THE TECHNICAL UNIVERSITY OF ŁÓDŹ
Faculty of Electrical and Electronic Engineering

Master of Engineering Thesis

SOFTWARE IMPLEMENTATION OF MECHANISMS IMPROVING THE RELIABILITY OF DSP SYSTEMS IN THE RADIOACTIVE ENVIRONMENT

Marcin Wojtczak

Student's number: 106133

Supervisor:
Grzegorz Jabłoński, PhD

Auxiliary supervisor:
Dariusz Makowski, MSc

Łódź, 2005

1. INTRODUCTION 3

2. RADIATION PROTECTION OF THE DSP SYSTEMS 5

2.1 RADIATION INFLUENCE ON ELECTRONIC SYSTEMS 5
2.2 HARDWARE PROTECTION METHODS 8
2.3 SOFTWARE PROTECTION METHODS 10
2.3.1 Parity control 10
2.3.2 Two dimensional parity control 10
2.3.3 Forward error correction codes 12
2.3.4 Reed-Solomon codes 19
2.3.5 Voting techniques 23

3. DSP-PC COMMUNICATION SYSTEM 25

3.1 INTRODUCTION 25
3.2 THE TMS320C6713 DSP OVERVIEW 25
3.3 HARDWARE PART OF THE COMMUNICATION SYSTEM 28
3.3.1 DSP module 29
3.3.2 PC module 30
3.4 SOFTWARE PART 32
3.4.1 Software for the DSP 32
3.4.2 Software for the PC 38

4. IMPLEMENTATION OF RADIATION PROTECTION METHODS IN SOFTWARE 42

4.1 PARITY CONTROL 42
4.2 TWO DIMENSIONAL PARITY CONTROL 44
4.3 FORWARD ERROR CORRECTION CODES 48
4.4 REED-SOLOMON CODES 51
4.5 VOTING TECHNIQUES 52
4.5.1 Triple voting 52
4.5.2 Parallel calculations 52

5. EXPERIMENTAL PROCEDURES CARRIED OUT IN DESY 55

5.1 OVERVIEW 55
5.2 ANALYSIS OF INFLUENCE OF RADIATION ON THE DSP 56
5.2.1 EIA-485 transmission test 56
5.2.2. Internal RAM test 58
5.2.3. DSP ALU test 62
5.3 EXAMPLE APPLICATIONS PROTECTED AGAINST RADIATION 64
5.3.1. FFT filtering of a sound signal 64
5.3.2. Convolution filter (without DSP/BIOS) 68

6. PROJECT RESULTS 73

6.1 MEMORY PROTECTION 73
6.2 PROTECTION AGAINST ERRORS IN CALCULATIONS 74
6.3 SERIAL TRANSMISSION 76

7. CONCLUSIONS 78

REFERENCES 80

APPENDIX A: SCHEMATIC OF THE DSP MODULE 81

APPENDIX B: SCHEMATIC OF THE PC MODULE 82

APPENDIX C: MAIN PART OF THE DSP BOOTLOADER CODE 83

APPENDIX D: HAMMING CODE FUNCTION IN ASSEMBLY 85

1. Introduction

This master's project is carried out in cooperation with Deutsches Elektronen-Synchrotron (DESY) in Hamburg, Germany. DESY is a high-energy physics research centre, where several different accelerators have been built. Also a new linear accelerator is under construction. The accelerator will be controlled by electronic equipment, which will be located in the accelerator tunnel, and therefore it will be under influence of high radiation (mainly gamma radiation and neutrons). It is known that radiation can cause malfunctioning of the electronics and decrease its reliability. This gives raise to a need for further, detailed investigation of how radiation influences various types of electronic devices and how the electronics' reliability can be increased.

Among the different types of electronic devices taken into consideration in development of the control system for the accelerator are Field Programmable Gate Arrays (FPGA) and Digital Signal Processors (DSP). This project is focused on analysis of the radiation on DSP systems. The main aim of the research is to investigate if the reliability of the irradiated DSP system can be increased by using only software methods. More detailed explanation of the influence of radiation on electronics and the theory behind all the software methods implemented in this project are described in Chapter 2.

The DSP system used in this project is based on a commercial off-the-shelf (COTS) equipment consisting of a DSP Starter Kit (DSK) board from Texas Instruments. The board includes:

- a Texas Instruments TMS320C6713 DSP operating at 225 MHz,
- an AIC23 stereo codec,
- 8 MB of synchronous Dynamic Random Access Memory (DRAM),
- 512 KB of non-volatile Flash memory.

The DSK6713 board is provided with Code Composer Studio (CCS) software which is an integrated development environment (IDE) designed specifically to be used with the TI's digital signal processors. This software package includes a compiler, assembler, linker, debugger, profiler and many other tools which enable us to fully take advantage of all the features of the DSP.

For the purposes of the research, a communication system between the DSP placed in the accelerator tunnel (radiation environment) and a PC computer located outside the tunnel had to be developed. The system consisting of two printed circuit boards (PCB) has been fabricated in the Department of Microelectronics and Computer Science (DMCS) at the Technical University of Lodz (TUL). All the hardware and software issues connected with the design of this communication system are described in Chapter 3.

Among the software radiation protection methods implemented in this project are parity calculation algorithms, forward error correction codes (FEC) (for example: Hamming and Reed-Solomon codes) and different voting techniques. The theory behind all these algorithms is explained in Chapter 2 while the implementation details are described in Chapter 4.

The developed system with all the implemented software methods have been tested in DESY in April 2005. The tests were carried out in the Linac II tunnel. Linac II is a linear accelerator, which is used as a source of positrons for the main DESY accelerator – HERA. The electron-to-positron converter located in the Linac II tunnel is a source of high gamma radiation and neutrons. The DSP board was placed approximately 3 m away from the converter, which ensured high system exposure to radiation. The detailed description of the tests is presented in Chapter 5, while all the project results are summarized in Chapter 6.

2. Radiation protection of the DSP systems

In this chapter, radiation influence on electronic systems is briefly explained in general. Next, some most popular hardware protection methods are described. In the last part of the chapter, the theory behind different software methods is explained.

2.1 Radiation influence on electronic systems

In the accelerator tunnel there are two main radiation types that have a substantial influence on all the electronic devices located in the tunnel. The first one is gamma radiation and the second one is neutron radiation.

Gamma radiation is a high energy electromagnetic radiation produced by nuclear transitions. It is a form of ionizing radiation, which means that individual photons that constitute the gamma radiation have enough energy to ionize atoms or molecules. The penetrating capabilities of gamma radiation are significant. To shield from the gamma rays, materials of large atomic number and high density are needed. The effects caused by an ionizing particle striking an electronic system are described later in this chapter.

Neutron influence. Neutron is a particle which has no net electric charge. Together with protons neutrons constitute the nuclei. Because of the fact, that neutrons have no charge, they have very high penetration capabilities. Charged particles and electromagnetic radiation (such as gamma rays) lose their energy when they pass through the matter by ionizing the atoms of the material they pass through. This loss of energy slows them down and eventually stops them. However, neutrons do not take part in the ionization process. They can be stopped only when they hit an atomic nucleus. The probability of such a collision is very small therefore neutrons can travel a long way deep into the material before they are stopped. When a collision of a neutron and a nucleus finally takes place, the atom which was hit is displaced. This causes damage to the silicon lattice of an integrated circuit. As a result, traps or other defects are created. However, high energy neutrons can also cause nuclear reactions within the semiconductor. In these reactions alpha particles are produced. These short range and heavily ionizing products deposit the energy and charge which can cause a single event upset [1].

Generally, the effects of radiation on electronic circuits can be divided into two main categories [2]:

- Total Ionizing Dose (TID),

- Single Event Effects (SEE).

Total Ionizing Dose is a long-term radiation effect caused mainly by electrons and protons or other ionizing particles. It involves a permanent degradation of an electronic device subject to radiation. It is an effect of a cumulative charge deposition in the IC material and it applies to all kinds of electronic devices (e.g. CMOS and bipolar). The degradation of performance can be caused by a change in the threshold voltage (CMOS), increase in current consumption or decrease in transistor gain (bipolar technology).

Single Event Effects are caused by single incidents, when an ionizing particle going through the IC deposits enough energy to cause a change in device operation. SEE can be divided into two main groups:

- soft-errors: Single Event Upsets (SEU), Single Event Functional Interrupt (SEFI),
- hard errors: Single Event Latch-up (SEL), Single Event Burnout (SEB) or Single Event Gate Rupture (SEGR).

Single Event Upsets happen when a given ionizing particle deposits high enough charge in the electronic device to change its state [3]. This is illustrated in the following figure:

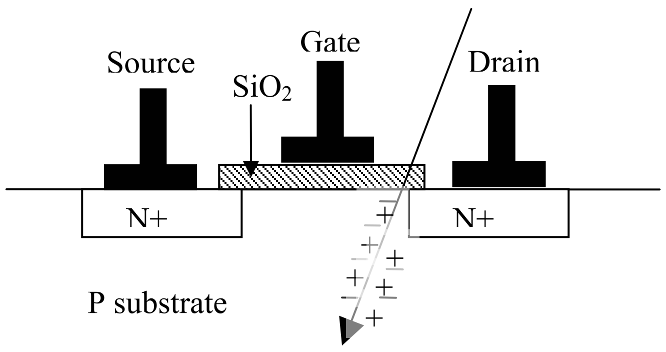


Fig.2.1 An ionization particle in an NMOS transistor

Ionization creates electron-hole pairs in the substrate. This leads to a current spike that can have an important influence on the electronic system. For example, in a SRAM memory cell, this can cause a bit-flip. Figure 2.2 shows a typical memory cell schematic [4]. It is designed in such a way, that there are two possible states of operation. When a '1' is stored, transistors Q1 and Q4 are 'on' while transistors Q2 and Q3 are 'off'. When a '0' is stored Q1 and Q4 are 'off' while Q2 and Q3 are 'on'. Always two transistors are enabled and two are disabled. When radiation particle hits one of the disabled transistors, the states of the transistors change and the memory bit is flipped.

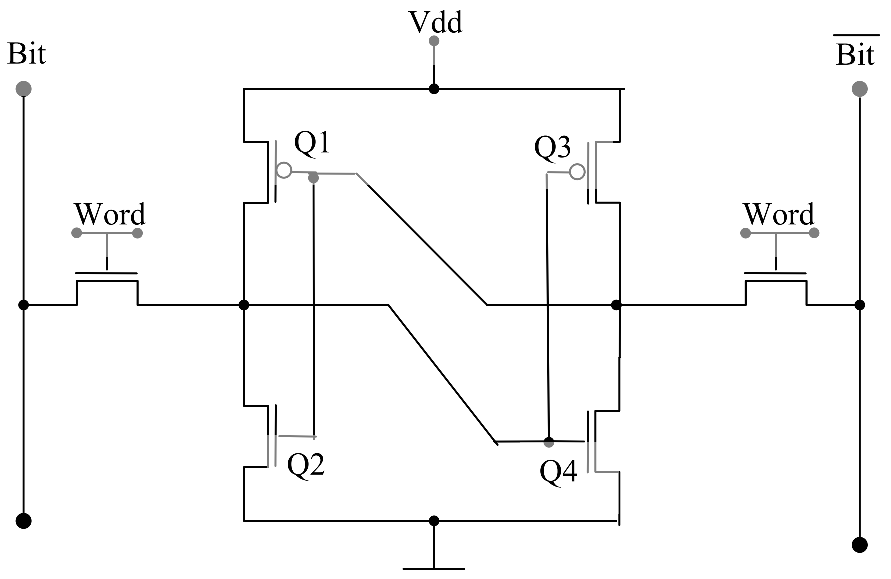


Fig.2.2 Memory cell schematic

SEUs do not damage the device. They only change its state (bit-flip) and therefore after memory rewriting the device works correctly. A SEFI is a severe type of error, in which a SEU in the device's control circuitry puts the device into an undefined state or a halt. This requires a power reset to recover.

Hard errors can damage the electronic device permanently. The most common hard error example is the Single Event Latch-up. This phenomenon is connected with the internal parasitic elements of a CMOS device, which are presented in Figure 2.3 [5].

A p-n-p-n parasitic thyristor can be distinguished. It comprises of two parasitic transistors. The PMOS source, n-substrate and p-well correspond to the emitter, base and the collector of the lateral p-n-p transistor, respectively. The NMOS source, p-well and n-substrate are the emitter, base and collector of the vertical n-p-n bipolar transistor, respectively. The base of each transistor is driven by the collector of the other one and this forms a positive feedback loop. When a radiation particle hits the CMOS structure it deposits a charge which creates a current pulse. If the current gain product of the two parasitic transistors ($\alpha_{npn}\,\alpha_{pnp}$) is larger than 1, latch-up is induced. This produces a large current flow from the power supply to the ground contact, which can damage the device (due to excessive heating) if the power is not turned off immediately.

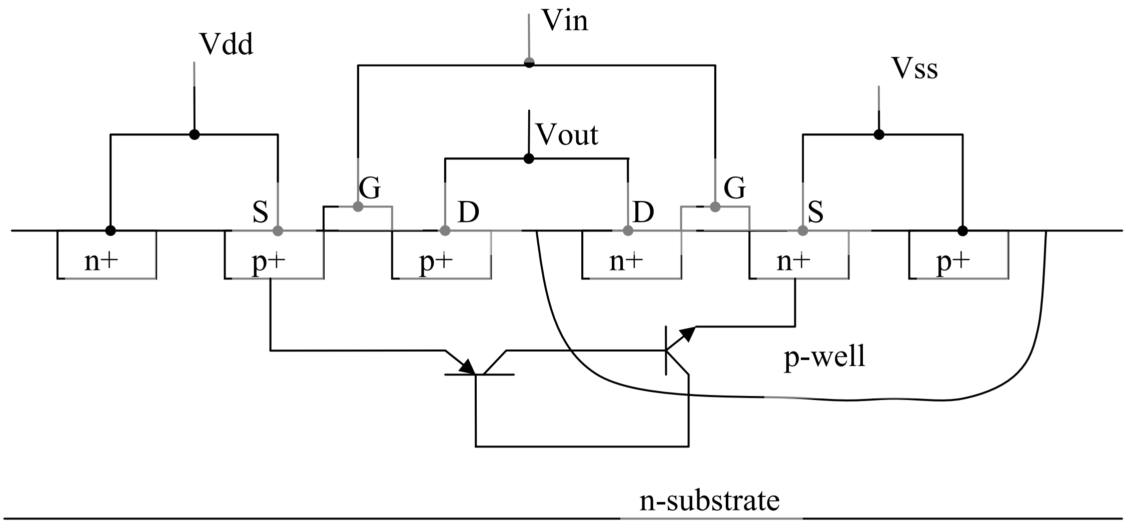


Fig.2.3. CMOS inverter with parasitic transistors

Another type of hard error, Single Event Burnout, can be triggered in a power MOSFET biased in the 'off' state when a heavy ion passing through deposits enough charge to turn the device on. This causes destruction of the device. A power MOSFET may also be subject to a Single Event Gate Rupture, which is a formation of a conducting path (a localized dielectric breakdown) in the gate oxide. This results in a destructive burnout.

2.2 Hardware protection methods

There are several different ways of protecting the electronic equipment against radiation. The simplest one is to use shielding. Shielding can reduce the particle flux considerably but it does not eliminate it completely. The main problem is that shielding is not capable of stopping the neutron radiation. Therefore some additional techniques need to be used.

One of the solutions is to adjust the process technology to produce rad-hard chips. For example an epitaxial bulk CMOS process can avoid the latch-up problem. Even further improvement can be achieved by using the Silicon On Insulator (SOI) technology. This technology process involves building of the transistors on a thin silicon layer, which is placed on top of an insulator. This reduces the capacitances and enables much higher speeds of the devices. It also reduces power consumption and what is most important in terms of radiation tolerance, it eliminates the possibility of a latch-up completely. The thin layer of silicon on top of the insulator also helps to protect the bulk from charged particles, reducing the SEU effect. The main drawback of this technology is the cost.

There are also other possibilities to adjust the device fabrication process to increase radiation immunity. For example, the gate oxide thickness can be decreased. This will decrease the probability of creation of radiation induced trapped charges in the oxide layer and greatly reduce the total ionizing dose effect [6]. Other techniques used in production of rad-hard integrated circuits include low temperature oxidation, oxygen enrichment [7], usage of retrograde wells and guard rings.

Other hardware methods include adjustments in the design of the systems to incorporate protection against radiation. This is usually accomplished by triple modular redundancy (TMR). In this solution the vulnerable parts of the system (for example memory cells, registers, or single flip-flops) are triplicated and additional radiation immune circuit is used to decide which output is the correct one. Figure 2.4 shows an example of a D-type flip-flop realized using triple modular redundancy.

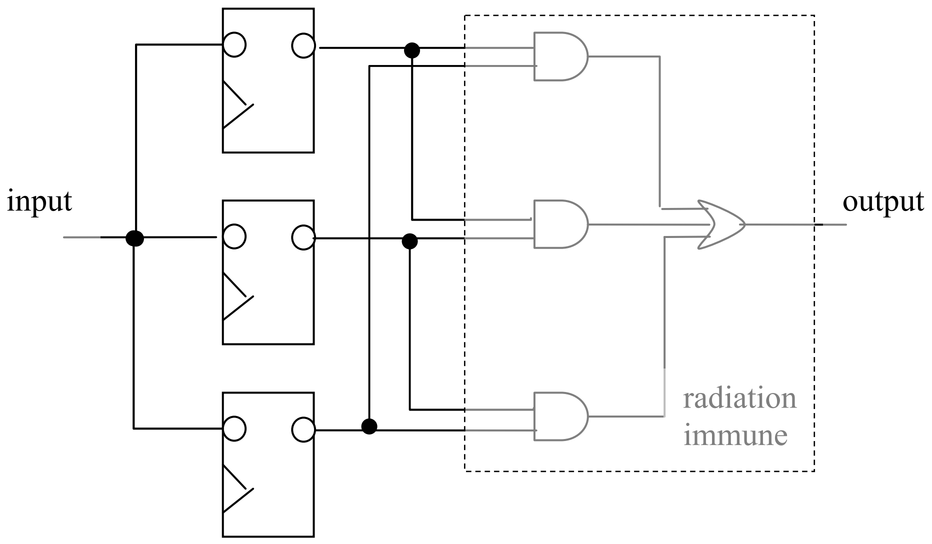


Fig. 2.4 D-type flip-flop with triple modular redundancy and voting

One of the main drawbacks of the solution is a great increase in the number of transistors and silicon area used. The voting circuit itself is also not immune to radiation.

This was just a brief review of some of the simplest hardware radiation protection methods. There exist several other, more sophisticated hardware solutions (for example: hardened memory cells). However, this project is focused on improving the reliability of commercial off-the-shelf (COTS) equipment using only software and therefore these specific, high-cost hardware solutions are not described in detail.

2.3 Software protection methods

The main focus of all the software protection methods is to control the SEUs caused by radiation. SEUs usually affect the memory or internal registers of a processor. Therefore most of the presented methods are focused on protection of a given memory block, which can store either data or code of an application.

2.3.1 Parity control

This is the simplest method and at the same time it is the fastest one. Single parity means adding 1 additional bit to the information bits. The value of the added bit should be such that the sum of all 1's in the information bits along with the one added bit should be even. Therefore, when an error occurs and a single bit is flipped, the total number of 1's becomes odd. This method is capable of detecting an odd number of bit errors. This is illustrated by the following table:

Table 2.1 Single parity control examples

	Information bits with parity bit	**Total number of ones**	**Errors**
original signal:	1 0 0 0 1 0 1 0 1	4 (even)	no errors
1 error introduced:	1 0 0 $\underline{1}$ 1 0 1 0 1	5 (odd)	detected
2 errors introduced:	1 $\underline{1}$ 0 $\underline{1}$ 0 0 1 0 1	4 (even)	undetected

As shown in the figure, if there are two errors, then the total number of 1's is still even and the errors go undetected. To calculate the number of 1's in a given memory block a XOR operation can be performed. This simple technique is very widely used in telecommunication and electronics. It is also implemented in this project.

This method is undoubtedly the fastest of all that have been implemented in this project. Unfortunately, it does not have the ability to correct any errors. Thus it is not well suitable for protection of data, since even if the errors are detected, there is no way of retrieving the original data. However, this method can be useful for protection of code, because when the errors are detected, the code can be copied from the Flash memory.

2.3.2 Two dimensional parity control

The idea of single parity presented in the previous section can be easily extended to achieve better error protection. An interesting way of modification of the original single parity calculation algorithm is to calculate the parity checksums in two directions: vertical and horizontal direction. This is explained on the Figure 2.5. The memory bits are arranged in 32 bit words. The parity bits are calculated for each word (row) and for each column.

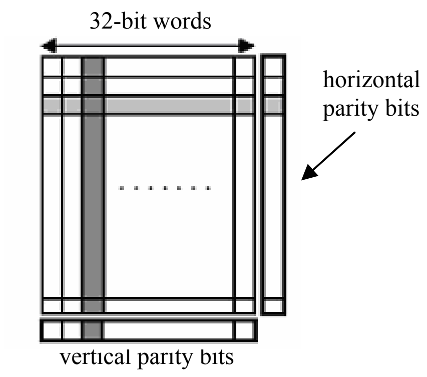


Fig.2.5 Two dimensional parity

Now, there is a much greater number of additional parity bits. Beside error detection, there is also a possibility to correct a one-bit error. Detection of errors that were previously undetectable is now also possible. For example, if there are 2 bits flipped in 1 column, the parity bit for this column does not indicate an error, but two horizontal parity bits show that an error has occurred. This is shown on the figure:

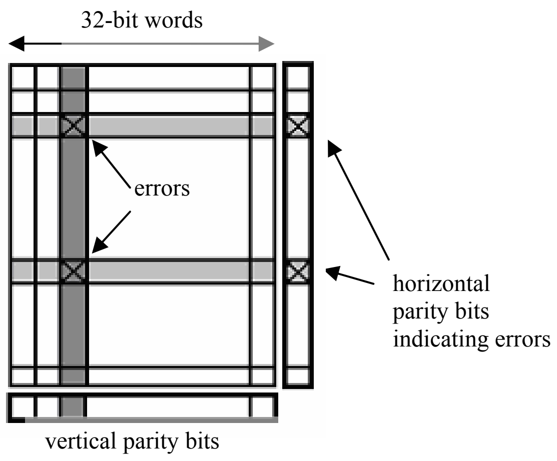


Fig.2.6 2D parity example

The 2D parity method can be implemented in a very fast way, which is explained in section 4.2. Its main advantages are the speed, small redundancy and the ability to correct single bit errors.

2.3.3 Forward error correction codes

A much more sophisticated method for memory protection is known as Forward Error Correction (FEC) codes. The main idea behind FEC codes is to add redundant bits to the available information bits and thus obtain a message word in which some bit errors can be detected or even corrected. To explain how these codes work, **a finite field arithmetic** needs to be introduced [8], [9].

The finite field arithmetic is denoted by GF(q) (Galois Field of order q). A field is an arithmetic structure in which after performing addition, multiplication and division on the members of the field, obtained results are also members of that field. Because there are only finite numbers of elements in a given field, the rules for addition and multiplication have to be changed. In computer memory binary numbers are used. They have only two elements: 0 and 1. Therefore the order q of the field is equal to 2 and the used field is denoted by GF(2). Table 2.2 shows how addition and multiplication is defined in the GF(2) arithmetic:

Table 2.2 Addition and Multiplication in GF(2) arithmetic

Addition		
+	0	1
0	0	1
1	1	0

Multiplication		
x	0	1
0	0	0
1	0	1

As it can be seen from Table 2.2 the addition operation is the same as a XOR operation and multiplication stays the same as ordinary multiplication. Subtraction in this finite field arithmetic is performed by addition. This can be easily explained. In normal arithmetic, subtraction from a number is achieved by addition of the additive inverse of the number to be subtracted. And the additive inverse of a number is defined as a number which, when added to the number itself, results in zero.

For example, to subtract *a* from *b* the inverse of *a* is added:

$$b - a = b + c$$

where *c* is the inverse of *a*:

$$a + c = 0 \qquad => \qquad a = -c$$

In GF(2) arithmetic both members of the field (0 and 1) are their own inverses as when added to themselves they result in zero:

$$1 + 1 = 0$$

$$0 + 0 = 0$$

In case of division, in classic arithmetic to divide a number *A* by *B* we multiply *A* by a multiplicative inverse of *B*. The multiplicative inverse of a number is defined as a number by which multiplication gives a result equal to 1. As it can be seen from the Table 2.2 a multiplicative inverse of 1 is equal to 1 and the multiplicative inverse of 0 is not defined (the same as in classic arithmetic).

Similarly, scalar product can be also defined in GF(2) arithmetic. Let there be two vectors *x* and *y* which are two binary *n*-tuples. The elements of these vectors belong to the GF(2). The vectors are denoted as follows:

$$x = [x_1, x_2, ..., x_n] \text{ and } y = [y_1, y_2, ..., y_n]$$

The scalar product of the two vectors is defined as:

$$x.y = x_1 . y_1 + x_2 . y_2 + ... + x_n . y_n$$

where addition (denoted by "+") is the modulo-2 addition and multiplication (denoted by ".") is performed bit by bit without carry according to the rules from Table 2.2.

There are two important terms that need to be defined before the FEC codes are explained in detail. The two terms are a weight of a code and a Hamming distance.

Weight of a binary codeword *c* is defined as the number of 1's in the codeword and is denoted as $w(c)$. For example, weight of x=100111 is $w(x)$=4.

Hamming distance is defined as the number of bit positions in which two binary sequences differ. For example, vectors x_1=10010 and x_2=00111 have a Hamming distance of 3.

Hamming distance between two vectors can also be calculated as a weight of a sum of the two vectors. In general, for a given FEC code the minimum Hamming distance between any of the codewords of that code satisfies the following relationships:

$$d_{\min} \geq t_1 + 1$$
$$d_{\min} \geq 2t_2 + 1$$

where:

$d_{\min}$ – minimum Hamming distance
t_1 – number of errors that can be detected
t_2 – number of errors that can be corrected

There exist several different types of FEC codes. The most popular, and the ones used in this project are the **block codes**. Block codes are codes where *k* consecutive information bits are encoded into blocks of *n* bits where $n>k$ by adding $n-k$ bits, which are called the parity bits. If the parity bits are added to the end of the information bits, then the given code is called a **systematic** block code. If the parity bits are inserted between the information bits, such a code is called **non-systematic** block code.

In this project, the systematic codes are used. This is mainly due to the fact that the memory region which is protected by the codes may contain an executable code of the DSP and inserting parity bits in this code would stop the DSP from proper operation.

The ability of FEC codes to correct and detect errors is governed by the following equation:

$$2^{n-k} \geq \sum_{i=0}^{t} C(n,i) \qquad \text{wher e} \;\; C(n,i) = \frac{n!}{(n-i)! \cdot i!}$$

where:

k – number of information bits in a block
n – total number of bits in a block (information + redundant bits)
t – number of bit errors, that a given code can correct
$2t$ – number of bit errors, that can be detected
The presented inequality is called the **Hamming Bound**. If the equality is satisfied, the given code is called a **perfect code**. One of the most popular FEC codes is the **Hamming** code, which is a single error correcting perfect code. Since the Hamming code can correct

one error, thus t = 1 and substituting it into the Hamming Bound equation gives the following relation:

$$n = 2^{n-k}$$

A given FEC code is usually denoted as an (n, k) code. For example, the most popular Hamming code is a (7, 4) code, which means that for every 4 bits, there are 3 more parity bits added which yields codewords of 7 bits. In such a codeword 1 error can be corrected or 2 errors can be detected.

To explain the idea of FEC codes, let us assume that a given message which is to be protected from errors is a k-element vector d. The codeword which is created by adding additional parity bits to the original message is an n-element vector c. In order to obtain the codeword vector c from the data vector d the following operation must be performed:

$$c = d \cdot G$$

where G is a k by n matrix called the **Generator matrix** defined as follows:

$$G = \underbrace{\left[\begin{array}{cc} \overbrace{1000\ldots0}^{k\ \text{elements}} & \overbrace{p_{11}p_{21}\cdots p_{(n-k)1}}^{(n\text{-}k)\ \text{elements}} \\ 0100\ldots0 & p_{12}p_{22}\cdots p_{(n-k)2} \\ 0010\ldots0 & p_{13}p_{23}\cdots p_{(n-k)3} \\ \ldots\ldots\ldots\ \ldots & \ldots\ldots\ldots\ \ldots\ldots\ldots\ . \\ \underbrace{0000\ldots1}_{I_k} & \underbrace{p_{1k}p_{2k}\cdots p_{(n-k)k}}_{P} \end{array}\right]}_{n\ \text{elements}}$$

Fig.2.7 The Generator Matrix

The equation is called the **coding equation**. This kind of Generator matrix is used in the systematic codes. It can be partitioned into two matrices: I_k and P:

$$G = [I_k, P]$$

where I_k is a (k by k) identity matrix and P is a (k by (n-k)) parity matrix.

The **decoding equation** is given as follows:

$$c \cdot H^{\mathrm{T}} = 0$$

where H^{T} is the transpose of the **parity check matrix *H*** defined as:

$$H = [P^{\mathrm{T}}\, I_{(n-k)}]$$

To define an FEC code, it is enough to define either *G*, *H* or *P* matrix, as the other two matrices can be derived from it. There is no systematic way of designing the matrices. However there are some conditions that have to be met to obtain a working FEC code. For a 1 error correcting and 2 errors detecting code, the minimum distance $d_{\min}$ of the code must be equal or greater than 3. Since all the codewords of the code are generated by multiplying the data bits by the *G* matrix, each row of *G* matrix must have at least three 1's. Therefore the *P* matrix must have at least two 1's in each row. Furthermore, each row of the *P* matrix must be different.

The decoding equation is very important as only the valid codewords (codewords without any errors) satisfy it. Upon receiving a message which may contain errors, the received codeword *r* should be substituted into the equation:

$$s = r \cdot H^{\mathrm{T}}$$

where *s* is called the **syndrome** and *r* is the received codeword which is equal to the transmitted codeword plus the errors which may have occurred in the transmission medium:

$$r = c + e$$

and substituting this into the syndrome equation gives:

$$s = (c + e) \cdot H^{\mathrm{T}} = c \cdot H^{\mathrm{T}} + e \cdot H^{\mathrm{T}}$$

From the decoding equation it is clear that the first term on the right hand side is equal to zero, thus the syndrome is equal to:

$$s = e \cdot H^{\mathrm{T}}$$

Therefore, it is clear that the syndrome depends only on the errors introduced. If the syndrome calculated on the receiver side is equal to 0, it means that there were no errors in the message. If the syndrome is not equal to 0, then errors are detected. However, it may happen that the number of errors was greater than the detectable number of errors for a given code and the erroneous codeword is the same as one of the valid codewords. Then the syndrome would be equal to 0 and the errors would be undetected.

It is possible to prepare a table, in which for a given error vector, a corresponding syndrome value is calculated. Then, upon receiving a message, the syndrome is calculated. Then, there are 4 possible cases:

1) there were no errors in the message => syndrome is equal to zero
2) there were some errors and they can be corrected => syndrome is not equal to zero, but it can be found in the prepared table and then the corresponding error vector can be added to the received message; this will correct the errors and yield a valid codeword
3) there were some errors, too many to be corrected, but they can be detected => syndrome is not equal to zero and it cannot be found in the prepared table
4) the number of errors was greater than the number of detectable errors => syndrome may have any value, the errors are not detected nor corrected

An example (7, 4) Hamming code can be defined by the following *P* matrix:

$$P = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

This yields the following *G* and *H* matrices:

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \quad H = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

And a following table of the syndrome values for given error vectors can be prepared:

Table 2.3 Error vectors and corresponding syndromes for (7, 4) Hamming code

Error vector	**Syndrome value**
1000000	011
0100000	101
0010000	110
0001000	111
0000100	100
0000010	010
0000001	001

The table contains only 1 bit error vectors, because the (7, 4) code can correct only 1 bit errors. Now, assuming that a data vector d = [0011] is given, the codeword can be calculated:

$$c = d \cdot G = [0011001\ ]$$

If the same codeword is received, then syndrome is equal to zero. However, if the second bit in the codeword is flipped, the received vector r=[0111001] and the syndrome is equal to:

$$s = r \cdot H^{\mathrm{T}} = [0111001\ ] \cdot \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = [101]$$

The calculated syndrome can be easily found in the prepared table, in the second row, and therefore the error vector is found: e=[0100000]. Adding the error vector to the received codeword yields the correct (transmitted) codeword:

$$e + r = [0100000\ ] + [0111001\ ] = [0011001\ ]$$

From the calculated codeword, the data bits can be extracted (first 4 bits from the left): 0011, which is exactly equal to the data vector d assumed at the beginning.

The presented example shows how a Hamming code can be used to correct errors. This code can correct 1 error and detect 2 errors (see the Hamming Bound equation). However, it cannot detect and correct errors at the same time. The user has to decide whether the errors should be corrected or detected.

There exists a solution that enables simultaneous detection of 2 and correction of 1 error. To achieve this, one additional parity bit must be added to the codeword. Such a code is called a modified or extended Hamming code. The additional parity bit value is set in such a way that the total number of 1's in the whole codeword (n bits) is even. Then, when a message is received, the following possibilities exist:

p = the total number of 1's in the received codeword

s = calculated syndrome

1) if p is even and s is equal to zero => no errors present
2) if p is even and s does not equal to zero => 2 errors are detected
3) if p is odd and s is equal to zero => 2 errors are detected
4) if p is odd and s does not equal to zero => 1 error is detected and can be corrected

The presented FEC codes can be implemented either in hardware or in software. They can be used for protection of memory or internal registers of a DSP. In this master's project, a (39, 32) code has been implemented in software, because of the 32bit registers of the DSP. The implementation details are described in Chapter 4.

2.3.4 Reed-Solomon codes

Reed-Solomon (RS) codes are very popular in telecommunication and electronics. Their main advantage over other types of error correcting codes is the ability to correct burst errors. This is mainly due to the fact that a Reed-Solomon code is word oriented rather than bit oriented. It treats all bit errors within 1 word as a single error. Therefore, if for example an 8-bit word RS code is used and it is capable of correcting 2 errors, then in the best case it can correct a total of 16 one-bit errors.

Reed-Solomon codes are a particular case of non-binary BCH codes, which belong to the family of cyclic FEC codes [10]. In these codes finite field arithmetic is used. Hamming codes presented in previous section used finite field arithmetic GF(2). In RS codes, Galois fields of order 2^m are used: GF(2^m). Usually, a field of order 2^8 is used.

An RS code can be specified by 2 parameters:

m – number of bits in a word
T – number of errors it can correct within 1 block
Usually m=8, therefore each word in a code consists of 8 bits (1 byte). The number of words in a block is equal to $N=2^8-1 = 255$. Out of these 255 words $2T$ words are used as parity (or check) words. The rest of the words are used to store data, the number of data words is denoted by K. Therefore $2T=N-K$. A compact notation for a given RS code is denoted as (N, K, T). For example, a (255, 223, 16) code is an Reed-Solomon code that in each 255 bytes block has 223 bytes of data and 32 parity bytes. This code can correct 16 errors or 32 erasures. Erasure is an error, which location is known. There exist also shortened RS codes. These codes are used when 255 byte blocks are too big and are not needed. Then a code of shorter block size can be defined. In such a situation, the remaining bytes are filled with zeros. For example, in an (208, 192, 8) code the remaining 255-208=47 bytes are not used and filled with zeros on the encoder and decoder side.

Reed-Solomon codes are based on Galois field arithmetic for fields of order 2^m, denoted as GF(2^m). To create elements of such a field, a primitive polynomial must be chosen. A primitive polynomial is an irreducible polynomial of degree m which divides $x^{2m-1}+1$. A most popular primitive polynomial for m=8 is $f(x)=x^8+x^4+x^3+x^2+1$. Assuming that α is a root of the primitive polynomial, all non-zero elements of the field can be represented as 2^m-1 consecutive powers of α: $1, \alpha, \alpha^2, \alpha^3, \dots , \alpha^{255}$.

Addition in GF(2^8) is carried out by a XOR operation. However, multiplication involves much more computations. To multiply two element of GF(2^8), Log and Antilog tables need to be used, because a product of two values is the exponent of the mod (GF-1) sum of their logarithms.

To encode a given data sequence using Reed-Solomon code, the k data bits are treated as a data polynomial of order k-1 with its coefficients equal to 1 or 0. For example, data bits 10011 can be expressed as a polynomial $d(x) = x^4+x+1$. A codeword is equal to the data polynomial multiplied by the generator polynomial, which order is equal to n-k. However, this kind of encoding produces a non-systematic code. To achieve a systematic encoding, the data polynomial must be multiplied by x^{n-k} and then the remainder of a division of the data polynomial raised to the x^{n-k} power by the generator polynomial must be added:

$$c(x) = d(x)x^{n-k} + (d(x)x^{n-k} \bmod g(x))$$

$c(x)$ – codeword polynomial

$d(x)$ – data polynomial

$g(x)$ – generator polynomial

Therefore every valid codeword is divisible by the generator polynomial.

The Reed-Solomon decoding process is quite complicated and comprises of the following steps:

1) Syndrome calculation

   A Reed-Solomon codeword has $2T$ syndromes that depend only on errors (not on the transmitted code word). The syndromes can be calculated by substituting the $2T$ roots of the generator polynomial $g(x)$ into the received polynomial $r(x)$.

   If all the syndromes are equal to zero, then no errors are present in the received message and the decoding is finished. If any of the syndromes is not equal to zero, then the next steps need to be processed.

2) Finding the locations of the errors

   This procedure involves solving simultaneous equations with T unknowns. This process can be divided into two sub-steps:

   a) Finding an error locator polynomial $\lambda(x)$

   The most efficient way of doing this is to use the Berlekamp-Massey algorithm. Another approach is to use the Euclid's algorithm, which is easier to implement, but less efficient. The Berlekamp-Massey algorithm for finding the error locator polynomial $\lambda(x)$ consists of the following steps:

   1. Let the syndromes be denoted S_1, S_2, S_3, ..., S_{2T}

   2. Initialize the algorithm variables: k=0, $\lambda^{(0)}(x) = 1$, $L = 0$, and $T(x) = x$, where k is the degree of $\lambda(x)$ at this iteration.

   3. Set $k = k+1$, Compute the discrepancy $\Delta^{\mathrm{k}}(x)$ as follows:

$$\Delta^k = S_k - \sum_{i=1}^{L} \lambda_i^{k-1} S_{k-1}$$

4. If $\Delta^k = 0$, then go to step 8

5. Modify λ polynomial as follows: $\lambda^k(x) = \lambda^{(k-1)} - \Delta^k T(x)$

6. If $(2L >= k)$ then go to step 8

7. Set $L = k$ and $T(x) = \lambda^{(k-1)}(x)/\Delta k$

8. Set $T(x) = x.T(x)$

9. If present iteration k is $< 2T$ then go to step 3

b) Finding the roots of the λ polynomial

The roots of the error locator polynomial are in fact the reciprocals of the error locations. Finding these roots is done using the Chien search algorithm. This algorithm simply substitutes each of the elements of the field into the error locator polynomial until all the roots are found.

3) Finding the values of errors

In this step, a Forney algorithm is usually used. In this algorithm, an error magnitude polynomial is defined:

$$\Omega(x) = [1 + S(x)]\lambda(x)$$

where:
$\Omega(x)$ – error magnitude polynomial
$\lambda(x)$ – error locator polynomial
$S(x)$ – syndrome polynomial defined as: $S(x) = S_1x+S_2x^2+\ldots+S_{2t}x^{2t}+S_{2t+1}x^{2t+1}+\ldots$
And the error magnitudes are given by the following equation:

$$e_{ik} = \frac{-X_k\Omega(X_k^{-1})}{\lambda'(X_k^{-1})}$$

where:
$\lambda'(x)$ – first derivative of the error locator polynomial
$\Omega(x)$ – error magnitude polynomial
X – error locations
Conceptually, the Reed-Solomon code encodes the message as points in a polynomial plotted over a finite field. The coefficients of this polynomial are the data symbols of the block. The plot overdetermines the coefficients, which can be recovered from subsets of the plotted points. In the same sense as the eye can recognize and correct a couple of "bad"

points in a smooth curve, Reed-Solomon code can correct errors in a block of data and recover the coefficients of the originally plotted polynomial.

2.3.5 Voting techniques

Another method of protection of a DSP from the effects of radiation is voting. This method can be used either for protection of memory or to ensure proper result of calculations. The idea behind this method is very simple. The data or calculations to be protected need to be repeated and then the final result is obtained by voting.

The most commonly used configuration of voting is the triple modular redundancy. In this case the calculations are repeated 3 times. Another possible approach is to repeat everything 5 times. Increasing the number of repetitions even further is possible but rather inefficient. Another approach is to repeat the calculations only once, and if the two results are different, then repeat the procedure once again. Figure 2.8 shows a schematic diagram of the triple voting and double calculations:

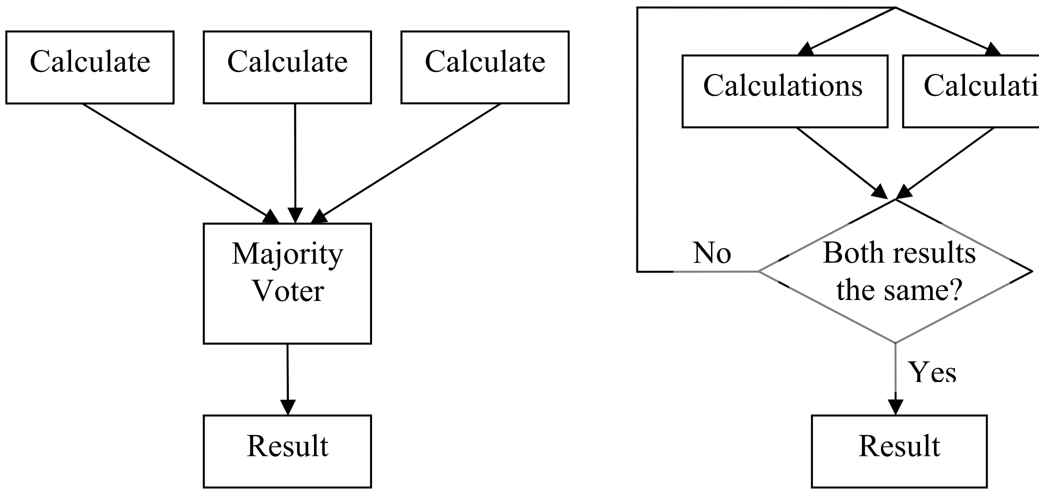


Fig.2.8 Triple voting and double calculations schematic diagrams

The main drawback of these methods is that if an error is introduced in the voter or during the results comparison, an incorrect result can be obtained even if all the calculations were correct. Therefore, the voter should be immune to radiation. However, total radiation immunity cannot be achieved in software.

In the case of triple voting, two different techniques can be used. One approach is to compare the 3 values treating each of them as a whole entity. In this case, if all three of them are different, then voting cannot be performed and this technique fails. Another solution is to compare the three values bit by bit. There are only two possible values for a bit: 0 or 1. Therefore on each bit position, there will be always a situation where two of the three bits are the same. This technique will always produce a result and never will the result be undecided.

An example comparing the two approaches:

Assume that the three calculation results are as follows:

A = 19 (binary: 00011001), B = 91 (binary: 10010001), C = 18 (binary: 00011000).

In this case voting method comparing the whole values will not yield a result, as all three values are different. However, the bit-by-bit approach will produce a result equal to 25 (binary: 00011001).

Some implementation issues concerning the voting techniques are described in section 4.5, while example programs that used these techniques and were tested in the radioactive environment are presented in section 5.3

3. DSP-PC communication system

3.1 Introduction

Analysis of the radiation influence on the digital signal processor requires that the processor itself should be put inside the accelerator tunnel, where the radiation is present. Because of this, a system that would enable remote control of the DSP was needed. This is depicted in Figure 3.1.

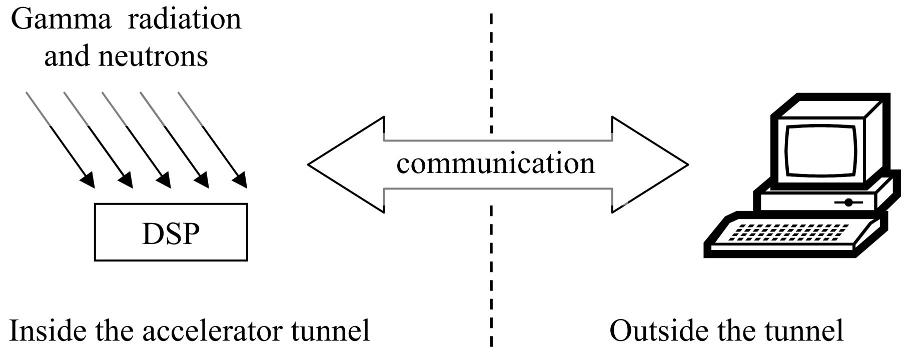


Fig.3.1 DSP-PC communication system

The main requirements for the communication system between the DSP and a PC computer are as follows:

- remote control of the DSP (turning it on and off),
- remote reset of the DSP,
- execution of any given program on the DSP,
- implementation of a watchdog,
- gathering information about the performance of the DSP in the radioactive environment,
- the communication distance of about 50 m.

This chapter describes all the details about the developed system (hardware and software).

3.2 The TMS320C6713 DSP overview

The DSP used in this master's project is a C6713 processor, which is one of the most powerful floating-point digital signal processors produced by Texas Instruments. It has a very interesting architecture consisting of a total of 8 independent functional units:

- Two ALUs (Fixed-Point),
- Four ALUs (Floating- and Fixed-Point),
- Two Multipliers (Floating- and Fixed-Point).

It also has 2 sets of 16 32-bit registers. This is illustrated in Figure 3.2. [11]

The main benefit of such architecture is the fact that this processor can carry out up to 8 different instructions in parallel in 1 CPU cycle. This can be achieved if each of the 8 instructions utilizes a different functional unit. The following is an example of an assembly code illustrating this:

```
       ADD.L1  A0,A1,A2       ; A2 = A0+A1 – addition using L1 unit
||     ADD.L2  B0,B1,B2       ; B2 = B0+B1 – addition using L2 unit
||     SUB.S1  A3,A4,A5       ; A5 = A3-A4 – subtraction using S1 unit
||     SUB.S2  B3,B4,B5       ; B5 = B3-B4 – subtraction using S2 unit
||     LDW.D1  *A6,A7         ; load a 32-bit word from address A6 into A7 – unit D1
||     LDW.D2  *B6,B7         ; load a 32-bit word from address B6 into B7 – unit D2
||     MPY.M1  A8,A9,A10      ; A10 = A8*A9 – multiplication using M1 unit
||     MPY.M2  B8,B9,B10      ; B10 = B8*B9 – multiplication using M2 unit
```

To perform instructions in parallel the "||" characters have to be placed at the beginning of the line. If the functional unit is not specified, the assembler will allocate a proper unit automatically.

Another feature worth explaining is the possibility to write conditional assembly instructions. This is realized by putting a register in square brackets before the instruction:

```
[A0]   MVK.S1  A0,B0          ; this operation will be performed if A0 is different from 0
[!B0]  ADD.S2  B1,B2,B3       ; this is executed if B0 is equal to 0
```

In such conditional statements only A1, A2, B0, B1 and B2 registers can be used. In a group of instructions performed in parallel, each instruction can have a different condition.

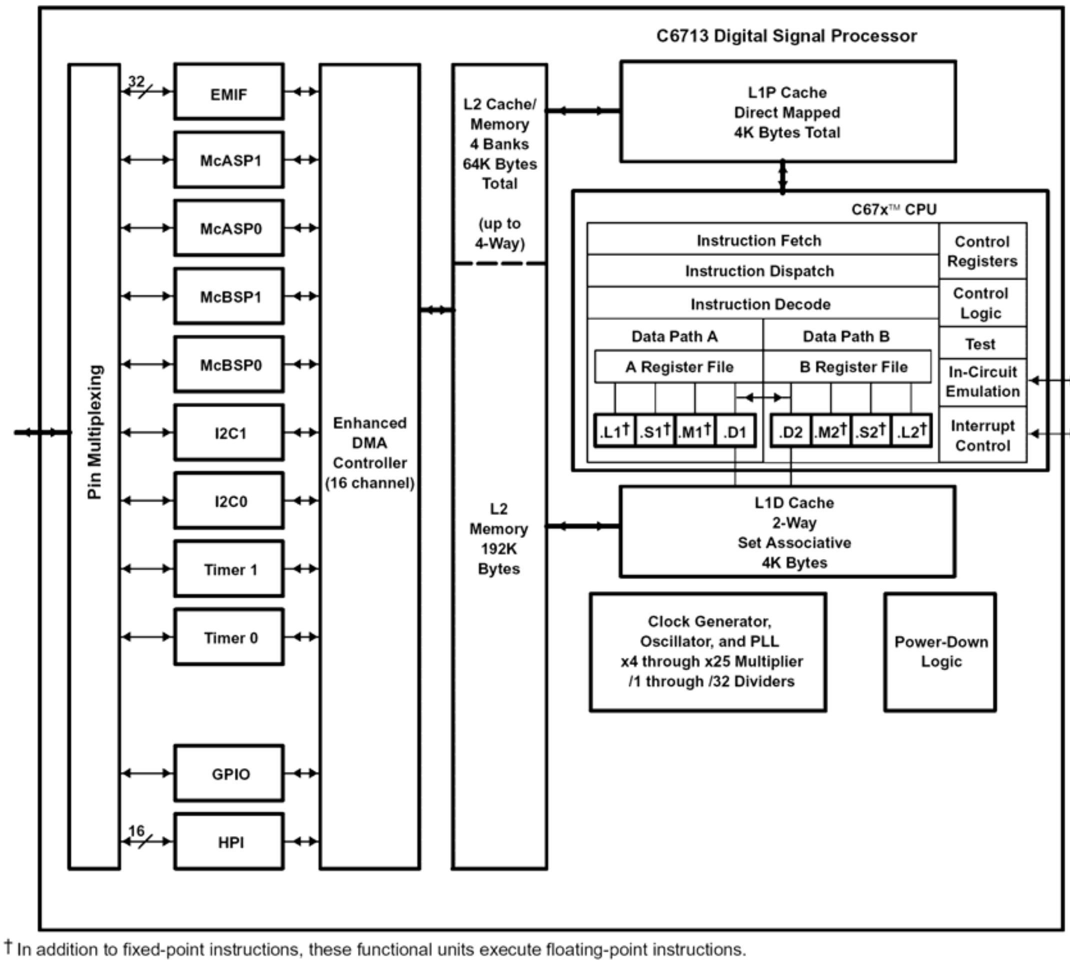

C6713 Digital Signal Processor
Pin Multiplexing
32
16
EMIF
McASP1
McASP0
McBSP1
McBSP0
I2C1
I2C0
Timer 1
Timer 0
GPIO
HPI
Enhanced DMA Controller (16 channel)
L2 Cache/ Memory 4 Banks 64K Bytes Total (up to 4-Way)
L2 Memory 192K Bytes
L1P Cache Direct Mapped 4K Bytes Total
C67x™ CPU
Instruction Fetch
Instruction Dispatch
Instruction Decode
Data Path A
Data Path B
A Register File
B Register File
.L1† .S1† .M1† .D1
.D2 .M2† .S2† .L2†
Control Registers
Control Logic
Test
In-Circuit Emulation
Interrupt Control
L1D Cache 2-Way Set Associative 4K Bytes
Clock Generator, Oscillator, and PLL x4 through x25 Multiplier /1 through /32 Dividers
Power-Down Logic
† In addition to fixed-point instructions, these functional units execute floating-point instructions.

Figure 3.2 C6713 functional block and CPU (DSP core) diagram

Another issue that should be explained is the fact that some instructions take more than 1 cycle to execute. In such cases the result of an instruction is available a few cycles later. These are called "delay slots". For example, a branch instruction has 5 delay slots, which means that the jump in the program is made 5 cycles after the command has been processed:

```
MVK     0xFF,A2
B       LABEL1          ; branch to LABEL1
MVK     0x0,A0          ;
MVK     0x20,A1         ;
ADD     A0,A1,A1        ;
SUB     A1,A5,A2        ;
NOP                     ; branch takes place here
MV      0x0,A2          ; this instruction is not processed
```

In the above code all 5 instructions (MVK, MVK, ADD, SUB, NOP) are executed and then the program counter jumps to the LABEL1 label. Numbers of delay slots required by the most frequently used instructions are listed in Table 1.1.

Table 3.1 Delay slots for some example instructions

Instruction	Description	Delay slots
LDW	load from memory	4
B	branch	5
MPY	integer multiplication	1
ADDSP	single-precision floating-point addition	3
MPYSP	single-precision floating-point multiplication	3
ADDDP	double-precision floating-point addition	6
MPYDP	double-precision floating-point multiplication	9

3.3 Hardware part of the communication system

To fulfill all the requirements of the communication system, both serial and parallel ports of a PC computer had to be used. To meet the 50 m distance requirement it was necessary to use the EIA485/422 serial transmission standard instead of the EIA232 standard which is commonly used in all PC serial ports, but can work only over a limited distance.

The system consists of two PCB boards. The first one is located in the accelerator together with the DSP, while the second one is located near the PC outside the accelerator. The two boards are connected using a 20-wire ribbon twisted cable. The PC-DSP communication system block-diagram is presented in Figure 3.3.

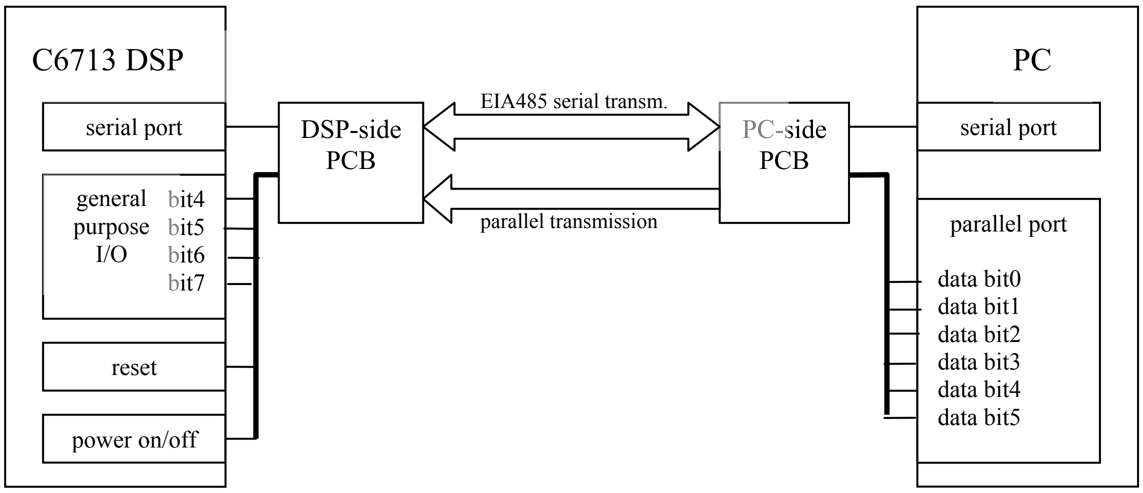


Figure 3.3 Communication system block diagram

The parallel communication is used for:

- turning the DSP on and off (parallel port data bit 5),
- remotely resetting the DSP (parallel port data bit 4),
- additional control signals (parallel port data bits 0-3).

While the serial transmission is used for:

- uploading new programs to the DSP,
- watchdog implementation,
- exchange of additional information (e.g.: about performance of the DSP under radiation, number of detected & corrected errors, etc.).

3.3.1 DSP module

The schematic of the circuit on the DSP side of the system (placed in the accelerator) is presented in Appendix A. Figure 3.4 presents a block diagram of that circuit and a picture of the fabricated printed circuit board connected to the DSK board is presented in Fig.3.5. The circuit connects to the DSK6713 board via two 80-pin connectors.

The first one - peripheral expansion connector – is used for the 4 parallel signals and the serial signals. The 5 V parallel signals are clipped to 3.3 V using the 3.3 V Zener diodes. The EIA422 signals are converted to TTL levels using MAX3485 ICs. The 3.3 V supply voltage for the ICs is taken from the DSK board via the peripheral expansion connector.

The second connector – HPI expansion connector - is used only for the active-low RESET signal.

Additionally, the circuit consists of a DC-DC converter, which has an input range of 9-18 V and output of 5 V (3 A). The converter is needed, because the DSK board has quite high supply current demand (up to 3 A) and therefore the losses would be too big if the 5 V supply voltage were sent to the DSK over the 50 m cable. Therefore, a higher voltage is sent over the cable reducing the current and then it is converted to 5 V using the DC-DC converter.

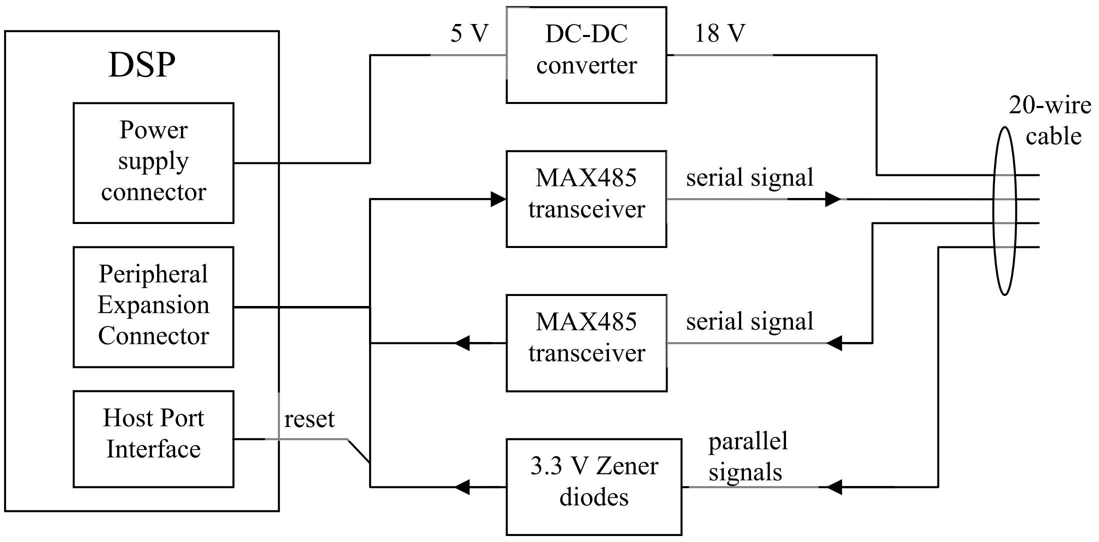


Fig.3.4 Block diagram of the DSP-module

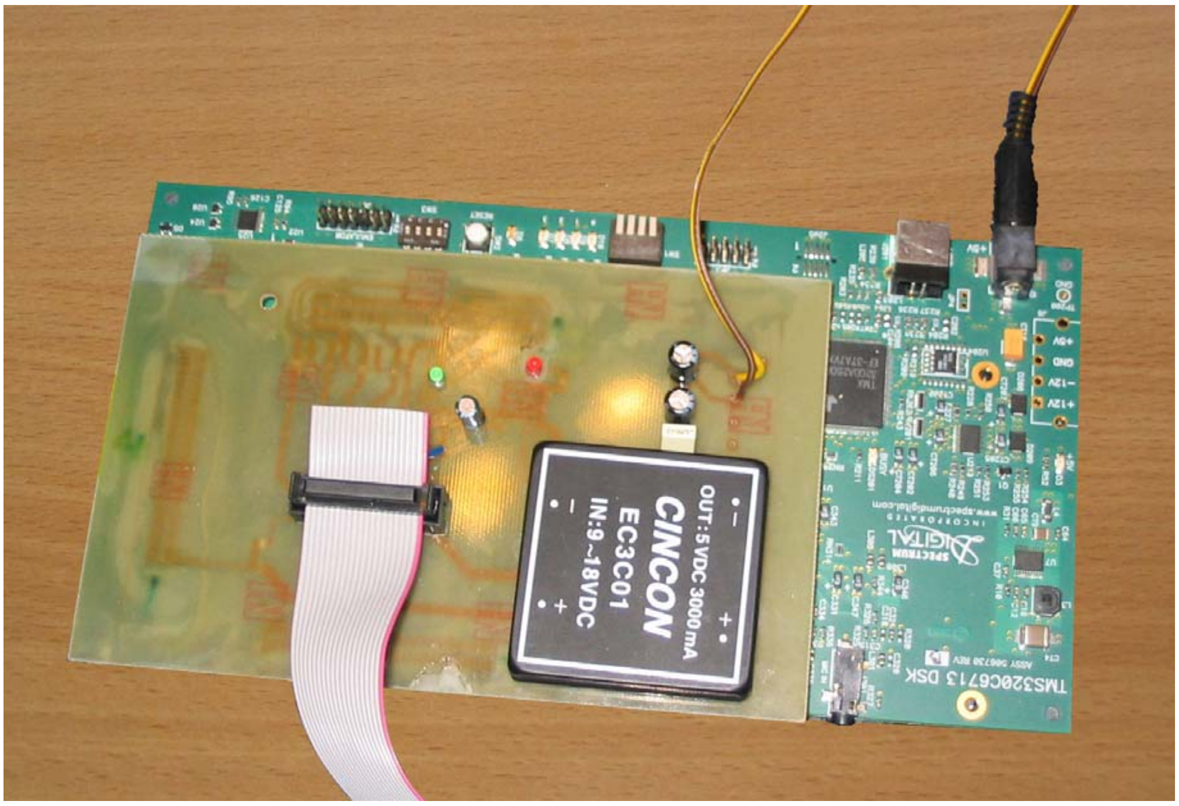


Fig.3.5 DSP-module connected to the DSK board

3.3.2 PC module

The circuit near the PC consists of a 74HC244 octal buffer, a relay, an NPN transistor and several transceivers. Its schematic is presented in Appendix B, while the block diagram is shown in Figure 3.6 and the photo of the fabricated board is presented in Figure 3.7.

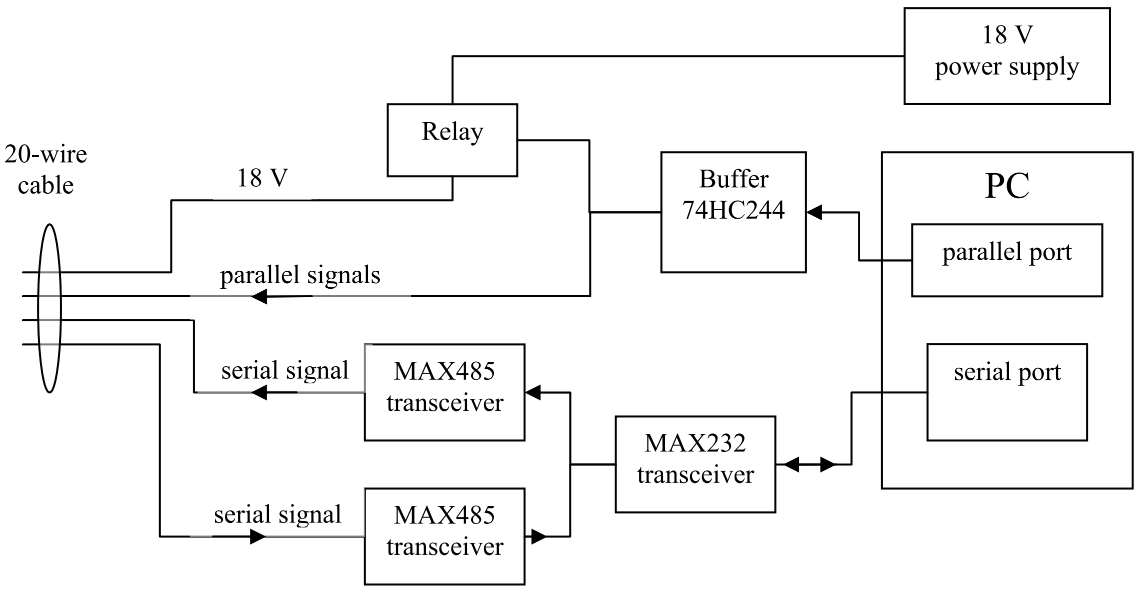


Fig.3.6 Block diagram of the PC-module

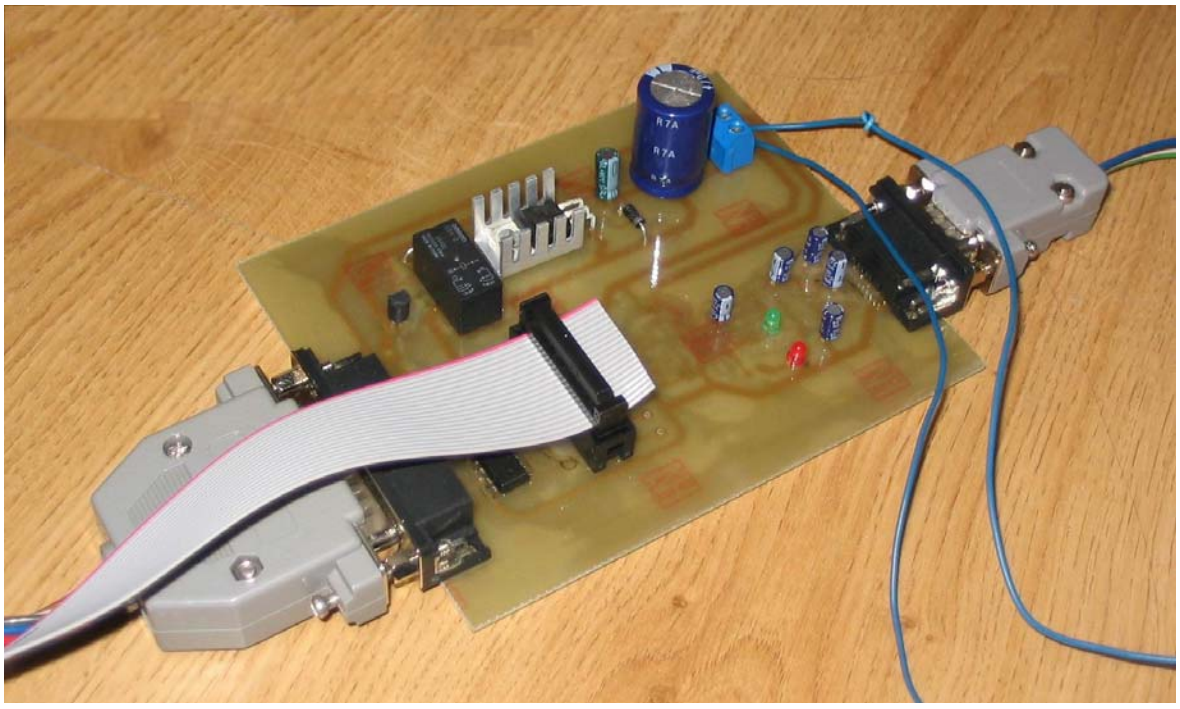


Fig.3.7 PC-module photo

The buffer is connected to the parallel port of the PC and drives the parallel signals to the DSP. Data bits 0-3 of PC's parallel port are used for communication with the DSP, while data bit 4 is used for the DSP's active-low RESET signal. Data bit 5 is used for remote switching of the DSK board on and off. This is accomplished by a relay that switches the supply voltage for the DSK on/off. The relay is controlled by a BC547 NPN transistor connected to one of the outputs of the buffer.

The serial port of the PC is connected to the MAX232 IC which converts the EIA232 signals to TTL levels. Then the signals are converted to EIA485 using the MAX485 integrated circuits. There is a separate pair of wires (and a separate transceiver) for each direction of the signals.

3.4 Software part

The software part also needs to be split between the DSP and the PC. The DSP part of the software is written in C and Assembler using the Code Composer Studio (CCS), while the PC part of the software was written in C++ Builder 6. The PC software works under Windows operating system, because the CCS programming environment also uses Windows.

3.4.1 Software for the DSP

One of the most important requirements for this DSP-PC communication system was to be able to execute any given program on the DSP. This should be achieved by uploading the program code through the serial port. To meet this requirement the Flash memory located on the DSK6713 board had to be used.

Flash was the best choice because it is a non-volatile memory, which means that a given program needs to be uploaded only once and it stays in the Flash until it is overwritten by a different program. The second very important advantage of the Flash is that it is highly resistant to the influence of radiation. The time after which the radiation can damage the contents of the Flash memory is much longer than the time of the tests carried out in this project. Therefore one can be sure that the code executed from flash is not altered by radiation. However, Flash has also some disadvantages. The main drawback is that in order to write some data into the Flash, it has to be erased first. The erasure cannot be performed on a single byte or word. A whole block of Flash must be erased. The Flash memory used in the DSK6713 board consists of 32KB blocks. Another disadvantage is that Flash memory is much slower compared to RAM. Therefore, the time-critical parts of the programs need to be copied to RAM for their execution.

The first 32KB block of the Flash is used for storing the code of a bootloader and the remaining part of the Flash is used to store the code and data of the programs that will be executed on the DSP. This solution perfectly fits with the default boot mode of the C6713 DSP. In this boot mode, right after reset, DSP copies 1KB of memory from the beginning of the Flash to the beginning of the internal RAM (address 0x0) and then branches to that 0x0 address and begins execution of the code. Figure 3.8 presents the Flash memory divided into the 32KB blocks.

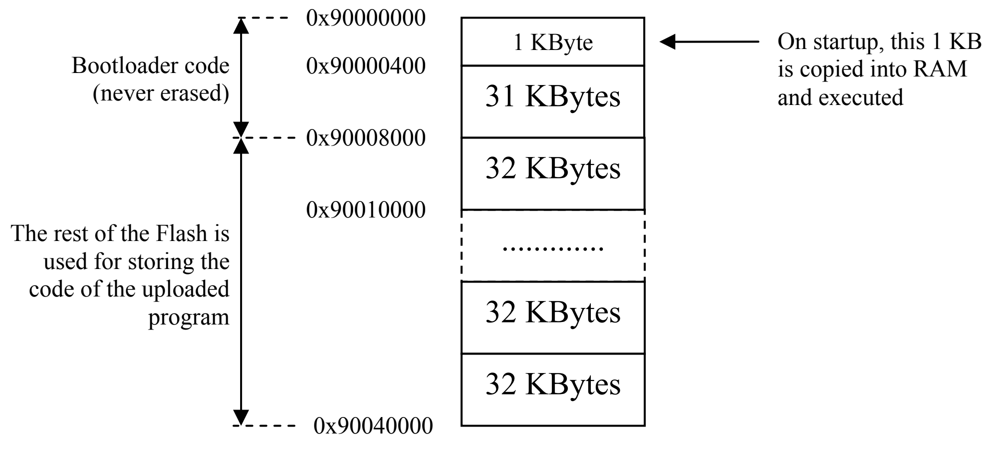

0x90000000
0x90000400
0x90008000
0x90010000
0x90040000
Bootloader code (never erased)
The rest of the Flash is used for storing the code of the uploaded program
1 KByte
31 KBytes
32 KBytes
....
32 KBytes
32 KBytes
On startup, this 1 KB is copied into RAM and executed

Figure 3.8 Flash memory organization

The first 1KB of Flash holds a simple code written in assembly. This code configures the external memory interface, which is used to communicate with SDRAM and FLASH, and configures the general purpose input/output (GPIO) port. Then it reads the fourth bit of the GPIO port. The input pin associated with that bit is connected through the communication system with bit 0 of the PC's parallel port. If the bit is set, then DSP branches to the bootloader code. If the bit is cleared, DSP branches to the address stored at 0x90008000. Therefore, if a user wants to upload a new program to the DSP, he simply needs to turn it on with the bit0 of the parallel port set to 1. Then the bootloader is executed. If the user wants to run a program that has already been uploaded, then the DSP needs to be started with parallel port's bit0 cleared. Then DSP loads the programs entry point from the 0x90008000 address and jumps to the entry point. If the flash is erased and no program is uploaded then the value at 0x90008000 is equal to 0xFFFFFFFF and the DSP enters an infinite loop. This is the most important part of the discussed code (without the EMIF configuration):

```
; GPIO configuration
MVKL 0x0,A0           ;
MVKH 0x01B00000,A0    ; A0 = GPIO address
ZERO A3
MVK 0x00FF,A3
STW .D1T1 A3,*A0++    ; GPEN = 0xFF
MVK 0x0,A3            ; clear all the other GPIO registers:
STW .D1T1 A3,*A0++    ; GPDIR
STW .D1T1 A3,*A0++    ; GPVAL
STW .D1T1 A3,*A0++    ; GPDH
STW .D1T1 A3,*A0++    ; GPHM
STW .D1T1 A3,*A0++    ; GPDL
STW .D1T1 A3,*A0++    ; GPLM
```

```
       STW .D1T1 A3,*A0++    ; GPGC
       STW .D1T1 A3,*A0++    ; GPPOL

       ; read GPIO bit 4
       MVK .S1 0x08,A0              ;
       MVKH .S1 0x01B00000,A0       ; A0 = GPVAL address
       LDW .D1T1 *A0++,A3           ; load GPVAL value
       NOP 4                        ; 4 delay slots for LDW
       EXTU .S1 A3,0x1b,0x1f,A1     ; extract bit0

 [A1]  mvkl    _c_int00,A0          ;
 [A1]  mvkh    _c_int00,A0          ; if (bit0 != 0)
 [A1]  b       A0                   ;      branch to the bootloader
 [!A1] mvkl    PROG_ENTRY_ADDR, A3  ; if (bit0 == 0)
 [!A1] mvkh    PROG_ENTRY_ADDR, A3  ;      load program's entry point address
       nop     3
       ldw     *A3, A2              ; Load entry point
       nop     4
       xor     A2,0xffffffff,A1     ; check if it equals 0xffffffff
 [A1]  b       A2                   ; jump to the program in flash
       nop     5

wait:  b wait                       ; infinite loop
       NOP 5
```

The main part of the source code of the bootloader is presented on the following page. The whole code is placed in the Flash. However, the Flash memory is very slow compared to internal random access memory (IRAM) of the processor; therefore some of the time-critical functions are copied into IRAM during run-time and executed from there. All the parts of the code that need to be run from IRAM are marked as the "fast" section. Then, during linking, this section is linked as if it was placed in IRAM; however it is actually placed in Flash. It is the program's responsibility to copy that section into an appropriate address in RAM. This is simply achieved by the following function call:

```
memcpy( (void *)&run_fast, (void *)&load_fast, (int)&size_fast );
```

where:

run_fast – pointer to the "run address" in IRAM,

load_fast – pointer to the "load address" in Flash,

size_fast – size of the "fast" section.

All these 3 variables are declared and defined automatically by the CCS linker. In the bootloader all the functions responsible for the serial communication, erasing and writing to Flash are placed in the "fast" section.

The main loop of the bootloader consists of copying the "fast" section code from Flash to IRAM, reading data on the serial port and writing that data into Flash. The data is sent from the PC to the DSP in blocks. After each block, a crc32 checksum of the block is sent. DSP calculates the crc32 of the received block and compares it to the received crc32. If these two are not equal, it does not write the block into flash and does not send the acknowledgement (ACK) and waits for a retransmission of the block from the PC. If the crc32's are the same, the data block is written into Flash. Then, crc32 of the data in Flash is calculated again and compared with the received crc32 to make sure that Flash has been programmed correctly. When all the blocks have been correctly received, PC clears the parallel port data bit0 and the DSP goes into an infinite loop. Then the DSK board can be reset and the newly uploaded program can be started. Crucial variables used in the bootloader, such as *serial_speed* and *flash_dest_addr* are stored in 3 copies and always triple voting is carried out on them before they are used. If the triple voting fails (all 3 copies are different) then an appropriate message is sent to the PC over the serial port and the program upload process is stopped. All the messages sent over the serial port are also repeated 3 times to make sure that the correct message reaches the PC. The most important part of the bootloader code is presented in Appendix C.

The DSP program from Appendix C uses the following three functions responsible for the communication over the serial port:

```
unsigned int SoftUartSpeedDetect();  // initialization and detection of transmission speed
char SoftUartInchar( unsigned int serial_speed );          // reading the serial port
SoftUartOutchar(unsigned int serial_speed, char out );     // writing to the serial port
```

The *SoftUartSpeedDetect*() function must be called first in order to configure the DSP's multichannel buffered serial port (MCBSP) correctly and to detect the speed of the serial transmission. In order for this function to work correctly, the PC must first send one byte that begins with a "1" bit followed by a "0" bit. This is because the function measures the time between two consecutive high-to-low transitions in the serial transmission line. This is illustrated on the following figure:

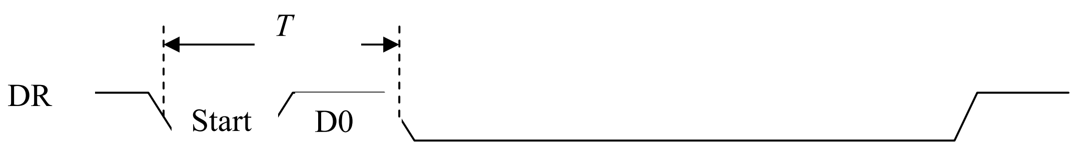


Figure 3.9 Serial transmission speed detection

The first transition occurs at the beginning of the start bit, the second at the end of the first data bit. Therefore *T* is equal to a time for transmission of 2 bits and dividing it by 2 the necessary delay needed to read and write bits over the serial connection is obtained.

The other two functions are used for reading and writing to the serial port with the speed obtained from the *SoftUartSpeedDetect*() function. All three functions are written in assembler and use the MCBSP port in general purpose I/O mode. They are taken from the Texas Instruments (TI) Application Report [12]. An important fact is that the original function *SoftUartSpeedDetect*() from [12] had an error in it. This error made the function completely useless, as it was always returning 0. The error has been located and repaired. It involved changing two assembly instructions. The following is the code of the corrected *SoftUartSpeedDetect*() function with the location of the error marked in comments:

```
_SoftUartSpeedDetect:
;** ---------------- function prolog -------------------------------------*
;** preserve ”save-on-call” registers
       SUB B15, 4, A0
       STW .D2 A10, *B15--[2] ; f
||     STW .D1 B10, *A0--[2] ; f
       STW .D2 A11, *B15--[2] ; f
||     STW .D1 B11, *A0--[2] ; f
       STW .D2 A12, *B15--[2] ; f
||     STW .D1 B12, *A0--[2] ; f
       STW .D2 A13, *B15--[2] ; f
||     STW .D1 B13, *A0--[2] ; f
||     MVC .S2 CSR,B13 ; f
       STW .D2 A14, *B15--[2] ; f
||     STW .D1 B14, *A0--[2] ; f
||     AND .L2 -2,B13,B13 ; f
       STW .D2 A15, *B15--[2] ; f
||     STW .D1 B3, *A0--[2] ; f
||     MVC .S2 B13,CSR ; f disable global interrupts
;** ----------------------------------------------------------------------*
       MVK .S1 0x8,A0 ; set offset to SPCR register
       MVKH .S1 0x18c0000,A0 ; takes McBSP0 port address
       LDW .D1T1 *A0,A3 ; load SPCR register
       NOP 4
       CLR .S1 A3,0x10,0x10,A3 ;
       AND .L1 0xfffffffe,A3,A3 ;
       STW .D1T1 A3,*A0 ; store new SPCR config value
||     MVK .S1 0x24,A0 ; set offset for PCR register
       MVKH .S1 0x18c0000,A0 ; takes McBSP0 port address
       LDW .D1T1 *A0,A3 ; load PCR register
       NOP 4
       SET .S1 A3,0xc,0xd,A3 ; set bit 12&13 for I/O mode
       STW .D1T1 A3,*A0 ; store new PCR config value
       NOP 5
       LDW .D1T1 *A0,A3 ;
       NOP 4
       EXTU .S1 A3,0x1b,0x1f,A1 ; wait while DEIATAT is high
;** ----------------------------------------------------------------------*
       .align 32
L1:    [ A1] B .S2 L1 ;
||     [ A1] LDW .D1T1 *A0,A3 ;
||     EXTU .S1 A3,0x1b,0x1f,A1 ; wait while DEIATAT is high
||     [!A1] ZERO .L2 B4 ; initialize counter
       NOP 5 ; for StartBit measurement
;** ----------------------------------------------------------------------*
       .align 32
L3:    [!A1] B .S2 L3 ;         originally it was:L3:[A1] B .S2 L3         <- error
||     [!A1] LDW .D1T1 *A0,A3 ; originally it was:|| [A1] LDW .D1T1 *A0,A3 <- error
||     EXTU .S1 A3,0x1b,0x1f,A1 ;
||     [!A1] ADD .L2 0x1,B4,B4 ; increment counter while
       NOP 5 ; DEIATAT bit is low
;** ----------------------------------------------------------------------*
       .align 32
L31B:  [ A1] B .S2 L31B ;
||     [ A1] LDW .D1T1 *A0,A3 ;
||     EXTU .S1 A3,0x1b,0x1f,A1 ;
||     [ A1] ADD .L2 0x1,B4,B4 ; increment counter while
       NOP 5 ; DEIATAT bit is low
;** ----------------------------------------------------------------------*
       .align 32
       SHRU .S2 B4,0x1,B4 ;
       MVK .S2 0x0b,B0 ;
       SET .S1 A3,0x5,0x5,A3 ; set DXSTAT bit to 1
||     MV .L1X B0,A4 ;
       MPYLHU .M1X A4,B4,A3 ;
||     STW .D1T1 A3,*A0 ; store new PCR config value
       MPYU .M2 B0,B4,B0 ;
       SHL .S1 A3,0x10,A3 ;
       ADD .L2X B0,A3,B0 ;
;** ----------------------------------------------------------------------------*
       .align 32
waitcnt:      [ B0] B .S1 waitcnt ;
||     [ B0] SUB .L2 B0,0x1,B0 ;
||     [ B0] LDW .D1T1 *A0,A3 ; Dummy load
       NOP 5
; BRANCH OCCUEIA ;
;** ----------------- function epilog ----------------------------------------*
;** restore preserved by call registers
       SUB B15, 4, A0
       LDW .D1 *++A0[2], B3 ; f
||     LDW .D2 *++B15[2], A15 ; f
||     MVC .S2 CSR, B13 ; f
       LDW .D1 *++A0[2], B14 ; f
||     LDW .D2 *++B15[2], A14 ; f
||     OR .L2 B13, 1, B13 ; f
       LDW .D1 *++A0[2], B13 ; f
||     LDW .D2 *++B15[2], A13 ; f
||     MVC .S2 B13,CSR ; f enable global interrupts
       LDW .D1 *++A0[2], B12 ; f
||     LDW .D2 *++B15[2], A12 ; f
       LDW .D1 *++A0[2], B11 ; f
||     LDW .D2 *++B15[2], A11 ; f
||     B .S2 B3 ; f return();
||     MV .L1X B4,A4 ;
       LDW .D2 *++B15[2], A10 ; f
||     LDW .D1 *++A0[2], B10 ; f
       NOP 4 ; f
;** ----------------------------------------------------------------------------*
```

The other two functions are very similar. All the three functions are used repeatedly in all other DSP programs that involve communication with the PC using the serial port.

Another important aspect of the presented DSP application for uploading programs into Flash is the crc32 calculation. This ensures that the received blocks of data are correct, which is very important, as the EIA485 transmission may not be totally radiation immune. In case of large numbers of errors reported during transmission, the speed of transmission can be decreased or some additional protection like Hamming codes or triple redundancy can be introduced to ensure correct transmission. The crc32 calculation function used in the program is as follows:

```
unsigned int crc32_calc(char *fp,int length)    /* calculate the crc value */
{
    register unsigned int crc;

    crc = 0xFFFFFFFF;
    while (length--)
        crc = (crc>>8) ^ crc_table[ (crc^*fp++) & 0xFF ];

    return( crc^0xFFFFFFFF );
}
```

It uses a lookup table which is generated by the following function:

```
static unsigned int crc_table[256];
```

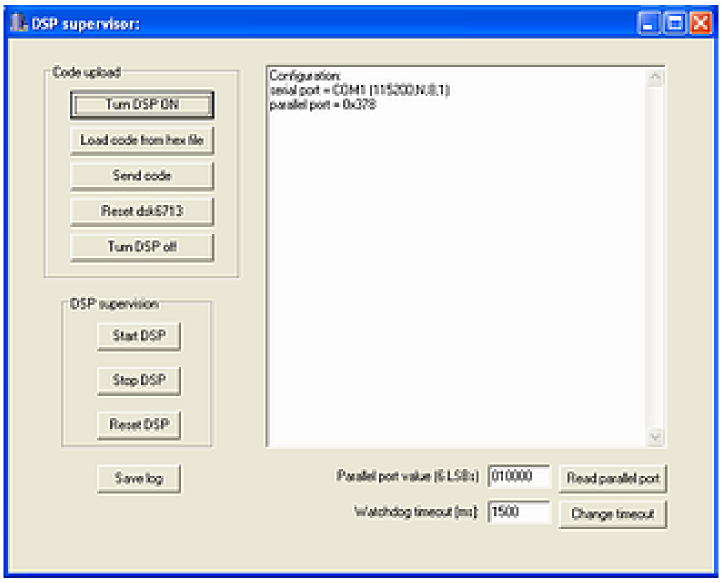
```
void crc32_gen_table(void)                /* build the crc table */
{
    unsigned int crc, poly;
    int i, j;

    poly = 0xEDB88320;
    for (i = 0; i < 256; i++)
    {
        crc = i;
        for (j = 8; j > 0; j--)
        {
            if (crc & 1)
                crc = (crc >> 1) ^ poly;
            else
                crc >>= 1;
        }
        crc_table[i] = crc;
    }
}
```

In the final implementation of the bootloader the *crc_table* is declared as constant and its values are defined in the header file. The two crc32 functions were taken from the internet [13] and needed only small modifications to incorporate into the project.

In brief, crc32 is an algorithm that treats the input message as a very large polynomial, divides it by another large polynomial and the remainder of this division is the crc32 value. The polynomials are created by representing a given decimal value in a binary form and treating each bit as a binary coefficient of a polynomial. This kind of algorithm is very effective, because even very little changes in the message change the division remainder (the crc32 value). Therefore, this method is very useful for ensuring that a given message has not been altered by any errors. The presented implementation of crc32 calculation is a very fast, optimized method. It is based on a lookup table which speeds up the process considerably.

3.4.2 Software for the PC

The software part for the PC is written in C++ Builder 6 and runs under Windows operating system. This is mainly because the Code Composer Studio IDE for the DSP provided by Texas Instruments is also designed to run in Windows environment. The main purpose for the PC software is to remotely control the DSP, upload new programs to the DSP and run them remotely. During the execution of the programs, a watchdog is used to control whether the DSP is working and additional information about detected or corrected errors is gathered. A screenshot of the program created with C++ Builder is presented in Figure 3.10:



```
Configuration:
serial port = COM1 (115200,N,8,1)
parallel port = 0x378
```

Parallel port value (6 LSBs): 010000
Watchdog timeout [ms]: 1500

Fig.3.10 Screenshot of the PC application

There are two main groups of buttons as there are two modes of operation:

a) uploading new programs to the Flash of DSP

In this mode, when a user clicks the "Turn DSP ON" button, the DSP is turned on and a signal via parallel port DATA0 bit is set. Then DSP starts the bootloader and awaits the new application code on the serial port. The user can load the appropriate Hex file with the code to be uploaded. The Hex files can be easily created using the hex6x.exe tool from TI. Finally, the code is sent to the DSP when the user clicks the "Send code" button. The process of sending the code is very simple. The code is divided into smaller blocks and after each block a crc32 value is sent. The function used for calculation of the crc32 is the same as the one used in the DSP and described on the previous pages.

b) running the programs from DSP's Flash memory

In the second mode, when the user clicks "Start DSP" button, DSP is turned on, but the parallel port DATA0 bit is cleared and so the DSP starts execution of a program that is already in Flash. The PC initializes serial transmission with the DSP by sending a 0x0D byte over the serial port. This enables the DSP to calculate the transmission speed (using the *SoftUartSpeedDetect* function). PC waits for an acknowledgement signal from the DSP. The signal should consist of three bytes of value 0xAA. Then, the PC starts a watchdog timer. The DSP should periodically send a special signal over the serial port to reset this watchdog timer. DSP can also send other signals, for example when using Hamming codes, it can send information about the amount of errors corrected or detected.

All this information is displayed on the PC screen. The values of the message codes that a DSP can send are listed in Table 3.2.

Table 3.2 Message codes for DSP-PC communication

Byte value	Meaning
0xF0	Watchdog signal
0x0F	Errors have been corrected by the hamming code. Next, the number of corrected bits is sent.
0xFF	Errors have been detected by the hamming code.
0x3F	Errors have been corrected by the 2D parity. Next, the number of corrected bits is sent.
0xFC	Errors have been detected by the 2D parity.
0x01	Triple voting has failed (all 3 values were different)
0xC3	The two parallel calculations produced different results.
0x3C	Triple voting has been used (1 of the 3 values was different from the others)

Additionally, there is an edit box that lets the user change the watchdog timer interval, and also a button that allows reading of the current parallel port value. There is also a "Save log" button. Pressing it allows the user to save the contents of the log to a file.

In the PC software part, a DLL library for parallel port communication is used. The library file name is inpout32.dll and the whole library along with the source codes can be downloaded for free from the internet [14]. The DLL consists of 2 main functions:

```
short Inp32( short portaddr );
void Out32( short portaddr, short data );
```

The *Inp32* function reads the port of address *portaddr*, while the *Out32* function writes the value of *data* to the port of address *portaddr*.

For the serial port programming, the Win32 API functions are used. The following is a part of the source code responsible for opening the serial port and configuring it:

```
// ##########  SERIAL PORT initialization  ##########

DCB dcbCommPort;

// open the serial port
hComm = CreateFile(serial_port,GENERIC_READ | GENERIC_WRITE,
                   0,0,OPEN_EXISTING,0,0);
```

```
// check if it opened correctly
if (hComm == INVALID_HANDLE_VALUE)
        {
         As.printf("Error opening serial port %s", serial_port);
         Application->MessageBox(As.c_str(),"Error",MB_OK);
         Application->Terminate();
         }

// set the comm timeouts
GetCommTimeouts(hComm,&ctmoOld);
ctmoNew.ReadTotalTimeoutConstant = 1000;
ctmoNew.ReadIntervalTimeout = MAXDWORD;
ctmoNew.ReadTotalTimeoutMultiplier = MAXDWORD;
ctmoNew.WriteTotalTimeoutMultiplier = 0;
ctmoNew.WriteTotalTimeoutConstant = 0;
SetCommTimeouts(hComm, &ctmoNew);

// configure the serial port
dcbCommPort.DCBlength = sizeof(DCB);
GetCommState(hComm, &dcbCommPort);
BuildCommDCB(com_conf, &dcbCommPort);
SetCommState(hComm, &dcbCommPort);
```

Then, to read or write to the port, the *ReadFile* and *WriteFile* functions are used. To send a single byte to the serial port, a *TransmitCommChar* function is used. This function is mainly used for initialization of the connection with the DSP, as the DSP's *SoftUartSpeedDetect*() function needs an arrival of one byte of value 0x0D to calculate the connection speed. On program shutdown, the *CloseHandle* function is used to release the serial port handle.

Programming the serial and parallel ports in C++ Builder is easy and straightforward. The whole program for the PC is quite simple therefore its source code is not presented. One very important fact is that in order for the port communication to work, the program must be run by a user with administrator's privileges on the Windows system.

4. Implementation of radiation protection methods in software

This chapter covers all the implementation issues concerning the software methods described in Chapter 2. All the methods in this master's project were implemented in C or assembly on the TMS320C6713 DSP from Texas Instruments.

4.1 Parity control

This is the simplest method to implement, and therefore it is also the fastest one. Since the C6713 DSP has 32-bit registers, the best way to implement the single parity method is to treat the memory as an array of 32-bit values and calculate the parity bits "vertically". This is illustrated on the Figure 4.1.

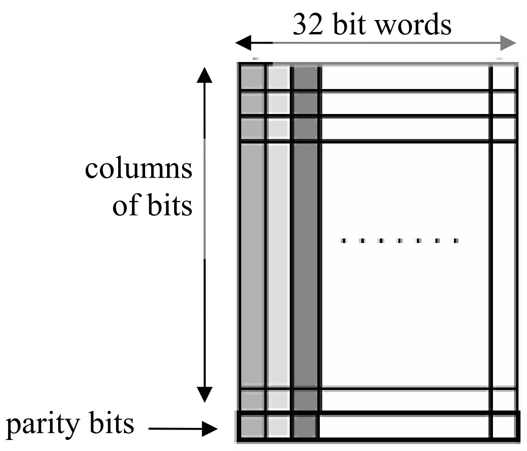


Fig.4.1 Parity control - "vertical" implementation

This kind of implementation has two big advantages:

- the parity of 32-bit columns is calculated in parallel,
- errors in different columns are detected independently.

The presented implementation enables detection of any odd number of bit flips occurring in one column. Each column is treated independently, so for example, 32 bit errors can be detected, if each one of them is placed in a different column.

Since a xor operation requires only 1 CPU cycle, the total number of cycles required to calculate the parity bits for a given memory region is approximately equal to the size of that region given in 32-bit words. The following is the hand optimized assembly code for calculation of the parity bits:

```
_parity_calc
               LDW    *A4++,A1       ; A1 = data pointer
               LDW    *A4++,B2       ; B2 = length
               B      loop
               B      loop
               B      loop
||             ZERO A7
               B      loop
||             ZERO A5
               B      loop
||             SUB B2,1,B2           ; decrease counter by 1

loop           LDW *A1++,A5          ; load the data
||      [B2]   SUB B2,1,B2           ; decrease the counter
||      [B2]   B      loop           ; check if it is the end
||             XOR    A5,A7,A7       ; xor with A7

               B      B3             ; return
               STW    A7,*A4         ; parity_bits = A7
               NOP 4
```

And this is the code of a function checking whether errors occurred:

```
_parity_check
               LDW    *A4++,A1       ; A1 = data pointer
               LDW    *A4++,B2       ; B2 = length (loop counter)
               LDW *A4,A7            ; A7 = parity_bits
               B      loop2
               B      loop2
               B      loop2
||             ZERO A4               ; return value = 0
               B      loop2
||             ZERO A5
               B      loop2
||             SUB B2,1,B2           ; decrease counter by 1

loop2          LDW *A1++,A5          ; load the data
|| [B2]        SUB B2,1,B2           ; decrease the counter
|| [B2]        B      loop2          ; check if it is the end
||             XOR    A5,A7,A7       ; xor with A7

               B      B3             ; return (after 5 cycles)
               MV     A7,A2          ; A2 = result xored with parity_bits
 [A2]          MVK -1,A4             ; if (a2!=0) return -1
               NOP 3
```

The declarations of the two functions are as follows:

```
typedef struct {
int *pointer;          // pointer to the memory region to protect
int length;            // length of the region in 32bit words
int parity_bits;       // parity bits calculated by parity_calc function
} Tparity;

extern parity_calc(Tparity *);
extern parity_check(Tparity *);
```

Both functions use a pointer to a *Tparity* structure as a parameter. This structure is used for description of a memory region that needs to be protected. The region can either consist of some data or a code of the DSP program. Similar structures are used in all other memory protection methods used in this project. The *parity_check* function returns 0 if there are no errors, and -1 when errors are detected.

4.2 Two dimensional parity control

The general idea behind the two dimensional parity is explained in section 2.3.2. The method is based on calculation of the parity bits in two directions: vertical and horizontal. However, this kind of calculation scheme is difficult to implement in software, as calculating horizontal xor operations is very slow (much slower than vertical). Fortunately, there is a solution that enables to keep all the benefits of the 2D parity and still manage to implement it in software in a very fast manner [15]. The method is illustrated in Figure 4.2:

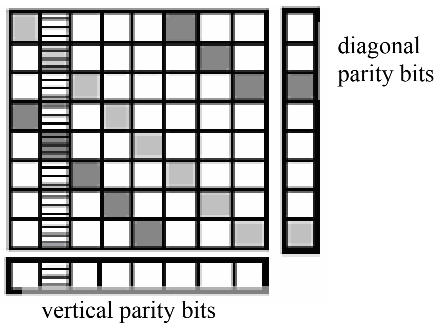


Fig.4.2 2D parity with diagonal bits calculation

In this case the parity bits are calculated in vertical and diagonal direction. The number of additional parity bits and all the benefits are the same as in vertical and horizontal approach, but the software implementation is much easier.

Using this approach, it was possible to implement the method in assembly using only 2-cycle loops. It is possible to write them in 1-cycle loops, but an instruction for bit rotation is needed to calculate the diagonal parity. The TI's C67xx digital signal processors do not have such an instruction; therefore a combination of 3 instructions to perform a bit rotation had to be used. This is explained in Figure 4.3.

However, the TI's digital signal processors from the C64xx family have a bit rotation instruction and therefore the 2D parity functions can be written with 1-cycle loops and work two times faster as the ones for C67xx processors.

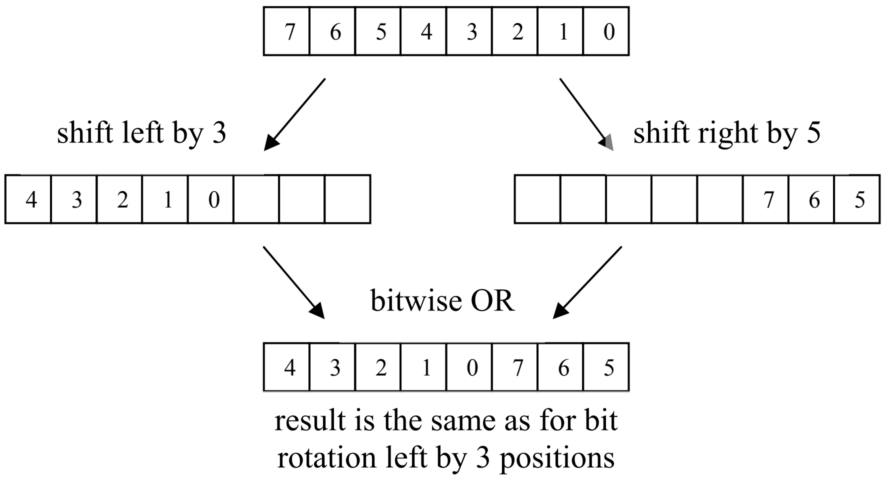


Fig.4.3 Bit rotation example

In the implementation of the algorithm used in this project the protected memory region is divided into smaller blocks of size 32x32 = 1024 bits (128 bytes). Every such block has its own 64 parity bits (32 vertical + 32 diagonal). Therefore in each of the 128 bytes blocks the program can correct 1 single-bit error. However, this error must be inside the memory block and not among the parity check bits. If it is in one of the parity bits, then the error is detected but not corrected.

The 2D parity calculation and check functions have been written in assembly and optimized by hand. The functions prototypes along with the definition of the structure associated with each memory region protected using this method are as follows:

```
typedef struct {
int *pointer;          // pointer to the memory region to protect
int *xor_bits;         // pointer to the memory where parity bits are stored
int length;            // length of the protected region in 32bit words
} Tparity2D;

extern parity2D_calc(Tparity2D *);
extern parity2D_check(Tparity2D *);

int parity2D_init(Tparity2D *block, int length);
void parity2D_close(Tparity2D *block);
```

The two additional functions *parity2D_init* and *parity2D_close* are needed for dynamic memory allocation/deallocation for the parity bits. This is due to the fact that number of parity bits depends on the size of the memory region to protect. The size of this memory region must be a multiple of 4 bytes due to the nature of the algorithm. The following is the source code of the 2D parity functions:

written in C:

```
int parity2D_init(Tparity2D *block, int length) // length is the size of memory in bytes
```

```
{
    div_t temp;

    temp=div(length,128);  // divide by 128 bytes (one block size)
    if ( temp.rem>0 ) temp.quot++;
    block->xor_bits = (int *)malloc(2*temp.quot*sizeof(int)); //allo. mem. for parity bits
    block->length = length >> 2;
    if (block->xor_bits == NULL) return 0;
       else return 1;
}

void parity2D_close(Tparity2D *block)
{
   free (block->xor_bits);
}
```

written in assembly:

```
; ###############################################
_parity2D_calc
               LDW    *A4++,A3       ; A3 = protected memory region pointer
               LDW *A4++,B4          ; B4 = pointer to parity bits
               LDW *A4,A1            ; A1 = length
               NOP 3

big_loop       MVK 32,B0
||             ZERO A7               ; vertical xor will be stored in A7
||             ZERO B7               ; diagonal xor will be stored in B7

               CMPGTU A1,B0,A2       ; if length>32
 [A2]          SUB A1,B0,A1          ;      then { length-=32; b0=31; }
 [!A2]         MV     A1,B0          ;   else {
 [!A2]         MVK 0,A1              ;              length = 0;

               SUB B0,1,B0           ;              b0 = length-1;}

               B   loop3             ; 2x branch, becuase it's a 2 cycle loop

               ZERO B2
||             ZERO A9

               B   loop3
||             MVK -3,A2             ; used for bit rotation (shl)

               ZERO B6
||             ZERO A5
||             MVK 35,B1             ; used for bit rotation (shru)

;2 cycle loop
loop3          LDW.D1  *A3++,A5      ; load the data
||             ADD.L1  A2,1,A2
||             SUB.L2  B1,1,B1
||             OR.S2   B2,A9,B6
|| [B0]        SUB.D2  B0,1,B0       ; decrease the counter
|| [B0]        B.S1    loop3         ; check if it's the end of the loop


               XOR.L1  A5,A7,A7      ; vertical xor
||             XOR.L2  B6,B7,B7      ; diagonal xor
||             SHL.S1  A5,A2,A9      ; SHL,SHR and OR from 1st cycle
||             SHRU.S2 A5,B1,B2      ; make the bit rotation

;epilogue
               SUB     B1,1,B1
||             OR      B2,A9,B6

               XOR     B6,B7,B7
||             SHL.S1  A5,A2,A9
||             SHRU.S2 A5,B1,B2

 [!A1]  B      B3                    ; if (length==0) return
|| [A1] B      big_loop
               STW     A7,*B4++      ; vertical = A7
               STW B7,*B4++          ; diagonal = B7
               SUB A3,8,A3           ; set the data pointer
               NOP 2


; ##############################################
_parity2D_check

               LDW    *A4++,A3       ; A3 = memory to protect pointer
               LDW *A4++,B4          ; B4 = pointer to parity bits
               LDW *A4,A1            ; A1 = length
               ZERO B5               ; B5 = number of corrected errors
               NOP 2

big_loop2      MVK 32,B0
||             LDW    *B4++,A7       ; A7 = vertical xor
               LDW *B4++,B7          ; B7 = diagonal xor

               CMPGTU A1,B0,A2       ; if length>32
 [A2]          SUB A1,B0,A1          ;      then { length-=32; b0=31 )
 [!A2]         MV     A1,B0          ;   else { b0= length-1;
 [!A2]         MVK 0,A1              ;          length=0; }

               SUB B0,1,B0

               B   loop4             ; 2x branch because it's a 2 cycle loop
||             ZERO A4               ; return value

               ZERO B2
||             ZERO A9
||             MVK -3,A2             ; used for bit rotation (shl)
||             MVK 35,B1             ; used for bit rotation (shru)

               B   loop4

               ZERO B6
||             ZERO A5
||             MV     A3,B8          ; copy of the data pointer

;2 cycle loop
loop4          LDW.D1  *A3++,A5      ; load the data
||             ADD.L1  A2,1,A2
||             SUB.L2  B1,1,B1
||             OR.S2   B2,A9,B6
|| [B0] SUB.D2 B0,1,B0               ; decrease counter
|| [B0] B.S1   loop4                 ; check if it's the end of the loop


               XOR.L1  A5,A7,A7      ; vertical xor
||             XOR.L2  B6,B7,B7      ; diagonal xor
||             SHL.S1  A5,A2,A9      ; SHL,SHR and OR from 1st cycle
||             SHRU.S2 A5,B1,B2      ; make the bit rotation

;epilogue
               SUB    B1,1,B1
||             OR     B2,A9,B6

               XOR    B6,B7,B7
||             SHL.S1  A5,A2,A9
||             SHRU.S2 A5,B1,B2

; check if there was an error and repair it:
               OR  A7,B7,A2          ; if vertical=0 & diagonal=0
 [!A2]  B      next                  ; then go to next
               MVK 32,A0
               MV  A7,A9             ; copy of the vertical xor
               LMBD 1,A7,A5          ; left-most bit detection for vertical
||             LMBD 1,B7,B6          ; left-most bit detection for diagonal
               SUB A5,B6,A8          ; diagonal-vertical
||             AND    A5,A0,A2
||             AND    B6,A0,B2
               ADD A5,1,A5
||             ADD B6,1,B6
||             OR     A2,B2,B2
               SHL A7,A5,A7
||             SHL B7,B6,B7
```

```
               OR      A7,B7,A2
               OR      B2,A2,A2       ; if error is in the parity bits
 [A2]   B      B3                     ; or more errors occurred then return -1
               MVK -1,A4
||             CMPLT A8,0,A2          ; if H-V<0 then add 32
 [A2]          ADD A8,A0,A8

               SHL A8,2,A5            ; multiply by 4, because it's 32bit data array
               ADD B8,A5,B8
               LDW *B8,A6             ; load the errored data
               NOP 4
               XOR A6,A9,A6           ; change the bit (correct the error)
||             ADD B5,1,B5
               STW A6,*B8             ; save corrected data
;next
next [A1]      B big_loop2            ; if (length>0) next loop
|| [!A1]       B  B3                  ; else return B5 (number of corrected errors)
               MV B5,A4
               SUB A3,8,A3
               NOP 3
```

To summarize, the 2D parity algorithm is an interesting solution, because it is quite fast and its code is short but has much bigger capabilities then the single parity method. On the C64xx family DSPs the two dimensional parity method can be as fast as the one dimensional method. In cases when errors caused by radiation occur only as single bit flips and they are separated in memory by a large distance (so that they are located in different 128 byte memory blocks) this method seems to be the best solution due to its high speed.

4.3 Forward Error Correction codes

The Forward Error Correcting (FEC) codes which have been explained in section 2.3.3 have been implemented in two ways. The first one uses matrix multiplication to calculate the parity bits. This way it is very easy to define any given Hamming code by simply defining its G or H matrix. This solution enables very easy testing of Hamming codes of different lengths. However its main drawback is that the amount of calculations is very big and therefore the speed of the parity bit calculations is very slow. According to the CCS Profiler, for a memory block of 128 bytes, this method is about 23 times slower than the 2D parity algorithm. However, there is a much better way of implementing the Hamming codes. It does not use the matrices, it simply involves hard-coding the necessary xor operations. This method has been used to implement a (38, 32) code in assembly and it proved to be about much faster than the "matrix implementation". The amount of CPU cycles needed to protect a given memory region was comparable to the number of cycles needed by the 2D parity algorithm.

The chosen (38, 32) code has the ability to correct 1 error or detect 2 errors. However, it cannot do both at the same time. The user has to decide to focus on either error correcting and then be able to correct 1 error, or focus on error detection and detect 2 errors.

However, the code was extended by adding 1 parity bit, which is calculated over all 38 codeword bits and the extended code is able to correct 1 error and detect 2 errors simultaneously.

Both implementations are based on the same idea to divide the protected memory region into blocks of 32x32 = 1024bits (128 bytes) and to calculate the hamming code vertically. This means that for every column of 32 bits 6 parity bits are added (because a (38, 32) code is used). This yields that for a 128 bytes block, additional 8 parity bytes (64 bits) are needed. This kind of implementation has some very important advantages. Firstly, the 32 bit columns are treated as separate messages, thus 1 error can be corrected in each bit column. This gives a total of 32 correctable 1-bit errors in the whole block (assuming that every error is in a different column). The second advantage is of course the speed of calculations, because the 32 codes are calculated and checked in parallel.

The following is the matrix implementation of the (38, 32) code with one additional parity bit. The function that performs the same error checking, but has the xor-ing hardcoded in assembly is presented in Appendix D.

```
int hamm_init(Thamm *block, int length)  // length is the size of protected memory region
{                                        // in bytes, and must be multiple of 4*K
    int rem, quot;

    rem = length % (4*K);
    quot = length / (4*K);

    if ( rem )
    {
        block->no_of_blocks = 0;
        return 0;     // error, wrong length
    }
    block->parity_bits = (int *)malloc((N-K+1)*4*quot); // memory alloc. for parity bits
    if (block->parity_bits == NULL)
    {
        block->no_of_blocks = 0;
        return 0;
    } else
    {
        block->no_of_blocks = quot;
        return 1;
    }
}

void hamm_close(Thamm *block)
{
    free (block->parity_bits);
}

void hamm_calc(Thamm *block)
{
    int j,i,k;

    for (i=0; i<(N-K+1)*block->no_of_blocks; i++)
        block->parity_bits[i] = 0;

    for (k = 0; k<block->no_of_blocks; k++)         // for each block
    {
        for (j = 0; j<(N-K); j++)           // perform the matrix multiplication
```

```
            for (i = 0; i<K; i++)
                block->parity_bits[k*(N-K+1)+j] ^=( block->data[k*K+i] & Ht[i][j] );

        // additional parity bit calculation:
        for (i = 0; i<N; i++ )
            if (i<K) block->parity_bits[k*(N-K+1)+N-K] ^= block->data[k*K+i];
                else block->parity_bits[k*(N-K+1)+N-K] ^= block->parity_bits[k*(N-K+1)+i-K];
    }
}

int hamm_check(Thamm *block)
{
    int j,i,k,error,tot_error,k1,k2;
    unsigned int temp,t,parity;
    int S[32];  // 32 syndromes because 32 parallel hamming codes are calculated

    error = 0;
    tot_error = 0;

    for (k = 0; k<block->no_of_blocks; k++) // for each block
    {
        k1 = k*K;
        k2 = k*(N-K+1)-K;
        parity = block->parity_bits[k2+N];

        for (i=0; i<32; i++) S[i] = 0;

        for (j = 0; j<N-K; j++)      // do the matrix multiplication
        {
            temp = 0;

            for (i = 0; i<N; i++)
                if (i<K)
                {
                    temp^=( block->data[k1+i] & Ht[i][j] );
                    if (j == 0) parity ^= block->data[k1+i];
                } else
                {
                    temp^=( block->parity_bits[k2+i] & Ht[i][j] );
                    if (j == 0) parity ^= block->parity_bits[k2+i];
                }

                for (i = 0; i<32; i++)      // change S from vertical to horizontal
                {
                    t = ((temp >> i) & 0x01);
                    if (t)
                    {
                        S[i] |= ( t << j );
                        error = 1;
                    }
                }
        }
        if (error)    // check if there were any errors
        {
            error = 0;
            for (i = 0; i<32; i++ )  // check all 32 syndromes
                if (S[i])            // if syndrome != 0 -> error
                    if ((parity & ( 0x1 << i )) > 0 )      // 1 error to correct
                    {
                        parity ^= (0x1 << i); // turn off the parity bit

                        j = 0;
                        while (COSET[j++] != S[i])
                            if (j == N) return -1; // more than 1 error occurred
                        j--;
                        // correct 1 error
                        if (j<K) block->data[k1+j] ^= (0x01 << i);
                            else block->parity_bits[k2+j] ^= (0x01 << i);
                        tot_error++;
                    } else return -1; // 2 errors detected
        }
        if (parity) return -1; // not all errors have been corrected
    }
    return tot_error; // return number of corrected errors
}
```

4.4 Reed-Solomon codes

Because of the complexity of the Galois field arithmetic, software implementation of Reed-Solomon codes is not fast. These codes can be quite efficiently implemented in hardware using rather simple circuits with shift registers, but software implementation is much more complicated. The main problem is the implementation of multiplication in GF(2^m). As it was described in section 2.3.4, to multiply two elements of the field, 3 table look-ups and 1 modulo addition must be performed.

The Reed-Solomon encoding and decoding functions used in this project are based on the source code from literature [10].

Because of the extensive number of calculations that need to be performed, the encoding/decoding software routines are very slow. The time needed to check a given memory region for errors (in the case when there are no errors) is comparable to the time needed to copy this memory block from the Flash. If errors are present and are corrected, then the execution time is much longer than the time needed for copying the data from Flash.

Another major drawback of the RS codes, is that their software implementation uses several big arrays to hold some temporary data. In the case of a DSP running in a radioactive environment this is a very serious disadvantage. The RS code is supposed to be used for protection of a given memory region. If the RS code algorithm itself needs to use a large block of memory for its own data, then it becomes vulnerable to the radiation and cannot be used as a reliable method. The 2D parity method does not use memory for storing data at all and the FEC codes use only a few bytes of memory for additional data.

To summarize, the Reed-Solomon codes, although very popular in telecommunication and in hardware implementations, are not an efficient method for software implementation to protect a DSP in a radioactive environment. Their main disadvantages are slow speed compared to the other methods, high usage of memory for temporary data, and large code size. However, the digital signal processors from the C6400 family have the Galois field arithmetic implemented in them. In these processors, just a single assembly instruction is needed to perform multiplication in GF(2^8). Therefore, software implementation of the Reed-Solomon codes on the C6400 DSPs can be much faster and efficient. Texas Instruments presents the RS codes implementation on C6400 processors in the application note [16].

4.5 Voting techniques

4.5.1 Triple voting

As it was described in section 2.3.5, triple voting can be implemented in 2 ways: comparing the 3 values bit-by-bit, or treating them as individual entities (for example: integers). Although the bit-by-bit approach is more effective, in all the programs in this project triple voting was implemented using the second approach. This is due to the fact, that this solution can give more information about the DSP behavior in the radioactive environment. Whenever the voting was actually used (one of the three compared values was different), a "Voting used" message was sent to the PC. The voting routine itself was not protected in any way. The code of the routine and amount of CPU cycles it used was significantly smaller than the resources used by the calculations. Therefore, probability that an error would occur during voting was negligible. However, if during the tests at DESY a need for protection of the voting routine had arisen; this could have been achieved by performing the voting twice.

An example application that utilized triple voting was implemented. This application performs FFT filtering of a sound signal. The program and its performance in radioactive environment are described in section 5.3.1.

Generally, implementation of triple voting is straightforward. The voting part of the program is very simple and fast. The main drawback of this method is that all the operations must be performed 3 times and this introduces a large load to the CPU. Therefore, this method can only be implemented in programs, where the calculation functions are very fast and it is possible to repeat them 3 times and still meet all the performance requirements.

4.5.2 Parallel calculations

In section 2.3.5 a different method of voting is described. In this approach, the calculations are performed only twice. If the two results are different, then the procedure is repeated.

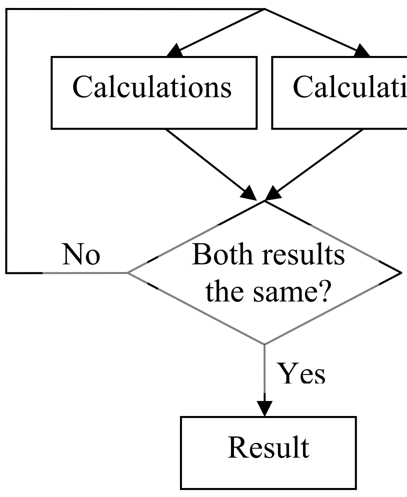


Fig.4.4 Parallel calculations diagram

The architecture of the C6713 DSP allows performing the two calculations in parallel. This way, theoretically, the execution time of operations performed using this method should be comparable to the execution time of single calculations. In reality, situation is a bit different. When the DSP performs only 1 calculation operation at a time (for example calculates the convolution of a signal with a filter's unit impulse response) the optimized assembly code is written in a way to use all the DSP ALUs and resources to enhance the speed of calculations. When the calculations are performed twice in parallel, it is not possible to optimize the code to such a great extent. Therefore, in some cases, it can happen that performing the calculations twice, one after another may be faster than performing them at the same time (in parallel). More detailed analysis of this problem is presented in Chapter 6, where the CPU cycles needed by an example filtering function implemented using this method are presented.

The part of the program that is the most vulnerable to the errors caused by radiation is the comparison of the two results, because this operation is done only once. However, the assembly instructions that perform the comparison of the two results take only 2 CPU cycles. Therefore, probability that an error corrupts this operation is very small.

One very important disadvantage of this method is the fact, that writing the function to perform the same calculations in parallel requires the function to be written in assembly. Therefore, the time needed to write the function is much longer than in the case of writing the function using C language.

An example program that uses this method of protection against calculation errors is presented in section 5.3.2. This program performs filtering of a sound signal by convolution in time-domain.

5. Experimental procedures carried out in DESY

5.1 Overview

The DSK6713 board was tested for radiation influence at the Deutsches Elektronen-Synchrotron (DESY) centre in Hamburg, Germany in April 2005. The board was placed in the Linac II tunnel. Linac II is a linear accelerator, in which positrons are created. When the positrons leave the Linac II tunnel, they enter a PIA (Positron Intensity Accumulator) ring where they are bunched into packets. Afterwards they are sent to the DESY II accelerator. The DSK6713 board was placed approximately 3 m away from the electron-to-positron converter, which is the main source of gamma radiation and neutrons in the Linac II tunnel.

Before putting the DSK board into the accelerator, the system was tested in the laboratory in DESY. During these tests, an unexpected problem was found. The board would reset itself from time to time (approximately every 20-30 minutes). This problem did not exist when the system was tested in Łódź. After further investigation, it turned out that there was some kind of interference on the reset signal which was connected to the parallel port of the PC. After connecting the oscilloscope to the PC's parallel port it was clear that this interference is in the port itself. Figure 5.1 shows the signal shape on the oscilloscope, while a constant value of '1' is sent over the investigated port pin.

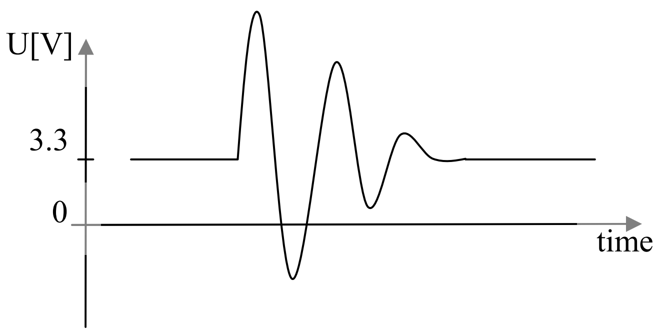


Fig.5.1 Disturbance on the PC's parallel port

The disturbance in the signal was very short (a few nanoseconds). However it was enough to reset the board. What was strange about this phenomenon is that the interference seemed to be caused by sudden movements of people around the PC. For example, every time when a person sitting near the PC would stand up, the signal was disturbed. Touching the PC, or even shaking it, did not trigger the signal distortion, while moving the chair which was nearby – did. The same disturbance was observed on two different PC computers. A

100nF capacitor was connected to the system to filter out the signal, but this solution did not solve the problem. Therefore, a decision was made to disconnect the reset signal. The PC software was changed in such a way, that instead of a reset, the board was turned off for 5 seconds and turned on again.

After these minor changed, the board was placed in the Linac II tunnel. It was kept there for about a week. First, a few programs that tested the influence of radiation on the DSP system were run. Later, two example applications (protected by different software methods) were tested.

5.2 Analysis of influence of radiation on the DSP

5.2.1 EIA-485 transmission test

Introduction

The purpose of this test was to investigate the behaviour of the EIA-485 serial transmission in the radioactive environment. Reliability of the serial connection between the DSP and a PC was crucial, as all the information about the performance of the DSP under radiation is sent to the PC using this connection. The main idea behind this test is quite simple. The DSP awaits an incoming byte on the serial link and sends back the received byte to the PC as soon as it is received. The PC sends a byte to the DSP and listens on the serial port. If it receives the same byte it had previously sent, then it sends another byte and the loop continues.

DSP part of the application

At the beginning of the program the DSP reads the value of the PC's parallel port bit2 (DSP's GPIO bit5). If the bit value is 0 then the whole DSP application code is kept in Flash and executed from there. If the bit is set to 1, the code is copied into RAM and executed from RAM. The main difference between these two modes of operation is that the code in Flash can achieve a maximum transmission speed of about 2000 bps, while the code executed from RAM is much faster and can easily work at a rate of 115200 bps. However, the code kept in RAM is not protected in any way, thus it may hang up if errors in RAM are caused by radiation.

The whole application was written in assembler and consists mainly of the *SoftUartInChar* and *SoftUartOutChar* functions presented in Chapter 3.

PC part of the application

The PC program enables the user to choose in which mode (slow – from Flash, or fast – from RAM) the DSP program will be launched. Then the user can initiate the serial connection by sending the 0x13 value needed by the *SoftUartSpeedDetect* function to calculate the connection speed. The program enables the user to send 1 byte of any value and checks whether the same value is sent back by the DSP. It can also work in a loop, where each time the sent value is increased by 1. All the encountered errors are displayed on the screen and can be logged to a file.

Performance in the accelerator environment

First, the test program was executed in slow-mode - that is a mode in which the whole code is kept in Flash and serial transmission speed is 2048 bps. The program was running for about 12.5 hours (from 6.04 20:52 to 7.04 9:25) and no errors were reported. Then the program was launched in the fast mode, where the DSP application code is placed in internal RAM of the DSP and transmission speed is 115000 bps. The program was running for a total of 6 hours (in the times: 9:38 - 13:35 and 17:38 - 19:25) and again no transmission errors were found. The time of this test was relatively short because the code of the DSP had to be placed in RAM to enable fast serial transmission and it was not protected in any way. Then the PC program was launched in a mode, where the PC just listens on the serial port, to see if the radiation induces any error bits in an idle EIA-485 transmission line. Again, no errors were reported.

To sum up, all the serial transmission tests lasted a total of 21.5 hours and during that time, not even a single bit error was found. Therefore a conclusion was made that the EIA-485 transmission is reliable enough to be used in the next tests which focus on the influence of the radiation on the digital signal processor itself and its internal memory. It can also be concluded that the EIA-485 transmission can be used as a reliable medium in an accelerator environment in future projects.

5.2.2. Internal RAM test

Introduction

The purpose of this test was to check how radiation influences the internal memory of the DSP. The program simply scans the whole IRAM continuously to see if any changes are present.

DSP part of the application

The whole program was written in assemby and is contained in Flash. It does not use the IRAM or external SDRAM at all. All the necessary variables are kept in internal registers; therefore the whole IRAM can be used only for testing the influence of radiation. Additionally, important variables, such as the serial connection speed, are kept in 3 registers and triple voting is used on them. The program first fills the whole 256 KB of IRAM with a test value of 0xFFFF0000. Then it scans the RAM and checks whether the read values are equal to the test value. After the whole memory is checked, which takes about 3.5s (because the code is kept in Flash), a watchdog signal is sent to the PC. If a value read from the memory is not equal to the test value, then immediately a message is sent to the PC. The message contains the address of the error, and a new value read at this address. The whole message is sent 3 times to ensure proper reception on the PC side. The program flow diagram is depicted in Figure 5.2.

PC part of the application

The PC program simply starts the DSP, initiates the serial transmission and listens on the serial port for incoming messages. There is a watchdog timer which resets the DSK board if no signal is received within 4 seconds. Triple voting is carried out on all the received messages and the outcome is displayed on the screen and can be logged into a file.

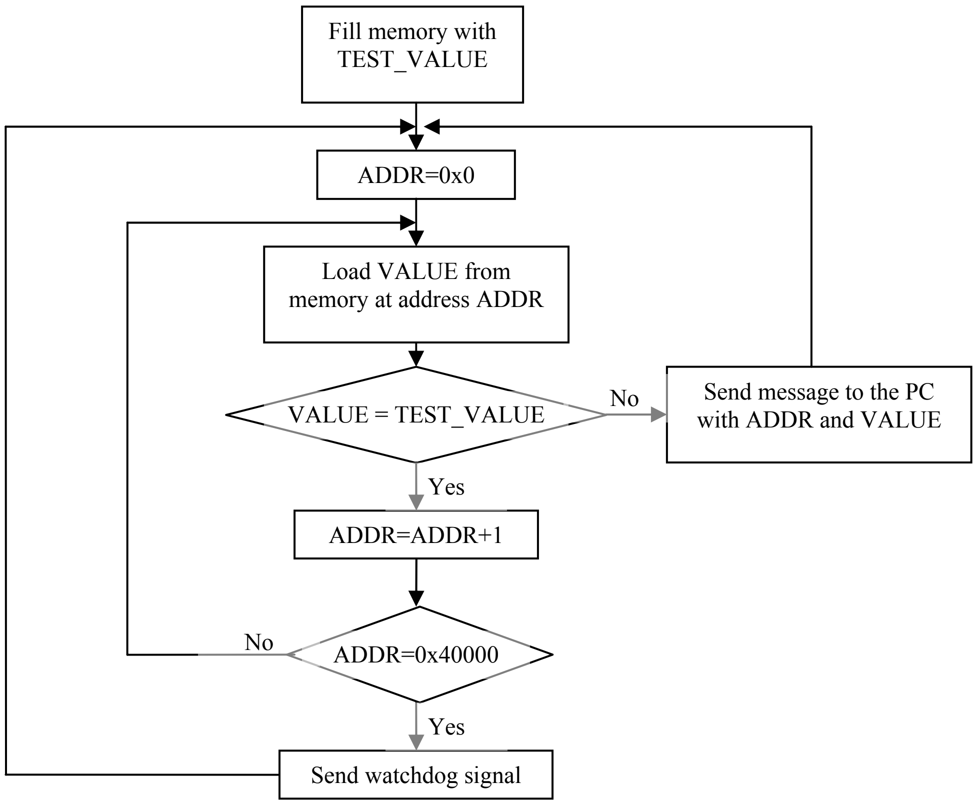

Fill memory with TEST_VALUE
ADDR=0x0
Load VALUE from memory at address ADDR
VALUE = TEST_VALUE
No
Send message to the PC with ADDR and VALUE
Yes
ADDR=ADDR+1
No
ADDR=0x40000
Yes
Send watchdog signal

Fig.5.2 DSP program flow diagram

Performance in the radioactive environment

The program which tested the influence of radiation on the internal RAM of the DSP was launched 2 times. The first time, test lasted for 42 hours and 25 minutes (the test started on 7-04 at 19:27 and ended on 9-04 at 13:52). The observed errors (bit-flips in memory) are listed in Table 5.1.

Table 5.1 Observed memory bit-flips

Date & time of error	Bits flipped	Memory address
7-04 19:45	1	0x00013BD0
8-04 9:07	1	0x0000B424
8-04 17:10	1	0x000077A0
8-04 17:23	1	0x00034410
8-04 17:41	1	0x00032FA8
8-04 18:52	1	0x00029C6C
9-04 3:49	1	0x0000B480
9-04 4:55	1	0x00039ED0

There were two very short breaks in the test (of approx. 20 minutes of total length) because some network settings of the PC had to be changed and this involved restarting the Windows operating system.

The easiest way to quantify Linac II activity during the test is to measure the PIA current, which corresponds to the number of electrons hitting the electron-to-positron converter [17]. A graph showing the PIA current during the test and the moments of bit-flips marked in pink is presented in Fig.5.3.

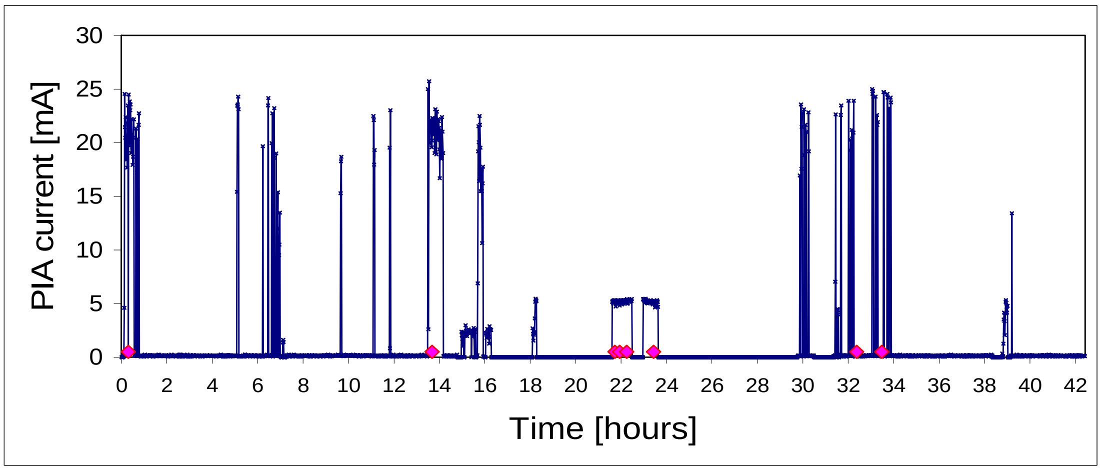


Fig.5.3 Linac II activity during the test (bit-flips marked in pink)

The second test was performed a few days later, when the activity of the accelerator was much higher. The test lasted for 10 hours (from 14.04 23:14 to 15:04 9:14). The following is a graph showing the Linac II activity and a table with all the reported memory bit-flips:

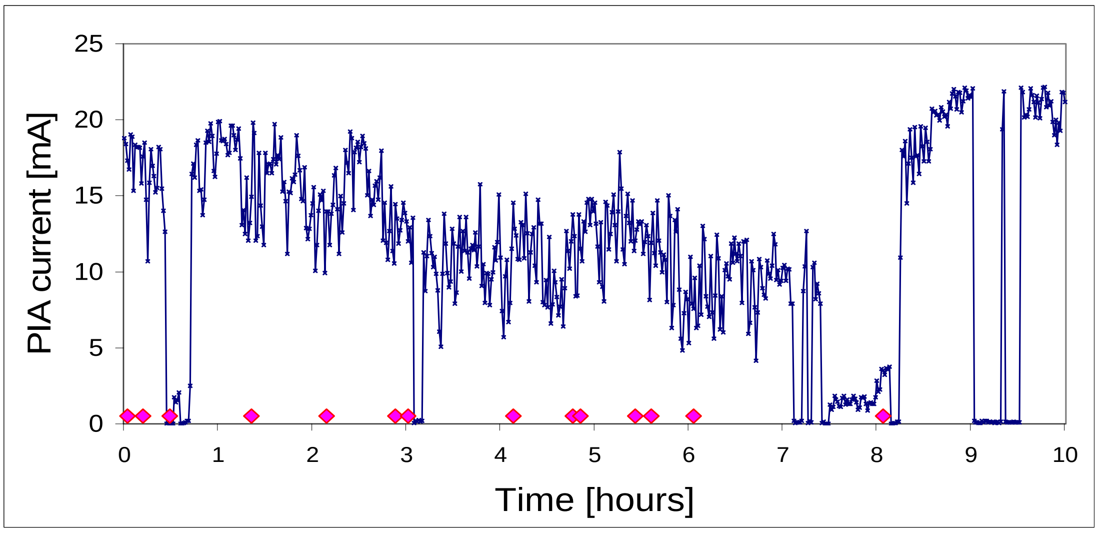


Fig.5.4 Linac II activity during the test (bit-flips marked in pink)

Table 5.2 Observed memory bit-flips

Date & time of error	Bits flipped	Memory address
14-04 23:16:49	1	0x0003CB68
14-04 23:26:48	1	0x0001F9D4
14-04 23:43:23	1	0x00006E9C
15-04 00:35:35	1	0x0003E058
15-04 01:23:56	**1**	**0x000083B8**
15-04 01:23:57	**1**	**0x000084B8**
15-04 02:07:39	1	0x0003F5D8
15-04 02:15:53	1	0x0000F3FC
15-04 03:22:15	1	0x000234B0
15-04 04:00:14	1	0x000306B8
15-04 04:05:26	1	0x000264FC
15-04 04:40:20	1	0x0003204C
15-04 04:50:48	1	0x00016E84
15-04 05:17:52	1	0x0001F8C8
15-04 07:18:36	1	0x00009424

It can be easily noticed that during the second test PIA current was relatively high most of the time. Therefore radiation was much higher and the number of bit-flips is much bigger than in the first test. There were 15 errors reported in 10 hours of testing compared to only 8 errors during the 42 hours of the first test. The most important observation that can be made is that all the errors involved changing only 1 bit at a time. When the radiation was high, errors were reported every couple of minutes. There was one situation, where two errors occurred within one memory scanning loop which lasts for about 4 seconds. The two errors were present at memory locations separated by 0x100 (256) bytes. If Hamming or 2D parity memory protection methods had been used, then the two bit-flips would have been placed in two different memory blocks and each of them could have been corrected by the algorithms. Generally, during both tests, there were no burst errors reported. Each time a bit-flip occurred, its memory address was uncorrelated with addresses of previous bit-flips.

5.2.3. DSP ALU test

Introduction

The purpose of this test was to check how radiation influences the DSP itself and if it causes any wrong results in the calculations made by the ALU modules.

DSP part of the application

The program for the DSP was written in assembly and placed in Flash. The whole program is just one loop, where the DSP performs some calculations. The C6713 DSP has all its modules duplicated; therefore it can perform many arithmetic operations in parallel. In this test, the DSP makes two parallel calculations on two different sets of internal registers.

The calculations start with initial value of 0x1200 from which 0x0C00 is subtracted. Then the result is multiplied by 0x03 which should give a final result of 0x1200 (the same as initial value). Then the calculations are repeated. Both, the subtraction and multiplication are performed twice in parallel by two independent processor modules. If any ot the two results is not equal to 0x1200, a message is sent to the PC. The calculation loop is repeated 0xC000 times and afterwards the results (even if they are correct) are sent to the PC and the loop is started again. The correct results after the loop are used as a watchdog signal.

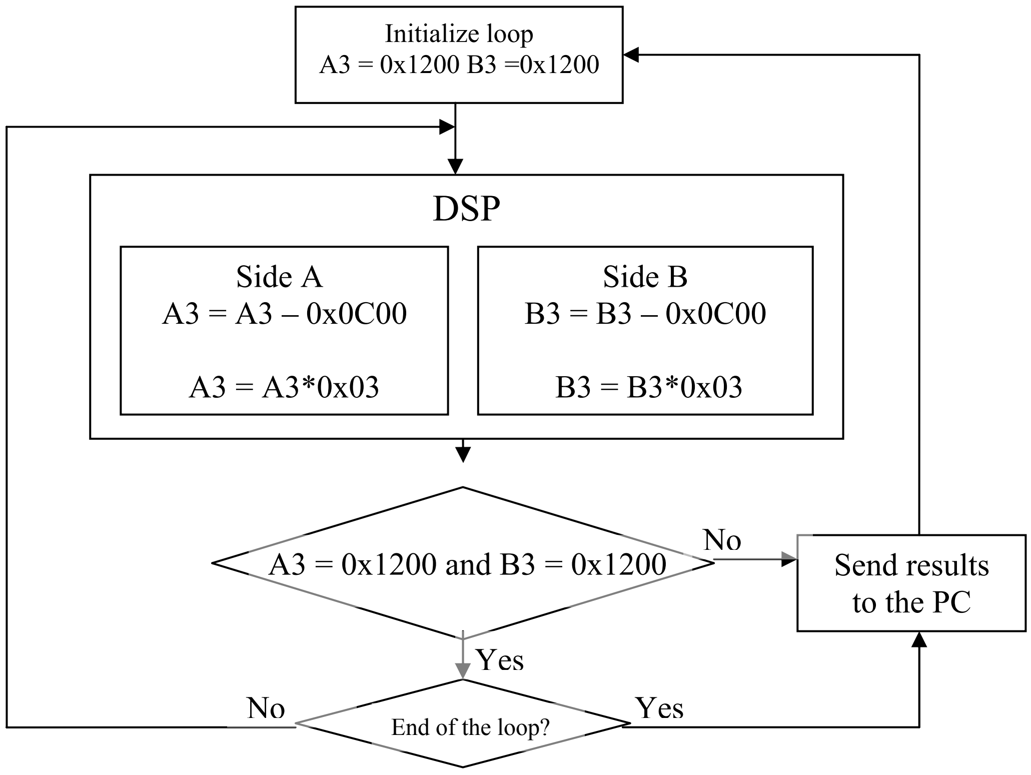


Fig.5.5 DSP program flow diagram

The following is the most important part of the source code:

```
start:                                      ; A4 = serial_speed
              MV           A4,A8
||            MV           A4,A10
||            MV           A4,B10           ; A8 = A10 = B10 = serial_speed
              MVKL         LOOP_COUNT,B2  ; loop counter
              MVKH         LOOP_COUNT,B2  ; loop counter

loop_start:                                 ; set initial values
              MVK          AA,A3            ; A3 = initial value
||            MVK          AA,B3            ; B3 = initial value
              MVK          BB,A5            ; A5 = value to be subtracted
||            MVK          BB,B5            ; B5 = value to be subtracted
              MVK          CC,A4            ; A4 = multiplication coeff
||            MVK          CC,B4            ; B4 = multiplication coeff

main_loop:
 [B2]         B            main_loop
|| [B2]       SUB          A3,A5,A3         ; side "A" subtract
|| [B2]       SUB          B3,B5,B3         ; side "B" subtract
 [B2]         MPY          A4,A3,A3         ; side "A" multiply
|| [B2]       MPY          B4,B3,B3         ; side "B" multiply
 [B2]         MVK          AA,A1            ; load correct values
|| [B2]       MVK          AA,B1
 [B2]         XOR          A3,A1,A1         ; check A-side result
|| [B2]       XOR          B3,B1,B1         ; check B-side result
||            MVK          BB,A5            ; A5 = value to be subtracted
||            MVK          BB,B5            ; B5 = value to be subtracted
 [B2]         OR           A1,B1,B0         ; B0>0 if any of the two results is wrong
||            MVK          CC,A4            ; A4 = multiplication coeff
||            MVK          CC,B4            ; B4 = multiplication coeff
 [B0]         MVK          0x0,B2           ; if B0>0 stop loop
|| [!B0]      SUB          B2,1,B2          ; if B0==0 decrease loop counter

              MV           A3,B11           ; b11 = A-side result
||            MV           B3,B12           ; b12 = B-side result

send_msg:
              ; here is the code that sends the message to the PC
              ; and then jumps to loop_start
```

PC part of the application

The PC program starts the DSP, initiates the serial connection and listens on the serial port for the calculation results. It also activates a watchdog timer that resets the board if no messages are received for 4s. The program enables the user to input an expected results value (default value = 0x1200) and displays on the screen only the messages that contain results not equal to the expected one.

Performance in the radioactive environment

The test program was running for a total of 36 hours (one period of about 10h and later for 26h). Unfortunately, the Linac II activity was rather small during the tests. During that time all the calculation results were correct. There was no error in the calculations made by the DSP. However, there were two occasions when the watchdog timer on the PC expired and the DSK board was reset. This means that radiation has hanged the DSP or influenced it in such a way, that it stopped running and did not send any message to the PC. The graphs showing PIA current in the tunnel with the moments of watchdog reset marked in pink are presented in the following figure:

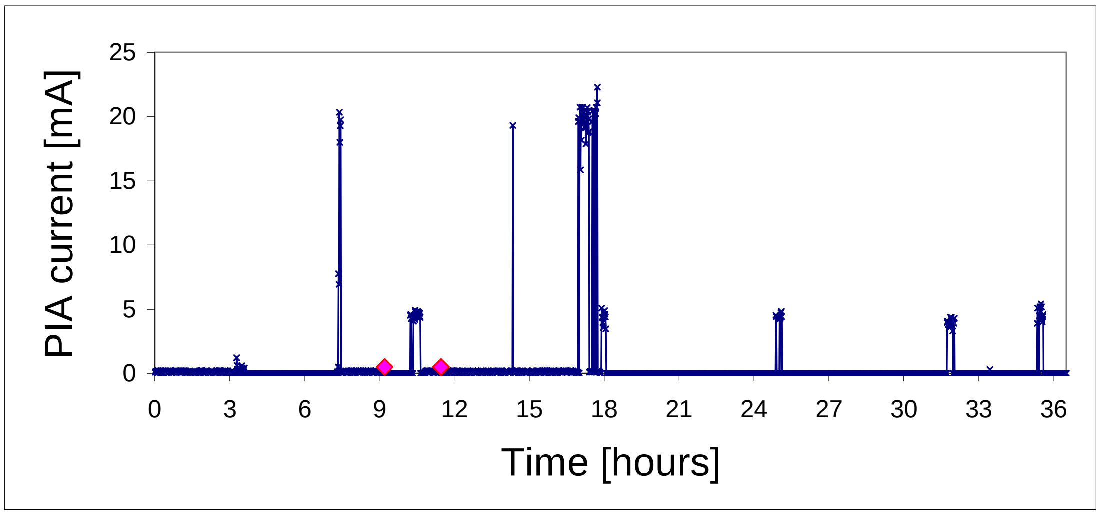


Fig.5.6 Linac II activity during the test (watchdog resets marked in pink)

The most interesting thing about these graphs is that both times when the DSP hanged and the watchdog reset had to be used occurred when PIA current was almost 0. This means that radiation can be quite high and have important influence on the DSP also when the PIA current is very low.

5.3 Example applications protected against radiation

5.3.1. FFT filtering of a sound signal

Introduction

In this test an example application program was loaded into the DSP's Flash memory and executed. The program performs the FFT transform of an 8kHz input sound signal, filters it (using complex multiplication in the frequency domain), then calculates inverse FFT and outputs the result to the audio codec.

DSP program

The program is quite big and complicated. It uses EDMA and "ping-pong" buffering to increase its performance. It also uses the TI's operating system called DSP/BIOS. Many of the DSP/BIOS modules do not work correctly when they are placed in Flash. Therefore all the DSP/BIOS code is placed in RAM along with the chip support library (csl6713.lib) and run-time support library (rts6700.lib). Of course, the most important and time-critical parts of the program (e.g. FFT calculation, filtering) are also placed in RAM. The total amount of code placed in RAM is 41 KB. All this code is protected using the 2-dimensional parity method. Each time, when an interrupt is triggered, the code is checked for errors before the interrupt routine is executed. If more errors are detected than can be corrected, then the code is copied from Flash to RAM again. After the memory check, the appropriate interrupt routine is executed. The 2D parity function that performs the memory check is also placed in RAM, because of the need for fast execution. Therefore, it is vulnerable to the effects of radiation. However, the code of the function is very small. It is less then 300 bytes, while the Hamming code function's code is more than 1 KB. As a result, only 300 bytes of the RAM memory are actually vulnerable to radiation. If the program were not protected by the 2D parity method, then the whole 41 KB of application code stored in RAM would be vulnerable. So, usage of the 2D parity method decreases the probability that bit-flips in internal memory corrupt the application code by about 41000/300=136.67 times.

All the calculations performed in the program are repeated 3 times and voting is used to determine the final result. Also, all the buffers, filter coefficients and other tables (e.g. coefficients used in FFT calculation) are protected using the Hamming code. Whenever errors are detected or corrected, or voting needs to be used, an appropriate message is sent to the PC. A diagram showing the general concept of how the program processes the input signal is presented on Figure 5.7:

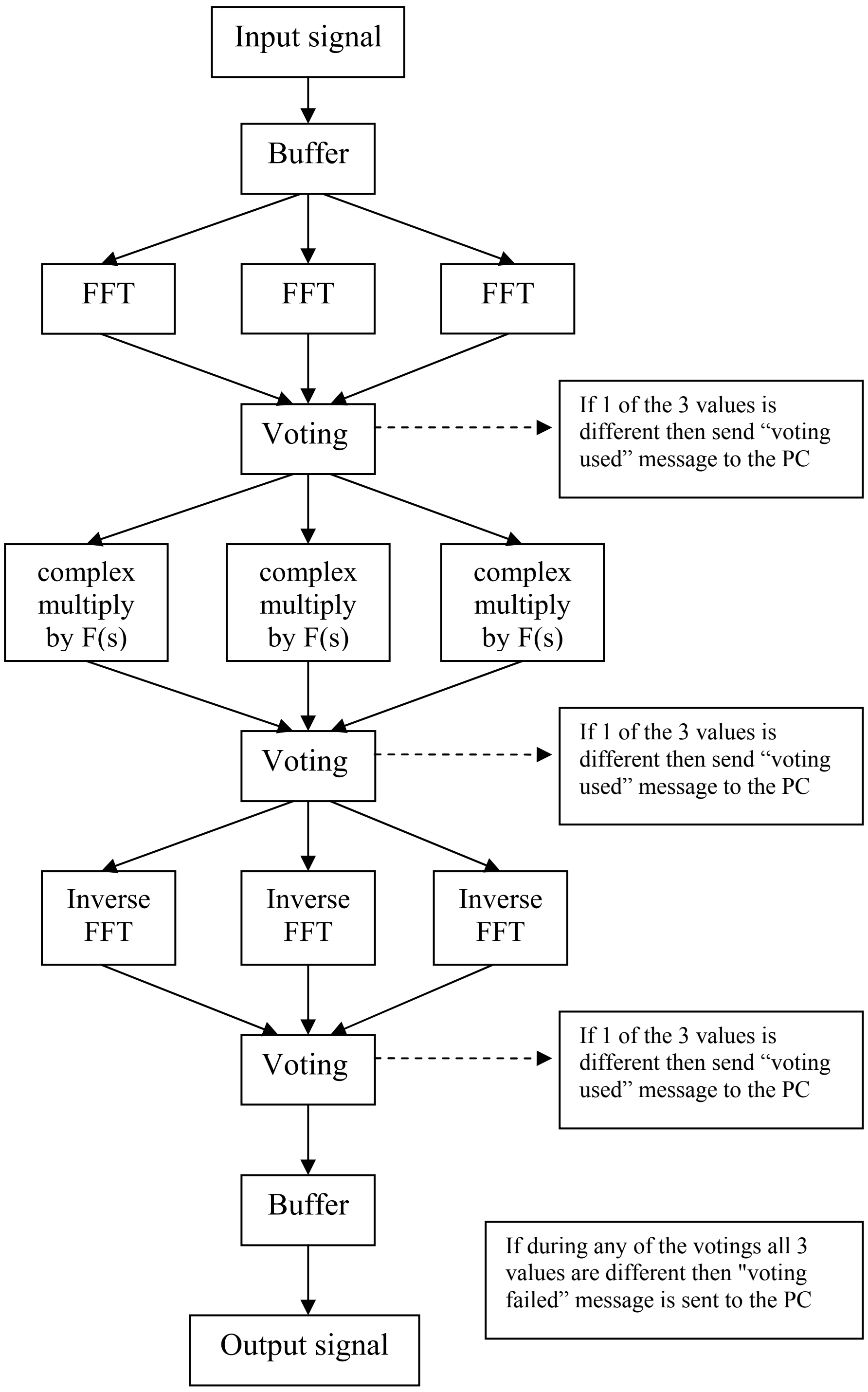

Input signal
Buffer
FFT
FFT
FFT
Voting
If 1 of the 3 values is different then send "voting used" message to the PC
complex multiply by F(s)
complex multiply by F(s)
complex multiply by F(s)
Voting
If 1 of the 3 values is different then send "voting used" message to the PC
Inverse FFT
Inverse FFT
Inverse FFT
Voting
If 1 of the 3 values is different then send "voting used" message to the PC
Buffer
If during any of the votings all 3 values are different then "voting failed" message is sent to the PC
Output signal

Fig.5.7 DSP program flow diagram

PC program

The PC program is just the one explained in the chapter about the DSP-PC communication system. It listens on the serial port for messages from the DSP. There is a watchdog timer, which resets the DSK board if no signal is received in a specified time.

Performance in the radioactive environment

First, the program was launched on 11-04 at 16:55. At that time the watchdog would reset the board almost all the time (every 2-3 minutes). Changing the watchdog timer value from the default 1.5s to 2s fixed the problem. It is hard to find a reasonable explanation for this, because the DSP was programmed to send the watchdog signal every second. However, after the change, watchdog stopped resetting the board and the program was running smoothly. The total time of execution of the program was about 48 hours. During that period the following events happened:

- 12.04 at 22:20 - over 280 messages “voting used” were received and this was immediately followed by a watchdog reset. This could mean that the radiation has corrupted the DSP operation in such a big way that it achieved a large number of incorrect calculation results (which were fixed by triple voting), and it also stopped sending the watchdog signal, which caused the watchdog reset. However, the next day an error was found in the PC software, which could have caused the watchdog reset even if the watchdog signal was send by the DSP, if a large number of other messages were sent from the DSP at the same time when the watchdog signal should arrive. Therefore it is impossible to decide what really happened 12.04 at 22:20. It could have been one of the two possibilities: either the radiation influenced the DSP in such a way that it did not send the watchdog signal, or it only caused a large number of incorrect results that were fixed by triple voting and the reset of the DSP was caused by an error in PC software.
- 2 times “voting used” message was received (on 13.04 at 1:02 and at 8:19) – this means that one of the calculations produced wrong result, but triple voting managed to correct it and produced the right result.
- 4 times single bit error was corrected by the 2-dimensional parity algorithm in the code of the program (it happened on 13.04 at 1:58, 6:02, 9:30 and 17:20).

A graph showing the activity of the accelerator by means of the PIA current is shown on Figure 5.8 (the radiation induced events are marked in pink):

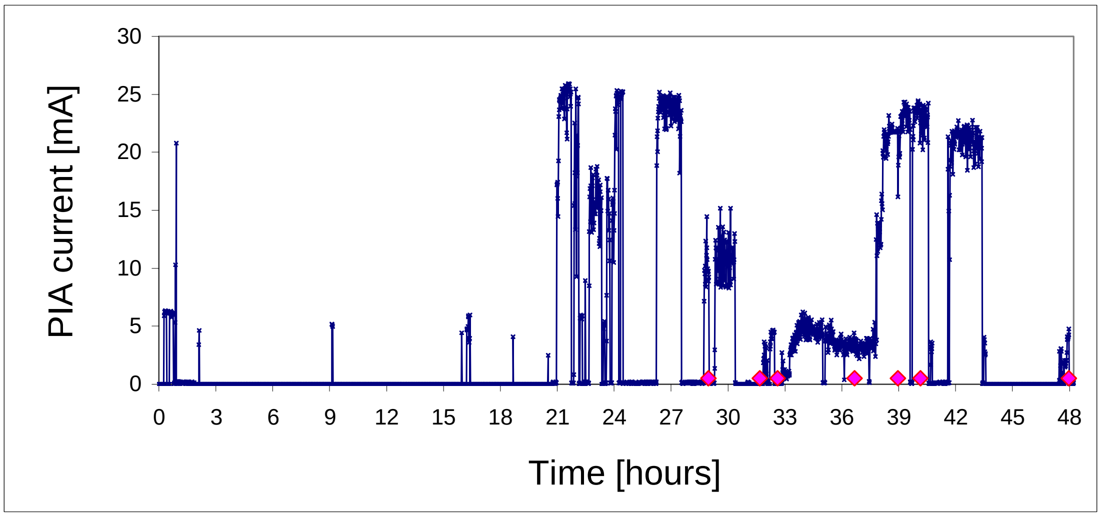


Fig.5.8 Linac II activity during the test (SEUs marked in pink)

To sum up, during 48 hours of program execution there were 6 times when the implemented software radiation protection methods helped in keeping the DSP properly running and producing correct results. Four of them involved correction of the bits in memory that were flipped by radiation and two times triple voting helped in obtaining correct results. However, there was one time, when the software did not manage to keep the DSP running properly despite the effects of radiation and the board was reset by the watchdog. All of the events happened in the periods were the PIA current was above 0mA. In the times when the accelerator activity was much smaller, no events were reported.

5.3.2. Convolution filter (without DSP/BIOS)

Introduction

In this test the example application that was running on the DSP was a simple filter. The program carried out the filtering by convolution calculation. The main difference compared to the previous program is that this time DSP/BIOS was not used and therefore the program code was considerably smaller.

DSP program

The main part of the program is the routine that calculates the convolution of the input signal with the filter’s unit impulse response. This function is written in assembly in such a way, that all the calculations are carried out twice in parallel. Whenever the two results are not equal to each other, the calculations are repeated and an appropriate message is sent to the PC. The block diagram is shown on Figure 5.9. Additionally, in the convolution calculation function, all important variables, such as pointers to filter coefficients and buffers, are kept in registers with triple redundancy. During each loop iteration triple voting is performed on them. This solution gives even more protection, as the function can work correctly even if radiation induces changes in the internal registers. The main drawback of this approach is the loss in speed. The function execution time is higher than execution time of a simple unprotected convolution calculation function repeated 2 times, and voting performed afterwards.

In this program, again the whole code that is placed in RAM is checked for errors before being executed. The error checking is done using the 2-dimensional parity algorithm which is executed before each interrupt service routine. However, because there is no DSP/BIOS, the size of the code in RAM that needs to be protected is only 6.5 KB. The small size of code is also caused by the fact that the algorithm used for filtering is much simpler than FFT. The most important problem with DSP/BIOS was that this system uses some areas of RAM for its own data and this was not protected in any way in the previous program. This time, when DSP/BIOS is not present, everything is under control and there are no areas of data or code that are not protected. Additionally, all the necessary data tables (e.g. filter coefficients) are protected by the Hamming code.

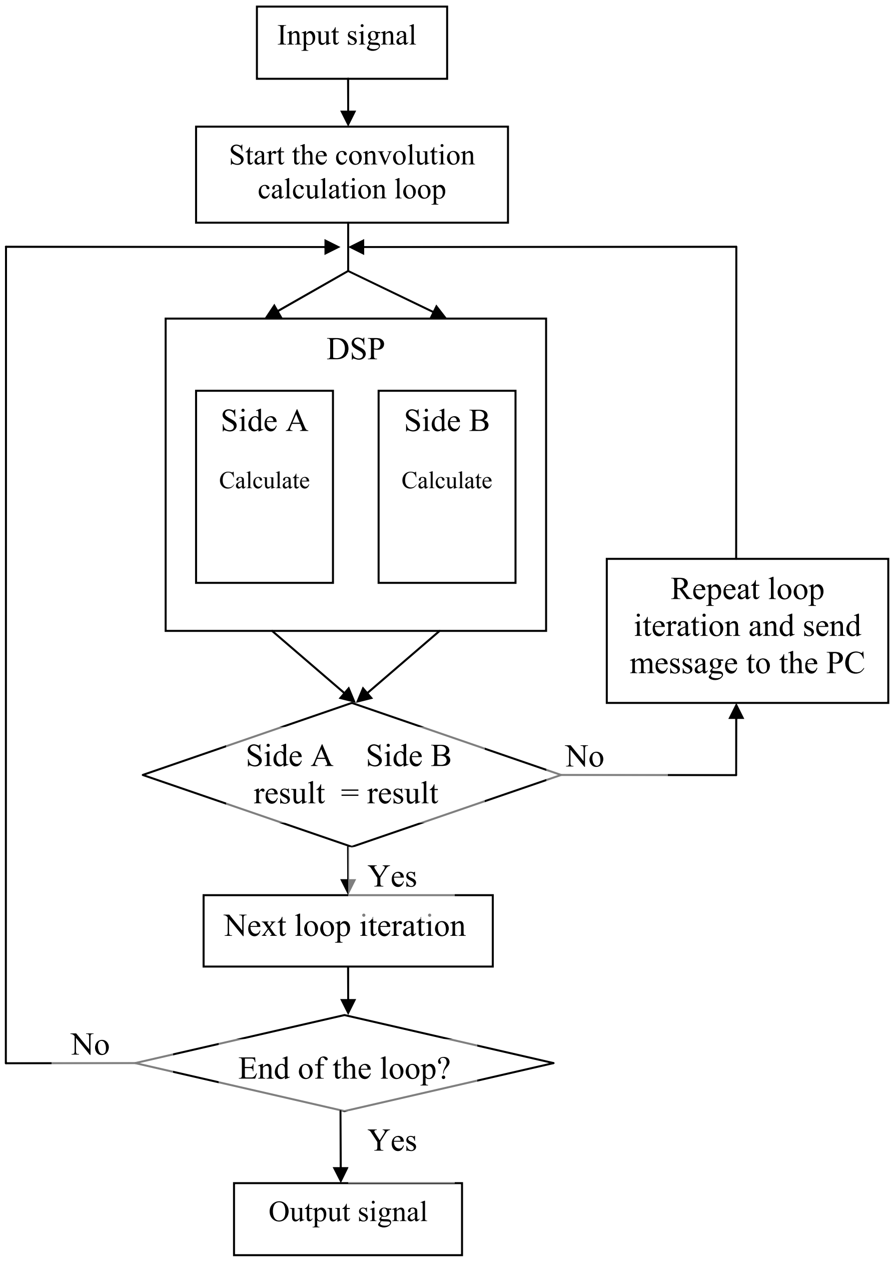

Input signal
Start the convolution calculation loop
DSP
Side A
Calculate
Side B
Calculate
Repeat loop iteration and send message to the PC
Side A result = Side B result
No
Yes
Next loop iteration
No
End of the loop?
Yes
Output signal

Fig.5.9 DSP program flow diagram

PC program

The PC program is the same as the one used in previous test.

Performance in the radioactive environment

The program was tested in the Linac II tunnel for about 29h.30min (from 17:42 13.04 to 23:16 14.04). During this time the activity of the accelerator was very high. A total of 4 events caused by radiation have been observed. Fig.5.10 shows the values of PIA current in the Linac II tunnel during the test with the events triggered by radiation shown by pink markers.

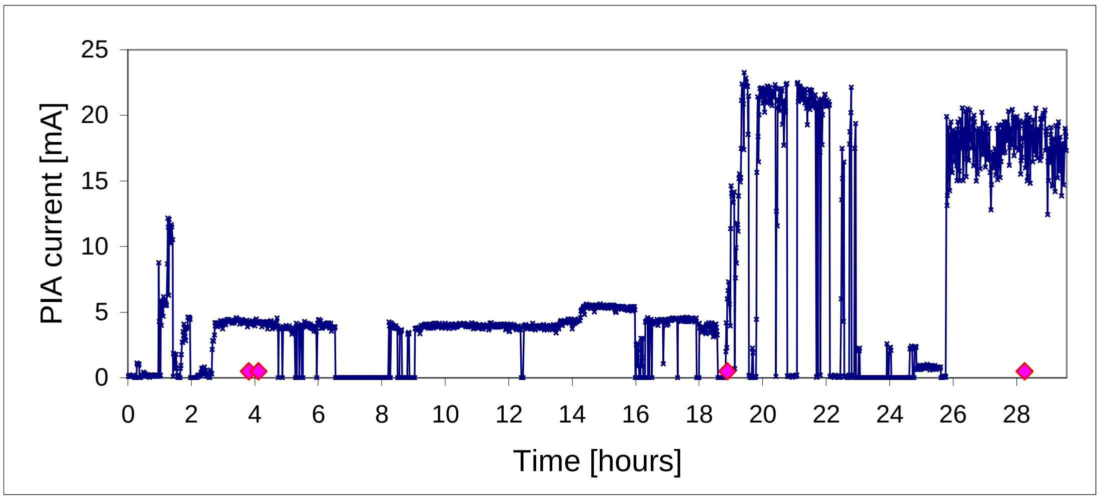


Fig.5.10 Linac II activity during the test (SEUs marked in pink)

Here are more details about the events caused by radiation:

- 13.04 at 21:30 - one bit error was corrected by the Hamming code. The error must have occurred somewhere in the data section of the program, as only the data was protected by Hamming code.
- 13.04 at 21:48 - another error in memory was corrected. This time 1 bit-flip occurred in the code of the program and was corrected by the 2D parity algorithm.
- 14.04 at 12:35 - 1 bit error was corrected by the 2D parity method
- 14.04 at 21:57 - the PC received 1 byte of value 0x0 over the serial port. However, this value is not equal to any of the valid message codes. One second later, the watchdog timer expired and the DSK board was reset. This is clearly a situation, where radiation stopped the DSP from proper operation and the software protection methods failed. A 0x0 received on the serial port means that the serial port output of the DSP malfunctioned, because normally, in idle state, the serial port output is set to logical '1'. An example situation when the serial port output is equal to '0' is when the DSP is turned off, or when it is just after reset. Probably, radiation caused the DSP to stop or even reset itself and this switched the serial port output to '0' and prevented the watchdog signal from being sent.

The most important fact is that during the whole test there was no situation when the two parallel convolution calculations would produce different results. Also, the triple voting on internal registers of the processor did not report any changes in the values stored in registers. Therefore, the method of performing parallel calculations and using triple modular redundancy to protect the internal register values does not increase the reliability of a DSP.

6. Project results

6.1 Memory protection

In general, the tests performed at DESY showed that radiation has an important influence on the DSP and its internal RAM memory. The most frequently observed type of single upset event caused by radiation was a change in the internal memory of the processor. In all the tests only 1 bit errors were found in RAM. This means that to protect the system's memory, 1 error correcting codes are sufficient.

In this master's project two main memory protection methods have been implemented. First one is the 2D parity control which can correct 1 bit error in every 128 bytes. The second one is the (39,32) extended Hamming code which can correct up to 32 1-bit errors in a 128 bytes block if every error is introduced in a different "bit column". Therefore both methods are sufficient to deal with the effects of radiation on the DSP's memory. However, it is important to remember, that memory protection methods can be useful only if the program memory is scanned for errors each time before execution of a given code. For example, if some interrupt service routine (ISR) needs to be protected then the memory containing the ISR code has to be scanned each time the interrupt is triggered. This produces relatively high additional CPU load. Both memory protection methods implemented in this project require about 800 CPU cycles to scan a 1 KB memory region for errors. If a faster routine is needed, then simple parity control can be used. This method requires only about 280 CPU cycles to scan a 1 KB memory block, however it cannot correct errors. Therefore, when an error is detected the code needs to be copied from some other radiation immune source, for example the Flash memory. However, copying the code from Flash is dependent on the Flash memory speed and can be very time-consuming. On the other hand, the process of correcting 1 bit error using the 2D parity control method requires only about 20 extra CPU cycles. In the case of the Hamming code this process is a few times longer and depends on the location of the error.

Another important factor which has to be taken into account is the fact that there is a probability that an error occurs in the memory checking function itself. Therefore the memory scanning function should also be checked for errors or its code copied from Flash prior to its execution. Because of this fact the 2D parity control function seems to be a much better choice than the Hamming function, because its code is much shorter. The 2D parity checking function code size is about 280 bytes, while the Hamming memory checking function code size is over 1 KB. Therefore checking the code for errors or copying it from Flash is about 4 times faster in the case of 2D parity. If the function is not protected and not copied from Flash, the probability that an error occurs within its code is also 4 times smaller. Comparison of the code size and execution speed of all memory protection methods used in the test is presented in Table 6.1. The presented CPU cycles needed to scan 1 KB memory block are given for the case, when no errors are present.

Table 6.1 Comparison of memory protection methods

Method	**Code size**	**CPU cycles needed to scan 1 KB of memory**
Simple parity control	72 bytes	288
2D parity control	280 bytes	771
(38,32) Hamming code	934 bytes	750
(38,32) extended Hamming code	1160 bytes	796

Summarizing the presented data, it can be concluded that both methods implemented (2D parity and Hamming codes) are sufficient to provide radiation immunity. However the 2D parity control method is best suitable for protection of code which is frequently executed and its execution time is crucial for the system, for example the interrupt service routines. This is due to the small size of the 2D parity control function code and its very quick process of error correction. Methods based on the Hamming code can be used in all other cases and for protection of data due to their higher error correcting capabilities with approximately the same speed of execution as the 2D parity control.

6.2 Protection against errors in calculations

Another type of SEU's caused by radiation is when the processor itself is influenced. As an effect of this, some calculations and other DSP operations yield incorrect results. This can be fixed by repetition of the calculations and using voting methods to choose the correct result. In the test of the FFT filtering program 2 times voting has prevented the DSP from obtaining incorrect result. However, the other method which was implemented, the method involving parallel calculations in two independent ALU's of the DSP did not prove to be useful. There was no situation were the two parallel calculations would give different results. Of course, both methods introduce additional load to the processor and require additional CPU power to provide radiation immunity. The case of triple voting is very simple. Repetition of calculations 3 times means that the DSP can perform 3 times smaller number of operations than in the case when no protection is implemented. The additional delay introduced by voting is very small and negligible compared to the calculations (e.g. FFT). Also the code of the voting routine is very short. Therefore the probability that an error occurs during voting is much smaller than the probability that it occurs during the process of calculations. In the case of parallel calculations, the analysis is much more complicated. At first, one can think that performing calculations 2 times in parallel should take the same amount of time as normal process of calculations. However, this is not exactly true. When operations are not performed twice in parallel, all the CPU resources can be used for optimization of the calculation process and therefore the result may be obtained much faster. The exact additional CPU power needed by this radiation protection method is difficult to estimate and is different for every type of calculations algorithm. This method also needs additional CPU power to protect the values of the internal registers, for example by triple voting. This protection may be crucial for some types of algorithms. For example, if an error is introduced in the register holding the data address on the "A" side of the DSP, then the data loaded for calculations will be incorrect and therefore the "A" and "B" side results will be different. If the calculations are repeated without correcting the error in the register, then the two results will be always be different and the program will enter an infinite loop. To avoid such situations, the register value needs to be protected from errors. Another approach is to use the same register for data address for both calculations. This will avoid infinite loop possibility, but will not ensure radiation immunity, as the error introduced in the register will influence the results of calculations but will be undetected. In this project, the parallel calculations method was used for protection of convolution calculation. Table 6.2 presents the amount of CPU cycles needed by the normal (optimized) convolution routine and "twice in parallel" calculation function with and without triple voting on internal registers.

Table 6.2 CPU cycles needed by parallel calculations method

Method of convolution calculation	CPU cycles
normal C function, optimized by compiler	464
"twice in parallel" without additional register protection	765
"twice in parallel" with triple voting on all internal registers	2403

It can be seen from the Table 6.2 that the parallel calculation method is much slower than the normal calculation process; however it is not two times slower. Therefore, this solution is still faster then the case where the calculations would be simply repeated and then the two results compared. Another important fact is that if the calculations were just repeated and later compared, then additional memory (vulnerable to SEUs) would be needed for storing the results of both calculations. This memory is not needed if the parallel calculations approach is used. However, when additional protection of registers is used, the total execution time increases dramatically. In the case of the function used in this project, the number CPU cycles needed is 6 times bigger than in the case of no protection. However, this is in the extreme case when all internal registers are protected. Usually, the optimum solution requires protection of only some limited number of registers and then the number of CPU cycles needed is smaller. Another drawback of the parallel calculations method is the fact that the functions must be written in assembly and therefore the software development time is much longer. However, the most important disadvantage of this method is the fact that during the tests at DESY there was no occasion when the two parallel results would be different, or the internal register value changed. Therefore, the method is ineffective in real radiation environment.

6.3 Serial transmission

One very important observation made during the tests is that the EIA 485 serial transmission can be used as a reliable medium in a radioactive environment. During all the tests, which lasted over a week, no error in the serial transmission has been reported. This means that the EIA 485 transceivers are immune to radiation and this standard may be used in future, further research. The standard is well known for its interference immunity. This is mainly due to the fact that the signals are transmitted as voltage difference between two transmission wires rather than a single voltage value with respect to a common ground. Therefore, when some interference is introduced, it affects both wires in almost the same manner so the voltage difference between the wires is not changed and the signal can be read correctly. This feature is important also in the accelerator tunnel, because there is a large number of different kinds of electronic equipment placed in the tunnel and all these devices can interfere with the transmission.

7. Conclusions

The main goal of this project was to analyze the influence of radiation (mainly Gamma radiation and neutrons) on a DSP system and investigate if the reliability of the system can be increased by means of purely software methods. The processor used in the experiments was a C6713 DSP from Texas Instruments.

For the purposes of the research a communication system between the DSP placed in the radioactive environment (accelerator tunnel) and a PC located outside the radioactive area has been designed and fabricated. The system has been used in all the tests that have been conducted in DESY in April 2005.

In this project numerous software methods have been implemented in C and assembly language. The tests performed in DESY confirmed that radiation causes single effect upsets (SEU) in the digital signal processor. Most of the SEUs were connected with changes in the internal memory of the processor. The most important observation was that all of these changes are only 1 bit memory bit-flips. The main two methods of memory protection implemented in this project (2D parity control and Hamming codes) are able to correct all such errors. Another type of SEUs observed was a change of the DSP operation causing for example incorrect calculation results. This kind of errors have also been detected and corrected by triple voting. Another implemented method that was suppose to correct such errors was based on performing all DSP operations twice in parallel (utilizing the DSP modular architecture). However, this method did not produce satisfactory results. There was no situation during the tests, when the two parallel results would be different, therefore the method did not have any influence on improving reliability of the DSP.

Another important aspect of the research was that the EIA-485 serial transmission standard used in the communication system proved to be reliable and immune to radiation. During over 1 week of testing there have been no errors in the serial transmission reported. This gives optimistic perspective for any future research.

However, there have been a few occasions (a total of 4 during all the tests) when the implemented methods have not managed to detect and correct errors caused by radiation. Probably these situations occurred when the radiation particles hit the DSP core and stopped its normal operation. In these cases the DSP no longer executed its code and therefore all the software methods became useless. In such cases watchdog reset was performed.

Overall, this project proved that the reliability of a DSP system in a radioactive environment can be improved significantly by means of pure software methods. During the tests at DESY there were numerous occasions when the implemented methods detected and corrected errors induced by radiation. However, the improved reliability comes at a cost of performing additional operations like memory checking, repeating of calculations and voting. Therefore, during development of an application that will be running in a radioactive environment, detailed calculations must be performed to estimate how much processor power will be available for the implementation of the radiation protection. If only a small amount of CPU cycles are available, then only the simplest methods (e.g. simple parity control) can be used and errors can be only detected. If more CPU resources can be used for radiation protection, then more sophisticated methods (e.g. 2D parity control, Hamming codes, triple voting) can be used and errors caused by radiation will be corrected. However, all the software methods have a limit of their performance. When radiation particle hits the core of the CPU it can cause the processor to stop executing its software and then only an external method (e.g. a watchdog) or a hardware protection method can help.

References

[1] R. J. Peterson, "Radiation-induced errors in memory chips", Brazilian Journal of Physics, vol.33 no.2, 2003

[2] N. Sramek, "Radiation Hardened Electronics for Space Systems", The Aerospace Corporation, 12.06.2001

[3] F. Anghinolfi, "Radiation Hard Electronics", CERN/EP, 2000

[4] A. S. Sedra, K.C. Smith, "Microelectronic circuits", Oxford University Press, 1998

[5] S. M. Sze, "Semiconductor devices", John Wiley &Sons Inc, 2002

[6] M. Moll, "Radiation Hardening of Silicon Detectors by Oxygen Enrichment", CERN, 10. April 2000

[7] F. Giustino, "Radiation Effects on Semiconductor Devices", PhD thesis, Politecnico di Torino, 2001

[8] E. Olcayto, "Information theory" lecture material, University of Strathclyde

[9] M. Purser, "Introduction to Error-Correcting Codes", Artech House Inc., 1995

[10] R. H. Morelos-Zaragoza, "The Art of Error Correcting Coding", John Wiley & Sons Ltd, 2002

[11] TMS320C6713 Data Sheet, Texas Instruments

[12] T. Hiers, R. Ma, P. Malleth, S. Chen, "TMS320C6000 McBSP: UART", Application Report SPRA633B, Texas Instruments, 2004

[13] G. Rhoads, http://remus.rutgers.edu/~rhoads/

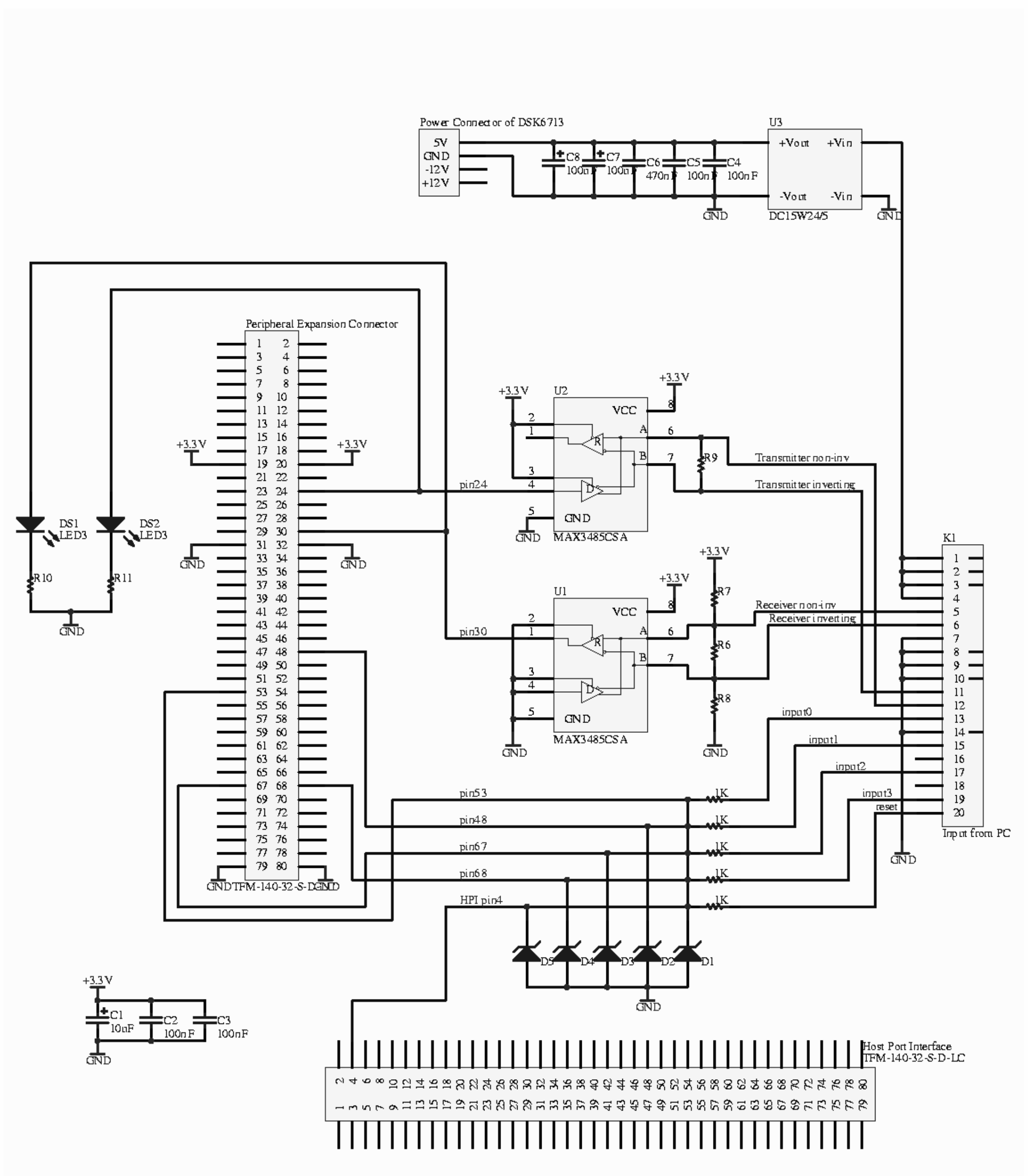
[14] Logix4u, http://www.logix4u.net

[15] P. Shirvani, N. Saxena, E. J. McCluskey, "Software-Implemented EDAC Protection Against SEUs", Stanford University, Stanford

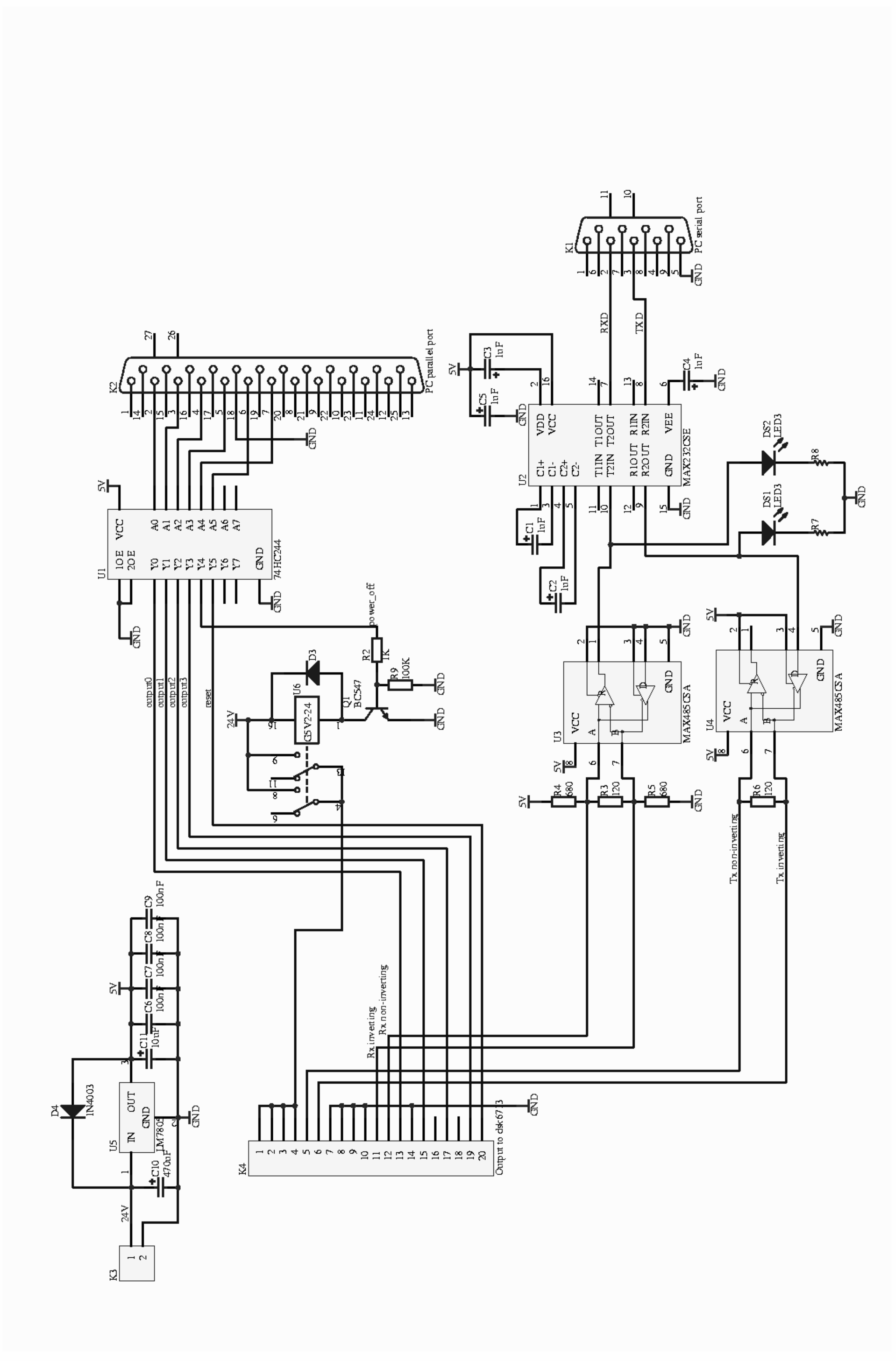
[16] J. Sankaran, "Reed Solomon Decoder: TMS320C64x Implementation", Application Report SPRA686, Texas Instruments, 2000

[17] D.K. Rybka, A. Kalicki, K. Pozniak, R. Romaniuk, B. Mukherjee, S. Simrock, "Irradiation Investigations for TESLA and X-FEL experiments at DESY"

Appendix A: Schematic of the DSP module



Appendix B: Schematic of the PC module



Appendix C: Main part of the DSP bootloader code

```
#define CHIP_6713 1

#include <csl.h>
#include <csl_gpio.h>

#include "dsk6713.h"
#include "dsk6713_flash.h"

#include "crc32.h"

// ########  macro definitions  ###############
#define BUFFER_SIZE 2048
#define ACK 0xF0
#define VOTE_FAILED 0x01
#define FLASH_FAILED 0x0F

#define COPY_CODE_FROM_FLASH \
       memcpy( (void *)&run_fast, (void *)&load_fast, (int)&size_fast )

#define VOTE(a,b,c) \
       if (!( a ^ b )) c = a;\
              else if (!( b ^ c )) a = b;\
                     else if (!( a ^ c )) b = c;\
                     else failure( VOTE_FAILED );

#pragma CODE_SECTION (ReadBlock,"fast"); // put func.ReadBlock into "fast" section

char buffer[BUFFER_SIZE];     // data buffer
unsigned int serial_speed[3]; // serial transmission speed

GPIO_Handle hGpio;    // for GPIO (PC-parallel connection)

extern int load_fast; // load address of "fast" section
extern int run_fast;  // run address of "fast" section
extern int size_fast; // size of "fast" section

unsigned int crc32_received,crc32_temp;

volatile unsigned int *LED_ptr = (volatile unsigned int*)(0x90080000); // on-board LEDs

// #######################  functions   ##########################

void send_msg(char val)
{
SoftUartOutchar( serial_speed, val );       // send the message 3 times
SoftUartOutchar( serial_speed, val );
SoftUartOutchar( serial_speed, val );
}

void ReadBlock(void)  // function reads a block of data & its crc32 through serial port
{
int i;
char *ptr;

// read the block of data
i = BUFFER_SIZE;
ptr = buffer;
while ( i-- )
       *ptr++ = SoftUartInchar( serial_speed[0] );

// read crc32 of the block
i = 4;
ptr = (char *)&crc32_received;
while ( i-- )
       *ptr++ = SoftUartInchar( serial_speed[0] );

}
void failure(char val)  // function is called when triple voting or flash crc32 fail
{
send_msg( val );
while(1);
}
```

```
// #################  MAIN  ######################
void main (void)
{
unsigned int flash_dest_addr[3];

// GPIO configuration
GPIO_Config   MyConfig = {
                0x00000000,  // gpgc
                0x000000F0,  // gpen pins 4-7 are set to read mode
                0x00000000,  // gdir
                0x00000000,  // gpval
                0x00000000,  // gphm
                0x00000000,  // gplm
                0x00000000   // gppol
                };

hGpio = GPIO_open(GPIO_DEV0,GPIO_OPEN_RESET);
GPIO_config(hGpio,&MyConfig);

/* Initialize the board support library */
DSK6713_init();

*LED_ptr = 0x1;
DSK6713_rset(DSK6713_MISC, 0x1); // switch McBSP0 to peripheral expansion connector

// copy the "fast" functions from flash to internal RAM
COPY_CODE_FROM_FLASH;

serial_speed[0] = SoftUartSpeedDetect();    // detect the serial transmission speed
serial_speed[2] = serial_speed[1] = serial_speed[0];
send_msg( ACK );  // send ACK

*LED_ptr = 0x9;
// erase the flash and set the start address of flash to be written
flash_dest_addr[2] = DSK6713_FLASH_BASE+0x8000;
flash_dest_addr[0] = flash_dest_addr[1] = flash_dest_addr[2]; // used for triple voting
DSK6713_FLASH_erase(flash_dest_addr[0], DSK6713_FLASH_PAGESIZE*7);
*LED_ptr = 0x1;

send_msg( ACK ); // inform about flash erased

while (1) {
       // read block of data
       if ( !((GPIO_read (hGpio,0x00F0) >> 4) & 0x01) ) break; //check if GPIO bit 4 is set

       COPY_CODE_FROM_FLASH;
       *LED_ptr = 0x03;
       VOTE(serial_speed[0],serial_speed[1],serial_speed[2]); // voting on serial_speed
       ReadBlock();

       // calculate crc32
       crc32_temp = crc32_calc( buffer, BUFFER_SIZE );

       if ( crc32_temp == crc32_received ) {       // if ok then write to the flash
              *LED_ptr = 0x5;
              // voting on flash_dest_addr
              VOTE(flash_dest_addr[0],flash_dest_addr[1],flash_dest_addr[2]);

              DSK6713_FLASH_write((Uint32)buffer, flash_dest_addr[0], BUFFER_SIZE);
              crc32_temp = crc32_calc( (char *)flash_dest_addr[0], BUFFER_SIZE );
              if ( crc32_temp != crc32_received )
                     failure( FLASH_FAILED );

              send_msg( ACK );      // send acknowledgement

              flash_dest_addr[0] += BUFFER_SIZE;
              flash_dest_addr[2] = flash_dest_addr[1] = flash_dest_addr[0];
              }
       }

GPIO_close(hGpio);     // the end
*LED_ptr = 0xf;        // light up all the diodes
while (1) ;            // infinite loop
}
```

Appendix D: Hamming code function in assembly

This function detects and corrects errors in a memory block. The argument of the function is a pointer to a structure *Thamm* that describes the protected memory block.

Thamm declaration in C:

```
typedef struct {
int *data;
int *parity_bits;
int no_of_blocks;
} Thamm;
```

Function code in assembly:

```
N              .equ    38
K              .equ    32


               .sect "fast"

COSET:         .byte 48, 24, 12, 6, 3, 40, 20, 10
               .byte  5, 36, 18, 9, 34, 17, 33, 56
               .byte 28, 14, 7, 52, 26, 13, 44, 22
               .byte 11, 60, 30, 15, 42, 21, 45, 51
               .byte  1, 2, 4, 8, 16, 32

_hamm_check_asm:
               stw            a10, *b15--     ; push a10 to stack
               ldw            *a4++,a0        ; a0 = pointer to data
               ldw            *a4++,a5        ; a5 = pointer to parity bits
               ldw            *a4,a1          ; a1 = no_of_blocks
               mvk            64,b4
||             mvk            0x0,a3          ; a3 = total errors
               mvk            0x0,a2
               add            b4,a0,b0        ; a0 -> data[0], b0 -> data[16]

               add            12,a5,b5        ; a5 -> parity[0], b5 -> parity[3]
 [!a1]         b              end                    ; if (no_of_blocks == 0) return 0

loop:
               ;load parity bit values
               mv             a0,a6           ; a6 -> data[0]
||             mv             a5,b6           ; b6 -> parity[0]
               ldw            *a5++,a7        ; a7 = temp_S[0]
||             ldw            *b5++,b7        ; b7 = temp_S[3]
               ldw            *a5++,a8        ; a8 = temp_S[1]
||             ldw            *b5++,b8        ; b8 = temp_S[4]
               ldw            *a5++,a9        ; a9 = temp_S[2]
||             ldw            *b5++,b9        ; b9 = temp_S[5]
               ldw            *b5++,a10       ; a10 = temp_S[6] - additional parity
               nop
               xor            a7,b7,b4
               xor            a8,b8,a4
               xor            a4,b4,b4
||             xor            a9,b9,a4
               xor            a4,b4,b4
               xor            b4,a10,a10

               ldw            *a0++,a4        ; load data[0]
||             ldw            *b0++,b4        ; load data[16]
               nop     3
               ldw            *a0++,a4        ; load data[1]
||             ldw            *b0++,b4        ; load data[17]
               ; a4 = data[0], b4 = data[16]
               xor            a4,b8,b8
||             xor            b4,a9,a9
||             xor            a4,a10,a10
               xor            a4,b9,b9
```

```
||              xor             b4,b7,b7
||              xor             b4,a10,a10
                xor             b4,b8,b8
                ldw             *a0++,a4        ; load data[2]
||              ldw             *b0++,b4        ; load data[18]
                ; a4 = data[1], b4 = data[17]
                xor             b4,a8,a8
||              xor             a4,b7,b7
||              xor             a4,a10,a10
                xor             b4,a9,a9
||              xor             a4,b8,b8
||              xor             b4,a10,a10
                xor             b4,b7,b7
                ldw             *a0++,a4        ; load data[3]
||              ldw             *b0++,b4        ; load data[19]
                ; a4 = data[2], b4 = data[18]
                xor             b4,a7,a7
||              xor             a4,a9,a9
                xor             b4,a8,a8
||              xor             a4,b7,b7
||              xor             b4,a10,a10
                xor             b4,a9,a9
||              xor             a4,a10,a10
                ldw             *a0++,a4        ; load data[4]
||              ldw             *b0++,b4        ; load data[20]
                ; a4 = data[3], b4 = data[19]
                xor             a4,a8,a8
||              xor             b4,b8,b8
||              xor             a4,a10,a10
                xor             a4,a9,a9
||              xor             b4,b9,b9
||              xor             b4,a10,a10
                xor             b4,a9,a9
                ldw             *a0++,a4        ; load data[5]
||              ldw             *b0++,b4        ; load data[21]
                ; a4 = data[4], b4 = data[20]
                xor             b4,a8,a8
||              xor             a4,a10,a10
                xor             b4,b7,b7
||              xor             a4,a7,a7
||              xor             b4,a10,a10
                xor             b4,b8,b8
||              xor             a4,a8,a8
                ldw             *a0++,a4        ; load data[6]
||              ldw             *b0++,b4        ; load data[22]
                ; a4 = data[5], b4 = data[21]
                xor             a4,b7,b7
||              xor             b4,a9,a9
||              xor             a4,a10,a10
                xor             a4,b9,b9
||              xor             b4,a7,a7
||              xor             b4,a10,a10
                xor             b4,b7,b7
                ldw             *a0++,a4        ; load data[7]
||              ldw             *b0++,b4        ; load data[23]
                ; a4 = data[6], b4 = data[22]
                xor             b4,a9,a9
||              xor             a4,b8,b8
||              xor             a4,a10,a10
                xor             b4,b7,b7
||              xor             a4,a9,a9
||              xor             b4,a10,a10
                xor             b4,b9,b9
                ldw             *a0++,a4        ; load data[8]
||              ldw             *b0++,b4        ; load data[24]
                ; a4 = data[7], b4 = data[23]
                xor             a4,a8,a8
||              xor             b4,b8,b8
||              xor             a4,a10,a10
                xor             a4,b7,b7
||              xor             b4,a8,a8
||              xor             b4,a10,a10
                xor             b4,a9,a9
                ldw             *a0++,a4        ; load data[9]
||              ldw             *b0++,b4        ; load data[25]
```

```
               ; a4 = data[8], b4 = data[24]
               xor          b4,a7,a7
||             xor          a4,a9,a9
               xor          b4,a8,a8
||             xor          b4,a10,a10
               xor          b4,b7,b7
||             xor          a4,a7,a7
||             xor          a4,a10,a10
               ldw          *a0++,a4       ; load data[10]
||             ldw          *b0++,b4       ; load data[26]
               ; a4 = data[9], b4 = data[25]
               xor          a4,a9,a9
||             xor          b4,b8,b8
||             xor          a4,a10,a10
               xor          a4,b9,b9
||             xor          b4,a9,a9
||             xor          b4,a10,a10
               xor          b4,b7,b7
||             xor          b4,b9,b9
               ldw          *a0++,a4       ; load data[11]
||             ldw          *b0++,b4       ; load data[27]
               ; a4 = data[10], b4 = data[26]
               xor          b4,a8,a8
||             xor          a4,b8,b8
||             xor          a4,a10,a10
               xor          b4,a9,a9
||             xor          a4,a8,a8
               xor          b4,b7,b7
||             xor          b4,b8,b8
||             xor          b4,a10,a10
               ldw          *a0++,a4       ; load data[12]
||             ldw          *b0++,b4       ; load data[28]
               ; a4 = data[11], b4 = data[27]
               xor          a4,a7,a7
||             xor          b4,b7,b7
||             xor          a4,a10,a10
               xor          a4,b7,b7
||             xor          b4,a7,a7
||             xor          b4,a10,a10
               xor          b4,a8,a8
||             xor          b4,a9,a9
               ldw          *a0++,a4       ; load data[13]
||             ldw          *b0++,b4       ; load data[29]
               ; a4 = data[12], b4 = data[28]
               xor          b4,a8,a8
||             xor          a4,b9,b9
||             xor          a4,a10,a10
               xor          a4,a8,a8
||             xor          b4,b7,b7
||             xor          b4,a10,a10
               xor          b4,b9,b9
               ldw          *a0++,a4       ; load data[14]
||             ldw          *b0++,b4       ; load data[30]
               ; a4 = data[13], b4 = data[29]
               xor          a4,a7,a7
||             xor          b4,b8,b8
||             xor          a4,a10,a10
               xor          a4,b8,b8
||             xor          b4,a7,a7
||             xor          b4,a10,a10
               xor          b4,a9,a9
               ldw          *a0++,a4       ; load data[15]
||             ldw          *b0++,b4       ; load data[31]
               ; a4 = data[14], b4 = data[30]
               xor          b4,a7,a7
||             xor          a4,b9,b9
||             xor          a4,a10,a10
               xor          b4,a9,a9
||             xor          b4,b9,b9
||             xor          b4,a10,a10
               xor          b4,b7,b7
||             xor          a4,a7,a7
               nop
               ; a4 = data[15], b4 = data[31]
               xor          a4,b7,b7
```

```
||              xor             b4,a7,a7
||              xor             b4,b9,b9
||              xor             a4,a10,a10
                xor             a4,b8,b8
||              xor             b4,a8,a8
||              xor             b4,a10,a10
                xor             a4,b9,b9
||              xor             b4,b8,b8

                or              a7,a8,a2
||              or              b7,b8,b2
||              mvk             64,a4
                or              a2,a9,a2
||              or              b2,b9,b2
||              add             a0,a4,a0        ; a0 points to data[0] of next block
||              add             b0,a4,b0        ; b0 points to data[16] of next block
                or              a2,b2,a2        ;
                or              a10,a2,a2       ; check if any temp_S != 0
||              add             a5,16,a5        ; a5 points to parity[0] of next block
||              add             b5,12,b5        ; a5 points to parity[3] of next block

 [a2]           b               correct_errors
                sub             a1,1,a1         ; decrease loop counter
                nop             4
end:
 [a1]           b               loop
 [!a1]          b               b3
||[!a1]         ldw             *++b15,a10      ; pop a10 from the stack
                nop             4
                mv              a3,a4


correct_errors
                ; first calculate S[i]
                mvk             31,a0           ; i = 31
||              mvk             31,b0
syndrome_loop:
                shru            a7,a0,a5        ; temp_S[0] >> i
||              shru            b7,b0,b5        ; temp_S[3] >> i
                and             a5,0x1,a5       ; a5 & 0x1
||              and             b5,0x1,b5       ; b5 & 0x1
                shl             b5,3,b5         ; b5 << 3
                or              a5,b5,a2
                shru            a8,a0,a5        ; temp_S[1] >> i
||              shru            b8,b0,b5        ; temp_S[4] >> i
                and             a5,0x1,a5       ; a5 & 0x1
||              and             b5,0x1,b5       ; b5 & 0x1
                shl             a5,1,a5         ; a5 << 1
                shl             b5,4,b5         ; b5 << 4
                or              a5,b5,a4
                or              a4,a2,a2
||              shru            a9,a0,a5        ; temp_S[2] >> i
||              shru            b9,b0,b5        ; temp_S[5] >> i
                and             a5,0x1,a5       ; a5 & 0x1
||              and             b5,0x1,b5       ; b5 & 0x1
                shl             a5,2,a5         ; a5 << 2
||              mvk             0x0,b1
                shl             b5,5,b5         ; b5 << 5
                or              a5,b5,a4
||              mvk             0x1,b2
                or              a4,a2,a2        ; a2 = S[i]
||              shl             b2,b0,b2        ; b2 = (0x1 << i)

                and             a10,b2,b2
 [a2]           mvk             0x1,b1          ; b1 = 1 if (a2>0), b1 = 0 if (a2 == 0)
                shru            b2,b0,b2        ; b2 = corresponding parity_bit
                and             b1,b2,b1
||              xor             b1,b2,b2
 [b1]           b               find_coset ; if ((S[i]>0) && (parity_bit>0)) correct 1 error
 [b2]           b               b3          ; if ((S[i]>0) && (p==0)) || ((S[i]==0) && (p>0))
 [b2]           mvk             -1,a4           ;    then return -1 (can't correct the errors)
||[!a2]         sub             a0,1,a0
||[b2]          ldw             *++b15,a10      ; pop a10 from the stack
 [!a2]          sub             b0,1,b0
                cmplt           b0,0x0,b1
```

```
              nop          2
 [!b1]        b            syndrome_loop
syndr_loop_end:
              mvk          28,a2
              add          b6,a2,a5       ; a5 -> parity[0] of next block
 [b1]         b            end
              add          12,a5,b5       ; b5 -> parity[3] of next block
              mvk          128,b1
              add          a6,b1,a0       ; a0 -> data[0] of next block
||            mvk          192,b2
              add          a6,b2,b0       ; b0 -> data[16] of next block
              mv           a0,a6          ; a6 = a0
||            mv           a5,b6          ; a5 = b6

find_coset:
              mvkl         COSET,b1
||            mvk          0x0,a5
              mvkh         COSET,b1
              mvk          -1,b5
              mvk          N,b0           ; number of cosets
coset_loop:
              ldb          *b1++,b2       ; load coset
              sub          b0,1,b0
 [b0]         b            coset_loop
              nop          2
              xor          a2,b2,b2       ; compare coset with S[i]
 [!b2]        mv           a5,b5          ; b5 = index of matching coset
              add          a5,1,a5

              cmplt        b5,0,b2               ; check if b5<0
 [b2]         b            b3                    ; if yes then return -1
||[b2]        ldw          *++b15,a10     ; pop a10 from the stack
 [b2]         mvk          -1,a4
              mvk          K,a2
              cmplt        b5,a2,b2              ; check if (found index < K)
 [!b2]        sub          b5,a2,b5
              shl          b5,2,b5        ; multiply by 4, because int = 4 bytes

 [b2]         add          a6,b5,b4
 [!b2]        add          b6,b5,b4
              ldw          *b4,b2         ; load value to correct
||            sub          a0,1,b0
              mvk          0x1,a2
              cmplt        b0,0x0,b1
 [b1]         b            syndr_loop_end
 [!b1]        b            syndrome_loop  ;next syndrome
              shl          a2,a0,a2       ; a2 = 0x01 << i
              add          a3,1,a3        ; total_errors++
              xor          b2,a2,b2       ; repair the error
              stw          b2,*b4         ; store corrected value
              mv           b0,a0
```