Efficient protection against data errors in embedded control software

MICHAEL SHORT[1], MICHAEL SCHWARZ[2] and JOSEF BOERCSOEK[2]
[1]Department of Engineering,
University of Leicester,
University Road,
Leicester,
UK

[2] Department of Engineering,
Universitat Kassel,
Wilhelmshoeher Allee 73,
34121 Kassel,
GERMANY

Abstract: - This paper presents a novel approach to tolerating transient data faults that may affect the software executing on commercial-of-the-shelf (COTS) embedded processors. The main concept of the approach is the use of information redundancy, in which the program user data and user stack areas are duplicated byte-for-byte in areas of RAM known as mirror arrays. We also present a novel approach to implementing the management of this data duplication and fault detection/correction in software which results in highly readable, portable code. Preliminary results are reported for a matrix multiplication program which indicates the effectiveness of the technique.

Key-Words: - Software Fault Tolerance, Embedded Systems.

1 Introduction

Modern control systems are almost invariably implemented using some form of embedded digital computer system [1]. The dominance of digital systems in this field is a consequence of the low cost, increased flexibility, greater ease of use, and increased performance of digital control algorithms when compared with equivalent analogue implementations [2][3].

When such embedded systems are used in situations where their correct functioning is vital, special care must be taken to ensure that the system is both reliable and safe [4][5]. In particular, care must be taken to ensure that both transient and permanent memory faults - such as Single Event Effects (SEE's) caused by particle strikes - must not cause the program execution to veer from its desired trajectory and cause the system to enter potentially dangerous situations.

Previous research has demonstrated that SEE's may manifest themselves in a variety of ways. They may cause transient disturbances known as Single Event Upsets (SEU's) - manifested as random bit-flips in memory. They may also cause permanent stuck-at faults over an array of memory, caused by damage to the read/write circuitry or chip latchup. Failure rates for SEU's in ground-based installations are in the region of 10^{-9} - 10^{-8} failures per bit per hour [6] and permanent failures in the region of 10^{-8} - 10^{-6} failures per device per hour [7].

Recent years have seen the development of several software-based approaches to implementing transient fault detection on COTS processors, which due to their low-cost are not radiation hardened. These techniques are designed to detect errors caused by transient or permanent hardware faults by relying on specially crafted software, without resorting to special-purpose hardware.

Many of the techniques, for example [8][9][10], are based on similar approaches, in that modifications are made to the application code to detect deviations from the expected execution flow. If any deviations are detected, program execution is suspended and an error recovery procedure is called (oftentimes simply resetting the processor to a known state). They are based around data duplication, instruction duplication and control flow checking.

Although providing relatively high levels of fault-tolerance, such approaches are problematic (from the point of view of the embedded system developer) for two main reasons. The first is that when data duplication is employed, the developer has little control over where the duplicated copies reside in memory. For example, when temporary variables are declared, they are all placed by the compiler in the user stack area - next to one another - as such, they are at risk of a common failure should the memory chip implementing the user stack become faulted.

The second is the sheer complexity involved in applying the techniques. Many are only suitable for use with automatic code generators, which may be problematic from a safety perspective and may cause problems with certification. Conversely, when the techniques are applied by hand, the complexity

of the source code increases dramatically, and the basic meaning of the code itself becomes obscured - this can cause problems with maintainability and testing. For example, consider the segment of C code shown in figure 1.

```
01:     #define N (10)
02:     int i;
03:     int a[N],b[N];
04:     for(i=0;i<N;i++)
05:             {
06:             b[i]=a[i];
07:             }
```

Fig. 1: Un-hardened code

To most programmers, the meaning of the code is self-explanatory; to copy and array of 10 integers. Now, consider the same code, hardened using the technique suggested by Redaudengo et al. [10]. This is shown in figure 2 (without the required initialization code). The total code segment, including the initialization which must be called before each operation, and the XOR macro CHK, is in excess of 36 lines in length; the meaning of the code is also somewhat obscured. Additionally, the variable ***i*** in figure 2 remains un-hardened; and the arrays ***a0***, ***b0***, ***a1*** and ***b1*** all reside alongside each other in the same area of memory.

```
01:     #define N (10)
02:     int i ;
03:     int a0[N], b0[N] ;
04      int b1[N], b1[N] ;
05      int c0, c1 ;
06:     for ( i = 0 ; i < N ; i++ )
07:             {
08:             c0 = c0 ^ b0 ;
09:             c1 = c1 ^ b1 ;
10:             b0 [i] = a0 [i] ;
11:             b1 [i] = a1 [i] ;
12:             c0 = c0 ^ b0 ;
13:             c1 = c1 ^ b1 ;
14:             if ( a0 [i] != a1 [i] )
15:                     {
16:                     if (CHK ( a0, b0 ) == C0 )
17:                             {
18:                             a1 [i] = a0 [i] ;
19:                             c1 = c0 ;
20:                             }
21:                     else
22:                             {
23:                             a0 [i] = a1 [i] ;
24:                             c0 = c1 ;
25:                             }
26:                     }
27:             }
```

Fig. 2: Hardened code

In this paper, we will attempt to address these problems of code complexity, obscurity and variable location, and propose a novel methodology we have implemented for this purpose. The paper is organized as follows. In Section 2 we will describe the memory architecture of a typical COTS microcontroller. Section 3 will introduce the concept of the mirror array. In Section 4 we will describe how the redundancy management can be achieved in C and C++ programs, resulting in code that is highly readable and maintainable. Section 5 describes an experimental study performed to asses the effectiveness of the methodology. The paper is concluded in Section 6.

2 Embedded System Memory

Before describing the methodology in full, we will first describe the architecture of a typical embedded processor. Many embedded processors employ either a Harvard architecture – in which the code and data memory areas employ a different address space - or Von Neumann architecture, in which the code and data memory share a common address space. The techniques we describe are equally applicable to both architectures. It is common for a processor to have a small amount of on-chip (internal) RAM, termed the IRAM. It may or may not have a small amount of on-chip ROM or FLASH for code storage.

At the lowest address spaces, there will normally be an interrupt vector table, implemented in ROM. The lowest of these contains the reset vector – upon start/reset, the processor loads this vector from address 0h and jumps to the appropriate memory address where program execution commences. Following this vector table, the IRAM will commonly be implemented. The system designer is then free to use the address space following the IRAM for implementation of either externally implemented RAM or ROM. A typical configuration for a Von Neumann style architecture is shown in figure 3.

The CPU registers, system stack and Special Function Registers (SFR's) will be implemented in the IRAM; normally there is space for a small amount of user variables in this area. Since most embedded systems utilize a scheduler (a small specialized RTOS), a developer will typically implement the scheduler data areas in the remaining IRAM to reduce overheads, as access to this data area is faster than XRAM. The user stack and all task data will then be implemented in XRAM. This flexibility in assigning areas of XRAM, possibly

even over multiple (physically separate) memory chips, can thus be exploited to increase reliability.

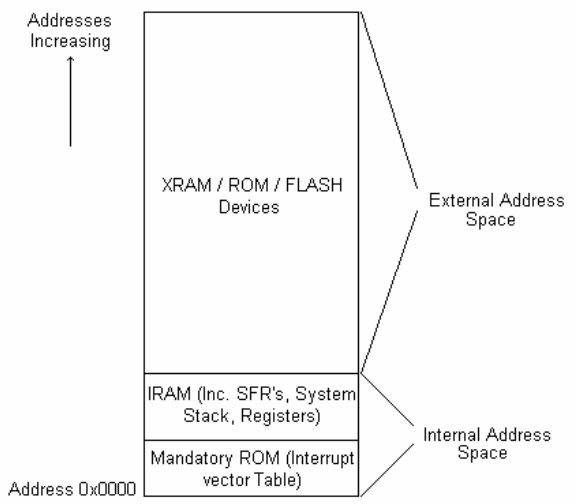

Fig. 3: Typical address space usage

Before describing this process, we will state a number of assumptions that will be used throughout the remainder of the paper:

- We assume that the designer wishes to duplicate data in the user stack and task data areas, both implemented in XRAM[1].
- The management of this duplicated data should be as straightforward as possible, with minimal alterations to the source code.
- For the remainder of this paper, we will use the notation ***tByte*** to represent an unsigned 8-bit value, ***tWord*** to represent an unsigned 16-bit variable and ***tFloat*** to represent a floating point variable.
- We assume that if an uncorrectable data inconsistency is detected, the processor is forced into a reset mode to run pre-programmed built-in-test routines - other behavior may be more appropriate depending on the application.

3 Mirror Arrays

The mirror array is an entirely replicated area of external memory, shifted from its base address by an offset value, which is written to and read from every time the corresponding primary area is written to or read from in the original. The advantage of using such an approach is that by choosing appropriate offset values, the programmer can specify absolute addresses for these mirrors to reside in; they can also reside in physically separate memory devices, from different manufacturers. This can potentially increase system reliability by avoiding common-mode failures in memory devices.

Such arrays can be used to provide data redundancy for global, persistent data, and since the user stack can also be mirrored, temporary variables can also be duplicated in the mirror.

Suppose a designer has created a working (standard) program which uses 200h bytes of XRAM, at addresses 0x10000 to 0x101FF. The user stack is located in the first 100h bytes, and the remaining data is global in scope. The entire contents of this data area can then be 'mirrored' at address space 0x11000 to 0x111FF - assuming the memory is physically present - shifted by a fixed offset of 1000h. This process can be repeated at other fixed offsets in memory, and the required level of data duplication can be achieved. This is shown in figure 4. For space reasons, in this paper we will primarily describe the duplex implementation. However, we have also considered the triplex implementation; this can be achieved with very little alteration to the underlying method.

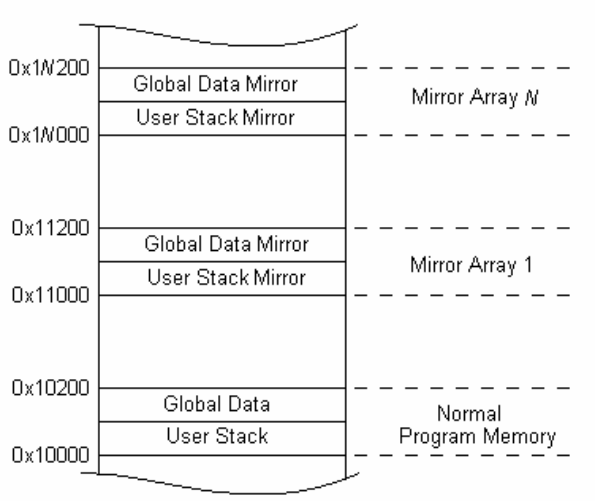

Fig. 4: Mirror array concept

In order for this methodology to work correctly, we require some way to manage this redundancy in the program code. If this can be achieved, data can be managed dynamically - as and when it is written to. Similarly, when a variable is read, we wish to also read the duplicated data areas and verify the data integrity. In section 4 we describe such a methodology for use with C/C++ compilers.

[1] Hardening of data in the IRAM areas is of course possible, but a description of such a process is not within the scope of the current paper

4 Implementation

Due to the unnecessary overheads associated with many C++ programs, many resource-constrained embedded programs are written only in C [11]. However, many C compilers have built-in C++ features - indeed these extra features are often implemented as a built-in pre-processor for a standard C compiler. If care is taken in the programming approach, objects may be compiled in C++ and exported to ordinary C programs using the ***extern "C"*** compiler directive, with little/no processing or memory overheads. The code described in this section was developed using the Keil embedded C/C++ compiler [12] – other compiler platforms may use slightly different terminology.

In order to implement the proposed approach, and make its use as transparent as possible to the programmer, we implemented three new basic data types as C++ classes. These new data types can then be exported to ordinary C programs if required. We created the data types ***duplex_tByte***, ***duplex_tWord***, and ***duplex_tFloat***, to be used, from the programmers perspective, identically to the basic data types representing a byte, word and float, with the exception that read and write operations to the basic data invoke code to implement the mirror arrays.

Each class contains a single private data declaration, ***Primary_Data***, corresponding to the basic simplex data type. We then create the required read and write operations on this data by defining new operator member functions using the ***operator*** keyword. By way of example, we show the member functions for the assignment and reference operations for the ***duplex_tByte*** data type in figure 5. Line 1 of this code defines an offset in memory of 1000h for the duplicated data. The ***inline*** keyword preceding the function deceleration of line 2 indicates that the code is to be executed inline, not as a function call – this minimizes the processing overheads. The remainder of line 2 indicates that the code should be executed when an assignment is made to the class, and passed a value of type ***tByte***. Line 4 features the statement ***_atomic_(0)***. This instructs the compiler (and microprocessor) to treat the following instructions, until the ***_endatomic_()*** statement is reached (line 6 in this case), as a single (uninterruptible) machine instruction. This is important to maintain data consistency in the mirror in pre-emptive systems. In line 5, we then assign the value passed to the class to both the primary data and the corresponding data in the mirror array. The function then exits.

```
01: #define MEMORY_OFFSET (0x1000)
02: inline void duplex_tByte::operator = ( tByte Value )
03:         {
04          _atomic_( 0 ) ;
05          Primary_Data =  * ( &Primary_Data +
            MEMORY_OFFSET ) = Value ;
06          _endatomic_() ;
07          }
08 inline tByte duplex_tByte::operator () ( void )
09          {
10          atomic_( 0 ) ;
11          If ( Primary_Data == *( &Primary_Data +
            MEMORY_OFFSET ) )
12                      {
13                      return ( Primary_Data ) ;
14                      _endatomic_() ;
15                      }
16          else
17                      {
18                      _trap_( 0x00 ) ;
19                      _endatomic_() ;
20                      }
```

Fig. 5: ***duplex_tByte*** operator functions

Line 6 indicates that the following function code should be executed, again inline, when the data type is referenced. In line 11, we compare the primary data to the corresponding data in the mirror array. If the data is consistent, we return the value and exit the function. If not, the statement ***_trap_(0x00)*** on line 18 executes a full system reset - this is equivalent to a reset by an external watchdog. In the triplex data type, the assignment and reference functions are suitably modified to incorporate the third data area; the reference function additionally performs a 2 from 3 vote if possible, and executes a reset if all the data are inconsistent.

On reset, the chip first enters a test mode where, among other things (such as ROM checksum tests), the RAM functionality is verified to detect faults, using a similar method as outlined in [13] before normal program execution commences. If a faulty RAM is detected, the system can enter a safe state, and perform appropriate external signaling to maintain system integrity.

Each of the C/C++ operators, such as ++, --, <=, and so on, were similarly implemented as inline member functions for the new data types. Thus, in combination, this ensures inter-operability of the new classes with each other and the basic data types; the implementation (and compiler consistency checking) of the duplicated data is completely hidden from the programmer.

Although we have implemented constructors for each data type to initialize the data in the mirror areas, it is beneficial to initialize the mirror areas as part of the initialization code. This can be achieved either in assembly, as part of the XRAM test, or

directly in C. The ***_at_*** or ***MARRAY*** specifiers can be used with the Keil compiler to specify an absolute area for direct memory access in C.

The resulting data types are highly portable, and do not obfuscate the meaning of the hardened code. With reference to figure 1, applying the code hardening methodology we have described produces the source code shown in figure 6. From this figure it can be observed that the code length is identical to the original and is also highly readable. Additionally, we note that the variable ***i*** is also hardened in this case.

```
01:     #define N (10)
02:     duplex_tByte i;
03:     duplex_tByte a[N],b[N];
04:     for(i=0;i<N;i++)
05:             {
06:             b[i]=a[i];
07:             }
```

Fig. 6: Hardened code

The methodology does not require the use of automatic code generators; this simplifies the process of code certification, as no 'extra' components need to be certified as the process of certifying the compiler will be required regardless. All that is required is for the programmer to have a basic understanding of the meaning of the new data types. It also allows a system developer to first implement the system in a simplex fashion, determine the memory requirements for the system, then harden the code when the required memory offset has been determined. The hardening procedure can be accomplished in seconds.

In the following section we describe preliminary experimental results we have obtained from applying the technique.

5 Experimental Results

To asses the effectiveness of the proposed methodology, we performed several fault-injection studies on an Infineon C167 microcontroller [14] executing a 4x4 matrix multiplication program. The experimental set-up is shown in figure 7.

During each experiment, we injected transient faults into the XRAM data area at random times, performing random bit-flips in all the user data areas. The fault injection was performed using a high-speed serial link and a small monitor program in the C167. The main program loop first initializes the source matrices with values hard-coded into the ROM. The matrix multiplication is then performed. The values contained in the result matrix are then compared with values coded into the program ROM. The process then repeats endlessly. Any failures, detected faults or corrected faults are reported to the host PC via the serial link.

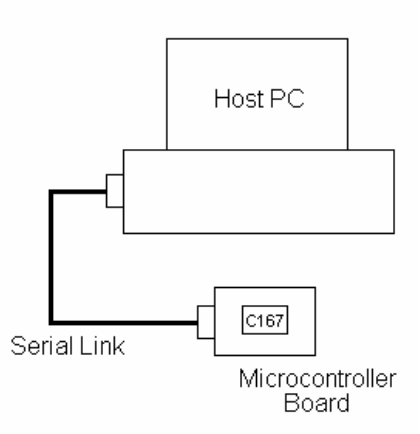

Fig. 7: Experimental setup

We considered three different implementations of the program; the un-hardened (simplex) case, and two hardened versions (duplex and triplex) of the program. The application of any such technique has an impact on the required system resources; we begin by describing the resulting code size, memory requirement and execution time of each iteration of the program. The results are shown in table I.

Table I: Required system resources

Resource	Simplex	Duplex	Triplex
Code Size (b)	1956	2044	2232
Memory Size (b)	204	408	612
Execution Time (ms)	0.918	2.16	2.61

In table II, we summarize the results we recorded during the fault injection experiments. In the duplex and triplex cases, we increased the number of faults injected to reflect the increased size of the program data areas. Fault effects are classified into one of four categories as follows.

- Effect-less: the fault does not result in a computation failure.
- Detected: the fault is detected but cannot be corrected – the iteration is restarted after processor reset.
- Detected and corrected: the fault is detected and has been corrected.
- Failure: the fault is not detected or corrected and results in an invalid computation output.

From table I, we can see an increase of approximately 4.5% and 14.1% increase in the code size for the duplex and triplex case

respectively, 100% and 200% increase in data memory, and a 135.3% and 184.3% increase in execution time. In terms of data memory increase, the duplex and triplex case invariably causes a doubling or trebling of each of the hardened variables, plus the user stack. The code size increase in this case is small; despite the use of inline function calls. The increase in execution time is almost doubled and tripled by the application of the technique; this is however to be expected as each hardened variable uses instruction replication and comparison. In cases where the software in not data intensive (unlike this example) or only portions of the data need to be hardened, then the increase in execution time will be must greatly reduced; we present here a 'worst-case' analysis of the method.

Table II: Fault injection results

	Simplex	Duplex	Triplex
Injected	10000	20000	30000
Effect-less	1289	2448	3744
Detected	0	17552	0
Detected/Corrected	0	0	26256
Failures	8711	0	0

Now considering table II, we can see that in each case approximately 12% of each of the injected faults was effect less. In the simplex system, the remaining faults all caused failures. In the duplex system, all remaining faults were detected; no failures occurred. In the triplex system, all remaining failures were detected and corrected; the system is fully fault-tolerant. From the perspective of safety critical systems, both the duplex and triplex system were capable of preventing failures resulting from the incorrect computations of the simplex case.

When compared to techniques such as [8][9][10], these results suggest that both the duplex and triplex techniques are comparable in terms of memory size increase, and favorable in terms of code segment increase and execution time. Additionally, these results show that high levels of fault detection and tolerance to permanent and transient failures in RAM can be achieved without the need for automatic code generators; and the impact on source code readability and maintainability is negligible.

6 Conclusion

In this paper, we have presented a novel approach to software implemented fault-tolerance that can be used in embedded systems. The approach relies on data and instruction duplication. We have shown how new data types can be implemented as C++ classes and exported into a C program. We have shown that the method is easily applied, results in readable code, and is able to tolerate 100% of the injected faults on the benchmark described. These results suggest that the method may be highly applicable to safety-related embedded system designs based on COTS processors.

References:

[1] C.T. Kilian, *Modern control technology: components and systems*, Delmar Thomson Learning, 2000.

[2] K. Astrom, B. Wittenmark, *Computer Controlled Systems: Theory And Design*, Prentice Hall.

[3] G.S. Virk, *Digital computer control systems*, McGraw-Hill, 1991.

[4] N. Storey, *Safety Critical Computer Systems*, Addison Wesley Publishing, 1996.

[5] N.G. Levenson, *Safeware: System Safety and Computers*, Reading, M.A., Addison-Wesley, 1995.

[6] E. Normand, Single Event Effects in Avionics, *IEEE Trans. on Nuclear Science*, Vol. 43, No. 2, 1996.

[7] P. Dodd, L. Massengill, Basic Mechanisms and Modeling of Single Event Upset in Digital Microelectronics, *IEEE Trans. on Nuclear Science*, Vol. 50, No. 3, 2003.

[8] N. Oh, P.P Shivani, E.J. McCluskey, Control Flow Checking by Software Signature, *IEEE Trans. On Reliability*, September 2001.

[9] A. Benso, S. di Carlo, G. di Natale, P. Prinetto, L. Tagliaferri, Control-Flow Checking Via Regular Expressions, *Proc. IEEE Asian Test Symposium*, pp. 299-303, 2001.

[10] M. Rebaudengo, M. Sonza Reorda, M. Violante, A new approach to software-implemented fault tolerance, *Proc. IEEE Latin American Test Workshop*, 2002.

[11] M. Pont, *Embedded C*, Addison-Wesley, 2002.

[12] Keil, *XC16x/C16x/ST10 Product Overview*, http://www.keil.com/c166/.

[13] S. Hamdioui, A. van der Goor, M. Rogers, March SS: A Test for All Static Simple RAM Faults, *Proc. Of the 2002 IEEE Intl. Workshop on Memory Tech., Design and Testing*, 2002.

[14] Phytec, *phyCORE 167CS Hardware Manual*, Phytec, April 2003.