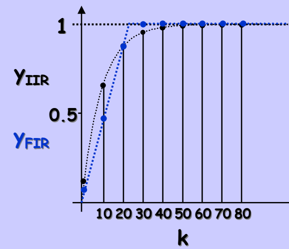
Considering, as an example, N=4, the filter step response (g = {1, 1, 1, … }) is given by sequence y = {0.25, 0.5, 0.75, 1, 1, 1, … }.

Thus, after only 4 steps the transient is over. A similar response cannot be achieved from any analog filter.

We may observe that, to make the two filter responses similar to each other, we need to take a much higher order for the FIR filter (or a much higher b value for the IIR filter).

FIR digital low pass filter

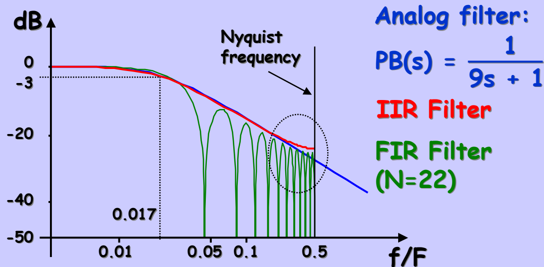
Considering, for instance, N=22, the two filters respond in a similar way.



Thus, the FIR filter requires a bigger number of operations to give a step response similar to the IIR filter's one (22 terms instead of 2). This always happens with FIR filters.

Frequency Response

Digital filters can be described also by means of their frequency response.



Analog filter:

$$PB(s) = \frac{1}{9s + 1}$$

IIR Filter

FIR Filter (N=22)

Frequency Response

Looking at the three frequency responses (taking into account only its magnitude) we see that the filters have a similar behavior.

The FIR filter exhibits frequency cancellation phenomena, due to the periodicity of its structure. The envelope of its frequency response magnitude, follows that of the IIR filter and of the reference analog filter.

The IIR filter and the analog one respond in practically identical ways (the digital filter is indeed the discretization of the analog one).

IIR digital high pass filter

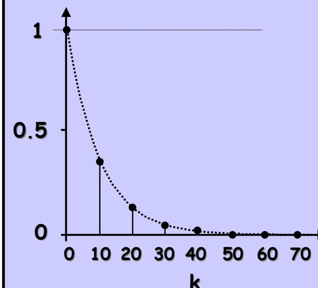
A simple high pass filter can be obtained using the following difference equation:

$$y(k) = x(k) - x(k-1) + a \cdot y(k-1)$$

Being a recursive equation, it corresponds to a IIR filter. Parameter *a* allows to tune the filter response.

Everything is OK if $0 < a < 1$, otherwise we may get (damped) oscillatory step responses or even unstable ones. Actually, it is better to take, at least, $a > 0.5$.

IIR digital high pass filter



Our IIR high pass filter responds to a step input as if it was the sampled version of a first order high pass analog filter.

The graph is obtained with $a = 0.91$. Lower *a* values produce faster responses.

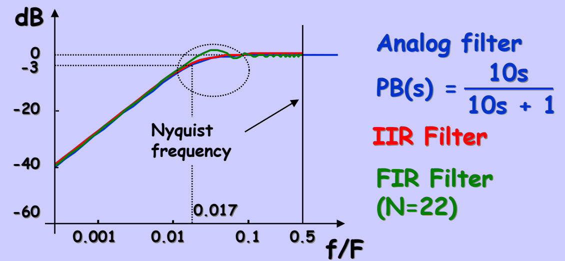
FIR digital high pass filter

Again, we can get a similar filter without using recursion. The following filter:

$$y(k) = \frac{N-1}{N} \cdot x(k) - \frac{1}{N} \cdot \sum_{i=1}^{N-1} x(k-i)$$

is a N-order FIR high pass filter. Its step response reaches the steady-state after N sampling periods. To get a similar response with respect to the IIR filter, we need to take relatively high N values (>20).

Frequency Response



In this case we get again very similar frequency responses: note that the filters have relatively low band pass frequencies.

Discretization

We can always use discretization techniques to turn a continuous time filter into an equivalent discrete time one.

The easiest way to do this is using a suitable approximation of the integral operator (1/s) in the discrete time domain, as for example that based on the Euler approximation:

$$\int_0^{nT} x \, dt \cong \sum_{k=1}^{n} x(k) \cdot T$$

where T is the so called integration step.

Discretization

We can then write:

$$Int_x(nT) \cong T \cdot [\underbrace{x(1)+x(2)+\ldots+x(n-1)}_{Int_x[(n-1)T]}+x(n)]$$

that is

$$Int_x(nT) = Int_x[(n-1)T]+T \cdot x(n)$$

From this we get:

$$\frac{1}{s} = \frac{T}{1-z^{-1}} \Rightarrow s = \frac{1-z^{-1}}{T}$$

(z^{-1}: Unity delay)

Discretization

Because this is an approximation process, the discretization does not maintain the filter frequency response unaltered. Indeed the original filter frequency response is perturbed and warping phenomena appear. We therefore need to be very careful when applying this method, to avoid *unexpected* digital filter behaviors. As a rule of thumb, we say that discretization results are accurate only up to frequencies equal to 1/10 of the sampling frequency.

Discretization

We may use even more sophisticated discretization methods, that allow us to obtain a better frequency response approximation. For instance:

$$s = \frac{2}{T} \cdot \frac{1-z^{-1}}{1+z^{-1}}$$

Trapezoidal integration, or Tustin transform.

Finally, we have several methods based on some kind of response invariance to a particular family of signals, like steps or ramps.

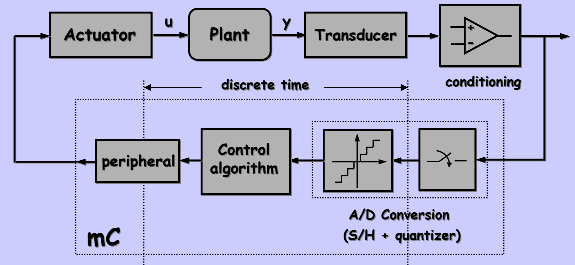
Real-time control

Closed loop digital control of systems or processes, requires the mC or DSP to elaborate signals taken from the plant, according to suitable algorithms, implementing different kinds of regulators.

The design of such regulators requires the application of discrete time automatic control theory.

Their implementation is done using the same signal processing techniques we described for digital filter synthesis.

Real-time control



Closed loop control system with mC

Real-time control

The synthesis of regulators can be done again following different strategies.

The simplest one consists in the discretization of regulators that have been previuosly designed in the continuous time domain.

In most cases, these are just simple PID regulators.

As an alternative, it is possible to use control algorithms that have no continuous time domain equivalent, such as, for instance, the various types of predictive controllers.

Real-time control

It is worth noting that, in most cases, the regulators adopted in industrial applications are just PID controllers.

PID controllers usually represent a very good trade-off between complexity and achievable performance.

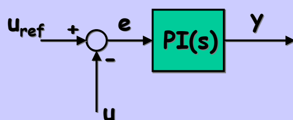
They are extremely robust and relatively easy to tune (small number of parameters).

Achievable performance is often more than satisfactory, even if always lower with respect to that offered by their analog counterparts.

PI Regulator

In the analog domain, a PI regulator is described by an input-output relation of the following type:

u_{ref} + → e → PI(s) → y, with u fed back (−)



$$\frac{Y(s)}{E(s)} = k_p + \frac{k_i}{s}$$

k_p = proportional gain

k_i = integral gain

PI Regulator

By direct discretization, it is possible to turn the continuous time PI regulator into a suitable control algorithm. Of course, k_p and k_i constants ought to be known already!

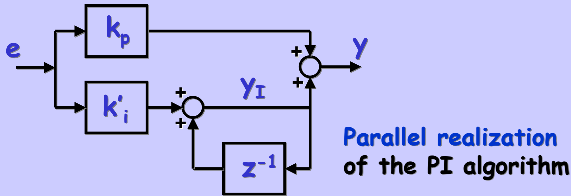
We then immediately find the following control equations:

$$\begin{cases} y(k) = k_p \cdot e(k) + y_I(k) \\ y_I(k) = k_i \cdot T \cdot e(k) + y_I(k-1) \end{cases}$$

$y_I(k)$ ← integral control

$k_i \cdot T$ ← k_i'

PI Regulator



Parallel realization of the PI algorithm

z^{-1} = unity delay

$k'_i = k_i \cdot T$ (T is the sampling period)

PI Regulator

Given the simplicity of the PI controller equations, the computation of y can be very fast. If we have a mC or DSP with MAC instruction, the algorithm may require only 3 clock cycles:

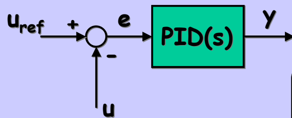
1. accumulator precharge with $y_I(k-1)$;
2. computation of $y_I(k)$, i.e. MAC $e(k), k'_I$;
3. computation of $y(k)$, i.e. MAC $e(k), k_p$;

In the end, the accumulator contains $y(k)$.

Of course, several things may go wrong in the process (overflow, quantization, …)!

PID Regulator

In the analog domain a PID regulator is described by an input-output relation like the following:



$$\frac{Y(s)}{E(s)} = k_p + \frac{k_i}{s} + s \cdot k_d$$

k_i = integral gain

k_p = proportional gain

k_d = derivative gain

PID Regulator

A purely derivative control action cannot be implemented in the analog domain (the corresponding transfer function is not proper), nevertheless it is possible to generate it numerically, for instance like this:

$$y_d(k) = k'_d \cdot [e(k) - e(k-1)], \qquad k'_d = k_d / T$$

The derivative action is very noise sensitive, actually it is a good noise amplifier.

We must use it with great care: a typical provision is to combine it with a series low pass filter.

PID Regulator

The derivative action is normally implemented according to the following algorithm:

$$y_d(k) = \frac{k_d}{T+\tau_L} \cdot [e(k) - e(k-1)] + \frac{\tau_L}{T+\tau_L} \cdot y_d(k-1)$$

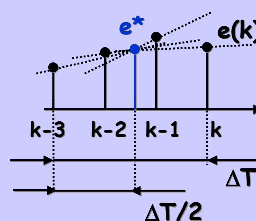
that corresponds to the following continuos time domain transfer function:

$$\frac{Y_d(s)}{E(s)} = \frac{k_d \cdot s}{1 + s \cdot \tau_L}$$

Low pass filter: limits the derivative action at high frequencies.

PID Regulator

As an alternative, we can use more complex derivative algorithms, based on the linear interpolation of several samples.



We create a virtual sample e* that is located at one half of the considered interval (4 samples, here) and whose value is the average of the considered samples.

PID Regulator

The derivative is then expressed as the average value of the incremental ratios computed among the considered samples and the virtual sample e*, that is:

$$\frac{de}{dt} \cong \frac{1}{4}\cdot\left[\frac{e(k)-e^*}{1.5T}+\frac{e(k-1)-e^*}{0.5T}-\frac{e(k-2)-e^*}{0.5T}-\frac{e(k-3)-e^*}{1.5T}\right]$$

where $e^* = \frac{1}{4}\cdot[e(k)+e(k-1)+e(k-2)+e(k-3)]$

We then find:

$$\frac{de}{dt} \cong \frac{1}{6T}\cdot[e(k)+3e(k-1)-3e(k-2)-e(k-3)]$$

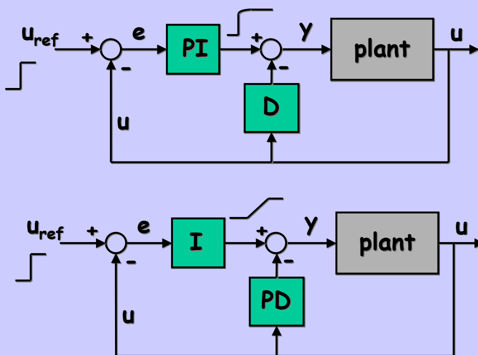
PID Regulator

It is possible to extend the average to a bigger number of samples, thus further reducing the sensitivity of the computation to noise. But, in this case, the speed of response becomes lower.

Extending the average to more than a few samples, as in our example, is often not advantageous.

Moreover, it is possible to use different configurations of the PID regulator, where the derivative action is treated differently from the proportional and integral ones.

PID Regulator



PI regulator with anti-wind-up

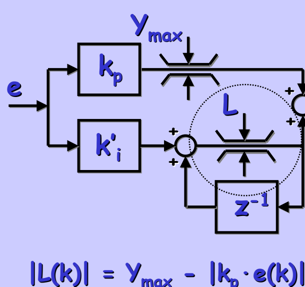
A serious problem with integral regulators is given by the integrator saturation during transients (or in the presence of other saturations in the system control loop).

The presence of a non-zero error at the integrator input for relatively long periods, unavoidably causes undesired desaturation transients, when the regulator comes back to normal operation.

This transient is often unacceptable. It can be removed, if we use a specific provision called anti-wind-up action.

PI regulator with anti-wind-up

The simplest way to operate the anti-wind-up action is the following.



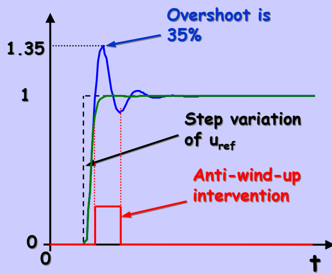
$|L(k)| = Y_{max} - |k_p \cdot e(k)|$

Each control period, we compute limit L. The current output of the integral controller is limited within ±L. This way y is always < Y_{max} in absolute value.

PI regulator with anti-wind-up

The anti-wind-up action complicates the PI algorithm quite a lot, since it needs the evaluation of L, that is of the difference between Y_{max} and the integral controller output at every control cycle. Besides, the limitation of the integral controller requires its comparison with limit L, and, depending on the result, different actions, i.e. the program will include conditional branches. Some mCs allow to reduce this complexity because their assembly include specifically designed instructions (e.g. saturated arithmetic).

PI regulator with anti-wind-up



Unity step responses of a closed loop system with 2 PI regulators. The first does not include anti-wind-up, the second instead does.

The anti-wind-up reduces the overshoots in the reference step variation responses.

Predictive Regulators

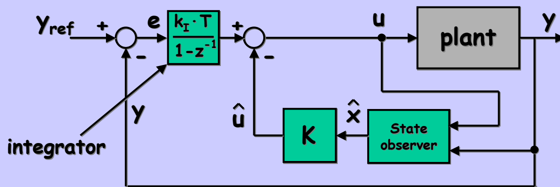
When a reliable model of the controlled plant is available, it is possible to implement predictive or dead-beat controllers.

These can only be implemented digitally, because they are based on a on-line plant model running internally to the controller.

The closed loop system dynamics (in terms of step response), in case of a perfect, ideal model (no model errors), can be made equal to those of a pure delay, of a certain minimum order.

Predictive Regulators

The general structure of a predictive regulator is the following:



The controller needs an estimation of the system state variables, that is used to close the control loop.

Predictive Regulators

Even if it is a very powerful control tool, predictive control is rarely used in industrial applications.

This is due to its relatively high complexity and to the consequent difficulty in the design of the controller parameters.

Moreover, reliable plant models are not always available (in this case, system identification is required).

Lastly, this type of controllers are relatively noise sensitive. Great care must be taken in signal conditioning.

Seminar 4

Summary

- Timing
- Data acquisition
- Scale factors
- Normalization
- Quantization effects
- Rounding coefficients
- Finite precision arithmetic
- Limit cycles

Seminar 4

Some references

1. D. Glover, J.R. Deller, "Digital Signal Processing and the Microcontroller", Prentice Hall, 1999.
2. A.V. Oppenheim, R.W. Schafer, J.R. Buck, "Discrete Time Signal Processing", Second Edition, Prentice Hall.
3. K. Ogata, "Discrete Time Control Systems", Prentice Hall, 1987.

Digital Controller Organization

When we use a mC or DSP to implement a digital controller, or just a simple digital filter, we always have to face some typical practical problems, that are not related to the design of the controller and/or filter, which takes place considering ideal conditions. At the moment of implementation though, processor characteristics (e.g. its arithmetic, word lenght) and peripheral unit features (e.g. those of the A/D converter) play a fundamental role, severely constraining the practical implementation.

Digital Controller Organization

Schematically, the various problems we may experience fall into one, or more (!), of the following categories:

1. algorithm timing;
2. data acquisition;
3. choice of scale factors;
4. quantization effects;
5. finite precision arithmetic effects;
6. occurrence of limit cycles.

Timing

The program timing, that is the distribution along time of its operations, in a closed loop digital controller or in a digital filtering algorithm is often a critical factor.

For example, the control system or digital filter performance are strongly dependent on data sampling frequency.

The minimum requirement for this parameter is just to be kept rigorously constant, to any value we may decide to choose.

Timing

If a data acquisition peripheral unit (A/D converter) is available on chip, keeping the sampling frequency constant is quite easy.

The problem of choosing the sampling frequency value is quite complicated, in general, since it directly affects the filter and controller design (the coefficients are frequency dependent).

In the standard practice, the sampling frequency is set at the highest possible value allowed by the A/D converter, but this is not always the correct choice!

Timing

All mCs and DSPs for real time embedded control application, always include an A/D converter on chip.

The use of this peripheral unit constrains the implementation of filters and controllers.

The main limitations are related to the speed of conversion and to the converter word lenght.

The typical conversion speeds are in the order of 1 μs, while the word lenghts are typically 8, 10 or even 12 bit.

Timing

The speed of conversion poses an upper limit to the sampling frequency (the lower limit is given by the system dynamics or by the band width required to the filter).

The limited word lenght causes distortion of input data, due to data value quantization.

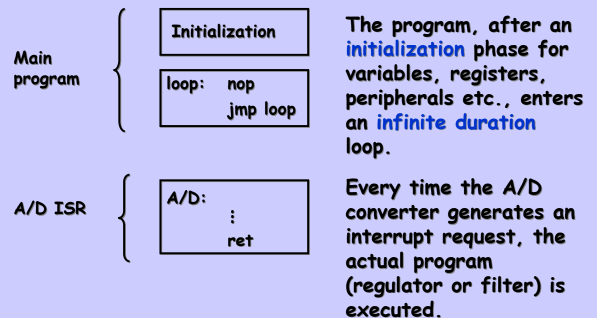
This, in turn, causes several problems that we will discuss in the following.

Timing

The availability of on-chip A/D converters makes it quite simple to guarantee that the controller, or the filter, operate at the desired frequency and that this is kept perfectly constant.

The more frequently adopted method consists in locating the control or filter algorithm within the ISR associated to the A/D converter. This can be configured to acquire data at a fixed frequency, related to that of the processor, and to generate an interrupt at the end of conversion.

Timing



The program, after an initialization phase for variables, registers, peripherals etc., enters an infinite duration loop.

Every time the A/D converter generates an interrupt request, the actual program (regulator or filter) is executed.

Timing

In this way, the only timing requirement is that the ISR duration, including latency times and other possible higher priority active interrupt sources, is in any case lower than the sampling period.

Otherwise, some data could get lost and, moreover, the control or filter algorithm could be operated at a variable frequency.

As a rule of thumb, it is better to associate the A/D converter, any time this is possible, to the highest priority interrupt.

Timing

It is not always easy to exactly estimate the duration of a given program segment with pencil and paper calculations, e.g. counting the instructions.

The safest method is to use an output pin of one of the I/O processor ports, to signal the start (asserting the bit) and the end (de-asserting it) of the program segment under test.

With the help of an oscilloscope, this allows to precisely measure the duration of the program and to detect its occasional variations (jitter or glitch phenomena).

Data acquisition

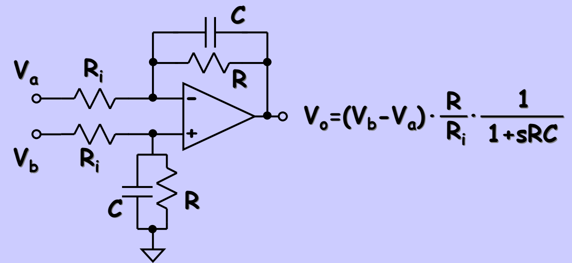
Data acquisition from the plant is always implemented interfacing the A/D of the mC or DSP with a transducer. This requires, at least usually, the implementation of suitable circuits for signal conditioning that allow us to:

a. adapt the transduced signal amplitude, so as to make it compatible with the input specifications of the A/D converter;

b. introduce filters to eliminate or reduce noise and to avoid undesired aliasing effects.

Data acquisition

Both requirements can be satisfied by simple circuits based on operational amplifiers. For example:



Data acquisition

The circuit allows to:

a. acquire a signal differentially;

b. acquire a single ended signal;

bb. summing to the signal an offset (useful for bipolar signals);

c. amplify or attenuate the signal;

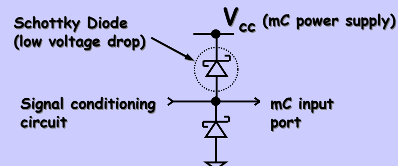
d. filter the signal with a low pass filter;

It may be used to interface a generic transducer to an A/D converter. Its low output resistance allows us to reduce to the minimum any possible loading effect.

Data acquisition

The signal conditioning circuits are always completed by suitable protection structures that keep, in any condition, the amplitude of input signals within acceptable limits.

For example:



Scale factors

The choice of scale factors for the input data of a digital controller (or digital filter) is a problem to be considered with great attention. The problem only exists for fixed point processors.

A wrong choice can determine the bad exploitment of the processor arithmetic (which can in turn cause undesidered effects related to quantization) or even the occurrence of overflow or saturation phenomena (which normally fatally affect the program).

Scale factors

This problem is particularly serious in the case of controllers, because scale factors influence the gain of the control loop.

Moreover, in the case of controllers, a bad use of the arithmetic may even cause the instability of the system, even if the design of the controller has been done correctly.

A typical case takes place when the regulator coefficients (e.g. those of a PI controller) have very different values (e.g. $k_P >> k'_I$). In this case, it is essential to make a proper use of scale factors.

Scale factors

A *correct approach* to the problem starts from the signal conditioning circuit design.

This has to be done so as to completly exploit the word lenght of the A/D converter i.e. making the maximum expected value of the signal (including a suitable safety margin, as needed to reveal possible malfunctions or out of range operating conditions) to be equal to the full scale range of the A/D converter.

In this phase, it is necessary to take into account the possible need for *reconstruction* of data sign.

Scale factors

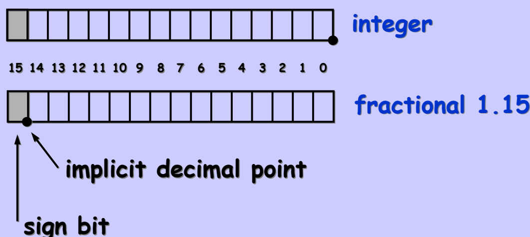
Normally, the processor word lenght is longer than that of the A/D converter. This problem, called data alignment, can be treated in two different ways:

a. right side alignment, i.e. the two LSBs are aligned;

b. left side alignment, i.e. the two MSBs are aligned (paying attention to sign!);

Each one determines a different normalization criterion for data, that may be integer (case a) or fractional (case b) type.

Normalization



Possible formats for numeric representation of data in fixed point processors (16 bit case). Negatives are represented with 2's complement.

Normalization

If we choose to adopt an integer normalization format, we implicitly set the decimal point to the right of the LSB. In practice, with this choice, any algorithm will operate only on signed integer numbers.

This mode of operation is quite easy to manage, but may complicate the scale factor manipulation: often regulator and filter coefficients are fractional quantities. To be normalized to integers, these need to be multiplied by suitable scale factors. This operation needs to be done with great care.

Normalization

Let's suppose we want to calculate:

$y = a \cdot x + b$, with $a = 0.12$, $b = 60$, $x = 1350$.

We know the final result is $y = 222$.

If we operate with integer normalization, we cannot use the true *a* value in the computation, because this is < 1, that is the minimum number we can actually represent.

We thus have to pre-multiply the *a* value by a constant that can make it an integer number (with some approximation). For instance, we may want to choose 1024.

Normalization

We get $a' = a \cdot 1024 \cong 123$. Now, executing the required calculation we find:

$a' \cdot x = y' = 166050$.

Of course, we cannot sum b to the result, because the two number are not represented on the same scale (the value of any bit in b is 1024 times lower than that of the same bit in y'). We thus need to divide y' by 1024 (that, very suitably, is a power of 2) to get the new $y'' = 166050/1024 \cong 162$ value.

In the end we get $y = 162 + b = 222$.

Normalization

Referring to this procedure, we may notice some interesting points.

Each multiplication by *a'* must be followed by a arithmetic shift to the right by 10 bits, before we can execute the sum with b. But if also *b* were the result of another product, taken with a different scale factor, before executing the sum it would be required to equalize the weight of the bits of each operand. Any time we perform an addition it is necessary to check if the operands are represented on the same scale!

Normalization

Another little problem, maybe less evident, is that we introduce a rounding error every time we substitute the exact coefficient with its scaled version.

This may have quite significant effects on the filter or regulator behaviour, because it determines a change in the allocation of its poles. In some cases, we may even get unstable behaviors.

A good provision against this, is to use the biggest possible scale factor, paying attention not to cause overflow problems.

Normalization

If we choose to adopt the fractional normalization, we implicitly set the decimal point on the right of the MSB (i.e. the sign bit). Any number must then be interpreted as a fraction, being 1 the maximum positive integer we can represent internally, for example by $7FFF_{16}$ in case of 16 bit arithmetic. This normalization is indicated as 1.15 or Q15 and is very often used.

Dealing with fractions is no longer a problem, but, of course, managing integer numbers > 1 (or <-1), will be a little more complicated.

Normalization

When using 1.15 normalization, the product always needs to be adjusted with one bit left shift, because the format of the result otherwise would be 2.30. Some processors, both DSPs and mCs, do the needed left shift automatically, quite transparently for the programmer. Sometimes a bit can be set in the processor status register to decide if the multiply has to be done as an integer (without left shift) or as a fractional one (with left shift). After the shift, the result is in the 1.31 format, that is easy to treat.

Normalization

The post normalization of fractional multiplies is performed directly by taking the most significative part of the result (bits from 16 to 31).

Attention: if we are dealing with integer multiplies, we just cannot procede the same way. The result has to be shifted right so as to set the correct weight of each bit.

Of course, also 1.15 format can cause some little problems when we need to deal with operands whose absolute value is bigger than unity.

Normalization

Let's suppose we need to compute:

$y = a \cdot x + b$, with $a = 3.5$, $b = 0.2$, $x = 0.2$.

We know the correct result is $y = 0.9$.

If we adopt the 1.15 normalization, we cannot use the true value of *a* in the computation, because this is > 1, which is the maximum positive number we can represent.

We then need to pre-divide the *a* value by some constant to make it fractional. We may want to choose 4, which is, conveniently, a power of 2.

Normalization

We find a' = a/4 = 0.875. Computation yelds:

a'·x = y' = 0.175

Now we cannot sum b, because the two numbers are not represented on the same scale (the value of a bit in b is 4 times higher than in y').

We therefore need to multiply y' by 4 to find y'' = 0.175·4 = 0.7.

As a consequence, we get y = 0.7 + b = 0.9.

Normalization

In the last example, there do not seem to be rounding errors. This is only because, for the sake of simplicity, all numbers were represented in base 10.

To pass to the binary representation, each positive fractional number has to be multiplied by 32768, rounded to the closest integer and turned into a binary number.

So, we always have some rounding error. This error is, at most, equal to 1/65536, and is often negligible, unless we are dealing with very small numbers.

Normalization

The 1.15 format is used a lot, since it simplifies the management of multiplies and, even without taking particular care, it generates small rounding errors.

On the other hand, a programmer may organize his or her program around any possible normalization (e.g. 5.11).

Thus, it's extremely important that the choice of the normalization strategy is explicitly declared in the program documentation, so as to make the alignment operations fully understandable to any experienced reader.

Overflow

Once we have chosen the best normalization strategy and adjusted the filter and/or regulator coefficients accordingly, we need to take into account the overflow problem.

It is essential that we make sure overflow never occurs in a routine implementing a filter or regulator, because it makes the computation totally useless.

To avoid overflow in multiplies, the simplest solution is to always use 1.15 fractional normalization.

Overflow

Variables that are involved in ripetitive sums, as, for instance, the integral part of a PI regulator, are often implemented in double word lenght.

The accumulator, that, especially in MAC units, has often double size with respect to the word processor, is then saved in memory by using two successive locations.

When it is summed to a single word variable, only one of the two parts is used (the most significative, if 1.15 format is being used).

Overflow

In some processors, the arithmetic unit can be operated in a saturated arithmetic mode. In case of overflow, the wrong result of the operation is replaced by the maximum (or the minimum, if negative) representable integer (tipically $7FFF_{16}$ or 8000_{16}).

This is equivalent to the introduction, after every sum, of a non linear saturation function that keeps the result within the acceptable range. The effects of such non-linearities in a complex algorithm are not easily predictable. We often have to rely on simulations.

Quantization

In a fixed point processor, quantization or rounding effects are often encountered.

Quantization and/or rounding errors are generated, for example, when:

a. an A/D conversion is operated;
b. filter and or regulator coefficients are calculated;
c. multiplies are executed and the result is trucated or rounded;
d. a D/A conversion is operated.

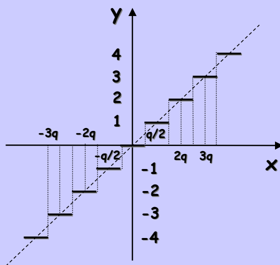
Quantization

Quantization phenomena are non-linear effects whose impact over any algorithm is, again, very difficult to predict with pencil and paper.

We do have some nice analitical tools, but they are quite difficult to apply in real cases. Once again, we have to rely on simulation tools.

Things get particulalry complicated when it comes to closed loop control systems, where the impact of quantization or rounding errors on the system stability can be very serious.

Quantization



Trans-characteristic of a “linear” quantizer with rounding.

Each value x such that: $(N-1/2)q < x < (N+1/2)q$ is associated to number N.

Parameter q is named quantization step.

The quantizer step q represents the minimum variation of x that will surely determine a variation of y by at least one bit.

Input quantization

Input quantization, e.g. due to the A/D converter, can severely affect any algorithm, since the A/D converter word is much shorter than the processor's one. The weight of q in terms of internal arithmetic is thus high.

A typical effect in closed loop regulators is the generation of a steady state error (for constant reference) that can be as high as q without being detected.

That's why it is advisable to make q smaller than the maximum acceptable error, which gives a good selection criterion for the ADC.

Output Quantization

Output quantization is mainly due to the process converting the output of the algorithm into a signal (DAC) or into the command for a transducer. Also this can cause steady state error, in a way similar to input quantization.

Moreover, the occurrence of a particular type of instability, typical of non-linear discrete time processes is also possible. This is called limit cycle.

To avoid this, we need to make sure that the resolution of the transducer or of the D/A converter is sufficiently high.

Coefficient Quantization

Also the rounding of coefficients required by the finite word lenght of the processor is, in fact, a kind of quantization error.

The main effect of this, is the so called pole shifting phenomenon, that affects the filters or regulators we are using in our algorithm.

A simple example can be given considering the following IIR low-pass filter:

$$y(k) = b \cdot x(k) + (1-b) \cdot y(k-1)$$

where we assume that $b = 0.01$.

Coefficient Quantization

Operating with e.g. 8 bit, C2, 1.7, the value of b we can internally represent is:

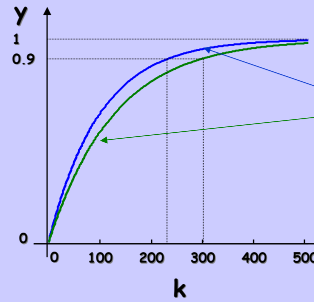
b_q = INT($b \cdot 2^7$) = 1 = 00000001_2.

We thus get a very big error with respect to the desired value. It is just as if we were using the value 2^{-7} = 0.0078 instead of 0.01.

As a consequence, the rounded value for 1-b is 01111110_2 that is equal to 0.9922.

We can now compare the two responses: that of the real filter, with quantized coefficients, and that of the ideal one, with exact coefficients.

Coefficient Quantization



Unity step response of the same filter with exact and rounded coefficients.

We can see that rise times (to 90% of the final value) change from about 220 sampling periods to about 300, with more than 36% increase!!

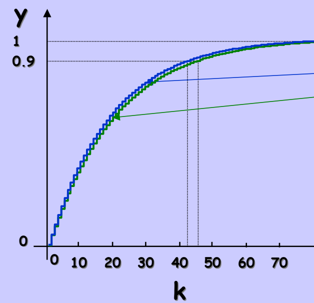
Effect of coefficient quantization

Coefficient Quantization

The problem is actually due to the fact that we are designing a filter with a relatively slow response with respect to the sampling period (rising time is higher than 100 cycles).

It is interesting to know that the performance can be improved by reducing the sampling frequency, because this will give us less rounding sensitive coefficient values. Supposing we can reduce the frequency to 1/5 of its original value, we will now design the filter for a rising time close to 40 sampling periods ($\Rightarrow$ b = 0.05, b_q = 0.047).

Coefficient Quantization



Unity step response of the same filter with exact and rounded coefficients and reduced sampling frequency.

As we see, rise times (to 90% of the final value) are now changing only from 42 sampling periods to about 46, with an error around 10% (almost ¼ of the previous case).

Coefficient Quantization

Differently from what we may expect, an increase of the sampling frequency does not always improve the performance of a digital filter or regulator, because it generally makes the effects of coefficient quantization more critical.

So, it is important that we choose the sampling frequency coherently with the processor word lenght.

High sampling frequencies require adequately long processor word lenghts.

Finite precision arithmetic

We have seen several times that the multiplication of two n bit numbers gives us a 2n bit result. We thus have the problem of making this result compatible with the other n bit data.

If the adopted normalization is 1.15, this is easily achieved by keeping only the most significative part of the result.

This operation corresponds to truncation (or rounding) of the result, that is, once again, to a quantization operation (internal to the algorithm itself).

Finite precision arithmetic

The effect of this last type of quantization upon the filter or regulator performance is the most difficult to analyse.

Indeed each multiply instruction within the algorithm implies quantization by truncation (or rounding), so that the number of non-linearities to take into account gets rapidly too large (we may think of a FIR filter, for example).

The only possible way to predict effects is to use simulation tools (e.g. the Fixed Point Blockset of Simulink).

Finite precision arithmetic

As can be expected, the effect of finite precision arithmetic is more severe upon closed loop controllers, where it may induce oscillations of the LSBs of the result, also known as limit cycles.

FIR filters, as a consequence, are immune to this kind of problems, since they do not use internal feedback (non-recursive filters).

Of course, if the FIR filter is used as a regulator inside a closed loop, the occurrence of limit cycles may still take place.

Limit Cycles

We can apply a theoretical approach to try and estimate frequency and amplitude of the limit cycles due to finite precision arithmetic (Tsypkin method or else the descriptive function method).

This is only viable for extremely simple cases, thus the best way to verify the occurrence of limit cycles, to evaluate their amplitude and frequency is by simulation.

Once the limit cycle is identified, we need to understand what is the main quantization effect behind it.

Limit Cycles

Nowadays, the use of 8 bit mCs for control applications or signal processing is quite rare. It is quite common, instead, to see 16 bit processor applications (mC o DSP).

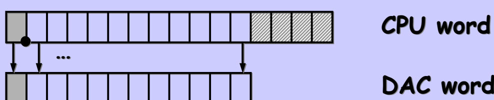
Because of that, the occurrence of limit cycles due to the processor arithmetic by itself is quite rare and can be normally taken care of with a suitable choice of scale factors.

Indeed, even if the limit cycle does not disappear, it is often possible to confine it to the lowest bits of the processor word.

Limit Cycles

The filter or controller output normally needs a lower number of bits with respect to the processor word.

A correct alignment of the two words can allow to remove the effects of a possible limit cycle.



Limit cycle *corrupted* bits are discarded.

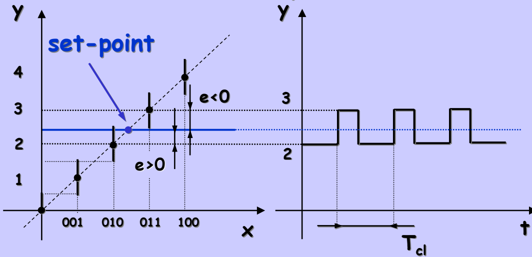
Limit Cycles

Even with an ideal algorithm, the finite resolution of the output quantizer can make it impossible for the system to get to the steady state.

If we do not have any integrator in the process (plant + controller), the effect of this can be a zero frequency limit cycle, that is a permanent error of the output with respect to the set-point.

If we do have an integrator, then we may experience the occurrence of persistent oscillations.

Limit Cycles



The system **does not reach** equilibrium and **oscillates** between the **two closest** output values, because of the integral action included in the plant or the regulator.

Limit Cycles

The **frequency** of the limit cycle, and also the relative duration of the two states, depend on the system **dynamics.**

The situation we just discussed is typical of systems controlled by **digital PWM.** In these modulators, the duty-cycle values are quantized. Every time the equilibrium duty-cycle value **does not coincide** with one of the possible values, we may experience an **oscillating limit cycle.** For this reason, it is very important to count on a high enough **resolution** for duty-cycle (8 bit at least).

Seminário 5

Uso do conversor A/D do μC LP2129 para a realização de um filtro numérico.

Sumário

- Especificações do filtro.
- Projeto do filtro.
- Configuração do conversor A/D.
- Configuração das interrupções.
- Desenvolvimento e teste do código no simulador.

Referências

1. Datasheet do microcontrolador Philips LPC2129.
2. Manual de uso do microcontrolador Philips LPC2129.
3. Simulador do processador ARM7.

Filtro digital

Pede-se projetar e realizar com o μC LP2129 um filtro digital do tipo passa-baixas com as seguintes especificações:

1. banda passante de 5 kHz;
2. atenuação de 20 ± 0.5 dB a 50 kHz;
3. ganho unitário na banda passante;
4. sinal de entrada proveniente de um transdutor com *range* de +/- 5V;
5. *clock* do processador de 12 MHz.

O código deve ser escrito em linguagem assembly e adequadamente comentado.

Projeto do filtro digital

Da análise das especificações fica claro que é suficiente um filtro IIR de primeira ordem.

Vimos que uma resposta equivalente se pode também realizar no modo FIR.

Porém a quantidade de cálculos requerida para uma realização FIR é sempre superior àquela necessária para um filtro IIR.

Por outro lado, o filtro FIR não apresenta problemas de estabilidade numérica, que podem se verificar com o filtro IIR, por causa do feedback interno.

Projeto do filtro digital

A realização do tipo FIR poderia ser adotada caso se estivesse trabalhando com um DSP, ou ainda se houvessem especificações sobre o defasamento da resposta em freqüência do filtro (os filtros FIR podem ter fase linear).

O uso de um μC, ainda que muito potente como aquele indicado, e o fato que não existem especificações sobre o defasamento, levam a considerar a realização do tipo IIR.

Trata-se então de realizar o algoritmo:

$$y(k) = b \cdot x(k) + a \cdot y(k-1)$$

Aquisição dos dados

Uma vez decidido o algoritmo, é necessário escolher como organizar a aquisição dos dados. O μC possui um conversor A/D interno, o que simplifica muito as coisas.

Como todos os periféricos do LP2129, também este é mapeado na memória.

É preciso aprender como programá-la e decidir uma estratégia de gestão (*polling* vs interrupção).

É importante recordar que para as aplicações de processamento de sinais a temporização é essencial.

Aquisição dos dados

É necessário que as amostras sejam adquiridas com uma frequência constante e conhecida.

Além disso, é necessário que o processamento do dado adquirido se conclua dentro do período de amostragem pré-fixado.

Estes fatos implicam em vínculos para a escolha dos parâmetros relativos ao conversor A/D.

Em outras palavras, a escolha da frequência de amostragem não dependerá apenas das especificações do filtro, mas também da qualidade (número de instruções, i.e. tempo de cálculo) do algoritmo.

Aquisição dos dados

Do manual do LP2129 tem-se que o ADC é controlado através de dois registros (ADCR e ADDR):

Name	Description	Access	Reset Value	Address
ADCR	A/D Control Register. The ADCR register must be written to select the operating mode before A/D conversion can occur.	Read/Write	0x0000 0001	0xE003 4000
ADDR	A/D Data Register. This register contains the ADC's DONE bit and (when DONE is 1) the 10-bit result of the conversion.	Read/Write	NA	0xE003 4004

O primeiro contém o conjunto de parâmetros de configuração (modo de funcionamento, frequência, …), o segundo, o dado convertido em 10 bits ADDR[15:6] e o bit (ADDR[31]) de final de conversão.

Aquisição dos dados

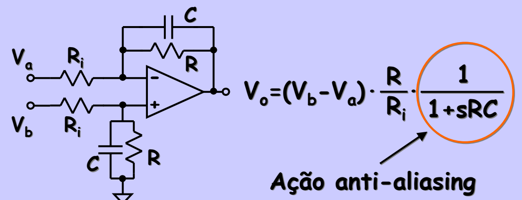
A tensão de alimentação do ADC, que representa o fundo de escala para os dados de entrada, é de 3.3V.

Devemos, porém, adquirir dados cujos valores podem estar entre –5 e +5 V.

Temos, necessidade, portanto, de um circuito de escalonamento do sinal de entrada, que o desloque e atenue até torná-lo compatível com o ADC.

O resultado da conversão representa a razão entre a tensão de entrada e o fundo de escala de 3.3V.

Aquisição dos dados



Fazendo V_a = -5 V e escolhendo a razão R/R_i igual a 3.3/10, obtemos o escalonamento necessário. Importante: a saída do OPAMP deve estar fisicamente muito próxima à entrada do ADC.

Aquisição dos dados

Após o circuito de aquisição dispomos de um sinal entre 0 e 3.3V.

Os valores menores que 1.65V serão correspondentes a tensões negativas no transdutor. Aquelas maiores, correspondem a tensões positivas.

Sendo importante manter a informação sobre o sinal, devemos prever um procedimento no programa desenvolvido.

Os dados de saída do ADC terão sempre valores positivos.

Conversão A/D

Ao término da conversão, o ADC escreve o dado no registro ADDR e coloca o bit 31 em 1.

Uma eventual rotina de *polling* pode aproveitar este fato para identificar o final da conversão.

A leitura do registro, com a conversão concluída, corresponde à leitura de um número < 0 (o bit de sinal é alto).

Se a conversão ainda estiver em curso, o número lido será positivo (ou nulo). A supervisão do periférico em *polling* é, assim, imediata.

Conversão A/D

Na realidade, ao se completar a conversão, o ADC envia sempre ao VIC uma solicitação de interrupção.

Se o correspondente bit do registro VICIntEnable for 1, esta solicitação é enviada ao processador, segundo a modalidade prevista pelo programador, ou seja, segundo a configuração do próprio VIC.

Caso contrário, a solicitação é ignorada.

É pois, possível gestir o periférico também por meio das interrupções.

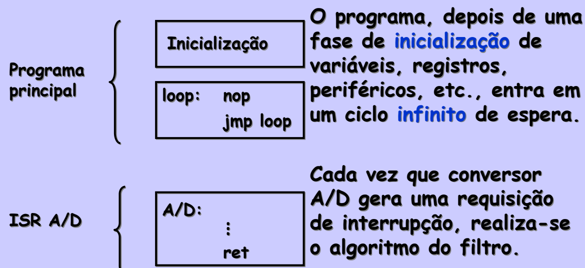
Conversão A/D

Neste exemplo, o filtro é a única atividade prevista para o µC. Esta é uma situação completamente irrealística.

Na verdade, o filtro deverá fazer parte de uma série mais ampla de atividades do µC, que deverá estar livre para realizar também outras funções.

Ainda que seja possível para nós, a gestão em *polling* não é aquela que seria adotada em uma aplicação real, na qual o uso das interrupções seria a escolha obrigatória.

Conversão A/D



O programa, depois de uma fase de inicialização de variáveis, registros, periféricos, etc., entra em um ciclo infinito de espera.

Cada vez que conversor A/D gera uma requisição de interrupção, realiza-se o algoritmo do filtro.

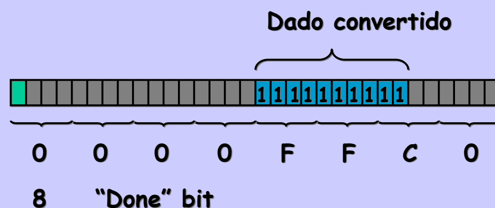
Organização do programa Filtro_PB

Conversão A/D

O dado convertido pelo ADC é escrito com 10 bits no ADDR[15:6]. O valor de fundo de escala será memorizado como:

ADDR = 0x0000FFC0

Dado convertido



8 "Done" bit

Projeto do filtro digital

O primeiro parâmetro que deve ser fixado no projeto de um filtro é a frequência de amostragem (f_C).

Usualmente esta é fixada pelas especificações, ou ainda é vinculada a outros aspectos do projeto.

No nosso caso, ao contrário, é um grau de liberdade do projeto.

Uma primeira estimativa pode se basear na banda passante do filtro. Sabemos que as técnicas de discretização fornecem bons resultados até 1/10 da frequência de amostragem.

Projeto do filtro digital

No nosso caso, estabelece-se um limite inferior da ordem de 50 kHz, valor abaixo do qual convém não descer.

O limite superior é dado pela potência de cálculo do µC. De fato, devemos ser capazes de completar os cálculos dentro de um período de amostragem.

Isto significa que devemos ser capazes de completar os cálculos e produzir um novo valor de y(k) no máximo em 20 µs.

Um outro limite superior para f_C poderia vir dos efeitos de quantização.

Projeto do filtro digital

- Four channel 10-bit A/D converter with conversion time as low as 2.44 μs.
- Multiple serial interfaces including two UARTs (16C550), Fast I²C (400 kbits/s) and two SPIs™.
- 60 MHz maximum CPU clock available from programmable on-chip Phase-Locked Loop.

Da análise do datasheet vê-se que o ADC do μC é capaz de adquirir dados com uma frequência máxima de cerca de 400 kHz.

Além disso, o período de clock do processador é igual a cerca de 83 ns, o que comporta a disponibilidade de cerca de 240 ciclos de clock no tempo máximo à disposição. Tem-se, pois, tempo para muitas instruções (mas não muitíssimas!).

Projeto do filtro digital

Esta rápida análise mostra que é possível resolver o problema dado pois tanto a velocidade de conversão do ADC quanto a frequência de trabalho da CPU são adequadas à realização do filtro.

Neste ponto podemos escrever um código hipotético e avaliar o tempo de execução.

Isto permitirá entender qual é a máxima frequência de amostragem f_C a qual podemos efetivamente pensar em atingir, sem perder dados.

Código do filtro digital

```
        LDR   r9, [r2]               /* Leio dado, endereço em r2  */
        BIC   r9,r9,#0x80000000      /* Limpo bit fim de conversão */
        LDR   r8, =PWMMR0            /* Endereço para o resultado  */
Filtro:                              /* O valor adquirido está em r9*/
        LDR   r10, =b_1              /* b_1 em r10                 */
        MUL   r7, r9, r10
        MOV   r9, r7, LSR #16        /* b_1*x(k) em r9             */
        LDR   r10, =a_1              /* a_1 em r10                 */
        MUL   r7, r5, r10            /* r5 = y(k-1)                */
        MOV   r7, r7, LSR #16        /* a_1*y(k-1) em r7           */
        ADD   r5,r9,r7               /* r5 = y(k)                  */
```

O algoritmo resulta muito simples: apenas 10 instruções.

Código do filtro digital

A análise do nosso código hipotético mostra que o número de instruções para o filtro passa-baixas é muito pequeno.

No pior caso, podemos contar 3 ciclos de *clock* para as instruções não MUL e 5 para as MUL (na realidade serão menos) para um total inferior a 35 ciclos de *clock*, ou ainda, com o nosso valor de f_{clk} (12 MHz), a 3 μs.

Se fosse apenas pelo algoritmo, a frequência f_C poderia ser aumentada até mais de 300 kHz.

Na prática, convém limitar-se a um valor menor, para não aumentar os efeitos de quantização.

Projeto do filtro digital

Uma primeira escolha de f_C pode ser:

$$f_C = 100 \text{ kHz}.$$

É suficientemente maior que a banda exigida para o filtro e ao mesmo tempo compatível com o tempo de cálculo estimado e a velocidade de conversão do ADC.

Pode-se agora encontrar os valores dos coeficientes do filtro, para obter a banda desejada. Um modo elementar de resolver o problema é através da discretização de um filtro analógico equivalente.

Projeto do filtro digital

O filtro a discretizar é:

$$F(s) = \frac{1}{1 + sT_p} \qquad T_p = 3.183 \cdot 10^{-5}$$

Com a Z-form:

$$s = \frac{1 - z^{-1}}{T_C}$$

Integração de Euler, $T_C = 1/f_C$

Encontramos:

$$a_1 = \frac{T_p}{T_p + T_C} \qquad b_1 = \frac{T_C}{T_p + T_C}$$

Projeto do filtro digital

Substituindo os valores obtemos:

a_1 = 0.760943 b_1 = 0.239057

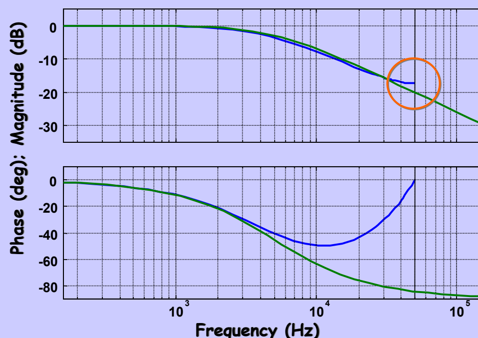
N.B: obviamente a_1 + b_1 = 1.

Devemos agora verificar se o projeto está correto.

Para tanto podemos recorrer a um programa como o Matlab, para obter a resposta em frequência e ao degrau do filtro.

Por enquanto, deixemos de lado todos os efeitos de quantização.

Projeto do filtro digital



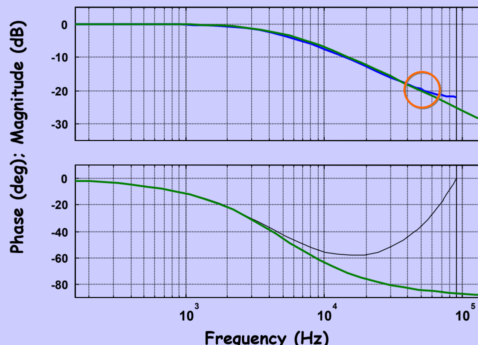
Projeto do filtro digital

A escolha de f_C leva a um erro na atenuação pedida em 50 kHz. Para reduzir este erro, mantendo a estrutura do filtro, é necessário aumentar a frequência de amostragem.

Repetimos o projeto diversas vezes aumentando sucessivamente o valor de f_C.

Para f_C = 180 kHz o erro se reduz a menos de 0.5 dB, que pode ser considerado aceitável.

Projeto do filtro digital



Projeto do filtro digital

Deve-se agora discutir a representação interna dos coeficientes do filtro.

O nosso μC dispõe de aritmética a 32 bits. No entanto, a execução de multiplicação a 32 bits requereria um acumulador de 64 bit, que não é disponível.

Portanto, a representação dos dados de entrada e dos coeficientes do filtro deve ser no máximo em 16 bits.

Pode-se aproveitar do fato que serão processados apenas números positivos.

Projeto do filtro digital

Os valores dos coeficientes (para f_C = 180 kHz) são:

a_1 = 0.851402 b_1 = 0.148598

Para representar em 16 bits sem sinal, aproveitando assim ao máximo a resolução disponível, podemos usar a relação:

1-LSB = 2^{16}-1 = 65535 = FFFF

Obtendo:

a_1_16 = 0xD9F5 b_1_16 = 0x260A

Projeto do filtro digital

Se também o conteúdo da parte baixa do registro ADDR for interpretado como dado inteiro a 16 bits (sem sinal), o valor em Volts do LSB é:

$$3.3V/2^{16} \cong 50.35\ \mu V.$$

A resolução efetiva do nosso filtro será, porém, muito mais imprecisa, por causa da quantização de entrada a 10 bits, que permite distinguir, no máximo, variações de

$$3.3V/2^{10} \cong 3.2\ mV.$$

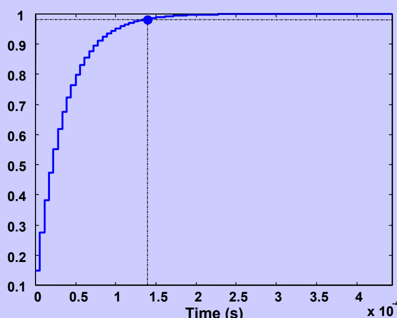
Projeto do filtro digital

Dado que usamos 16 bits para representar os valores dos coeficientes na sua representação binária, não há efeitos visíveis devido à quantização. A nossa resolução numérica é igual a $1/2^{16} = 1.53 \cdot 10^{-5}$.

Os diagramas de Bode (ideal/quantizado) resultam praticamente sobrepostos.

Qualquer efeito poderia haver devido ao truncamento implícito nas multiplicações com vírgula fixa, mas a sua avaliação pode ser feita diretamente com o sistema de desenvolvimento.

Projeto do filtro digital



Resposta ao degrau esperada para o filtro

Multiplicação

A escolha da representação dos dados tem efeito também na execução das multiplicações.

Ao final de cada multiplicação, é preciso transformar o resultado, que está em 32 bits, para 16 bits, tornando-o homogêneo com os outros dados. Isto é feito de modo a garantir que os bits tenham sempre o mesmo "peso", para salvaguardar a exatidão das eventuais somas calculadas na sequência.

No nosso caso, isto se obtém deslocando o resultado 16 bits à direita.

Multiplicação

Por exemplo: multipliquemos 0x4000 (que representa a fração 0.25, em 16 bits sem sinal) por 0xA000 (que representa a fração 0.625 sempre na mesma modalidade).

A multiplicação dos dois números (que o processador trata como inteiros com sinal mas a 32 bits) é igual a 0x28000000 (obviamente >0).

Na nossa notação o resultado esperado será:

$0.25 \cdot 0.625 = 0.1625$ ou seja 0x2800.

O valor correto, coerente com a normalização, se obtém com um deslocamento do resultado em 16 posições à direita.

Multiplicação

Em geral, este tipo de normalização é, em efeito, uma pré-multiplicação dos operandos por um fator de escala do tipo 2^p.

Ao invés de multiplicar x e y, de fato nós multiplicamos $x \cdot 2^p$ por $y \cdot 2^p$ obtendo

$$r = x \cdot y \cdot 2^{2p}.$$

Se deslocamos este número, à direita, de p bits, obtemos então

$$r' = x \cdot y \cdot 2^p,$$

que se encontra na mesma escala dos operandos iniciais.

Realização do filtro digital

Inicialmente devemos configurar o sistema de aquisição de modo que trabalhe em 10 bits e a 180 kHz. As sequências de bits:

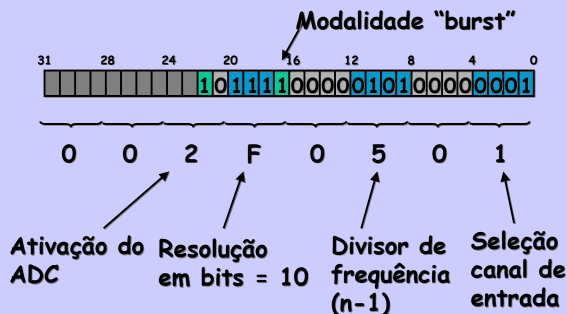
```
.equ  VPB_CFG,        0x00000001

.equ  AD_CFG,         0x002F0501
```

produzem este efeito. A primeira, uma vez escrita no registro VPBDIV, impõe uma frequência de *clock* para o bus dos periféricos (pclk) igual àquela da CPU (cclk). A segunda, uma vez copiada no registro ADCR, configura o ADC como se segue.

Realização do filtro digital

Configuração do registro ADCR:



Realização do filtro digital

Configuração do registro ADCR.

Dividindo a frequência do *clock* por 6 e lembrando que a conversão requer 11 ciclos da *clock*, tem-se que o conversor A/D, funcionando em modalidade "burst", terá os dados disponíveis para o processamento em uma frequência igual a:

$$f_C = f_{clk}/66 \cong 182 \text{ kHz}$$

que é um valor suficientemente próximo àquele de projeto.

Realização do filtro digital

Configuração do registro ADCR.

Teria sido possível fazer o conversor operar no modo "single shot", comandando o início da conversão na frequência requerida por meio do uso de um timer. A conversão deveria ser muito mais veloz, para deixar espaço ao processamento.

Isto é desvantajoso para a precisão da conversão e para o consumo de potência.

A solução adotada, além disso, utiliza menos recursos.

Realização do filtro digital

Configuração das interrupções.

É necessário sinalizar, na ocorrência de final de conversão, uma ISR, que deve ser alocada na posição correta do vetor de interrupções. Copiando a sequência de bits:

```
.equ  VICInt_CFG,     0x00040000
```

nos registros VICIntEnable e VICIntSelect obtém-se ativação da IR associada à ADC e a sua classificação como FIQ. às IR deste tipo é reservada a posição 0x0000001C

do vetor de interrupções

Realização do filtro digital

Configuração das interrupções.

Devemos alocar naquele endereço a nossa ISR ou ao menos uma instrução de salto à rotina.

```
Vectors:  LDR   PC, Reset_Addr
          LDR   PC, Undef_Addr
          LDR   PC, SWI_Addr
          LDR   PC, PAbt_Addr
          LDR   PC, DAbt_Addr
          NOP                      /* Reserved Vector */
          LDR   PC, IRQ_Addr
          LDR   PC, FIQ_Addr
```

Vetor das interrupções.
Vectors = 0x0000000,
endereço 0 da memória.

Realização do filtro digital

Configuração das interrupções.

A alocação da rotina no endereço correto é obtida no módulo de configuração do nosso programa:

```
FIQ_Addr:      .word   FIQ_Handler
FIQ_Handler:   B       Isr_AD
```

uma vez resolvido pelo assemblador, produzirão o salto à nossa rotina (ISR_AD) a cada ocorrência da Fast Interrupt reQuest.

Realização do filtro digital

Programação da ISR.

A ISR do ADC contém o verdadeiro e próprio programa. O algoritmo que realiza o filtro é aquele apresentado antes.

A este se adiciona um segmento de código que permite registrar os primeiros 512 valores da sequência de saída do filtro na memória do μC, de modo a poder, visualmente, verificar o seu correto funcionamento.

Neste ponto, o código pode ser testado e.g. Através do simulador. Sinal de teste: onda quadrada (1.65, 1200).

Seminário 6

Uso dos Timers do mC LP2129 para a realização de um "Phase Locked Loop" (PLL).

Sumário

- Descrição do problema.
- Projeto do algoritmo de controle.
- Configuração dos timers.
- Configuração das interrupções.
- Desenvolvimento e teste do programa usando simulador.

Referências

1. Datasheet do microcontrolador Philips LPC2129.
2. Manual de uso do microcontrolador Philips LPC2129.
3. Simulador do processador ARM7

Problema

O problema que devemos resolver consiste em usar o μC LP2129 para a construção de um sinal de onda quadrada cuja frequência seja 8 vezes a frequência de um sinal de referência (sempre a onda quadrada). A frequência de referência está entre 40 Hz e 60 Hz, com valor nominal de 50 Hz.

O tempo di sincronização deve ser o menor possível. Além disso, o sinal sintetizado deve permanecer em fase com o sinal de referência com o menor erro possível para o hardware considerado (f_{clk} = 12 MHz).

PLL

Um PLL permite regular a frequência e a fase de um sinal, colocando-o em uma relação conhecida e constante com um sinal de referência. A sua realização analógica é simplificada com o uso oportuno de circuitos integrados, como o CD4046.

A função do PLL é extremamente útil para diversas aplicações como multiplicador de frequência, síntese de frequências, circuitos de sincronização.

PLL

É importante ressaltar que obter a relação requerida entre as frequências não é, em si, difícil, mas a amarração de fase requer que a frequência do sinal gerado seja controlada em malha fechada.

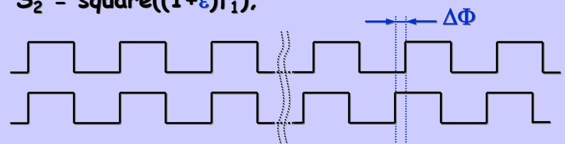
Uma realização em malha aberta, por causa das inevitáveis diferenças entre a frequência desejada e aquela efetiva, levaria a um defasamento continuamente variável entre os sinais de sincronismo e gerado. As duas formas de onda, na prática deslizam uma em relação à outra (ainda que muito lentamente).

PLL

Consideremos, por exemplo, os dois sinais:

S_1 = square(f_1);

S_2 = square($(1+\varepsilon)f_1$);



Por causa do erro ε na frequência, os dois sinais, que partem em fase, depois de um certo tempo aparecem defasados. O tempo integra a diferença entre as frequências!

PLL

A solução consiste em variar de modo controlado a frequência do sinal gerado de modo a impor o anulamento do erro de fase.

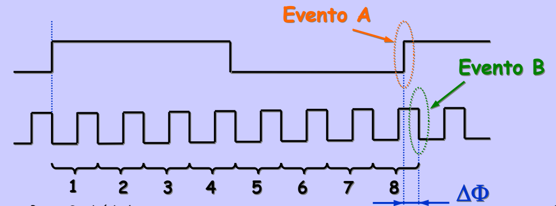
Um PLL realiza assim estas funções:

1. determinação do erro de fase;
2. regulação da frequência do sinal gerado para anular, com uma dinâmica controlada, o erro de fase.

Como todos os sistemas realimentados, também o PLL pode ser instável!

Algoritmo de controle do PLL

No nosso caso, devemos manter amarradas em fase duas ondas quadradas, uma externa (sincronismo) e uma construída por nós, com uma frequência de 8 vezes aquela original. Definamos em primeiro lugar o erro de fase:



Algoritmo de controle do PLL

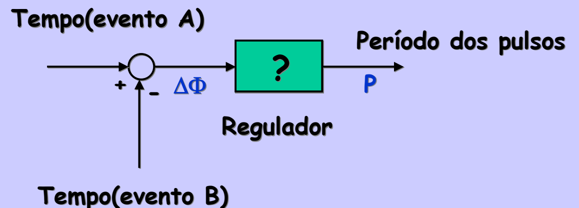
O erro de fase é o intervalo de tempo que transcorre entre os eventos A (fronte de subida do sinal de sincronismo) e B (final do oitavo impulso gerado).

Esta quantidade pode ser positiva ou negativa, porque a ordem entre os eventos não é conhecida a priori.

O objetivo do nosso sistema de controle é levar o erro de fase a zero em um certo tempo (o mais breve possível), modificando de modo oportuno o período dos impulsos gerados.

Algoritmo de controle do PLL

Configura-se assim uma estrutura de controle do tipo:



O problema é encontrar o regulador que obtenha o resultado ($\Delta\Phi = 0$) e garanta a estabilidade!

Algoritmo de controle do PLL

Primeira solução: tento compensar o atraso em um único período de controle.

$$P(k+1) = P(k) + \frac{\Delta\Phi(k)}{4}$$

O novo período é igual ao antigo, ao qual se soma uma correção. A correção é igual ao erro de fase dividido por 4. Deste modo o novo período será tal que anula o erro no passo sucessivo.

Mas está será uma situação de equilíbrio?

Algoritmo de controle do PLL

Consideremos o seguinte exemplo: suponhamos que o sinal de sincronismo tenha um período P_S igual a 8 p.u. (unidade arbitrária).

$P(k) = 0.9$

$\Delta\Phi(k) = 0.8$

$P(k+1) = 1.1$

O valor de regime deveria ser 1!

Depois de 8 períodos o erro $\Delta\Phi(k+1)$ será:

$$\Delta\Phi(k+1) = 8 + 0.8 - 1.1 \cdot 8 = 0!$$

P_S — $\Delta\Phi(k)$ — 8 períodos $P(k+1)$

Algoritmo de controle do PLL

O exemplo precedente mostra que se tentamos compensar de modo imediato o erro de fase, o sistema começará a oscilar porque o anulamento do erro implica atribuir ao período dos pulsos um valor diferente daquele de regime!

Devemos seguramente modificar a nossa estratégia de controle.

Idéia: separemos a compensação do erro de fase da construção do período P em regime, de modo que, com o erro compensado, o período possa ser aquele "certo".

Algoritmo de controle do PLL

Solução "em duas fases":

$$P_{Int}(k) = P_{Int}(k-1) + \frac{\Delta\Phi(k)}{8}$$

Aqui construo o valor de regime

$$P(k+1) = P_{Int}(k) + \frac{\Delta\Phi(k)}{8}$$

Aqui compenso o erro de fase

A variável $P_{Int}(k)$ memoriza as correções que são feitas a cada vez.

Algoritmo de controle do PLL

Consideremos de novo o exemplo precedente:

$P_{Int}(k-1) = 0.9$

$\Delta\Phi(k) = 0.8$

Segundo o novo algoritmo:

$P_{Int}(k) = 0.9 + 0.1 = 1$ ← valor de regime!

$P(k+1) = 1 + 0.1 = 1.1$

Ao final do período sucessivo teremos erro de fase nulo (porque $P(k+1)$ é 1.1 como antes), mas agora o sistema está em regime porque em $P_{Int}(k)$ está memorizado o valor correto.

Algoritmo de controle do PLL

Observemos que o nosso algoritmo final tem a estrutura de um regulador PI discretizado, no qual as constantes proporcional e integral são iguais entre si e valem 1/8.

Esta particular escolha confere, todavia, ao algoritmo características muito vantajosas: este é capaz de anular o erro de regulação em apenas um passo!

Este tipo de regulador é chamado "dead-beat". Seu comportamento deriva do posicionamento do pólo do sistema em malha fechada na origem do plano complexo.

Realização do PLL

Para implementar o algoritmo devemos estabelecer uma estratégia para a medida do erro de fase, que, como visto, é um intervalo de tempo.

Naturalmente devemos usar os timers do µC.

Um deve operar no modo "capture" (atribuindo ao seu pino de entrada o sinal de sincronismo) e outro no modo "compare" (ou match), com a tarefa de medir o período dos pulsos e de gerar em seu pino de saída.

Realização do PLL

O problema seguinte é quando avaliar o erro de fase. Existem pelo menos duas possibilidades:

1. quando ocorre o evento A;
2. quando ocorre o evento B.

A escolha correta é a segunda, porque permite modificar o período dos impulsos sempre no início de uma nova sequência de 8, como previsto pelo algoritmo.

Se agíssemos no evento A, poderíamos modificar o período de uma sequência já iniciada, coisa que o algoritmo não prevê!

Realização do PLL

Tal escolha, ainda que correta, coloca outros dois problemas de ordem prática (categoria: temporização):

1. tempo de cálculo;
2. cálculo do erro de fase quando o evento A segue B.

O tempo para realizar os cálculos é limitado. Idealmente, devemos ser capazes de atualizar o período dos impulsos antes que se complete o primeiro semiperíodo do primeiro impulso da nova sequência.

Realização do PLL

Em regime este tempo é:

$P_{Smin}/16 = 1/(60 \cdot 16) = 1.04$ ms

Tendo em conta a reduzida complexidade do algoritmo de controle, estamos em condições de garantir o funcionamento em regime.

No transitório porém é possível que, para anular o erro de fase, o controle imponha uma frequência dos impulsos ainda maior que a máxima esperada.

Os tempos do algoritmo devem ser avaliados no caso mais crítico.

Realização do PLL

Consideremos o caso em que a frequência da onda quadrada de sincronismo passe de 40 Hz a 60 Hz em degrau, imediatamente depois que o período dos impulsos foi calculado.

Suponhamos que o sistema estivesse em regime e portanto parta de um erro de fase nulo.

Ao final do novo período da onda quadrada o erro de fase será:

$1/60 - 1/40 = -8.333$ ms

O controle reduzirá o período dos impulsos de modo a compensar o atraso de fase.

Realização do PLL

O novo período dos impulsos será:

$P(k+1) = 1/320 - 8.333 \cdot 10^{-3}/4 = 1.041$ ms

que corresponde a uma frequência de aproximadamente 960 Hz.

O cálculo deve ser concluído em um tempo máximo de $P(k+1)/2 = 0.52$ ms.

Trata-se de um tempo compatível com a velocidade do nosso μC, como se pode verificar avaliando a parte do programa que corresponde ao algoritmo de controle.

Realização do PLL

```
PLL_routine:
    MOV    r2, #0              /* reset do contador de ciclos         */
    LDR    r0, =CCR0           /* carrega endereço capture register   */
    LDR    r8, [r0]            /* lê valor CCR0                       */
    LDR    r0, =T0TC           /* carrega endereço TC0                */
    LDR    r9, [r0]            /* lê valor TC0                        */
    SUB    r10, r8, r9         /* erro de sincronismo                 */
    LDR    r0, =PERIODO_0      /* uso o periodo medido                */
    LDR    r6, [r0]            /* para evitar saturação               */
    MVN    r6, r6, ASR #1      /* - semiperiodo in r6                 */
    CMP    r10, r6             /* r6 contém o semiperiodo (negado)    */
    SUBLE  r10, r10, r6, LSL #1 /* corrige medida erro                */
    MOV    r10, r10, ASR #4    /* divido por 16                       */
    ADD    r4, r4, r10         /* integral das correções              */
```


Realização do PLL

As instruções que acessam a memória são 6, às quais se somam 3 instruções de deslocamento de dados entre registradores e 4 instruções aritméticas.

No caso pior, são realizadas um número de ciclos igual a:

$N = 6 \cdot 3 + 3 \cdot 2 + 4 \cdot 1 = 28$

que corresponde a um tempo estimado de execução de 2.33 μs.

Tem-se uma margem muito ampla no tempo de execução. Pode-se garantir o funcionamento até a frequências bem maiores!

Realização do PLL

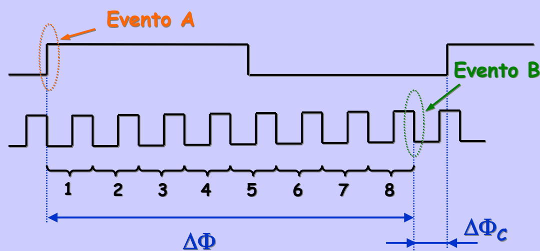
O segundo problema da realização da medida do erro de fase deve-se ao fato que nunca seremos capazes de obter valores > 0.

No caso de avanço de fase, o evento B precede o evento A. No momento dos cálculos, o tempo no evento A não pode, portanto, ser conhecido.

Na prática, o algoritmo utilizará o antigo valor do tempo do evento A.

Devemos, então, reconstruir o valor correto através de uma correção da medida.

Realização do PLL



O valor esperado para o erro de fase pode ser reconstruído somando ao erro encontrado (<0) o valor do período da onda de sincronismo.

Realização do PLL

Para garantir o correto funcionamento da correção, é necessário que:

1. esta venha aplicada apenas quando o erro de fase supera metade do período do sinal de sincronismo;
2. o período do sinal de sincronismo seja conhecido.

A segunda condição, dado que o sinal de sincronismo pode ter frequência variável, impõe medir o período, coisa que pode ser feita usando a interrupção associada à condição de "capture".

Realização do PLL

O comportamento transitório do regulador se ressentirá do fato que as variações do período de sincronismo podem ser obtidas apenas com um atraso de um período.

De fato, até que o período não seja completo não podemos medir-lhe a duração.

As eventuais correções dos erros de fase realizadas com valores de período não atualizados serão, portanto, erradas.

No pior caso, pode-se esperar um atraso de amarração de 2 sequências de 8 impulsos.

Problema

Consideremos a seguinte situação: com o PLL em regime, a frequência do sinal de sincronismo é reduzida em "degrau".

Segue-se um erro de fase negativo, porque o trem de 8 pulsos se conclui antes do evento A correspondente.

O mecanismo de correção descrito reconstrói o erro de fase, mas o algoritmo não produz qualquer correção do período dos pulsos.

A que se deve este comportamento?

Problema

No momento da atualização do período dos pulsos, o valor memorizado para o período do sinal de sincronismo ainda é aquele antigo!

Assim a reconstrução do erro produz um erro de fase final praticamente igual a 0, o que não leva a qualquer correção do período dos pulsos.

Apenas sucessivamente o período do sinal de sincronismo será atualizado, e então apenas no ciclo sucessivo se poderá ter qualquer efeito.

Problema

Felizmente, no entanto, o erro de fase resultante ao sucessivo instante de controle, será compensado como se fosse obtido todo no ciclo de controle imediatamente precedente.

Isto, no entanto, é errado, porque na realidade isto deriva de dois ciclos de controle nos quais a compensação não ocorreu.

A correção que será realizada será portanto errada. Isto gerará um transitório que pedirá ainda outros ciclos para se estinguir.

Isto reduz significativamente a velocidade de resposta.

Solução

A solução consiste em identificar os aumentos de período do sinal de sincronismo e, caso ocorra, modificar a lei de controle de modo a evitar o surgimento do transitório.

É suficiente somar ao termo integral uma quantidade igual ao incremento do período medido dividida por 8. Além disso, no momento da correção seguinte, a atualização da parte integral não deve ser feita.

Isto permite levar o sistema ao regime com um atraso máximo igual a três sequências de 8 pulsos.

Realização do PLL

Assim estruturado, o PLL garante o menor tempo de amarração possível (3 ciclos) e a faixa de captura desejada (de 40 Hz a 60 Hz).

A máxima resolução do erro de fase pode ser diminuída até o limite de um período de contagem do timer utilizado como "base dos tempos", isto é, aquele em modalidade capture. Levando este valor a T_{clk}, teremos também a máxima precisão possível com o nosso μC (83.3 ns).

Trata-se agora de configurar os timers no modo adequado.

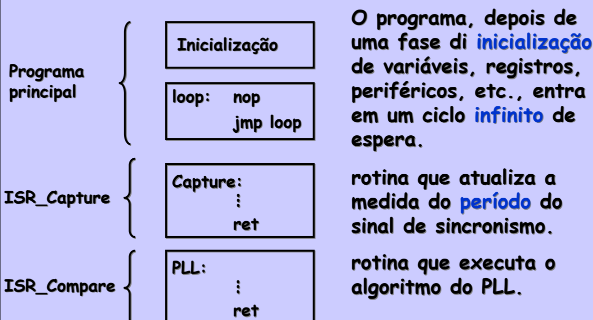
Realização do PLL

A organização mais conveniente para a aplicação requer o uso de interrupções.

O primeiro timer (TIMER0) pode servir de base de tempo, operar em modalidade capture e medir o período do sinal de sincronismo. A sua atividade pode ter menor prioridade em relação àquela do TIMER1.

Este é usado na modalidade match (compare) de modo a produzir os pulsos. A sua ISR deve ter a máxima prioridade. Cada oitava IR devemos executar o algoritmo do PLL, de modo a atualizar o período.

Diagrama de blocos do PLL



O programa, depois de uma fase di inicialização de variáveis, registros, periféricos, etc., entra em um ciclo infinito de espera.

rotina que atualiza a medida do período do sinal de sincronismo.

rotina que executa o algoritmo do PLL.

Organização do programa PLL

Realização do PLL

Neste ponto deve-se verificar para cada ISR a condição de intervalo.

Para este exemplo, o tempo disponível é muito maior que a duração de cada ISR. O problema não é crítico.

Assim, no pior caso, a ISR com máxima prioridade poderá ocasionalmente ser atrasada por aquela associada ao TIMER0, mas como esta é muito breve, o atraso é desprezível. Não há violação da condição de intervalo.

Configuração das interrupções

Escrevendo no registro VICIntEnable a sequência de bit:

```
.equ  VICInt_CFG,          0x00000030
```

Habilitam-se os dois timer a produzir uma IR quando se verificar o evento de capture e a condição de match respectivamente.

Escrevendo no registro VICIntSelect a sequência:

```
.equ  VICSel_CFG,          0x00000020
```

Qualifica-se TIMER0 como IRQ e TIMER1 como FIQ.

Configuração dos timer

Ocorre como em exemplos já tratados (geração de onda quadrada, medida de período).

Dadas as especificações é oportuno inicialmente fixar pclk=cclk atuando sobre o controlador do bus dos periféricos.

TIMER0 é configurado para identificar bordas positivas no pino C0.0.

TIMER1 é configurado para gerar uma onda quadrada no pino M1.0.

Ambos os pinos devem ser preliminarmente conectados a um dos pinos de I/O do μC.