SINGLE EVENT UPSET ERROR PROTECTION FOR SOLID STATE DATA MEMORY ON MICROSATELLITES

M.S. Hodgart, C.I. Underwood, J.W. Ward
Surrey Satellite Technology Ltd.
Centre for Satellite Engineering Research
University of Surrey, Guildford, Surrey GU2 5XH, UK

Many microsatellite missions rely upon large arrays of CMOS static RAM for data storage. Even a satellite with strict power and volume limitations can carry several Mbytes of SRAM. Experience has shown that these memory arrays are useful for scientific data collection, image storage, store-and-forward message switching and spacecraft telemetry monitoring. This paper describes the authors' design and implementation of error protection codes which protect large CMOS RAM arrays from radiation-induced single event upsets.

Introduction

Many microsatellite missions require that large amounts of data be stored on-board. In some cases, the storage area is used to buffer data generated at a high rate so that it can be transmitted at a lower rate. This is particularly true of space science missions. Store-and-forward communications and remote imaging microsatellites must store data until the downloading station comes within the satellite footprint. Data storage time ranges from seconds to days, and requirements of up to tens of megabytes are not uncommon.

In large satellites, these storage requirements are usually met by data recorders using moving magnetic tape. Microsatellite designers, however, have traditionally shied away from power-hungry, continuously-moving mechanisms such as those found in tape recorders. When faced with multi-megabyte data storage requirements, they have turned to CMOS static RAM (SRAM). CMOS SRAMs are compact, consume little power, and can be assembled easily into flexible data storage systems.

One disadvantage of CMOS SRAMs is their sensitivity to radiation. Total ionising dose gradually shifts the thresholds of the CMOS FETs inside SRAMS, causing increased power consumption and eventual catastrophic failure.[1] Also, cosmic particles passing through or stopping in SRAMs can deposit sufficient charge to change the contents of memory cells - called *single event upset (SEU)*. For any particular mission, these radiation risks must be evaluated and, if necessary, the memories or data must be protected from radiation effects.

This paper describes an effective method of detecting and correcting SEUs in data stored in SRAMs on-board spacecraft. The technique described will protect

---

1. This paper will not address total-dose effects, except to note that after 7 years in polar orbit, UoSAT-2s on-board memory systems show no appreciable signs of total dose failure.

data which is both written to and read from memory in *blocks*. It is not suitable for protecting programs or data which are to be accessed on a byte-by-byte basis directly by a central processing unit (CPU). The error-detection and correction (EDAC) block code described here is implemented in software on the UoSAT-2, UoSAT-3 and UoSAT-5 microsatellites.

Constraints

We began developing this code to protect 96 kbytes of SRAM on the UoSAT-2 store-and-forward Digital Communications Experiment (DCE). Although the RCA 1802 on-board computers of both UoSAT-1 and UoSAT-2 had SEU-protected program memory, the DCE was the first UoSAT on-board computer to use dense SRAMs with large storage words. These devices are *byte wide* - storing 8 data bits at each address. They are also dense, with up to 64 kilobits per chip. These SRAMs form a bank-switched "RAM Unit" used for message storage (not to run programs from).

This was the first time that we had attempted to protect *byte-wide* RAMs used for bulk data storage. For some time, we and others had been protecting small 1- and 4-bit wide dynamic and static RAMs using hardware implementations of the Hamming (12,8) code. The literature carried no information concerning SEU effects in high-capacity, byte-wide SRAMs, and we feared that a single cosmic particle striking a dense memory device might alter several adjacent bits. To protect our data from these multi-bit SEUs, we needed a code which could restore an entire byte missing from the code block.

Code block size effects encoding/decoding efficiency and code overhead (the ratio of the useful data bytes to redundant bytes). The size of the code block also becomes the size of smallest addressable data element in the storage system. Storage allocations must be an integral number of code blocks. With an (n,k) coding system, up to (k-1) slack bytes may have to be added to a storage allocation to meet this constraint. Similarly, to change one data byte, an entire code block must be read, decoded, altered, encoded and re-written.

We first implemented the byte-correcting code to run on a Z-80 with a 0.8 MHz clock. This CPU also handled store-and-forward communications links at 1.2 kbits/second, so efficient block encoding and decoding algorithms were absolutely necessary. The code has since been implemented on 80C186 CPUs in multi-tasking systems, where there are similar CPU-loading constraints. The primary novelty of the coding system presented here is in the design of an encoder and decoder which can be implemented efficiently on general purpose CPUs.

The (255,252) Error Detection and Correction Code

The coding system was designed for speed and simplicity in implementation. Data are handled in blocks of 252 bytes. Whenever a data block is written to memory, an additional 3 parity bytes are derived from the data and appended to the block. Consequently a code block of structure (255,252) has been created.

At some later time, on reading back all 255 bytes from memory, the entire block is analysed for errors. Our choice of code allows one entire byte to be in error and fully correctable. Alternatively, if there should be two bytes in error, this situation is distinguishable, although the location and nature of these actual errors cannot be calculated.

The structure of the code will be recognised as a standard Reed Solomon code. However, the non-standard encoding process we have used is of interest. This encoding algorithm (also used in the decoding) involves a *remainder transform* described in the following paragraphs. The encoder was designed to be implemented in software using small look-up tables, but a hardware implementation is also envisaged.

Construction of Encoder Look-up Tables

Assuming that the entire encoding and decoding process is to be implemented in software, we require six different, pre-calculated look-up tables. Two tables have 255 bytes each, three have 256 and one has 72, for a total table space of 1350 bytes. These tables are generated once, by initialisation routines. They can be generated on the ground and uploaded to the satellite or generated as part of the initialisation of the on-board data handling software. The algorithms for constructing the tables are described here.

Five of the tables $F_0[]$, $F_1[]$, $F_2[]$, IGF[] and GF[] are common in coding theory. The sixth table C[] is quite different, and possibly unique to our implementation.

The basis for table generation is a software realisation of the standard shift-register with feedback through EX-OR gates, embodying the primitive polynomial

$$p(x) = 1 + x^2 + x^3 + x^4 + x^8 \qquad \text{(Eqn. 1)}$$

Tables GF[] and IGF[]

GF[] is the Galois Field table and IGF[] is the inverse Galois Field. In generating both the these tables, we start with this shift register containing 1, and shift the register step-by-step through 255 increments. If we call the contents of the register after I shifts reg(I), then the two tables are generated as follows.

$$GF[I] \leftarrow reg(I) \qquad \text{(Eqn. 2)}$$

$$IGF[reg(I)] \leftarrow I \qquad \text{(Eqn. 3)}$$

The shift register repeats for I greater than 254, setting the limit to both tables. Table GF[] will have entries from GF[0] to GF[254], and IGF[] will be found to have entries from IGF[1] to IGF[255]. The table GF[] should be regarded as containing data bytes looked up by natural numbers, whilst IGF[] should be regarded as a table containing natural numbers looked up by data. Clearly, these tables can be filled by a single loop iterating I from 0 to 254.

Tables GF[] and IGF[][1]

i	GF[i] Bit field	GF[i] Byte
0	10000000	1
1	01000000	2
	...	
7	00000001	128
8	10111000	29
	...	
24	11110001	143
25	11000000	3
26	01100000	6
	...	
174	10001111	241
175	11111111	255
176	11000111	227
	...	
254	01110001	142

i	IGF[i] Bit field	IGF[i] Byte
1	00000000	0
2	10000000	1
3	10011000	25
	...	
6	01011000	26
	...	
29	00010000	8
	...	
128	11100000	7
	...	
142	01111111	254
143	00011000	24
	...	
227	00000001	176
	...	
255	11110101	175

General Coding Tables

The general look-up tables, used for encoding and decoding, are $F_0[]$, $F_1[]$ and $F_2[]$. These are generated from GF[] and IGF[] using the simple rule:

$$F_j[i] \leftarrow GF[\ (IGF[i] + j + 1)\ MOD\ 255\ ], \quad j \leftarrow 0,1,2 \qquad \text{(Eqn. 4)}$$

Which fills all entries except for the substitution

$$F_j[0] \leftarrow 0 \qquad \text{(Eqn. 5)}$$

The total storage required for each table is just 256 bytes.

Note that the byte-wide EXOR addition of any two or more addresses maps to the byte-wide EXOR addition of the table contents at those addresses. For example,

$$F_j[k \oplus l \oplus m] = F_j[k] \oplus F_j[l] \oplus F_j[m] \qquad \text{(Eqn. 6)}$$

where k, l and m are arbitrarily chosen data bytes.

This general property of linearity is of immediate significance to an alternative implementation by a discrete logic matrix. It is not difficult to formulate a matrix of EXOR gates to which the application of a specific byte yields the correct F_j output. This would be the route to follow in a hardware implementation.

We believe that the sixth table C[] is unique to our work, and it will be described later.

---

1. Here and in following tables, bit fields are presented with the least significant bit on the left.

Factored Decoder

We start with a description of the standard factored decoder for a (255,252) block. Whilst the decoding algorithm does not involve any novelties, the corresponding factored encoder described later relies on the remainder transform. For this discussion, assume that we have a correctly encoded (255,252) block which may now contain an error.

Within a given code block, the notation V'_i refers to the i^{th} byte in the (possibly erroneous) block. The indices i from 0 to 2 identify the parity bytes, previously created by an encoder, while the indices in the range from 3 to N-1 identify the data bytes. For our code, the maximum value of N is 255.

Factored decoding in software is straightforward. Three syndrome bytes are derived: S_0, S_1 and S_2. Derivation of each syndrome involves use of one look-up table and an EXOR feedback mechanism. The highest order data byte is applied to the look-up table. The output of the table is then EXORed with the next most significant byte and the result used as a new address into the table. This is repeated until all bytes in the code block have been consumed. Exactly the same procedure is used to derive each syndrome; only the look-up table is changed.

In pseudo code, the decoder action can be expressed as:

$S_0 \leftarrow 0$
$S_1 \leftarrow 0$
$S_2 \leftarrow 0$
For i ← N-1 downto 0
    $S_0 \leftarrow V'_i \oplus F_0[S_0]$
    $S_1 \leftarrow V'_i \oplus F_1[S_1]$
    $S_2 \leftarrow V'_i \oplus F_2[S_2]$
Next i

(Eqn. 7)

The complete calculation requires 3*N look-ups and 3*N EXOR operations.

In terms of coding theory, each calculated syndrome is identical to the remainder which would be obtained by division of a polynomial representation of the code, using each polynomial factor. In the theory of Reed Solomon codes, deriving this set of syndromes is a standard first step towards decoding. Equivalently, these calculations find three 'harmonics' of a Fourier Transform of the code block (Sweeney 1990).

In the absence of error, which is the usual situation, all syndromes will calculate to zero, because no error has been detected. In this case, the error detection is complete.

If, however, all S_j are non-zero, then confirmation and location of a single correctable byte is as follows. The process requires the GF[] and IGF[] tables, and is likely always to be implemented in software. The process may be explained without reference to complex theory. Each syndrome pattern is used as an 8-bit address to the IGF[] table. As we saw, at each address in this table, there is a corresponding number - the exponent - in the range 0 to 254.

This IGF[] look-up table - addressed by the syndromes S_0, S_1 and S_2 - outputs exponents which are processed to give two estimates of the error location,

called i and j. The calculations are in conventional arithmetic:

$$i \leftarrow (IGF[S_2] - IGF[S_1] + 255) \text{ MOD } 255$$
$$j \leftarrow (IGF[S_1] - IGF[S_0] + 255) \text{ MOD } 255$$

(Eqn. 8)

If i is equal to j, then V'_i is the erroneous byte. The error pattern itself is calculated using GF[] as well as IGF[].

$$E \leftarrow GF[(IGF[S_0] - i + 255) \text{ MOD } 255]$$

(Eqn. 9)

The decoding process completes with the correction of the offending byte:

$$V_i \leftarrow V'_i \oplus E$$

(Eqn. 10)

If only some of the syndromes are non-zero, or the calculated values of i and j are unequal, then the decoder has detected an uncorrectable error pattern.[1]

The Factored Encoder

Having reviewed a standard method of factored decoding, we now describe the equivalent procedure for encoding. For the encoder we adopt a vector notation in which the data to be encoded are in the message vector, M. M_i is the data byte with index i (with i ranging from 0 to K-1). The maximum value of K is 252.

Using the same 'look-up with feedback' mechanism as in the decoder, we generate three remainders: R_0, R_1, and R_2. In pseudo code:

$R_0 \leftarrow 0$
$R_1 \leftarrow 0$
$R_2 \leftarrow 0$
For i ← K-1 downto 0
    $R_0 \leftarrow M_i \oplus F_0[R_0]$
    $R_1 \leftarrow M_i \oplus F_1[R_1]$
    $R_2 \leftarrow M_i \oplus F_2[R_2]$
Next i

(Eqn. 11)

This is seen to be almost exactly the same procedure as used for finding the syndromes in the decoder. The only difference is that the three fewer bytes are involved, since only the 252 data bytes exist at this stage.

The standard coding theory shows that we cannot leave the process at this point. To write the data plus these remainder bytes into the memory, although obvious and seemingly attractive, would be a mistake. The detailed

---

1. A decoder based on the (256,252) code will correct two-byte errors.

mathematical justification is beyond the scope of this paper, but specialists will know that the encoders are normally implemented quite differently using an 'n-k' encoder.

Our encoding process continues using the look-up table C[]. We must use C[] to transform R_0, R_1 and R_2 into the correct check bytes. This is a 24-bit to 24-bit transformation, as we start with R_0, R_1 and R_2 and end up with C_0, C_1 and C_2.

We will consider the three remainder bytes to form a single 24-bit word R24, with R_0 supplying the least significant 8 bits, R_1 the middle bits and R_2 the most significant bits. The bits in R24 are numbered from 0 (least significant) to 23 (most significant). Our final look-up table C[] contains one entry for each of these bit positions, and each entry in C[] is itself a 24-bit value. Thus, C[] is a 24 by 24 table:

Table C[]

i	C[i] Bit Field
0	01110110 10001010 10001100
1	00111011 01000101 01000110
2	10100101 10011010 00100011
3	11101010 01001101 10101001
4	01110101 10011110 11101100
5	10000010 01001111 01110110
6	01000001 10011111 00111011
7	10011000 11110111 10100101
8	10011110 00001000 10001010
9	01001111 00000100 01000101
10	10011111 00000010 10011010
11	11110111 00000001 01001101
12	11000011 10111000 10011110
13	11011001 01011100 01001111
14	11010100 00101110 10011111
15	01101010 00010111 11110111
16	11101010 10011110 01110110
17	01110101 01001111 00111011
18	10000010 10011111 10100101
19	01000001 11110111 11101010
20	10011000 11000011 01110101
21	01001100 11011001 10000010
22	00100110 11010100 01000001
23	00010011 01101010 10011000

To generate C_0, C_1 and C_2 from our 24-bit remainder R24, we first generate a 24-bit value C24. C24 is first set to 0. Then, considering each bit position in R24, if the bit at the selected position is a 0, no action is taken. If the bit is a 1, then the entry from the table C[] for that bit position is EXORed into C24. Defining $R24_i$ to be the i^{th} bit of R24, the pseudo code is:

```
C24 ← 0
For i ← 0 to 23
      if R24_i = 1 then C24 ← C24 ⊕ C[i]
Next i
```

(Eqn. 12)

The register C24 now contains the correct check bits for our message vector. We form 3 check bytes by taking the bits from C24: C_0 has the bits 0 to 7, C_1 bits 8 to 15 and C_2 bits 16 to 23.

We treat the block of message bytes and the three check bytes as the complete code block. Formally, this comprises the following shift:

$$V_{i+3} \leftarrow M_i \qquad (\text{for } 0 <= i < K) \qquad \text{(Eqn. 13)}$$

$$V_i \leftarrow C_i \qquad (\text{for } 0 <= i < 3) \qquad \text{(Eqn. 14)}$$

Theoretical Justification

We can implement this encoder efficiently using general purpose microprocessors. On average only 12 look-ups and 12, 14-bit wide EXOR operations are required for the final transform. This works because the subset of all possible message vectors (M), which yield a common set of remainders (R_i), uniquely maps to a common set of check bytes (C_i). It is required only to 'look up' the latter via the former. Suppose, for example, that the following R_i are derived from some given data M

$$R_0 = 10000000 \quad R_1 = 00000000 \quad R_2 = 00000000 \qquad \text{(Eqn. 15)}$$

Then the correct check sequence would be C[1], split into its three constituent bytes.

$$C_0 = 01110110 \quad C_1 = 10001010 \quad C_2 = 10001100 \qquad \text{(Eqn. 16)}$$

and this would be true for every data block M which generated those remainders. Perhaps more surprising is that an arbitrary 24-bit remainder pattern maps to a 24-bit check pattern by a linear combination of the entries in table C[].

It can be shown that this result is a specific application of the Chinese Remainder Theorem.

Summary of Coding and Decoding Action

The complete construction reduces to the following steps.

For encoding:

(E0) Assemble data into a block of bytes - M_0 to M_{252}

(E1) Derive remainders R_0, R_1 and R_2 using Eqn. 11 (either in software or hardware).

(E2) Form a 24-bit word with the remainders and transform this using Eqn. 12 into three check bytes C_0, C_1 and C_2.

(E3) Write the data and the check bytes into memory as V_0 through V_{254}.

For decoding:

(D0) Read 255-byte block into V'_0 to V'_{254}

(D1) Derive syndromes S_0, S_1 and S_2 as per Eqn. 7.

(D2) If an error is detected then process the syndromes further in order to try to locate the error (Eqn. 8, 9 and 10).

The steps D1 and E1 use the same software functions, just on a different number of bytes. The remainder transform in step E_2 transforms remainder bytes to valid check bytes in relatively few steps. The error computation (D2) could have been done in a number of well-known ways; a convenient method has been described here.

Practical Implementation

We designed the encoding and decoding algorithms described here so that we could easily implement them in efficient software. Hence, moving from the algorithms and pseudo code to software in high- or low-level programming languages is not complicated. Selecting the proper data representations and optimising code are, nevertheless, important.

The data are stored in byte arrays throughout the encoding or decoding process. In our implementation, the data are organised so as to avoid the 252-byte block move implied by Eqn. 13. Although this move makes sense in the notation of the encoding/decoding algorithms, translating it to a physical memory move is unnecessary and wastes processor cycles. The data are initially stored in array elements 3 through 254, and the proper transformation of indices is applied to the encoding loop of Eqn 12.

The EXOR feedback loop is common to both the encoder and the decoder, as it implements steps E1 and D1. The CPU must execute this loop for every byte of data stored in or retrieved from the mass memory. Because this loop is executed most frequently, it is the primary target for code optimisations. In our implementation for the 80C186, changing this loop from C-language code to code written in assembly language improved loop performance by 30%. This loop can be easily coded in assembly language, and the performance gain is worth the additional implementation effort.

Using the feedback loop function are the encoding function and the decoding function. In turn, these are called by functions which write and read 252-byte data blocks to and from the mass memory. In practice, the each 255-byte code block is assigned 256 bytes of storage in the mass memory, for ease of addressing. The extra byte of storage is unused in our current application, but is conveniently available for an extended (256,252) code which can correct two erroneous bytes in one block. These blocks are analogous to the *sectors* on a magnetic disk, as they are the smallest units which can be read and written effectively. In keeping with the disk analogy, 4 sectors are usually grouped into a *clusters* which stores 1008 data bytes. Higher level functions insulate the applications programmer from these details. The applications programmer uses the familiar C-language functions fopen(), fclose(), fread() and fwrite() which operate just like their disk-based counterparts.

The UoSAT implementation of the spacecraft data storage system with block-error protection is used on several satellites (UoSAT-3, UoSAT-5, AMSAT-OSCAR-16 and LUSAT-OSCAR-17). These satellites all use the Quadron multi-tasking operating system called qCF, and several tasks may need access to the data memory. A single file system server task fulfils data read and write requests and implements the block code described here. The server also periodically washes the code blocks to avoid multiple errors which would overcome our single-error correcting code.

Figure 1
Hierarchy of Functions

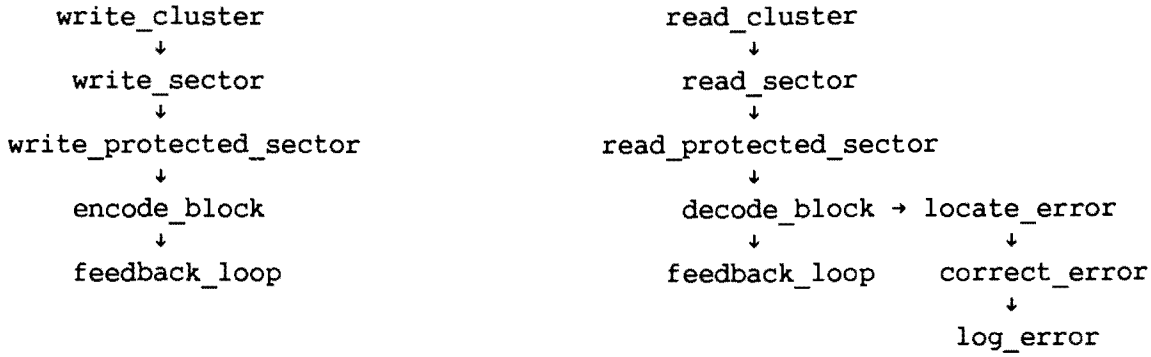
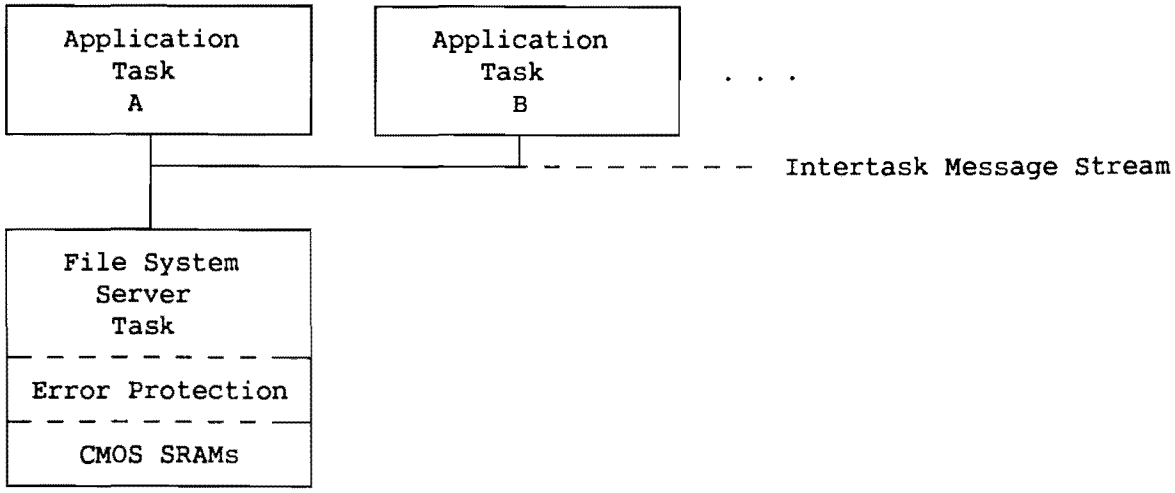


Figure 2
Microsatellite File System Software



This hierarchy provides the necessary error control to detect and correct SEUs in CMOS SRAMs, but hides these specialised functions from the application programmer. If necessary, the underlying codes can be altered (perhaps to add double-byte error correction) without effecting any of the applications programs. This insulation is important when satellite software has been implemented by several authors and is expected to serve a number of microsatellite missions.

Memory Wash Rates and Uncorrectable Error Probabilities

Ignoring for the moment the possibility that a single particle event can cause multiple bit errors spread over a number of bytes, the single-byte correcting code will still fail to eradicate errors if there is a significant chance that two or more independent events can occur in a block of memory in between accesses to that block. For this reason, it is required to 'wash' the memory (i.e. to read, correct and write back each block in turn) sufficiently often so as to minimise the chance of multiple bytes being affected within a block simply due to the accumulation of bit-errors. For a block size of 256 bytes, it may be reasonably assumed that this reduces to the problem of ensuring that the probability of two or more single particle events occurring in that block is acceptably small.

However, washing the memory implies an overhead in terms of processing time, and therefore we would wish to minimise the rate at which the memory is washed in order to maximise the processing time available for more useful tasks. It is therefore necessary to find the slowest wash rate which will still give acceptable protection against multiple-byte (and therefore uncorrectable) errors.

Suppose we split a large semiconductor memory system into 'n' blocks, each containing 'b' bits. Suppose further that the memory is constructed from devices which are susceptible to single-event upset with a underlying mean error-rate of 'p' SEUs per bit-day.

Each memory block is encoded with the error correcting code described in previous sections. This code can detect and correct any single byte-error, but is not able to correct two or more bytes in error.

If the wash period for a block is 't' days, then the time taken to cycle around the whole memory and re-visit any particular block is simply 'nt' days.

In-between washes, the block may be affected by one or more SEUs. The occurrence of SEUs should be a Poisson process, and so the probability of 'r' upsets occurring in a given 'observation time' is dictated by a single variable - the mean expected value 'u' :

$$P(r) = e^{(-u)} u^r / r! \qquad \text{(Eqn. 17)}$$

If we set the observation time to be 'nt' (the period between washes of any block), then

u = underlying SEU rate * number of bits per block * observation time

$$u = p\ b\ n\ t \qquad \text{(Eqn. 18)}$$

Thus, the probability of 'r' SEUs occurring in a block between washes is:

$$P(r) = e^{(-pbnt)}\ pbnt^r / r! \qquad \text{(Eqn. 19)}$$

Thus, the probability of zero errors is:

$$P(0) = e^{(-pbnt)} \qquad \text{(Eqn. 20)}$$

and the probability of a single error is:

$$P(1) = e^{(-pbnt)}\ pbnt \qquad \text{(Eqn. 21)}$$

Hence, the probability of two or more errors is :

$$P(>1) = 1 - [\ e^{(-pbnt)} + e^{(-pbnt)}\ pbnt\ ] \qquad \text{(Eqn. 22)}$$

i.e. this is the probability of seeing more than one error accumulate in any single block between washes : 'x'.

As the wash period is reduced, the probability of more than one error occurring in any block is also reduced. However, we are searching for the maximum acceptable period between washes so that we can cut down on the software overheads, hence the problem becomes one of maximising 't' the period between washing consecutive blocks, whilst still achieving an acceptably low probability of seeing an uncorrectable error in the entire memory over a given time.

Having calculated the probability of seeing an uncorrectable error occur in a block between washes (i.e. 'x'), we can find the expected number of uncorrectable errors which will occur in the entire memory of 'n' blocks as simply 'xn'. Of course, this is over an observation period of 'nt' days, which is a function of our variable 't' the wash period. We may normalise the probabilities to a 'per day' measure by simply dividing 'nt' into one day i.e. we may say that each period of 'nt' constitutes a trial and that there are 1/nt trials per day.

Thus, the expected probability of seeing a uncorrectable error occur in the entire memory over the course of one day is:

$$v = \text{probability of uncorrectable error in a block} * \text{No. of blocks} * \text{No. of trials per day}$$

$$v = (x\ n) / (n\ t)$$

$$v = x / t \qquad \text{(Eqn. 23)}$$

uncorrectable errors per day in the entire memory.

As the underlying mechanism for creating these errors is a Poisson process, the number of uncorrectable errors per day should also be a Poisson process, and so we may further find the probability of seeing 'q' such errors in time 'T' (say a week or a year) :

$$P(q) = e^{(-vT)}\ vT^{q} / q! \qquad \text{(Eqn. 24)}$$

Our acceptable limit may then be stated as follows:

There should be a 95% chance of seeing no uncorrectable errors over the period of T days (say a week), i.e.

$$P(0) = e^{(-vT)} >= 0.95 \qquad \text{(Eqn. 25)}$$

This puts a constraint of the maximum size of 't' the block wash period.

However, from a practical point of view, the minimum size of 't' is constrained by the time taken to wash the block. Thus, for a given size of memory ('nb' bits), and a given underlying error-rate ('p'), constrained so as to give a 95% probability of zero uncorrectable errors in 'T' days, it may not always be possible to find a value of 't' which is practical.

Using this procedure, it is possible to predict a suitable wash period for the UoSAT-3 PCE Memory.

The PCE memory consists of 4M bytes (33554432 bits) of memory, split into 256 byte (2048 bit) blocks:

b = 2048 bits (256 bytes x 8 bits)
n = 16384 blocks

The blocks are read every 't' seconds in a cluster of 4, so we may modify 'b' and 'n' as follows:

b = 8192 bits per cluster
n = 4096 clusters

The following tables show the probability of achieving no uncorrectable errors over a period (T) for different underlying error rates and wash times:

TABLE 1
Probability (%) of Zero Non-Correctable Errors (1E-6 SEU/Bit-Day)

Wash Period (t) / seconds	Observation Time (T) / days			
	1	7	30	365
0.5	99.67	97.75	90.69	30.46
1.0	99.35	95.54	82.25	9.28
2.0	98.70	91.29	67.66	0.86
4.0	97.43	83.34	45.79	0.01
8.0	94.93	69.48	21.00	0.00

TABLE 2
Probability (%) of Zero Non-Correctable Errors (5E-7 SEU/Bit-Day)

Wash Period (t) / seconds	Observation Time (T) / days			
	1	7	30	365
0.5	99.92	99.43	97.59	74.29
1.0	99.84	98.86	95.23	55.19
2.0	99.67	97.75	90.69	30.46
4.0	99.35	95.54	82.25	9.28
8.0	98.71	91.29	67.67	0.86

TABLE 3
Probability (%) of Zero Non-Correctable Errors (1E-7 SEU/Bit-Day)

Wash Period (t) / seconds	Observation Time (T) / days			
	1	7	30	365
0.5	99.99	99.98	99.90	98.82
1.0	99.99	99.95	99.80	97.65
2.0	99.99	99.91	99.61	95.36
4.0	99.97	99.82	99.22	90.93
8.0	99.95	99.64	98.44	82.68

Observations of the behaviour of the CMOS static RAMs on-board UoSAT-2 gave us an expected error-rate of around 5×10^{-7} SEU/Bit-Day. Thus, to meet the desired goal of only a 5% probability of a non-correctable error occurring in the entire memory in a week, a wash time of 4 seconds was deemed to be acceptable. Thus, the entire memory would be washed every 16384 seconds (i.e. 4 hours, 33 minutes).

In-Orbit Results from UoSAT-3

Regular washing of the PCE memory began in early April 1991. However, during routine monitoring,it was noticed that the occasional 'severe' (i.e. uncorrectable) error was occurring. In the light of this, the wash rate was increased by a factor of 4 to one cluster per second on 20th June 1991.

There has now been sufficient SEU data collected to make a preliminary analysis of the in-orbit performance of the memory.

Over the period 9th April 1991 to 20th June 1991, the daily SEU rate was recorded. Those days which involved software reloads were filtered out, leaving a 60 day observation period, during which time, 2107 SEUs occurred (of which 13 were 'severe'). This gives a mean observed error rate of 35.11 SEUs per day, or when normalised : 1.047×10^{-6} SEU/Bit-Day. This is consistent with a Poisson population whose mean has a 95% probability of lying between 8.047×10^{-7} and 1.400×10^{-6} SEU/Bit-Day. This figure is around double the rate observed on the CMOS SRAMs of UoSAT-2, but is consistent with other 32K x 8 bit memory devices on-board UoSAT-3 in the 1802-based On-Board Computer System.

On 20th June 1991, the wash rate was increased, and, by the end of July 1991, 1419 SEUs had occurred in 41 days of observation, of which 4 were severe. Again this is consistent with an error rate around 1×10^{-6} SEU/Bit-Day.

From these data, it is not possible to tell if there has been a significant change in the number of severe errors commensurate with the change in wash rate. However, given the observed underlying error rate of 1×10^{-6} SEU/Bit-Day, it is clear that the number of observed severe errors far exceeds the number expected from 'random chance':

SEU Rate : 1.05×10^{-6} , 4s per wash, 60 days observation:

Expected No. of Severe Errors : 1.7
Observed No. of Severe Errors : 13

SEU Rate : 1.05×10^{-6} , 1s per wash, 41 days observation:

Expected No. of Severe Errors : 0.3
Observed No. of Severe Errors : 4

Thus, the observed number of severe errors is not consistent with an underlying mechanism of single-bit errors occurring independently. This also implies that 'turning-up' the wash rate will have no significant effect, and that the single-byte correcting code is not sufficient to maintain an error-free memory.

This result, together with observations from the DCE and OBC of UoSAT-2 suggest that, although reasonably rare, some single particles are able to corrupt more than one bit on there passage through a memory device, and further that when this happens it is usually single bits in adjacent or nearly adjacent bytes which are affected.

The authors are currently implementing a code using the techniques described in this paper which will cope with this situation. Also, the nature of the particles which can give rise to these multiple-bit upsets is being investigated through the radiation environment monitoring payloads on UoSAT-3 and UoSAT-5.

Summary

The authors have extensive practical experience with the implementation of software error-detection and correction methods for CMOS memories on-board microsatellites. The (255,252) code, implemented using the algorithms reported here, is an efficient means of protecting large memory spaces. The final encoding step - in which the Chinese Remainder Theorem is implemented as a simple look-up - is thought to be unique to our work. This system now protects more than 30 Mbytes of CMOS SRAM on five microsatellites in low, polar orbits.

Since the (255,252) code only corrects single-byte errors, it is important to consider how practical memory wash rates alter the probability of two independent errors occurring in a single code block. The statistics thus far gathered from the UoSAT-3 memory bank (4 Mbytes), show that the observed double-byte errors are not consistent with independent Poisson arrival of SEUs. A single event, perhaps a heavy ion collision, has caused these multiple-byte upsets. Thus, we conclude that a double-byte correcting (256,252) code based on the techniques presented here will be necessary where extremely high data integrity is required, or in orbits where SEU rates are significantly greater.