THE MINISTRY of EDUCATION and SCIENCE of RUSSIAN FEDERATION
SAMARA STATE AEROSPACE UNIVERSITY

I. A. Kudryavtsev, D. V. Kornilin

Radio complexes for flight monitoring and control of the micro/nanosatellites

Electronic Lecture Notes

SAMARA
2011

Authors: **Kudryavtsev Ilya Alexandrovich,**
Kornilin Dmitry Vladimirovich

Kudryavtsev, I. A. Radio complexes for flight monitoring and control of the micro/nanosatellites = Радиотехнические комплексы контроля полета и управления микро/наноспутников [Electronic resource] : Electronic Lecture Notes / I. A. Kudryavtsev, D. V. Kornilin; The ministry of Education and Science of Russian Federation, Samara State Aerospace University. - Electronic text and graphic data. (1,1 Mb). - Samara, 2011. - 1 CD-ROM.

The notes are intended for the students, studying on the educational program 010900.68 (Applied mathematics and physics). Notes deal with the important aspects of the circuitry's design and programming issues. Notes are developed in the Interuniversity Space Research Department

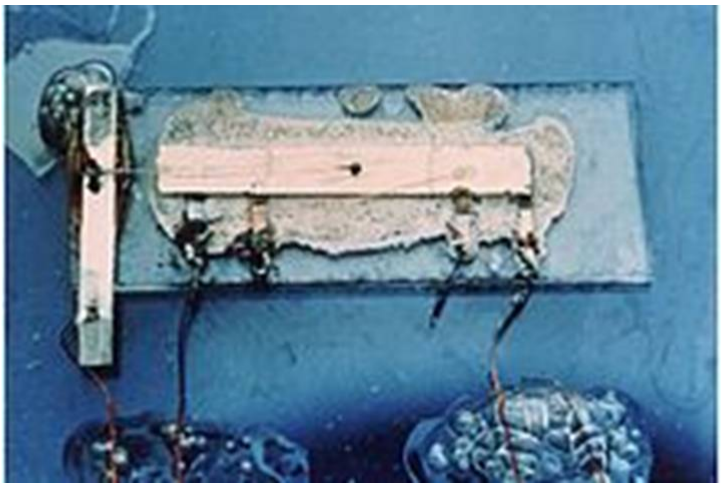
CONTENTS

1 Introduction

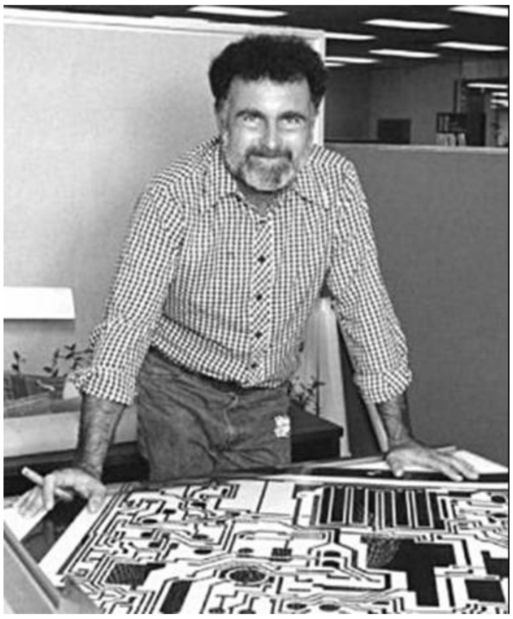
We all realize, that micro/nanosatellite without electronics inside is simply a box of metal. Microsatellite with electronic systems inside or without them will never return to the Earth, so reliability and operational functions of electronic systems determine its actual value.
We shall speak about these systems and subsystems, discuss main approaches, popular among the developers and some special tricks. Today we shall talk about microcontrollers, to help you successfully perform laboratory trainings, power systems, analog electronics and some aspects of circuitry development.

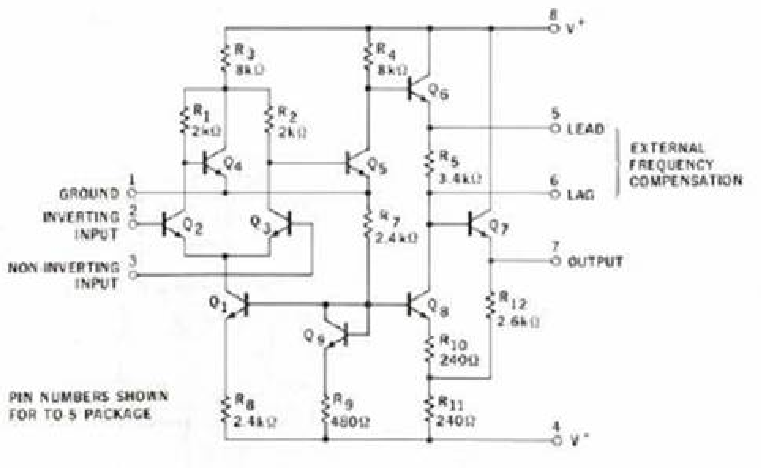
Here, in Russia we believe, that to understand smth., one needs to be familiar with the history of the question, at least briefly. Electronics is relatively young, so most of its main achievements were reached at last century. The main element transistor is invented by three scientists, working in Bell Labs: Here you can see also their brainchild.

William Schokley is known also for the idea of MOSFET, that was realized a bit later. Prof. Schokley was also known for his racist opinions, for which he was condemned by his colleagues. Nevertheless they all had won Nobel prizes for their contribution to the electronics.



John Kilby is known as an inventor of the germanium based integrated circuit, which was a generator. Robert Noyce had offered silicon based IC and a planar process, which was significantly more appropriate for the production.





Bob Widlar is mostly known as an inventor of the first integrated operational amplifier, whose circuit you can see here. Here, in Russia we had the analog of this circuit, and I had even a little experience, working with it. It was unstable, output stage could easily had been burnt, so I hadn't liked it, but soon we have got a lot of much more good chips. Bob Widlar is also known for his eccentric deeds, for example once he had brought a sheep in his Mercedes to trim the lawn near the National Semiconductors' office. Allegedly he had later sold the animal to the nearest restaurant. When something didn't worked properly, he usually had got a special hammer and had thoroughly destroyed the model into the dust. His colleagues named that process "wid-larising".

We can describe main typical functions, essential to this equipment as following:

1. Power control
2. Communication
3. Collecting, processing and storing data
4. Auxiliary functions, providing specific needs

Regardless of the destination of the equipment, it has to be reliable and compact, besides that developers usually try to make hardware, which consumes possibly less power and radiates less heat. Space conditions also require broad temperature range and a tolerance to low pressure.

2 Flight computer

Of course, satellite needs a control center, which aim is to coordinate the operation of different systems; usually we call it board computer. Hardware and software of this computer is extremely important issue to discuss and we shall start from it. The second reason is the timetable of our school, due to which you will have a special training just after lunch.

We can offer two main approaches to the development of flight computer. First one is to buy the ready processor board, which includes CPU, memory and necessary peripherals.

We can now find many different solutions here with various performance, price and features. It is a simple way, because it saves our efforts, but it has its own drawbacks. At first, we can hardly find a solution, completely corresponding to our purposes and conditions, like power consumption, size etc. and we should accommodate our solution to this hardware and software.

Secondly we should use hardware and software, recommended by vendor, and this fact can limit our opportunities. Here, for instance we have useful peripherals, powerful CPU, a lot of memory and seven useless (in space) USB ports. By the way, let's look at the first slide, and then at the last. Has anybody seen anything suspicious? Then once again and look at the temperature range.

Another way is to develop our own hardware/software, capable to interact with our peripherals and to solve our limited set of tasks. Here we can save power, make the PCB of any desired size and so on, but we need to develop the mainboard, which may be a separate problem.

We can also use a compromise solution – to use a so-called mezzanine board, including usually CPU, memory and only limited set of peripherals. Such boards have usually a little size and we can develop our own mainboard with necessary peripherals, perfectly satisfying our requirements. Some of you will deal with such board during laboratories.

If you want to develop flight computer, you should solve two main groups of problems: software and hardware. Hardware problems are usually not so various, so we'll begin from software.

Here are two main ways of thinking too. The first way is to develop our own software "ab ovo", and it will perform only limited set of tasks, necessary for our board equipment. This way provides us with an opportunity to use all the performance possibilities, that a processor have, but the drawback is poor flexibility. To add the feature or to remove can require full revision of the software. Advantage of this approach is a full control over your software and perhaps, exclusive efficiency (if you know, what are you doing certainly).

Another way is to use an operating system, like Windows Embedded, FreeRTOS and so on. Using OS simplifies the process of software development, but usually requires extra resources and limits user in its control over CPU. Nevertheless, when having modern CPUs, using also OS seems quite seducing. It is particularly evident, when we need to implement complicated interfaces, like TCP/IP, CAN etc. Using OS requires additional knowledge of its structure and features, so we shall limit our trainings by the first way of thinking.

You also need to know the main difference between GPOS and RTOS. RTOS can guarantee to some of your tasks an uninterruptible execution and GPOS cannot. Windows's task scheduler, for instance, provides a preemptive execution, which guarantee, that all the tasks get their time quanta of CPU's time, therefore no task cannot have an opportunity to be executed continuously. It is a good idea for powerful CPUs and home computers, but can be a problem for some specific missions.

Choice of the language is the most subjective problem. I know the software developer, who is involved in the creation of the satellites, who uses only Basic. The arguments for the choice of language are simple – having an experience with it, availability of the compiler, cooperation with colleagues, requirements to the software.

Software developer usually uses high-level languages (mostly C or C++ or Java) or assembler language. Assembler language allows making more perfect code, using minimum resources, but it also requires much more efforts from developer. Using C, we can solve many problems really faster and simpler, but we cannot guarantee that compiler software will translate our listing into CPU instructions perfectly and it can cause less efficiency and larger code size. Again, taking into account the characteristics of modern CPUs and compilers, most part of developers uses C/C++, but sometimes implementing part of your programs in assembler language can be very efficient.

Answer to this question is complicated due to the large choice of CPUs on the market. Actually, overwhelming majority of the engineers solve this problem subjectively. Main reasons are usually following: I have an experience in working with CPU/MCU, I have this CPU on my shelf, I have an integrated development environment for this CPU or maybe my chief insists on applying this CPU. For example, our laboratories are based on NXP CPUs only because we have these development boards. By the way, start with a development board significantly simplifies and accelerates the development. I can also add, that ARM7 is a widespread CPU platform, and having experience with one of these processors you can freely transfer your solution to another clone of ARM7.

ARM7TDMI (LPC2148) is 32-bit RISC MCU with rich set of peripherals almost for every purpose, which you can imagine. Its main drawback is large interrupt latency, which depreciates MCU's own performance. Another drawback, which has been annoyed many developers is a poor implementation of RS-485 mode in UART. Newer versions, named CORTEX, have improved the situation in both cases. CORTEX-M0, which is installed in laboratory trainings, is a low power version of CORTEX, which can be particularly useful in nano- and picosatellites with poor power capabilities.

Well, what do we need to know about these MCUs to develop software for our missions?

First of all, to create efficient software, we need to know MCU's features. We'll limit these features by only some general details of ARM processors.

ARM is a company, which invents and creates only pure architectures without creating a single processor. But it sells licenses on their cores to other companies – vendors of the chips. Essentially every vendor adds some features to their chips, leaving the core untouched. ARM architectures are ubiquitous and having an experience with one clone, you can migrate to another one without serious problems.

LPC2148 has a lot of on-chip peripheral modules, internal RAM and FLASH memory, connected by several buses. Please, pay your attention on the fact, that a lot of peripherals divide one separate bus, while memory is connected to the core via another bus. USB and vectored interrupt controller are connected to the third bus. Remark, that interrupt controller is not a part of core, which results in some disadvantages. CPU has a real-time clock with a separate clocking system and battery power. RTC can operate as a separate device, when CPU is switched off and it can awake CPU with its signal in the predetermined moments. Watchdog timer increases the reliability of processor operation. It is a counter, which can generate reset signal for the processor, if activated. The idea is in the regular clearing this counter by the software, when CPU operates properly. If some reason causes hanging of the CPU, WDT wouldn't be cleared in time and processor will be reset. I^2C, SPI, SSP modules support serial interfaces, USART provide standard operations with interfaces of type RS-232. ADC/DAC can be used for analog signal processing. Timers can be used for generating pulses of certain duration and measuring of time intervals, PWM modules can provide simple and efficient method of control for actuating devices. LPC2148 can run at frequencies up to 60MHz.

CORTEX-M3 has better bus performance and some advanced features, like USB Host/OTG support, DMA controller etc. LPC1768 can run at frequencies up to 100MHz. You can see, that here interrupt controller is already included in the core.

CORTEX-M0 is a truncated low-cost and low-power version of CORTEX-M3, running at 50MHz and keeping main advantages of CORTEX architecture.

3 ARM7TDMI architecture

ARM7TDMI is 32bit RISC general purpose processor with Von Neumann memory structure. It is realized on the base of v4T architecture.

Main CPU's features:

- 3-stage pipeline, which provides one clock execution time for sequential instructions (including multiplication);
- Two instruction sets: ARM and THUMB. In ARM mode CPU executes 32bit instructions, in THUMB mode — 16bit instructions;
- Load and Store architecture;
- Debugging interfaces - JTAG and ETM.

3.1 Pipelining

In order to improve performance a three stage pipeline is used, this allows multiple instructions to be processed simultaneously. The pipeline has three stages, Fetch, decode and execute. The hardware of each stage is designed to be independent so up to three instructions can be processed simultaneously. The pipeline is most effective in speeding up sequential code. However a branch instruction will cause the pipeline to be flushed marring its performance.

As the pipeline is a part of CPU it is transparent for the software developer. Nevertheless it is important to remember one thing: **Program Counter (PC) points to the prefetched instruction**, not to the executed, so developer should properly determine the offset.

3.2 Memory access

ARM7TDMI-S has Von Neumann architecture with a single 32bit data bus, used for the instructions and data transfers. Only load and save instructions can access memory.

Memory system uses one of the two following mapping schemes:

- *Big-endian*
- *Little-endian*.

Least significant byte of the word is kept in the lowest memory address (that is equal to the address of the low half-word and the whole 32bit word).

Most significant byte of the word is kept in the highest memory address (that is equal to the address of the low half-word and the whole 32bit word).

3.3 Instruction modes

ARM7TDMI has two modes:

ARM mode	CPU is executing 32bit ARM instructions.
Thumb mode	CPU is executing 16bit Thumb instructions.

The Thumb instruction set must always be entered by running a Branch exchange or branch link exchange instruction and NOT by setting the T bit in the CPSR. Thumb instructions are essentially a mapping of their 32 bit cousins but unlike the ARM instructions, they are unconditionally executed except though for branch instructions.

Thumb instructions only have unlimited access to registers R0-R7 and R13 – R15. A reduced number of instructions can access the full register set.

Thumb has a much higher code density than ARM code, needing some 70% of the space of the latter. However in a 32-bit memory, ARM code is some 40% faster than Thumb. However it should be noted that if you only have 16-bit wide memory then Thumb code will be faster than ARM code by about 45%. Finally the other important aspect of Thumb is that it can use up to 30% less power than ARM code.

Remark 1: In Thumb mode Program Counter (PC) is increased by 4 between instructions, unlikely increase by 8 in ARM mode.

Branching instructions (BX) are used to enter the 16-bit Thumb instruction set. Both the branch and branch-with-link (BLX) may perform an exchange between 32-bit and 16-bit instruction sets and vice versa.

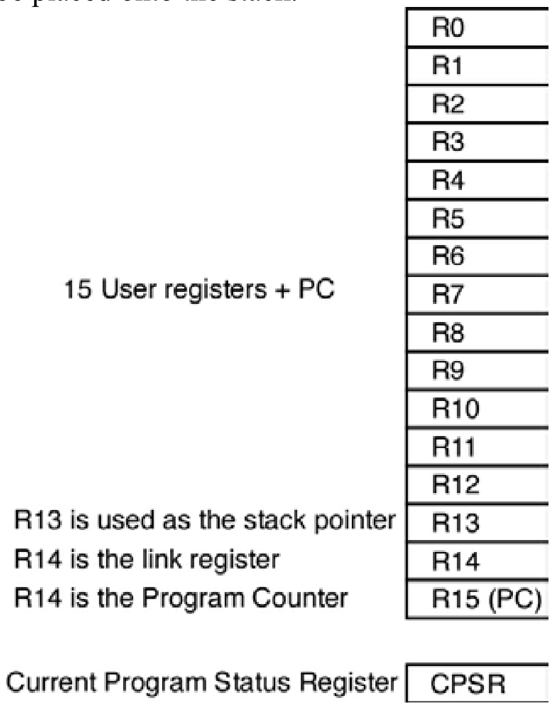
Remark 2: Switching between ARM and Thumb modes concerns neither processor's mode nor content of the registers.

Remarks 3: Exception's processing is being executed in ARM mode. When exception is occurred in Thumb mode, processor switches to ARM mode automatically. Switching to the Thumb mode back is performed also automatically, when exception processing is completed.

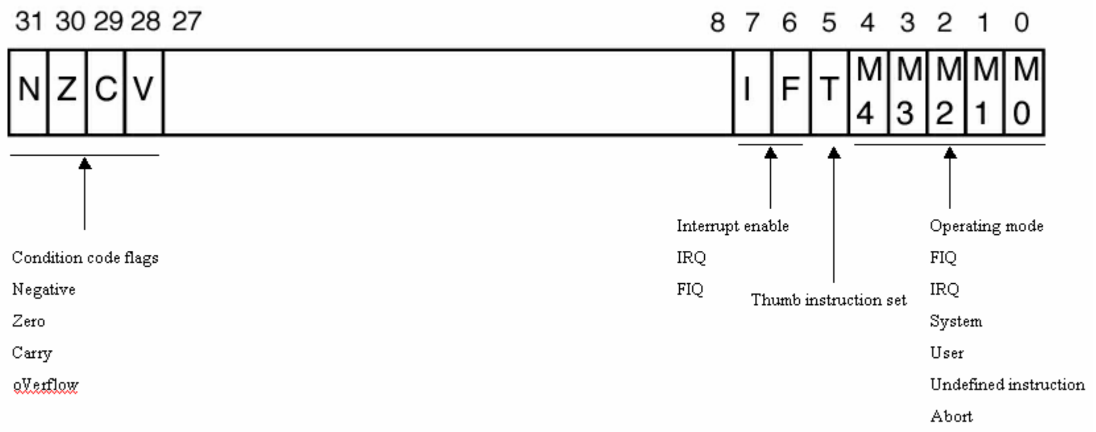
3.4 ARM registers

The programmer's model of the ARM 7 consists of 15 user registers, as shown in Fig. 3, with R15 being used as the Program Counter (PC). Since the ARM 7 is a load-and-store architecture, an user program must load data from memory into the CPU registers, process this data and then store the result back into memory. Unlike other processors no memory to memory instructions are available.

R15 is the Program Counter. R13 and R14 also have special functions; R13 is used as the stack pointer, though this has only been defined as a programming convention. Unusually the ARM instruction set does not have PUSH and POP instructions so stack handling is done via a set of instructions that allow loading and storing of multiple registers in a single operation. Thus it is possible to PUSH or POP the entire register set onto the stack in a single instruction. R14 has special significance and is called the “link register”. When a call is made to a procedure, the return address is automatically placed into R14, rather than onto a stack, as might be expected. A return can then be implemented by moving the contents of R14 into R15, the PC. For multiple calling trees, the contents of R14 (the link register) must be placed onto the stack.



In addition to the 16 CPU registers, there is a current program status register (CPSR). This contains a set of condition code flags in the upper four bits that record the result of a previous instruction, as shown in Fig 4. In addition to the condition code flags, the CPSR contains a number of user-configurable bits that can be used to change the processor mode, enter Thumb processing and enable/disable interrupts.

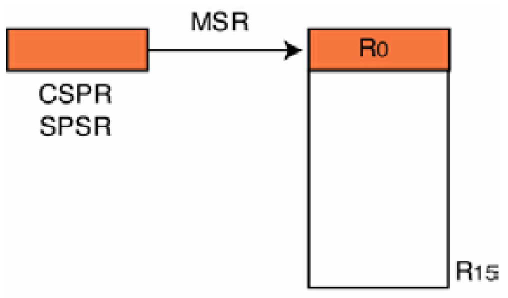


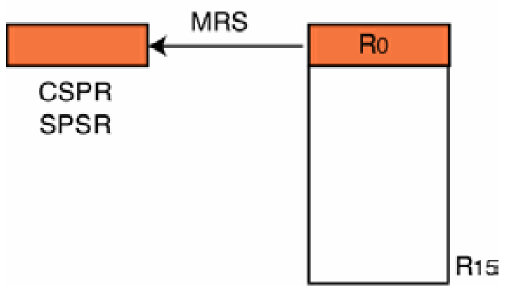
Saved Program Status Register – SPSR keeps the same set of the flags, as CPSR after any event, triggering an exception**.** If in this moment CPU was in User mode, the mode would change and CPSR would saved into SPSR. After completion of the exception’s processing CPSR is restored from SPSR, making possible the continuation of user’s routine.

Remark 4: SPSR is present in all modes, except User and System.

M4-M0 bits define six processor modes with specific register sets.

The CPSR and SPSR are only accessed by two special instructions to move their contents to and from a CPU register. No other instruction can act on them directly.





Thumb registers are the part of ARM registers.

In Thumb instructions only three bits are installed to point the number of register, so one can directly access only R0-R7 (low registers). R8-R12 (high registers) are available only via special load instructions, like MOV.

There are no MSP и MRS instructions in Thumb mode, so changing of CPSR and SPSR is only possible via indirect addressing. If it is needed to modify user bits in CPSR, one should switch to ARM mode.

3.5 Processor modes

The ARM 7 architecture has a total of six different operating modes, as shown below. These modes are protected or exception modes which have associated interrupt sources and their own register sets.

User: This mode is used to run the application code. Once in user mode the CPSR cannot be written to and modes can only be changed when an exception is generated.

FIQ: (Fast Interrupt reQuest) This supports high speed interrupt handling. Generally it is used for a single critical interrupt source in a system

IRQ: (Interrupt ReQuest) This supports all other interrupt sources in a system

Supervisor: A “protected” mode for running system level code to access hardware or run OS calls. The ARM 7 enters this mode after reset.

Abort: If an instruction or data is fetched from an invalid memory region, an abort exception will be generated

Undefined Instruction: If a FETCHED opcode is not an ARM instruction, an undefined instruction exception will be generated.

The User registers R0-R7 are common to all operating modes. However FIQ mode has its own R8 –R14 that replace the user registers when FIQ is entered. Similarly, each of the other modes has their own R13 and R14 so that each operating mode has its own unique Stack pointer and Link register. The CPSR is also common to all modes. However in each of the exception modes, an additional register - the saved program status register (SPSR), is added. When the processor changes the current value of the CPSR stored in the SPSR, this can be restored on exiting the exception mode.

3.6 Exceptions

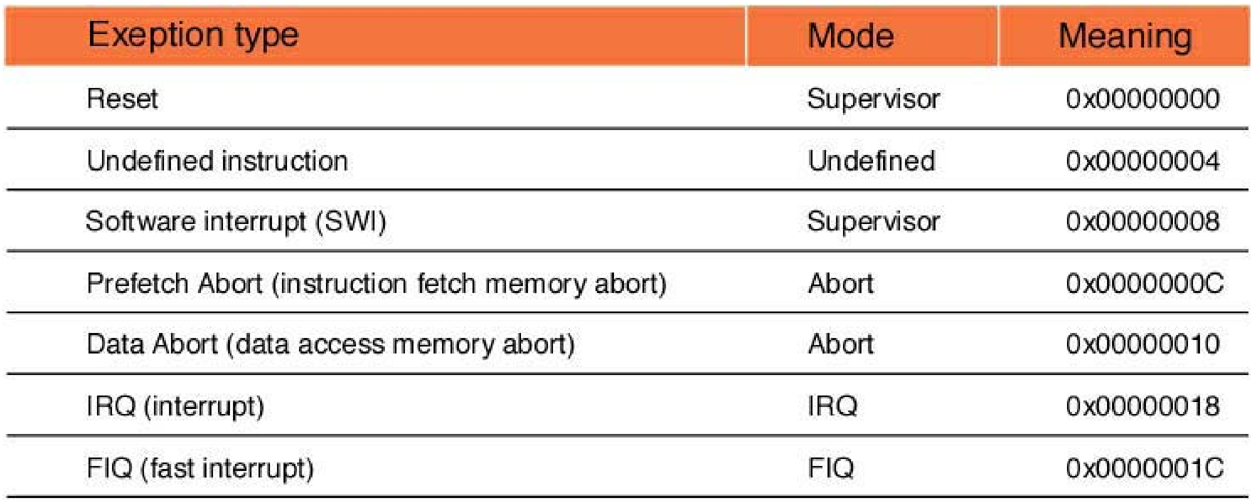
Entry to the Exception modes is through the interrupt vector table. Exceptions in the ARM processor can be split into three distinct types.

– Exceptions caused by executing an instruction, these include software interrupts, undefined instruction exceptions and memory abort exceptions
– Exceptions caused as a side effect of an instruction such as a abort caused by trying to fetch data from an invalid memory region.
– Exceptions unrelated to instruction execution, this includes reset, FIQ and IRQ interrupts.

Priorities of the exceptions are set as following:

Priority	**Exception**
Highest 1	Reset
2	Data Abort
3	FIQ
4	IRQ
5	Prefetch Abort
Lowest 6	Undefined instruction. SWI

Peripherals are served by FIQ and IRQ interrupts. Priorities of peripheral interrupts could be set inside these groups.



Exeption type	Mode	Meaning
Reset	Supervisor	0x00000000
Undefined instruction	Undefined	0x00000004
Software interrupt (SWI)	Supervisor	0x00000008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C
Data Abort (data access memory abort)	Abort	0x00000010
IRQ (interrupt)	IRQ	0x00000018
FIQ (fast interrupt)	FIQ	0x0000001C

Remark 1: You can see the “hole” in this table, because the vector with address 0x00000014 is absent. This address was used in earlier versions of ARM CPUs and is still left in ARM7 to make architectures compatible.
Remark 2: There are no vectors, connected with peripherals. Peripherals can arise FIQ or IRQ according the initial settings and the choice of actual reason is performed by interrupt routine.

In each case entry into the exception mode uses the same mechanism. On generation of the exception, the processor switches to the privileged mode, the current value of the PC+4 is saved into the Link register (R14) of the privileged mode and the current value of CPSR is saved into the privileged mode’s SPSR. The IRQ interrupts are also disabled and if the FIQ mode is entered, the FIQ interrupts are also disabled. Finally the Program Counter is forced to the exception vector address and processing of the exception can start. Usually the first action of the exception routine will be to push some or all of the user registers onto the stack.

Once processing of the exception has finished, the processor can leave the privileged mode and return to the user mode. Firstly the contents of any registers previously saved onto the stack must be restored. Next the CSPR must be restored from the SPSR and finally the Program Counter is restored by moving the contents of the link register to R15, (i.e. the Program Counter). The interrupted program flow can then restart.

Reset

When the **nRESET** signal goes LOW, ARM7TDMI abandons the executing instruction and then continues to fetch instructions from incrementing word addresses.

When **nRESET** goes HIGH again, ARM7TDMI:

1 Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The value of the saved PC and SPSR is not defined.

2 Forces M[4:0] to 10011 (Supervisor mode), sets the I and F bits in the CPSR, and clears the CPSR's T bit.

3 Forces the PC to fetch the next instruction from address 0x00.

4 Execution resumes in ARM state.

Abort

An abort indicates that the current memory access cannot be completed. It can be signalled by the external **ABORT** input. ARM7TDMI checks for the abort exception during memory access cycles.

There are two types of abort:

5. Prefetch abort occurs during an instruction prefetch.
6. Data abort occurs during a data access.

If a prefetch abort occurs, the prefetched instruction is marked as invalid, but the exception will not be taken until the instruction reaches the head of the pipeline. If the instruction is not executed - for example because a branch occurs while it is in the pipeline - the abort does not take place. If a data abort occurs, the action taken depends on the instruction type:

- Single data transfer instructions (LDR, STR) write back modified base registers: the Abort handler must be aware of this.
- The swap instruction (SWP) is aborted as though it had not been executed.
- Block data transfer instructions (LDM, STM) complete. If write-back is set, the base is updated. If the instruction would have overwritten the base with data (ie it has the base in the transfer list), the overwriting is prevented. All register overwriting is prevented after an abort is indicated, which means in particular that R15 (always the last register to be transferred) is preserved in an aborted.

The abort mechanism allows the implementation of a demand paged virtual memory system. In such a system the processor is allowed to generate arbitrary addresses. When the data at an address is unavailable, the Memory Management Unit (MMU) signals an abort. The abort handler must then work out the cause of the abort, make the requested data available, and retry the aborted instruction. The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort. After fixing the reason for the abort, the handler should execute the following irrespective of the state (ARM or Thumb):

```
SUBS PC,R14_abt,#4 for a prefetch abort, or
SUBS PC,R14_abt,#8 for a data abort
```

This restores both the PC and the CPSR, and retries the aborted instruction.

First interrupt (FIQ)

The FIQ (Fast Interrupt Request) exception is designed to support a data transfer or channel process, and in ARM state has sufficient private registers to remove the need for register saving (thus minimizing the overhead of context switching).

FIQ is externally generated by taking the **nFIQ** input LOW. This input can except either synchronous or asynchronous transitions, depending on the state of the **ISYNC** input signal. When **ISYNC** is LOW, **nFIQ** and **nIRQ** are considered asynchronous, and a cycle delay for synchronization is incurred before the interrupt can affect the processor flow. Irrespective of whether the exception was entered from ARM or Thumb state, a FIQ handler should leave the interrupt by executing

```
SUBS PC,R14_fiq,#4
```

IRQ
The IRQ (Interrupt Request) exception is a normal interrupt caused by a LOW level on the **nIRQ** input. IRQ has a lower priority than FIQ and is masked out when a FIQ sequence is entered. It may be disabled at any time by setting the I bit in the CPSR, though this can only be done from a privileged (non-User) mode. Irrespective of whether the exception was entered from ARM or Thumb state, an IRQ handler should return from the interrupt by executing

```
SUBS PC,R14_irq,#4
```

Undefined instruction
When ARM7TDMI comes across an instruction which it cannot handle, it takes the undefined instruction trap. This mechanism may be used to extend either the THUMB or ARM instruction set by software emulation. After emulating the failed instruction, the trap handler should execute the following irrespective of the state (ARM or Thumb):

```
MOVS PC,R14_und
```

This restores the CPSR and returns to the instruction following the undefined instruction.

Software Interrupt (SWI)
The software interrupt instruction (SWI) is used for entering Supervisor mode, usually to request a particular supervisor function. A SWI handler should return by executing the following irrespective of the state (ARM or Thumb):

```
MOV PC, R14_svc
```

This restores the PC and CPSR, and returns to the instruction following the SWI.

4 Instruction set

4.1 Addressing modes

Designation	Title
#Imm	Immediate
Rn	Register
Rn, shift #n	Register shifted
[Rn]	Indexed
[Rn,±Imm]	Preindexed with immediate offset
[Rn,±Rm]	Preindexed with register offset
[Rn,±Rm, shift #n]	Preindexed with scaled register offset
[Rn],±Rm	Postindexed with register offset
[Rn],±Rm, shift #n	Postindexed with scaled register offset

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15).

4.2 ARM Instruction set

ARM7TDMI-S is a RISC CPU, so the number of instructions is relatively low. The most important features are:

CPSR flags

The data processing operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15, see below) the V flag in the CPSR will be unaffected, the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0), the Z flag will be set if and only if the result is all zeros, and the N flag will be set to the logical value of bit 31 of the result.

Condition field

In ARM state, all instructions are conditionally executed according to the state of the CPSR condition codes and the instruction's condition field. This field (bits 31:28) determines the circumstances under which an instruction is to be executed. If the state of the C, N, Z and V flags fulfils the conditions encoded by the field, the instruction is executed, otherwise it is ignored.
There are sixteen possible conditions, each represented by a two-character suffix that can be appended to the instruction's mnemonic. For example, a Branch (B in assembly language) becomes BEQ for "Branch if Equal", which means the Branch will only be taken if the Z flag is set.
In practice, fifteen different conditions may be used: these are listed below.

In the absence of a suffix, the condition field of most instructions is set to "Always" (suffix AL). This means the instruction will always be executed regardless of the CPSR condition codes.

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

ARM instructions could be divided into following groups:

- **Branches**,
- **Load/store**,
- **Data processing**,

4.2.1 Branches

Mnemonic	Instruction	Operation
B	Branch	PC := PC + (rel<<1)
BL	Branch with link	LR := PC, PC := PC + (rel<<1)
BX	Branch and exchange	PC := Rn, T := Rn[0]
SWI	Software interrupt	Supervisor, LR := PC, PC := 0x0008

Branch instructions contain a signed 2's complement 24 bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes. The branch offset must take account of the Prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction.
Branches beyond +/- 32Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a Branch with Link type operation is required
Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. Note that the CPSR is not saved with the PC and R14[1:0] are always cleared. To return from a routine called by Branch with Link use MOV PC,R14 if the link register is still valid or LDM Rn!,{..PC} if the link register has been saved onto a stack pointed to by Rn. This instruction performs a branch by copying the contents of a general register, Rn, into the program counter, PC. The branch causes a pipeline flush and refill from the address specified by Rn. This instruction also permits the instruction set to be exchanged. When the instruction is executed, the value of Rn[0] determines whether the instruction stream will be decoded as ARM or THUMB instructions.
The software interrupt instruction is used to enter Supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change. The PC is then forced to a fixed value (0x08) and the CPSR is saved in SPSR_svc. If the SWI vector address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

4.2.2 . Load/store instructions

Mnemonic	Instruction	Operation
LDM	Load multiple registers	
LDR	Load register from memory	Rd := (address)
MOV	Move register or constant	Rd := Op2
MRS	Move PSR status/flags to register	Rn := CPSR (SPSR)
MSR	Move register to PSR status/flags	CPSR (SPSR): = Rm
MVN	Move negative register	Rd := HE Op2
STM	Store Multiple	
STR	Store register to memory	<address> := Rn
SWP	Swap register with memory	Rd := [Rn], [Rn] := Rm

Lock data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks which can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory

SR/MRS allow access to the CPSR and SPSR registers. The MRS instruction allows the contents of the CPSR or SPSR_<mode> to be moved to a general register. The MSR instruction allows the contents of a general register to be moved to the CPSR or SPSR_<mode> register.
The MSR instruction also allows an immediate value or register contents to be transferred to the condition code flags (N,Z,C and V) of CPSR or SPSR_<mode> without affecting the control bits. In this case, the top four bits of the specified register contents or 32 bit immediate value are written to the top four bits of the relevant PSR

he data swap instruction is used to swap a byte or word quantity between a register and external memory. This instruction is implemented as a memory read followed by a memory write which are “locked” together (the processor cannot be interrupted until both operations have completed, and the memory manager is warned to treat them as inseparable). This class of instruction is particularly useful for implementing software semaphores.

4.2.3 Data processing instructions

Mnemonic	Instruction	Operation
ADC	Add with carry	Rd := Rn+Op2+C
ADD	Add	Rd := Rn+Op2
AND	And	Rd := Rn AND Op2
BIC	Bit Clear	Rd := Rn AND (HE Op2)
CMN	Compare Negative	CPSR flags := Rn+Op2
CMP	Compare	CPSR flags := Rn–Op2
EOR	Exclusive OR	Rd := Rn XOR Op2
MLA	Multiply Accumulate	Rd := (Rm x Rs)+Rn
MUL	Multiply	Rd := Rm x Rs
ORR	OR	Rd := Rn OR Op2
RSB	Reverse Subtract	Rd := Op2–Rn
RSC	Reverse Subtract with Carry	Rd := Op2–Rn–1+C
SBC	Subtract with Carry	Rd := Rn–Op2–1+C
SUB	Subtract	Rd := Rn - Op2
TEQ	Test bitwise equality	CPSR flags:= Rn XOR Op2
TST	Test bits	CPSR flags:= Rn AND Op2

The multiply and multiply-accumulate instructions use an 8 bit Booth's algorithm to perform integer multiplication. The multiply form of the instruction gives Rd:=Rm*Rs. Rn is ignored, and should be set to zero for compatibility with possible future upgrades to the instruction set.
The multiply-accumulate form gives Rd:=Rm*Rs+Rn, which can save an explicit ADD instruction in some circumstances. Both forms of the instruction work on operands which may be considered as signed (2’s complement) or unsigned integers. The results of a signed multiply and of an unsigned multiply of 32 bit operands differ only in the upper 32 bits - the low 32 bits of the signed and unsigned results are identical. As these instructions only produce the low 32 bits of a multiply, they can be used for both signed and unsigned multiplies.

The multiply forms (UMULL and SMULL) take two 32 bit numbers and multiply them to produce a 64 bit result of the form RdHi,RdLo := Rm * Rs. The lower 32 bits of the 64 bit result are written to RdLo, the upper 32 bits of the result are written to RdHi.
The multiply-accumulate forms (UMLAL and SMLAL) take two 32 bit numbers, multiply them and add a 64 bit number to produce a 64 bit result of the form RdHi,RdLo := Rm * Rs + RdHi, RdLo. The lower 32 bits of the 64 bit number to add is read from RdLo. The upper 32 bits of the 64 bit number to add is read from RdHi. The lower 32 bits of the 64 bit result are written to RdLo. The upper 32 bits of the 64 bit result are written to RdHi.
The UMULL and UMLAL instructions treat all of their operands as unsigned binary numbers and write an unsigned 64 bit result. The SMULL and SMLAL instructions treat all of their operands as two's-complement signed numbers and write a two's complement signed 64 bit result.

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32 bit integer (either unsigned or 2's complement signed, the two are equivalent). If the S bit is set (and Rd is not R15) the V flag in the CPSR will be set if an overflow occurs into bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were 2's complement signed. The C flag will be set to the carry out of bit 31 of the ALU, the Z flag will be set if and only if the result was zero, and the N flag will be set to the value of bit 31 of the result (indicating a negative result if the operands are considered to be 2's complement signed).

5 Vectored Interrupt Controller

The Vectored Interrupt Controller (VIC) takes 32 interrupt request inputs and programmably assigns them into 3 categories, FIQ, vectored IRQ, and non-vectored IRQ.
The programmable assignment scheme means that priorities of interrupts from the various peripherals can be dynamically assigned and adjusted. Fast Interrupt reQuest (FIQ) requests have the highest priority. If more than one request is assigned to FIQ, the VIC ORs the requests to produce the FIQ signal to the ARM processor. The fastest possible FIQ latency is achieved when only one request is classified as FIQ because then the FIQ service routine can simply start dealing with that device. But if more than one request is assigned to the FIQ class, the FIQ service routine can read a word from the VIC that identifies which FIQ source(s) is (are) requesting an interrupt.

Vectored IRQs have the middle priority, but only 16 of the 32 requests can be assigned to this category. Any of the 32 requests can be assigned to any of the 16 vectored IRQ slots, among which slot 0 has the highest priority and slot 15 has the lowest. Non-vectored IRQs have the lowest priority.

The VIC ORs the requests from all the vectored and non-vectored IRQs to produce the IRQ signal to the ARM processor. The IRQ service routine can start by reading a register from the VIC and jumping there. If any of the vectored IRQs are requesting, the VIC provides the address of the highest-priority requesting IRQs service routine, otherwise it provides the address of a default routine that is shared by all the non-vectored IRQs. The default routine can read another VIC register to see what IRQs are active. All registers in the VIC are word registers. Byte and halfword reads and write are not supported.

Although multiple sources can be selected (VICIntSelect) to generate FIQ request, only one interrupt service routine should be dedicated to service all available/present FIQ request(s). Therefore, if more than one interrupt sources are classified as FIQ the FIQ interrupt service routine must read VICFIQStatus to decide based on this content what to do and how to process the interrupt request. However, it is recommended that only one interrupt source should be classified as FIQ. Classifying more than one interrupt sources as FIQ will increase the interrupt latency.

Following the completion of the desired interrupt service routine, clearing of the interrupt flag on the peripheral level will propagate to corresponding bits in VIC registers (VICRawIntr, VICFIQStatus and VICIRQStatus). Also, before the next interrupt can be serviced, it is necessary that write is performed into the VICVectAddr register before the return from interrupt is executed. This write will clear the respective interrupt flag in the internal interrupt priority hardware.

In order to disable the interrupt at the VIC you need to clear corresponding bit in the VICIntEnClr register, which in turn clears the related bit in the VICIntEnable register. This also applies to the VICSoftInt and VICSoftIntClear in which VICSoftIntClear will clear the respective bits in VICSoftInt. For example, if VICSoftInt = 0x0000 0005 and bit 0 has to be cleared, VICSoftIntClear = 0x0000 0001 will accomplish this. Before the new clear operation on the same bit in VICSoftInt using writing into VICSoftIntClear is performed in the future, VICSoftIntClear = 0x0000 0000 must be assigned. Therefore writing 1 to any bit in Clear register will have one-time-effect in the destination register.

If the watchdog is enabled for interrupt on underflow or invalid feed sequence only then there is no way of clearing the interrupt. The only way you could perform return from interrupt is by disabling the interrupt at the VIC (using VICIntEnClr).

Spurious interrupts are possible in the ARM7TDMI based microcontrollers such as the LPC21xx and LPC22xx due to asynchronous interrupt handling. The asynchronous character of the interrupt processing has its roots in the interaction of the core and the VIC. If the VIC state is changed between the moments when the core detects an interrupt, and the core actually processes an interrupt, problems may be generated. Real-life applications may experience the following scenarios:

1. VIC decides there is an IRQ interrupt and sends the IRQ signal to the core.

2. Core latches the IRQ state.
3. Processing continues for a few cycles due to pipelining.
4. Core loads IRQ address from VIC.

5.1 Bus architecture

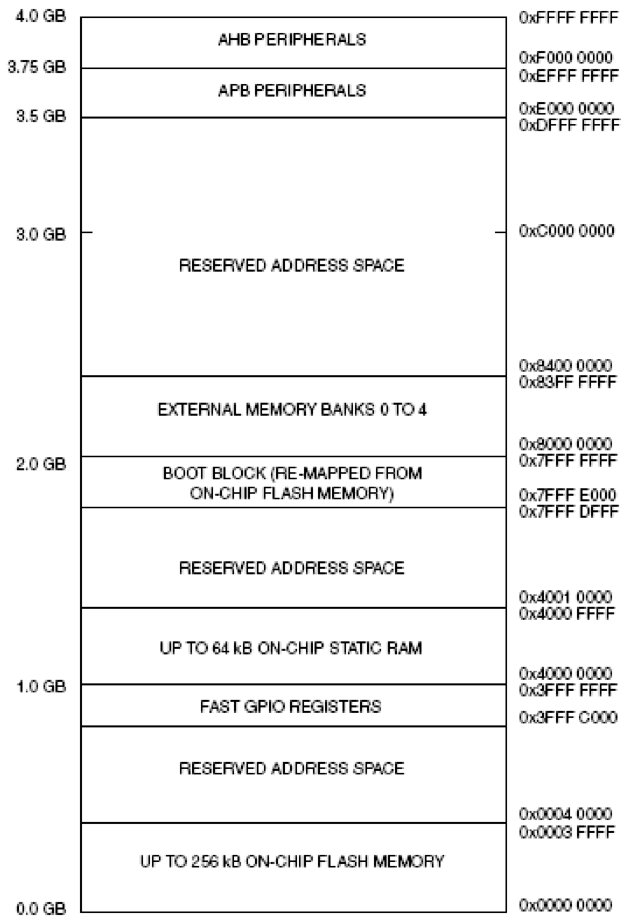
The LPC21xx/LPC22xx consist of an ARM7TDMI-S CPU with emulation support, the ARM7 Local Bus for interface to on-chip memory controllers, the AMBA Advanced High-performance Bus (AHB) for interface to the interrupt controller, and the ARM Peripheral Bus (APB, a compatible superset of ARM's AMBA Advanced Peripheral Bus) for connection to on-chip peripheral functions.

AHB peripherals are allocated a 2 megabyte range of addresses at the very top of the 4 gigabyte ARM memory space. Each AHB peripheral is allocated a 16 kB address space within the AHB address space. LPC21xx/LPC22xx peripheral functions (other than the interrupt controller) are connected to the APB bus. The AHB to APB bridge interfaces the APB bus to the AHB bus. APB peripherals are also allocated a 2 megabyte range of addresses, beginning at the 3.5 gigabyte address point. Each APB peripheral is allocated a 16 kB address space within the APB address space.

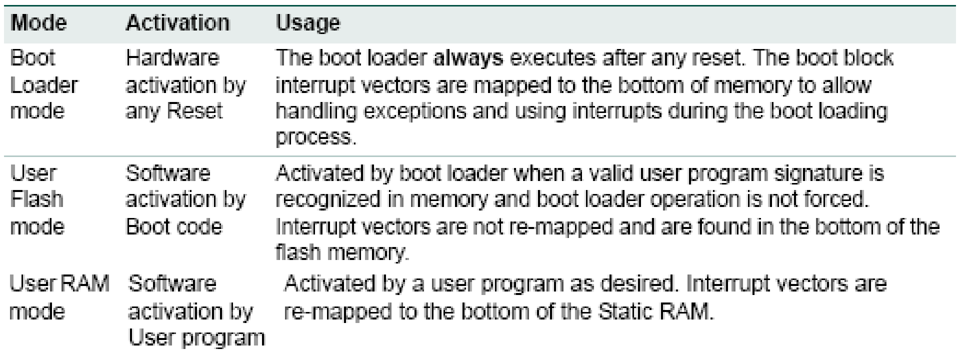
6 LPC21XX features

6.1 Memory map

The LPC21xx incorporates several distinct memory regions, shown in the figure below. The figure shows the overall map of the entire address space from the user program viewpoint following reset. The interrupt vector area supports address remapping, which is described later in this section.



The basic concept on the LPC21xx and LPC22xx is that each memory area has a "natural" location in the memory map. This is the address range for which code residing in that area is written. The bulk of each memory space remains permanently fixed in the same location, eliminating the need to have portions of the code designed to run in different address ranges. Because of the location of the interrupt vectors on the ARM7 processor, a small portion of the Boot Block and SRAM spaces need to be re-mapped in order to allow alternative uses of interrupts in the different operating modes described in the table below. Re-mapping of the interrupts is accomplished via the Memory Mapping Control features.



Mode	Activation	Usage
Boot Loader mode	Hardware activation by any Reset	The boot loader **always** executes after any reset. The boot block interrupt vectors are mapped to the bottom of memory to allow handling exceptions and using interrupts during the boot loading process.
User Flash mode	Software activation by Boot code	Activated by boot loader when a valid user program signature is recognized in memory and boot loader operation is not forced. Interrupt vectors are not re-mapped and are found in the bottom of the flash memory.
User RAM mode	Software activation by User program	Activated by a user program as desired. Interrupt vectors are re-mapped to the bottom of the Static RAM.

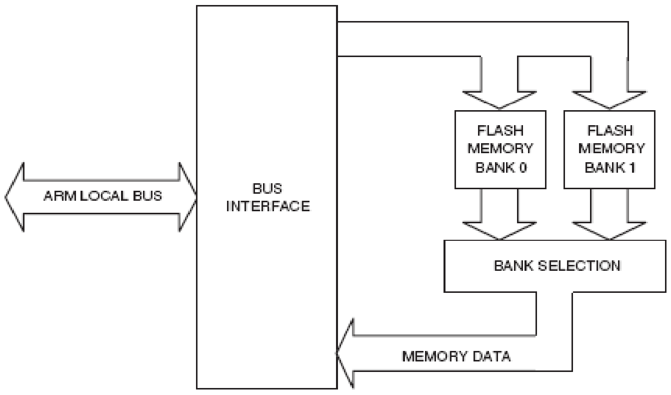
The portion of memory that is re-mapped to allow interrupt processing in different modes includes the interrupt vector area (32 bytes) and an additional 32 bytes, for a total of 64 bytes. The re-mapped code locations overlay addresses 0x0000 0000 through 0x0000 003F. The vector contained in the SRAM, external memory, and boot block must contain branches to the actual interrupt handlers or to other instructions that accomplish the branch to the interrupt handlers.

6.2 Memory acceleration module (MAM)

Simply put, the Memory Accelerator Module (MAM) attempts to have the next ARM instruction that will be needed in its latches in time to prevent CPU fetch stalls. The method used is to split the flash memory into two banks, each capable of independent accesses. Each of the two flash banks has its own prefetch buffer and branch trail buffer. The branch trail buffers for the two banks capture two 128-bit lines of flash data when an instruction fetch is not satisfied by either the prefetch buffer or branch trail buffer for its bank, and for which a prefetch has not been initiated. Each prefetch buffer captures one 128-bit line of instructions from its flash bank at the conclusion of a prefetch cycle initiated speculatively by the MAM.

Each 128 bit value includes four 32-bit ARM instructions or eight 16-bit Thumb instructions. During sequential code execution, typically one flash bank contains or is fetching the current instruction and the entire flash line that contains it. The other bank contains or is prefetching the next sequential code line. After a code line delivers its last instruction, the bank that contained it begins to fetch the next line in that bank.

Branches and other program flow changes cause a break in the sequential flow of instruction fetches described above. When a backward branch occurs, there is a distinct possibility that a loop is being executed. In this case the branch trail buffers may already contain the target instruction. If so, execution continues without the need for a flash read cycle. For a forward branch, there is also a chance that the new address is already contained in one of the prefetch buffers. If it is, the branch is again taken with no delay. When a branch outside the contents of the branch trail and prefetch buffers is taken, one flash access cycle is needed to load the branch trail buffers. Subsequently, there will typically be no further fetch delays until another such “Instruction Miss” occurs.



The flash memory controller detects data accesses to the flash memory and uses a separate buffer to store the results in a manner similar to that used during code fetches. This allows faster access to data if it is accessed sequentially. A single line buffer is provided for data accesses, as opposed to the two buffers per flash bank that are provided for code accesses. There is no prefetch function for data accesses.

After reset the MAM defaults to the disabled state. Software can turn memory access acceleration on or off at any time. This allows most of an application to be run at the highest possible performance, while certain functions can be run at a somewhat slower but more predictable rate if more precise timing is required.

Two configuration bits select the three MAM operating modes, as shown in the table below. Following Reset, MAM functions are disabled. Changing the MAM operating mode causes the MAM to invalidate all of the holding latches, resulting in new reads of flash information as required.

Bit	Symbol	Value	Description	Reset value
1:0	MAM_mode _control	00	MAM functions disabled	0
		01	MAM functions partially enabled	
		10	MAM functions fully enabled	
		11	Reserved. Not to be used in the application.	
7:2	-	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

The MAM Timing register determines how many CCLK cycles are used to access the flash memory. This allows tuning MAM timing to match the processor operating frequency. Flash access times from 1 clock to 7 clocks are possible. Single clock flash accesses would essentially remove the MAM from timing calculations. In this case the MAM mode may be selected to optimize power usage.

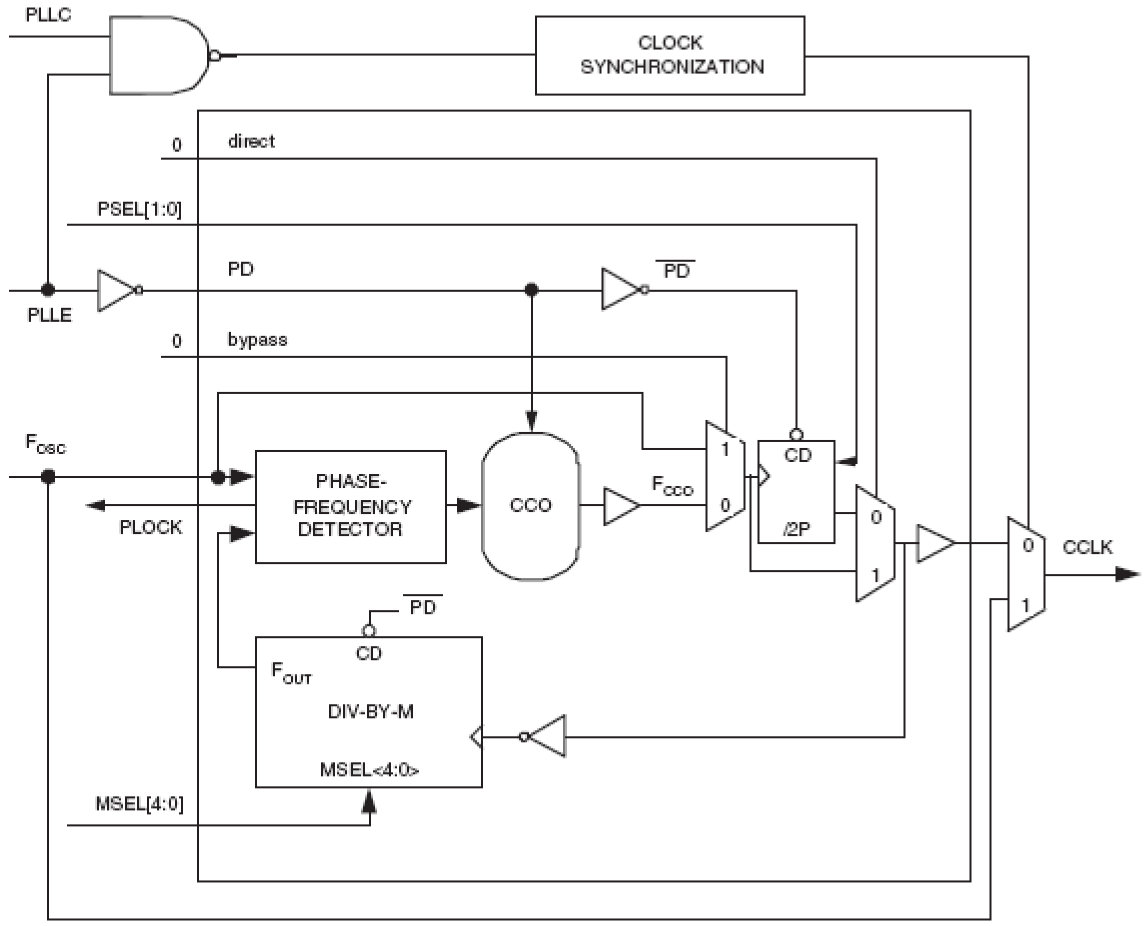
When changing MAM timing, the MAM must first be turned off by writing a zero to MAMCR. A new value may then be written to MAMTIM. Finally, the MAM may be turned on again by writing a value (1 or 2) corresponding to the desired operating mode to MAMCR.

For system clock slower than 20 MHz, MAMTIM can be 001. For system clock between 20 MHz and 40 MHz, flash access time is suggested to be 2 CCLKs, while in systems with system clock faster than 40 MHz, 3 CCLKs are proposed. For system clocks of 60 MHz and above, 4CCLK's are needed.

6.3 Phased Locked Loop (PLL)

The PLL accepts an input clock frequency in the range of 10 MHz to 25 MHz only. The input frequency is multiplied up the range of 10 MHz to 75 MHz for the CCLK clock using a Current Controlled Oscillators (CCO). The multiplier can be an integer value from 1 to 32 (in practice, the multiplier value cannot be higher than 7 on the LPC21xx/LPC22xx due to the upper frequency limit of the CPU). The CCO operates in the range of 156 MHz to 320 MHz, so there is an additional divider in the loop to keep the CCO within its frequency range while the PLL is providing the desired output frequency. The output divider may be set to divide by 2, 4, 8, or 16 to produce the output clock. Since the minimum output divider value is 2, it is insured that the PLL output has a 50% duty cycle.

PLL activation is controlled via the PLLCON register. The PLL multiplier and divider values are controlled by the PLLCFG register. These two registers are protected in order to prevent accidental alteration of PLL parameters or deactivation of the PLL. Since all chip operations, including the Watchdog Timer, are dependent on the PLL when it is providing the chip clock, accidental changes to the PLL setup could result in unexpected behavior of the microcontroller. The protection is accomplished by a feed sequence similar to that of the Watchdog Timer. Details are provided in the description of the PLLFEED register.
The PLL is turned off and bypassed following a chip reset and when by entering Power-down mode. The PLL is enabled by software only. The program must configure and activate the PLL, wait for the PLL to Lock, then connect to the PLL as a clock source.



The PLLCON register contains the bits that enable and connect the PLL. Enabling the PLL allows it to attempt to lock to the current settings of the multiplier and divider values. Connecting the PLL causes the processor and all chip functions to run from the PLL output clock. Changes to the PLLCON register do not take effect until a correct PLL feed sequence has been given.

The PLL must be set up, enabled, and Lock established before it may be used as a clock source. When switching from the oscillator clock to the PLL output or vice versa, internal circuitry synchronizes the operation in order to ensure that glitches are not generated.

Hardware does not insure that the PLL is locked before it is connected or automatically disconnect the PLL if lock is lost during operation. In the event of loss of PLL lock, it is likely that the oscillator clock has become unstable and disconnecting the PLL will not remedy the situation. A correct feed sequence must be written to the PLLFEED register in order for changes to the PLLCON and PLLCFG registers to take effect. The feed sequence is:

1. Write the value 0xAA to PLLFEED.

2. Write the value 0x55 to PLLFEED.

The two writes must be in the correct sequence, and must be consecutive APB bus cycles. The latter requirement implies that interrupts must be disabled for the duration of the PLL feed operation. If either of the feed values is incorrect, or one of the previously mentioned conditions is not met, any changes to the PLLCON or PLLCFG register will not become effective.

Power-down mode automatically turns off and disconnects activated PLL. Wakeup from Power-down mode does not automatically restore the PLL settings, this must be done in software. Typically, a routine to activate the PLL, wait for lock, and then connect the PLL can be called at the beginning of any interrupt service routine that might be called due to the wakeup. It is important not to attempt to restart the PLL by simply feeding it when execution resumes after a wakeup from Power-down mode. This would enable and connect the PLL at the same time, before PLL lock is established.

The PLL output frequency (when the PLL is both active and connected) is given by:

$$CCLK = M \times FOSC \quad CCLK = FCCO / 2 \times P$$

The CCO frequency can be computed as:

$$FCCO = CCLK \times 2 \times P \quad FCCO = FOSC \times M \times 2 \times P$$

The PLL inputs and settings must meet the following:

- FOSC is in the range of 10 MHz to 25 MHz.
- CCLK is in the range of 10 MHz to Fmax (the maximum allowed frequency for the microcontroller - determined by the system microcontroller is embedded in).
- FCCO is in the range of 156 MHz to 320 MHz.

If a particular application uses the PLL, its configuration may be determined as follows:

1. Choose the desired processor operating frequency (CCLK). This may be based on processor throughput requirements, need to support a specific set of UART baud rates, etc. Bear in mind that peripheral devices may be running from a lower clock than the processor.

2. Choose an oscillator frequency (FOSC). CCLK must be the whole (non-fractional) multiple of FOSC.

3. Calculate the value of M to configure the MSEL bits. M = CCLK / FOSC. M must be in the range of 1 to 32. The value written to the MSEL bits in PLLCFG is M − 1.

4. Find a value for P to configure the PSEL bits, such that FCCO is within its defined frequency limits. FCCO is calculated using the equation given above. P must have one of the values 1, 2, 4, or 8. The value written to the PSEL bits in PLLCFG is 00 for P = 1; 01 for P = 2; 10 for P = 4; 11 for P = 8.

7 Interfaces

Interfaces play an extremely important role in control systems. Let’s have a brief review of this issue. SPI is a serial interface, which is used to connect with different external devices, like ADC/DACs, memory, some sensors with digital output and so on. It consists of clock line and one or two data lines and provide point-to-point connection.

I^2C can be used to establish small local nets, including interboard connections. In such connections every node has its unique address and has to require to the requests of master. The typical configuration includes one master and several slaves, though multi-master mode is possible.

I^2S is an interface, intended for audio devices and can hardly be applied in onboard devices.

RS-232 interface is a standard for many typical devices, like navigational receivers, modems and so on. It main advantage is its simplicity and popularity. RS-232 is a point-to-point interface and RS-485 is multi-node version with higher baudrate.

USB is PC originated interface with a lot of useful features, but it is not suitable for autonomous devices, except OTG version. This version allows direct hot connection of typical USB peripherals like FLASH memory sticks, which can be sometimes useful.

Ethernet is high speed interface, which can be used for high-speed communications, connection with video cameras and similar devices.

CAN is a popular interface, which provide reliable connection with peripherals, working in hard EMI conditions. It has message based concept, where no node addresses needed and no rigid configuration is supported. Initially it was an automotive protocol, but it can be useful in space applications too.

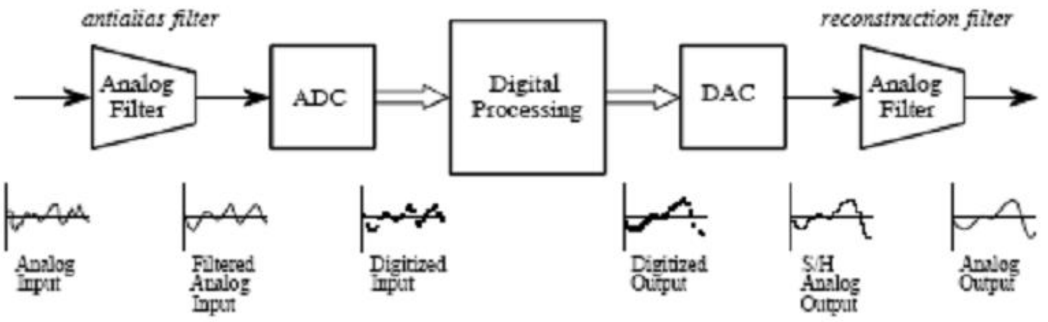
Certainly the most powerful is an assembler programming, but it is not necessary, when using modern CPUs. C compilers now can create compact and efficient code, which decreases the programmer's efficiency significantly. The advantages, which can be achieved, using assembler seem not so seducing in comparison with the opportunity to create and debug your software quite faster. This means, that we shall not pay a lot of time to the details of instruction set, but you should remember, that CPU executes only these instructions, regardless of your difficulties with understanding them. ARM7TDMI has two instruction sets – ARM and Thumb. Exception handlers are always written with ARM and the rest code could be written in ARM or Thumb. If you want to combine both types, you should tell the compiler about it (interworking strategy).

ARM instruction set has two features, which increase the overall efficiency. They are – conditional execution of some instructions and shifting capability.

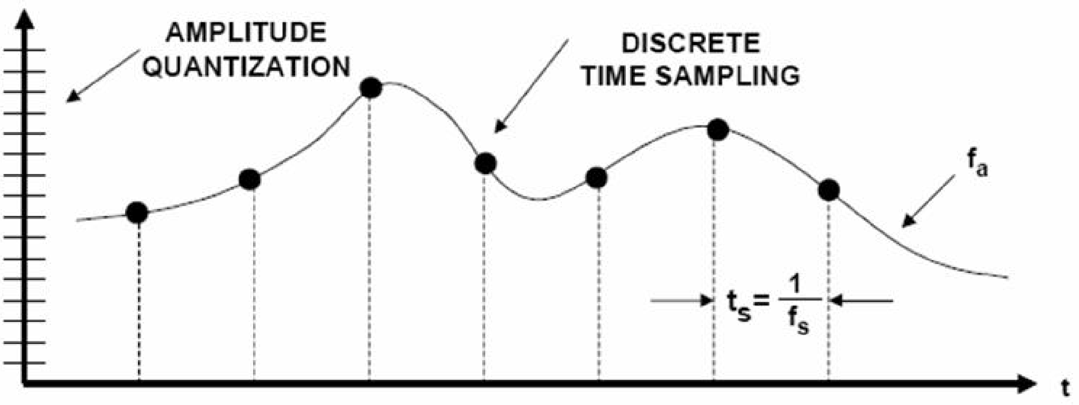
Using these features, we can implement complicated operations with the help of a single instruction, as shown here.

8 Signal processing

One of the key issues here is the process of analog-to-digital conversion along with some auxiliary subfunctions, like signal conditioning, amplifying and so on. First of all, the developer should define, what kind of information he needs to obtain. Main parameters here are amplitude range, frequency range and desired accuracy.



First question is a sampling frequency, or how many times per second we have to take samples. Here we need to take into account Nyquist's criteria.

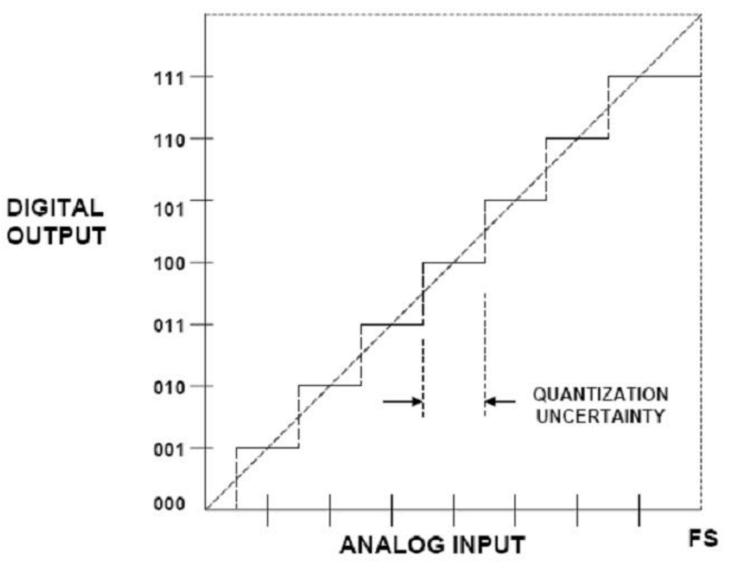


By the way, in Russia we prefer to say Kotelnikov's theorem, having the same meaning. I'm sure, that all of you had seen strange effects in the cinema, when carriage wheels or helicopter's airscrew is rotating with a wrong velocity or even in a wrong direction. This can be explained by improper sampling, when the time interval between adjacent frames is too large. Then the angle of rotation can be so significant, that we couldn't interpret it correctly. This all means, that you ought to be careful, choosing sampling rate for your application. ADC's capability is usually characterized in the terms of Samples Per Second (SPS).

You can also see similar effect o this slide. Some of you have some experience with the theory of Fourier transform and can interpret it in a frequency domain as spectrum aliasing.

Second question concerns conversion accuracy. Here ADC's transfer function plays the main role. As you can see here, ADC perceives adjacent values of analog signal as the same digital value, that cause an error, named quantization error. Ideal ADC has ±1/2LSB quantization error.

TRANSFER FUNCTION FOR IDEAL 3-BIT ADC



Besides that, real ADC has some extra sources of the error, like offset error and gain error. The idea is evident from the figure.

Non-linearity is more sophisticated. It id usually interpreted as the maximum deviation from the ideal characteristic, but some vendors prefer so-called "best-fit" estimate, which can mislead the developer.

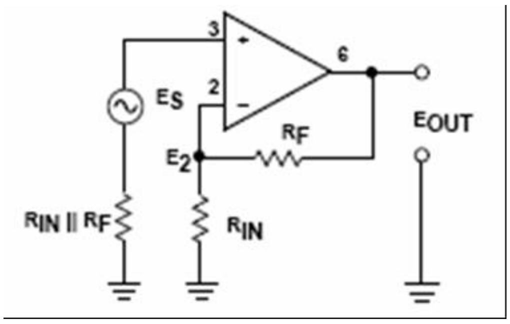
Quantization error is a source of so-called quantization noise, which can be treated as a difference between input signal and a discrete image. Then it can be shown, that SNR can be calculated with a following formula. Again we have not enough time to discuss ADC features in details, but I hope that you have already realized that ADC's choice requires attentive relation.

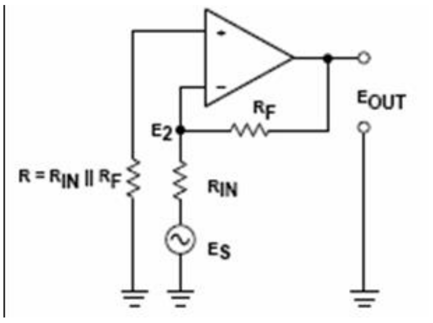
ADC control is usually quite simple: you have to start conversion, wait until it would be completed and read result. This can be done by different means, but the idea is usually the same.

Let me also briefly describe available ADC types. SAR (Successive Approximation Register) ADCs are the most popular. The idea of their operation is subsequent estimation of the signal with the help of the special procedure. Pipelined ADCs are intended for fast applications. The conversion is performed by the pipeline circuit. Sigma-Delta ADCs are the most precise ADCs, which can perform 24-bit accuracy, due to the complex algorithm of conversion.

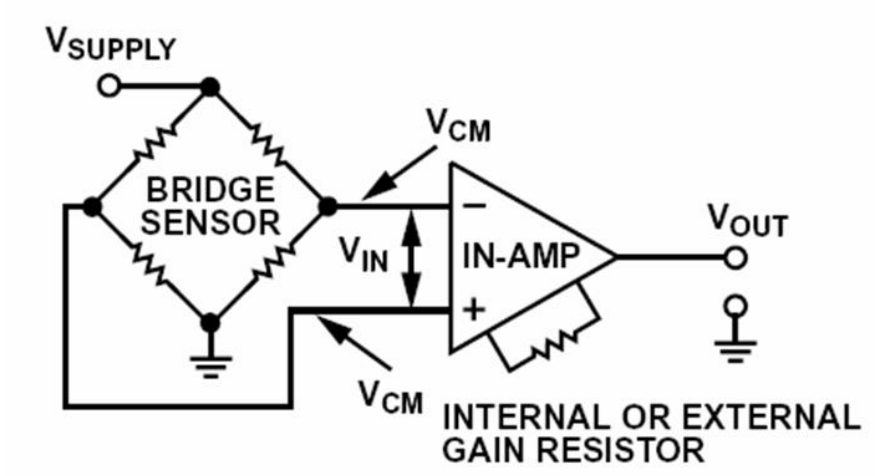
At the input of the ADC anti-aliasing filter is usually used. It is intended to attenuate the influence of the outband interference signals. If not rejected, these signals can affect useful signals, as shown on this figure.

Signals, obtained from the transducers can be quite weak, and to match the ADC's input requirements they are often need amplification. This operation can be performed by the operation amplifier. Theory of the operational amplifiers is not very simple, but we can be satisfied by several main ideas.





First of all, there are two main types of the amplifier circuits – inverting and non-inverting. Formulae for the gain of these circuits are very simple. Inverting amplifier inverts the signal, as implies from its name. Non-inverting amplifier has high input resistance, which is important, when you deal with a transducer with high impedance.



Operational amplifier is essentially differential, but previous circuits cannot provide really differential amplification. If you need it, for instance, while working with bridge-type sensors, you should apply instrumentation amplifier.

9 Communications

Communication systems are extremely important for the success of the satellite's mission, especially when we deal with microsatellites, which can't be landed. The idea of communication system is similar to the fundamental ideas, known to you from the basics of radioelectronics.

First of all we have to estimate power, required for the reliable reception of the signal. Typically we have relatively small transmitter aboard the satellite and a very simple antenna, while the ground stations can have better parameters of antennas and radio equipment. At any rate, the power equation is well known, as De Friis equation, shown below.

Friis's transmission equation: $\frac{P_r}{P_t} = \frac{G_t G_r \lambda^2}{(4\pi d)^2}$

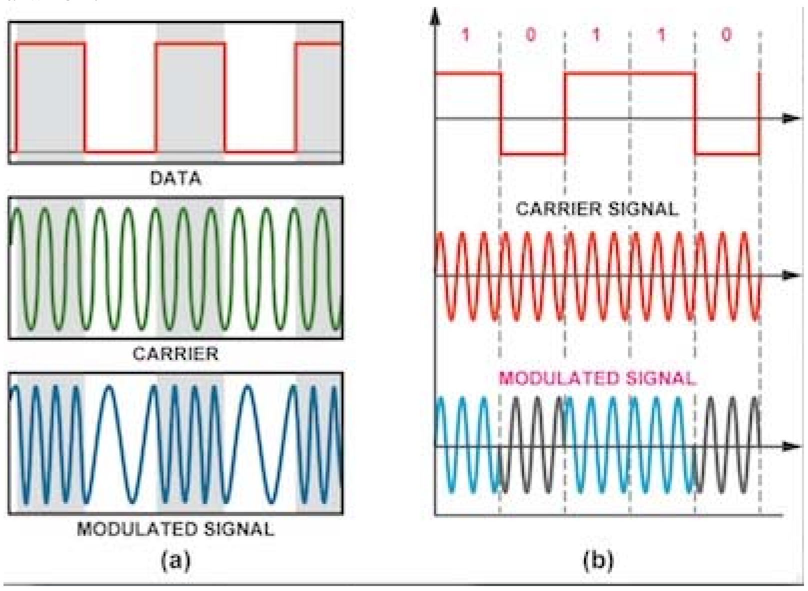
Friis's equation in decibel form: $10 \log \frac{P_r}{P_t} = G_{t(dB)} + G_{r(dB)} + 10 \log \left(\frac{\lambda}{4\pi d} \right)^2$

Spreading loss in satellite communications: $L = 33\,dB + \left[20 \log (d_{km}) \right] + \left[20 \log (F_{MHZ}) \right]$

Total noise in system: $T_{total} = T_{eq(rcvr)} + T_{eq(ant)}$

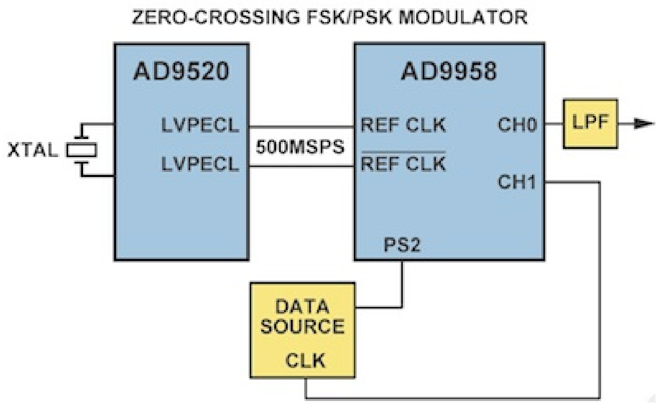
Second important issue is a modulation. The most popular kinds oa modulation used in microsatellite's communication systems, are PSK and FSK.

Frequency-shift keying (FSK) and phase-shift keying (PSK) modulation schemes are also used in digital communications, radar, RFID, and numerous other applications. The simplest form of FSK uses two discrete frequencies to transmit binary information, with Logic 1 representing the mark frequency and Logic 0 representing the space frequency. The simplest form of PSK is binary (BPSK), which uses two phases separated by 180°. Figure below illustrates these two types of modulation.

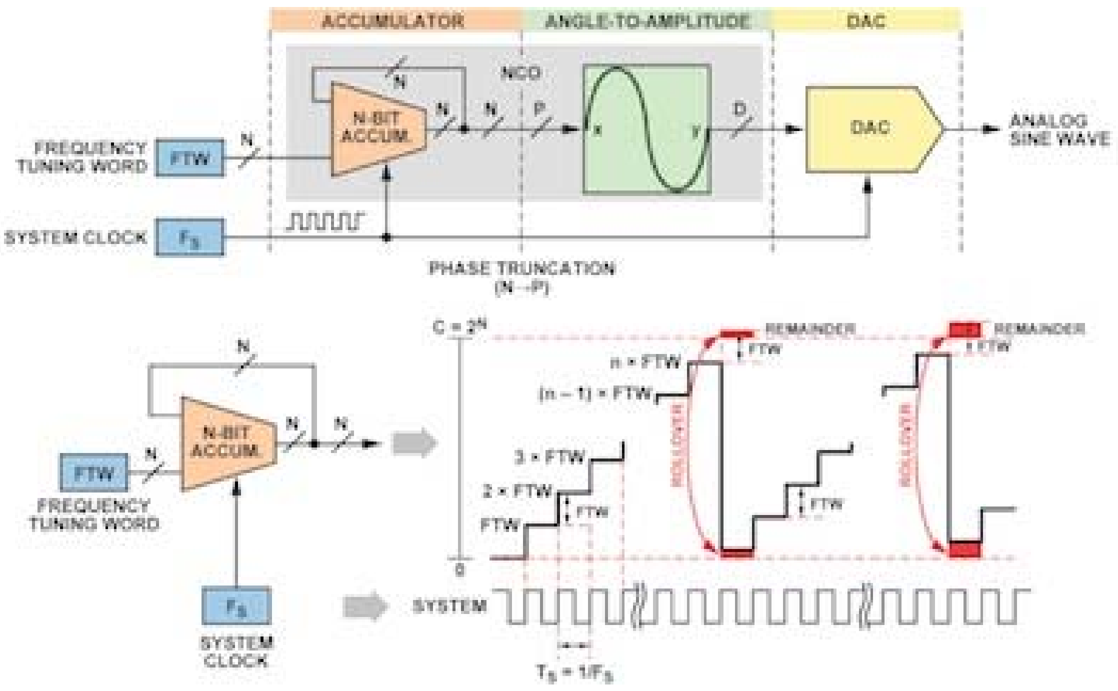


The modulated output of a direct digital synthesizer (DDS) can switch frequency and/or phase in a phase-continuous or phase-coherent manner, making DDS technology well suited for both FSK and PSK modulation. Two synchronized DDS channels can implement a zero-crossing FSK or PSK modulator. Here, the AD9958 two-channel, 500-MSPS, complete DDS is used to switch frequencies or phases at the zero-crossing point, but any two-channel synchronized solution should be capable of accomplishing this function. In phase-coherent radar systems, zero-crossing switching reduces the amount post processing needed for signature recognition of the target, and implementing PSK at the zero crossing reduces spectral splatter.

Although both of the DDS-channel outputs are independent, in a multi-channel DDS they share an internal system clock and reside on a single piece of silicon, so they should provide more reliable channel-to-channel tracking over temperature and power-supply deviations than the outputs of multiple, single-channel devices synchronized together. The process variability that may exist between distinct devices is also larger than any process variability you might see between two channels fabricated in a single piece of silicon, making a multichannel DDS preferable for use as a zero-crossing FSK or PSK modulator.



A critical element of any DDS is the phase accumulator, which, in this implementation, is 32 bits wide. When the accumulator overflows, it retains any excess value. When the accumulator overflows with no remainder, the output is precisely at phase 0, and the DDS engine starts over from where it was at time 0. The rate at which the zero-overflow is experienced is referred to as the grand-repetition rate (GRR) of the DDS.



The GRR is determined by the rightmost nonzero bit of the DDS frequency tuning word (FTW), as established by the following equation:

GRR = FS/2n,

where FS is the sampling frequency of the DDS, and n is the rightmost nonzero bit of the FTW. For example, suppose a DDS with a 1-GHz sampling frequency employs 32-bit mark and space FTWs with the binary values shown. In this case, the rightmost nonzero bit of either FTW is the 19th bit, so GRR = 1 GHz/219, or approximately 1907 Hz.

```
Mark   (CH0)    00101010 00100110 10100000 00000000
Space  (CH0)    00111010 11110011 11000000 00000000
GRR    (CH1)    00000000 00000000 00100000 00000000
```

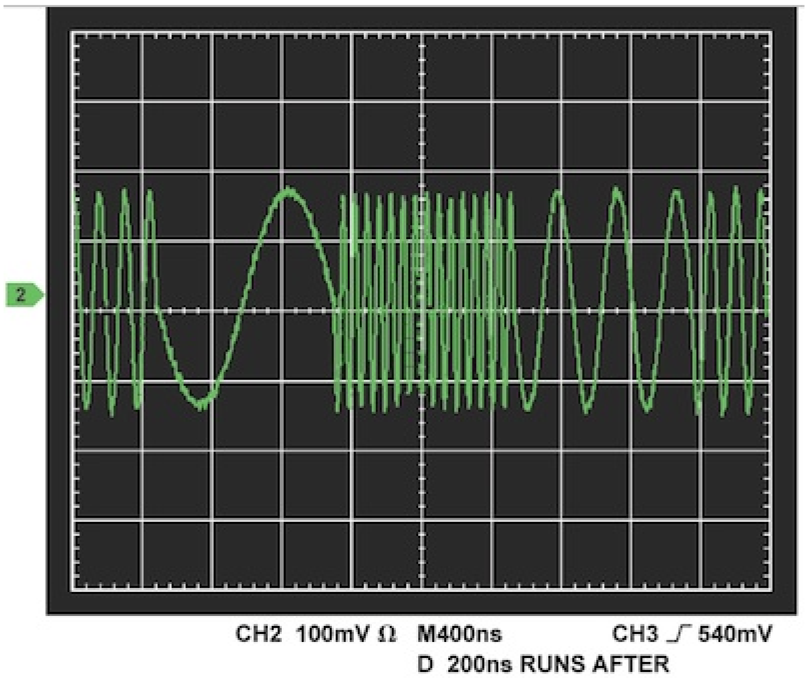
A DDS inherently switches frequency in a phase-continuous manner. This means that no instantaneous phase change occurs when the frequency tuning word changes. That is, the accumulator starts accumulating the new FTW from whatever phase position it was at when the new FTW was applied. Phase coherence, on the other hand, requires an instantaneous transition to the phase of the new frequency as if the new frequency had been present all along. Therefore, in order for a standard DDS to implement a phase-coherent FSK switch, the change from a mark frequency to a space frequency must occur when both frequencies have the same absolute phase. To implement a zero-crossing switch in a phase-coherent manner, the DDS must make the frequency transition at 0 degrees (that is, when the accumulator overflows with zero excess). Therefore, we must determine the instants at which phase coherent zero-crossings occur. If the GRR of the mark and space FTWs are known, the smaller of the two GRRs (if different) will indicate the desired phase coherent zero-crossing point.

Three criteria are necessary for implementing a phase-coherent zero-crossing switch:

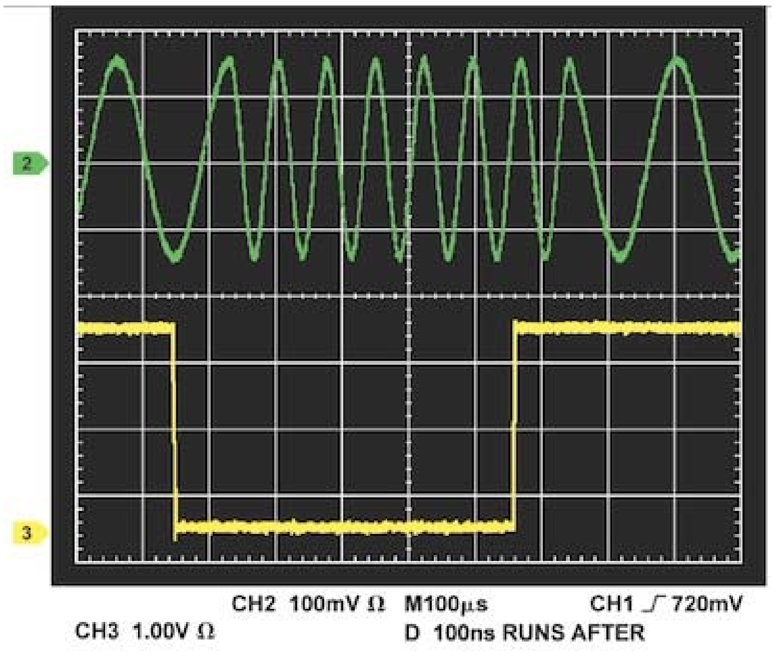
1. The ability to determine the smaller GRR of the mark and space FTWs associated with CH0.
2. A second DDS channel (CH1) synchronized to CH0 and programmed with an FTW having all zeros except for the one bit corresponding to the smaller GRR.
3. The capability to use the rollover of the second channel to trigger a frequency change on CH0.

Unfortunately, the latency between when a DDS accumulator hits zero and when that zero phase is represented at the output further complicates the solution. Fortunately, this latency is constant. The ideal solution necessitates that the auxiliary channel be phase adjusted to compensate for this latency. For instance, both channels on the AD9958 have a phase-offset word that can be used to fix this problem.

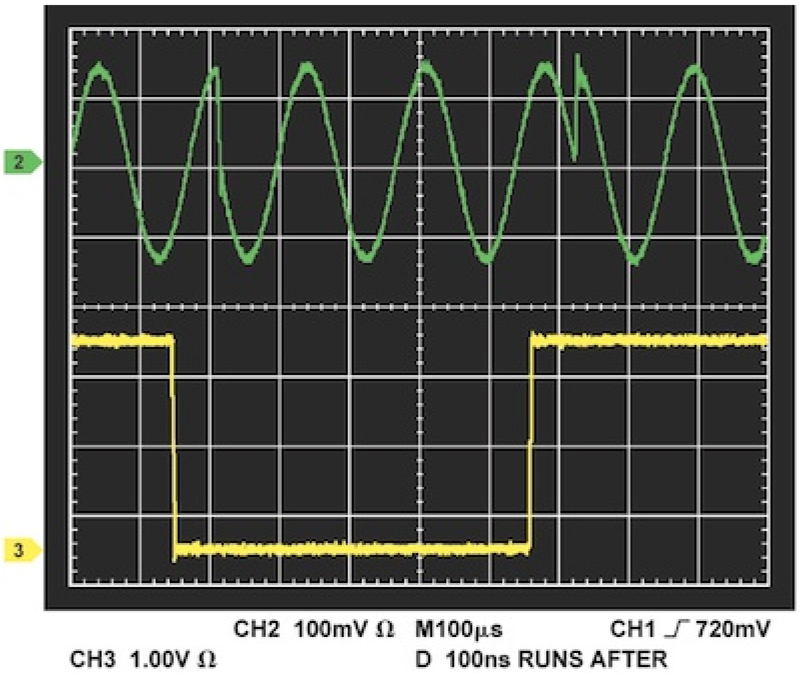
The AD9958 two-channel DDS produced the results shown below.



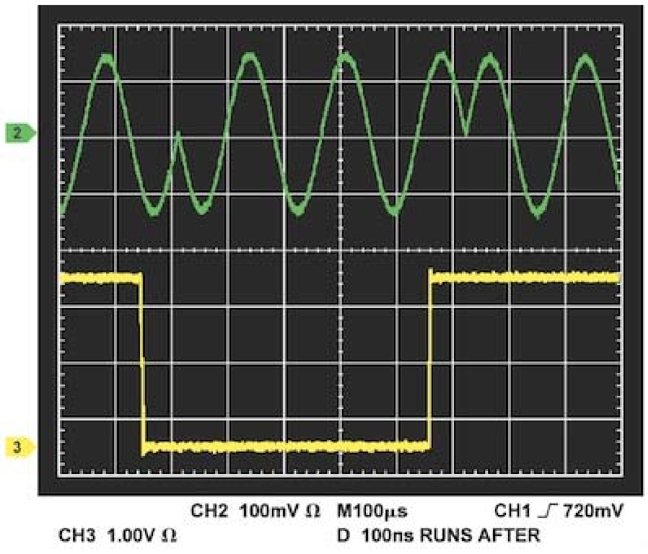
Phase-continuous FSK transition



Zero-crossing FSK transition

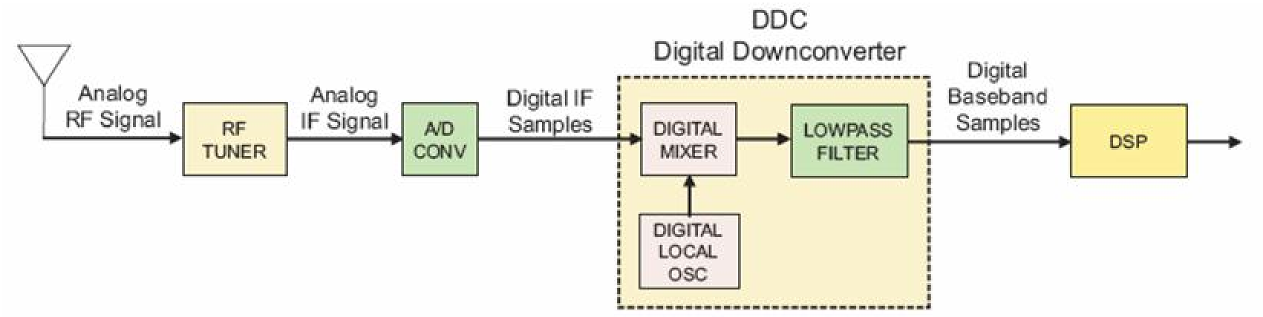


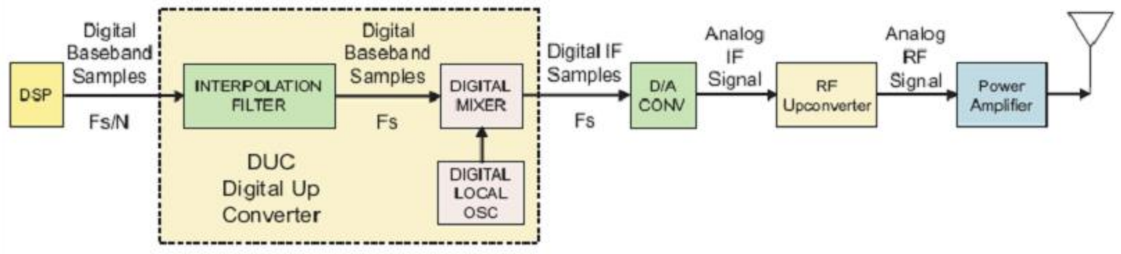
Phase-continuous BPSK transition



Zero-crossing BPSK transition

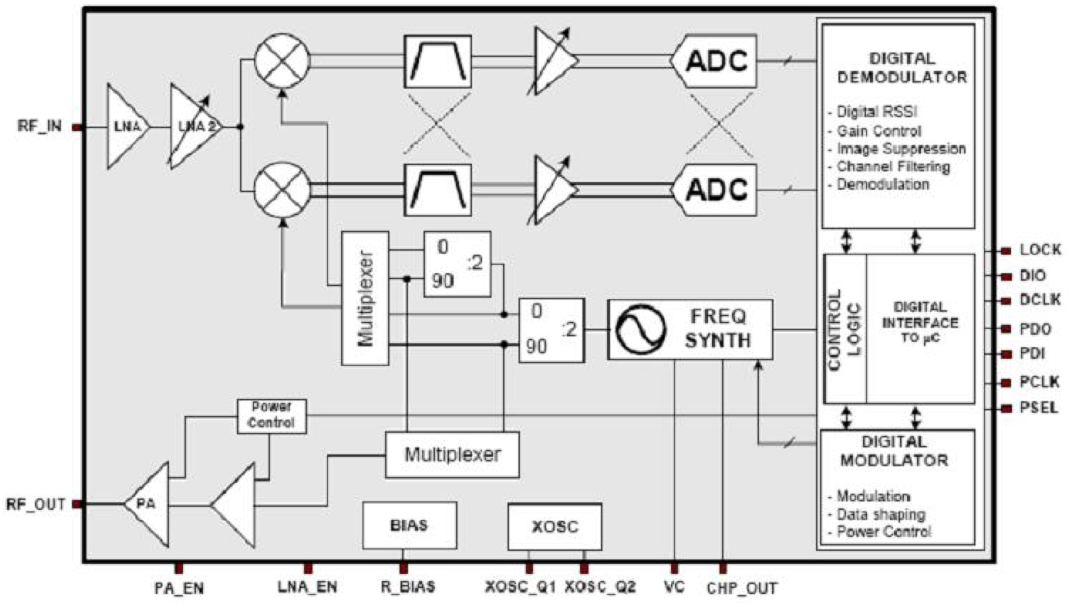
Nowadays Software Defined Radio concept becomes more and more popular due to its advantages - ease of design, reduced design-cycle time, quicker iterations, reduced costs associated with manufacturing and testing radios, multimode operation etc.





Most part of the main functions could be performed digitally, providing significant flexibility and reliability.

Sometimes, it worth to use some standard solutions, like integrated tranceivers.



Doing so, we can avoid some standard problems with tuning and reduce drastically the time of the development.

10 Interference immunity

Interference immunity may be very important, because EMI can affect the device operating characteristics. Honestly speaking, this chapter of the electronics is somewhere positioned close to the black magic. Regardless of this, developer should take into account these problems during all stages of the development. First of all we have to categorize interference.

Interference is a signal, which can get into your device as an electromagnetic wave or via some wiring. Besides that, improper composition of the device or poor PCB layout or some other similar factors can cause interference inside your device. The main rule is that we have to distinguish useful signal and interference. If it is not possible, our chances to win are poor.

For example, if the signal has a limited bandwidth, we can reject outband interference by filtering. If the interference id produced by the certain part of the circuit, we can isolate it by shielding or something like this.

If we can suppose the way, which is used by interference signal to get inside our circuit, we can oppose it by decreasing capacities or inductances or using shielded cables or making proper grounding. By the way, grounding of something on the board of the spacecraft or the aircraft is a typical reason for jokes, but it is a serious problem nevertheless.

We have to understand the impact, produced by the interference signal to our circuits as well. For example, it is well-known, that digital circuits have better immunity, than analog ones. This is true, but analog circuit "forgets" about the interference after it is completed and a digital circuit like memory or register can be switched without restoring previous state. That is why we can provide special protection for such cases.

A lot of issues are connected with improper layout and poor grounding strategy. These figures illustrate the situation.

A lot of problems can be avoided, when the elements for your design are chosen attentively. For example, electrolytic capacitors may reveal poor characteristics as a high frequency element due to its intrinsic structure. You can improve the situation, bypassing it with additional ceramic capacitor. General rule here – Attentively investigate datasheets for the elements to reveal such tips before they become the reason of the failure.

One of the issues, often missed is ESD problem. Electrostatic discharge is kind of interference, produced by the human. ESD can strongly affect electronics reliability. ESD is insidious effect, because it can occur during manufacturing period without any visible signs. You can see the damage caused by ESD on the photo.