Efficient implementation of fault-tolerant data structures 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

michael.short@le.ac.uk, m.schwarz@uni-kassel.de, j.boaercsoek@uni-kassel.de

Abstract: - This paper presents a methodology and small software library which is intended to reduce the impact of transient data errors that may affect the software executing on commercial-of-the-shelf (COTS) embedded processors. The methodology involves duplication of data in disparate areas of memory (referred to as “mirror arrays”), and the complexity of the processing required to manage these areas is hidden by means of a library exporting new basic data types. Results are reported from three case studies in which the library was employed (a matrix multiplication program, a list-sorting program and a real-time control application): the findings obtained suggest that the methodology is highly effective in the presence of memory errors, the code changes required in order to use the library are very limited, and the impact on code readability is minimal.

Key-Words: - Software Fault Tolerance, Embedded Systems, Critical 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 (latchup) failures in the region of 10^{-8} - 10^{-6} failures per device per hour [7].

In addition, memory devices may also fail due to normal electrical breakdown effects – failure rates for an individual device may be calculated using a methodology such as [16]. Such electrical or thermal failures and disturbances in memory devices may be highly unpredictable, manifesting themselves as complete device failures or stuck-at faults over part (or all) of the memory array. Memory devices are also susceptible to electromagnetic interference from a variety of sources. For example in a passenger vehicle, numerous devices such as electromechanical relays, alternators and ignition coils are all sources of noise that are capable of corrupting many electronic circuits [21][22].

Transient errors such as SEU’s in the data memory may manifest themselves in a variety of ways (e.g. [21][22]). For example, they may cause a illegal jump error, where the program execution makes an illegal jump inside the program area; they may cause an incorrect value to be output to a port or peripheral; and they may cause a further area of memory to be corrupted by indexing an array out of its normal bounds.

Many modern Commercial Off The Shelf (COTS) microprocessor designs now feature on-chip watchdog timers and exception handling routines that are specifically designed to detect erroneous program flow [15][18][23]. Such routines can detect, for example, stack overflow/underflow, illegal operands and illegal memory accesses. When used correctly, in conjunction with appropriate Software Based Self Test (SBST) routines for the CPU and RAM/ROM (e.g. [15][13]), these elements can give relatively high levels of control flow error detection and take the appropriate safety action if a permanent hardware (e.g. register) fault is diagnosed (e.g. [13][15][19][[22]). However, due to their nature, SBST techniques are generally not suitable for detecting and correcting transient errors in data memory [13][15].

Recent years have also seen the development of several software-based approaches to implementing transient fault detection on COTS processors: due to their low cost requirements, such devices 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 these techniques involve modifications to the application code in order to detect deviations from the expected program execution flow (e.g. [8][9][17][23][25]). If any deviation is detected, program execution is suspended and an error recovery procedure is called (this often involves resetting the processor to a known state: e.g. [25][10]). They are based around instruction counting [17], instruction/task duplication [25] and control flow checking using signatures and/or assertions [8][9]. Although such techniques are effective at detecting many control flow errors, systems which incorporate them may still be vulnerable to transient errors in data memory (which may not result in control-flow errors).

In order to address this vulnerability, some researchers have investigated the use of Single-Program Multiple Data (SPMD) techniques for data redundancy in both single and multi processor systems (e.g. [10][20]). However, such approaches (as with most software-based transient error detection techniques) are problematic from the point of view of the embedded system developer for two main reasons:

1. When data duplication is employed, the developer may have little control over where the duplicated copies reside in memory. For example, when redundant temporary variables are declared, they may all be placed (by the compiler) in adjacent locations in the user stack area: they may then be at risk of a common failure should the memory chip implementing the user stack become faulty.
2. When the techniques are actually applied, the complexity of the resulting source code can increase dramatically, and the basic meaning of the code can become obscured. This may have an impact on code development and subsequent code maintenance.

To illustrate this second point, consider the segment of C code shown in Fig. 1.

```
#define N (10)
int i;
int a[N],b[N];
for(i=0;i<N;i++)
 {
 b[i]=a[i];
 }
```

Fig. 1: Un-hardened code

For most programmers, this is "self documenting" code, and the meaning is clear (the programmer wishes to copy the contents of any array of ten integers to another array of the same size). Now, consider the same code, hardened using the technique suggested by Redaudengo et al. [10]. This is shown in Fig. 2 (note the required checksum initialization code and the XOR macro CHK have been omitted for space reasons). The total code segment, including this initialization (which must be called before each operation), and the CHK macro, is in excess of 36 lines in length; the meaning of the code is also somewhat obscured. In addition, the variable ***i*** in Fig. 2 remains un-hardened; and the arrays ***a0***, ***b0***, ***a1*** and ***b1*** are likely to be located alongside each other in the same area of memory. If the variable ***i*** were to be hardened, the meaning of the code would become further obscured, with the check-and-correct code for ***i*** embedded within the ***for*** loop construct; as more nested variables are hardened, the problem can soon become difficult to manage.

In this paper, a software-based methodology to complement the on-chip error detection facilities and existing SBST techniques for use with COTS microprocessors will be proposed. This technique is an implementation of an SPMD-like architecture to provide both fault detection and fault tolerance to both transient and permanent errors in data memory. This approach differs slightly from other methodologies in that errors in data memory that may lead to failures are detected as the program executes; the manifestation of a data error as a control flow error is not required for detection. In this approach, the problems of code complexity, obscurity and the locating of variables commonly associated with such a software-based approach to fault tolerance will attempt to be minimized.

```
#define N (10)
int i ;
int a0[N], b0[N];
int b1[N], b1[N];
int c0, c1;
for (i=0;i<N;i++)
 {
 c0=c0^b0;
 c1=c1^b1;
 b0[i]=a0[i];
 b1[i]=a1[i];
 c0=c0^b0;
 c1=c1^b1;
 if(a0[i]!=a1[i])
  {
  if(CHK(a0, b0)==C0)
   {
   a1[i]=a0[i];
   c1=c0;
   }
  else
   {
   a0[i]=a1[i];
   c0=c1;
   }
  }
 }
```

Fig. 2: Hardened code

The paper is organized as follows. Section 2 will describe the memory architecture of a typical COTS microcontroller. Section 3 will introduce the concept of the mirror array. Section 4 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 preliminary studies performed to asses the effectiveness of the methodology; following this, an in-depth case study is presented in Section 6. The paper is concluded in Section 7.

2 Memory In Embedded Systems

Before describing the methodology in full, the architecture of a typical embedded processor will first be described. 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 [14][18]. 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 [11][18].

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 external RAM or ROM. A typical configuration for a Von Neumann style architecture is shown in Fig. 3; in this case the architecture of the C167CR microprocessor is shown [14].

The CPU registers, system stack and Special Function Registers (SFR's) are normally implemented in an area of IRAM; this may even be within the normal address space. Typically there is also space for a small amount of user variables in the IRAM area. Since most embedded systems utilize a scheduler (a small form of specialized RTOS [11]), 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 external RAM (XRAM). The user stack and all task data will then typically 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 in SPMD architectures. Before describing how this can be achieved in software, a number of assumptions that will be used throughout the remainder of the paper will be stated.

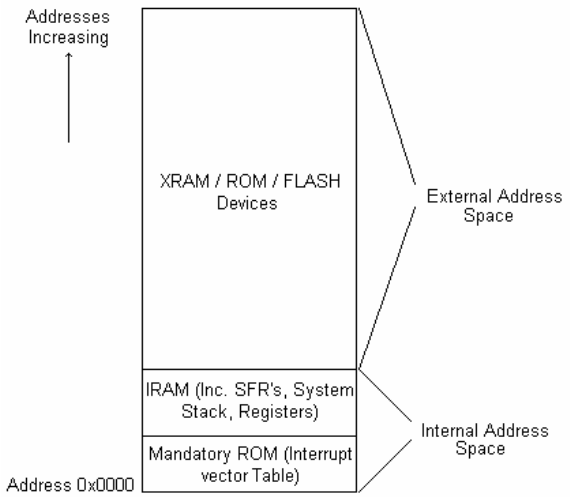
Addresses Increasing
XRAM / ROM / FLASH Devices
External Address Space
IRAM (Inc. SFR's, System Stack, Registers)
Mandatory ROM (Interrupt vector Table)
Internal Address Space
Address 0x0000

Fig. 3: Typical address space usage

- For the remainder of this paper, the notation ***tByte*** will be used to represent an unsigned 8-bit value, ***tWord*** to represent an unsigned 16-bit variable and ***tFloat*** to represent a floating point variable.
- It is assumed 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.

It is assumed that if an un-correctable data inconsistency is detected, the processor is forced into a reset mode to run pre-programmed SBST routines - other behaviour may be more appropriate depending on the application.

3 Mirror Arrays

A “mirror array” is defined as a replicated area of external memory, shifted from its base address by a user-defined offset value. The mirror array is accessed (for read or write operations) every time the corresponding primary area is accessed. 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 is intended to 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 also temporary variables and the user stack. 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 Fig. 4.

[1] Hardening of data in the IRAM areas is of course possible, but cannot be done with a mirror array. This is discussed further in Section 4.3.

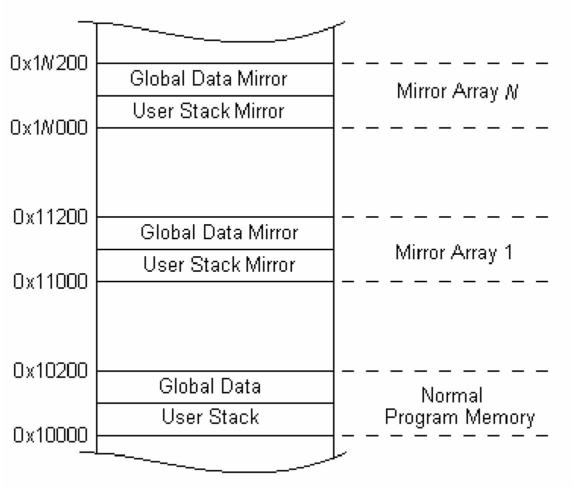

Fig. 4: Mirror array concept

For clarity and simplicity of description, this paper will primarily describe the duplex implementation. However, the triplex implementation has also been considered; this can be achieved with very little alteration to the underlying method.

4 Implementation

If the use of mirror arrays is to be effective, some way to manage this redundancy in the program code is required. Such an approach is described in this section.

Please note that the code described in this section was developed using the Keil C/C++ compiler [12] for the Infineon C16x family of microcontrollers [14]. Code for other compiler / processor combinations may vary slightly.

4.1 User-Defined Data Types

In order to implement the mirror arrays, and make their use as transparent as possible to the programmer, three new basic data types were implemented as C++ classes: ***duplex_tByte***, ***duplex_tWord***, and ***duplex_tFloat***. The intention is that variables of these types can be used, from the programmer’s perspective, in a manner identical to the basic data types representing byte, word and float variables. Each class contains a single private data declaration, ***Primary_Data***, corresponding to the basic simplex data type. The required read and write operations on this data were then created by defining new operator member functions using the ***operator*** keyword. By way of example, the member functions for both the assignment and reference operations for the ***duplex_tByte*** data type are shown in Fig. 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 intended to be executed inline, not as a function call (to minimize processing overheads).

```
01: #define MEMORY_OFFSET (0x1000)
02: inline void duplex_tByte::operator=(
tByte Value)
03:  {
04  IEN=0;
05
Primary_Data=*(&Primary_Data+MEMORY_OFFSET)
=Value;
06  IEN = 1;
07  }
08 inline tByte duplex_tByte::operator
tByte (void)
09  {
10  IEN=0;
11  if(Primary_Data==*(&Primary_Data+
MEMORY_OFFSET))
12   {
13   IEN=1;
14   return(Primary_Data);
15   }
16  else
17   {
18   _trap_(0x00);
19   }
```

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

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***. Variables of other type (e.g. float) that may be passed are automatically cast to this form by the compiler. Line 4 features the statement ***IEN = 0***. This instruction disables interrupts on the microprocessor until a corresponding ***IEN = 1*** statement is reached (Line 6 in this case). This is important to maintain data consistency in the mirrors in pre-emptive systems. In Line 5, the value passed to the class is written to both the primary data and the corresponding data in the mirror array. The function then exits.

Line 8 indicates that the following function code should be executed, again inline, when the data type is referenced (in this case it is an explicit reference – all references are treated by the compiler as ***tByte*** references). In Line 11, the primary data is compared to the corresponding data in the mirror array. If the data are consistent, the value is returned and the function exits. It is this returned value that is then further manipulated by the compiler. If not, the statement ***_trap_(0x00)*** on Line 18 executes a full system reset - this is equivalent to a full hardware or watchdog reset[2] in the C167 processor [14]. 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.

Each of the C/C++ operators that explicitly modify the contents of the duplicated data, such as ++, --, +=, and so on, were implemented in a similar manner. Thus, in combination, these operators ensure inter-operability of the new classes both with each other and with the basic data types; the implementation (and consistency checking) of the duplicated data is completely hidden from the programmer.

4.2 Initializing The Mirror Arrays

Although constructors have been implemented 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 SBST 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.

4.3 Hardening of IRAM Areas

By definition, the areas of microcontroller IRAM may not be directly replicated using the mirror array technique. When data in these areas is to be replicated – for example to provide tolerance to corruptions of internal scheduler data – then a slightly different methodology must be employed. The same basic framework as used for the duplex datatypes described above must be adapted to allow the redundant data fields to sit in adjacent areas of memory.

Each class now requires multiple private data declarations in internal RAM. For the duplex case this corresponds to ***Primary_Data*** and ***Secondary_Data*** declarations corresponding to redundant copies of the basic simplex data type. The code modifications required to implement these

[2] This behaviour may not be portable; in many microcontrollers a software reset may not be equivalent to a full hardware reset. In this case, it may be more suitable to freeze program execution and enforce a watchdog reset. Reliable techniques to enable this are discussed in [23].

changes is minimal; the new classes are given appropriate names, for example ***duplex_iByte***, and the compiler must now be instructed that the private class datatypes must now explicitly declared to reside in IRAM. The ***idata*** keyword can be used with the Keil compiler to perform this action. Finally, the overloaded operator functions must now be modified to act on the redundant copies of data, as opposed to the mirrored data.

4.4 Reset Mechanisms

On reset, the chip first enters a SBST mode where, among other things (such as CPU and ROM checksum tests [15]), 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 attempt enter a safe state, and perform appropriate external signaling to maintain system integrity. When used in conjunction with on-chip exception handling, a watchdog timer and SBST, the overall approach can be summarized as shown in Fig. 7.

4.5 Example

In Fig. 6, the code library described in this section is applied to the code example shown in Fig. 1. From Fig. 6 it can be seen that the length of the hardened code is identical to the original and is also highly readable. Additionally, it is noted that – unlike the code shown in Fig. 2 - the variable ***i*** is also hardened in this case.

```
#define N (10)
duplex_tByte i;
duplex_tByte a[N],b[N];
for(i=0;i<N;i++)
 {
 b[i]=a[i];
 }
```

Fig. 6: Hardened code

This library does not require the use of automatic code generators for its implementation: all that is required is for the programmer to have a basic understanding of the meaning of the new data types. The library 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(s) have been determined.

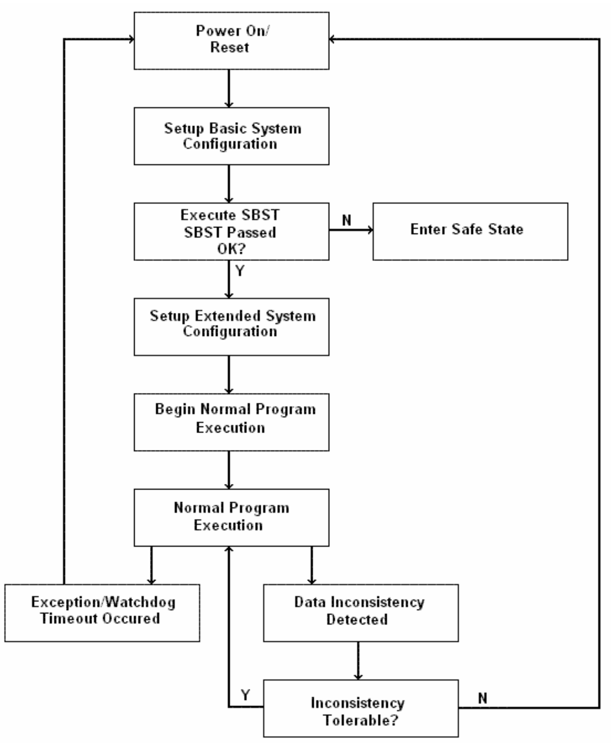
Power On/ Reset
Setup Basic System Configuration
Execute SBST SBST Passed OK?
N
Enter Safe State
Y
Setup Extended System Configuration
Begin Normal Program Execution
Normal Program Execution
Exception/Watchdog Timeout Occured
Data Inconsistency Detected
Y
Inconsistency Tolerable?
N

Fig. 7: Proposed approach to fault tolerance

The hardening procedure can be accomplished extremely rapidly; all that is required is the inclusion of the new data types into a project, and altering the basic data declarations that require hardening to either their duplex or triplex counterparts.

4.6 Mixing C and C++ Code

Due to the overheads associated with some features of the C++ language, many resource-constrained embedded programs are written only in C [11]. However, many C compilers have built-in support for a sub-set of C++ features - indeed these features are often implemented as a built-in pre-processor for a standard C compiler.

If care is taken in the programming approach, user tasks and functions (using the new data types) can be compiled in C++ and exported to other C programs (for example a scheduler implementation) using the ***extern "C"*** compiler directive, with little/no processing or memory overheads.

However, care must be taken when mixing C and C++ in safety-critical software. Although at the present time several working groups, such as MISRA [24], are cautious about this practice[3],

[3] It has been argued that C++ should not be used in UK road vehicle embedded software until its behaviour has been fully analyzed from a safety perspective, and which elements of the

software libraries may be used in a project if their behavior is well-defined and fully tested on a given platform. Additionally, research is currently underway to identify a safe subset for C++ which can be used in such systems. It is envisioned that the small amount of C++ that has been used in the creation of the new data types will be compatible with these findings. Until this can be verified, care must obviously be taken by designers when using the library, specifically when integrating and testing the code to ensure that the software behaves as expected on a given compiler / processor platform. Additionally, the library may also be implemented as a series of macro's / functions for use with a C program; however this requires slightly more source code modifications for implementation.

4.7 Impact Of Permanent Memory Errors

Considering the presence of permanent memory failures, it is assumed that physically separate memory devices have been used to implement the mirrors; it is also assumed that failures of the memory devices are isolated and do not occur instantaneously. In the duplex case, the system will reset upon the first detected memory mismatch in a memory read operation. This will force a reset and the subsequent SBST should diagnose the permanent memory failure.

In the triplex case, the failure of a single memory device will be tolerated by the voting mechanism; the system essentially defaults to the duplex case under these conditions. Upon a second, subsequent memory failure, the system will again reset on the first un-correctable memory read, and force the SBST. Again this should subsequently diagnose the permanent failure of multiple memory devices.

5 Preliminary Investigations

To assess the effectiveness of the mirror-array technique and associated code library, several preliminary fault-injection studies were performed on an Infineon C167CR microcontroller [14] executing two different programs: a 4x4 16-bit matrix multiplication program and a 100-element 16-bit list sorting exercise. The experimental set-up is shown in Fig. 8.

language produce reliable software. Such work was carried out for ANSI C, resulting in the publications regarding MISRA C [24]. Preliminary versions of such guidelines for C++ were proposed in 2007.

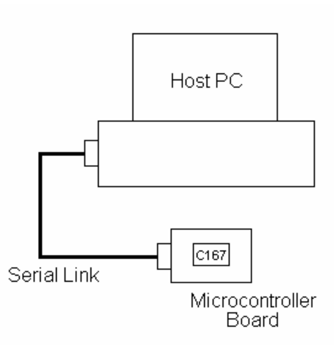

Fig. 8: Experimental setup

During each experiment, transient faults were injected into the XRAM data area at random times, performing random bit-flips in all the user data areas. The injection of faults into the CPU registers was not considered in this case, and is an area for future work. The fault injection was performed using a high-speed serial link and a small monitor program in the C167. In the matrix multiplication program (MATRIX), 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. In the list sorting program (LIST), the 100-element list is sorted into ascending order from an initial random assignment. The final list is compared to values hard coded into program ROM, before repeating. In both cases, any failures, detected faults or corrected faults are reported to the host PC via the serial link.

Three different implementations of both programs were considered; the un-hardened (simplex) case, and two hardened versions (duplex and triplex) of the programs. Since the application of any such technique has an impact on the required system resources, the resulting code size, memory requirements and execution times of each iteration of the programs are described. The MATRIX program requirements are shown in table I, and the LIST program requirements are shown in table II.

Table I: Required system resources for MATRIX

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

Table II: Required system resources for LIST

Resource	Simplex	Duplex	Triplex
Code Size (b)	2290	2392	2540
Memory Size (b)	464	928	1392
Exec. Time (ms)	13.65	29.57	42.84

In table III, the recorded results for the fault injection experiments for the MATRIX program are summarized. Similar results are shown for the LIST program in table IV. In the duplex and triplex cases, the number of faults injected was increased 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.

Table III: Fault injection results for MATRIX

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

Table IV: Fault injection results for LIST

	Simplex	Duplex	Triplex
Injected	10000	20000	30000
Effect-less	279	493	988
Detected	0	19507	0
Corrected	0	0	29012
Failures	9721	0	0

From table I, an increase of approximately 4.5% and 14.1% 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 can be seen. From table II, an increase of 4.45% and 10.9% for the code size, 100% and 200% increase in data memory size, and 116.6% and 213.8% increase in execution times by the application of the duplex and triplex techniques can be seen.

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 space. The code size increases in both these cases are small; despite the use of inline function calls. As can be seen the increase in execution time is more than doubled and almost tripled by the application of the technique to the MATRIX program; the increase is slightly worse for the LIST program.

An increase such as this is however to be expected, as each hardened variable uses instruction replication and a comparison or voting mechanism. These results indicate that the increase in code size and execution time is highly dependant on the application; the more memory read and write operations a program contains before hardening, the larger the impact of applying the technique. Due to the nature of the technique, the increase in data memory, however, is more predictable, with a doubling and trebling of used memory areas.

Now considering table III, it can be seen that for the MATRIX program 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 to the types of faults considered in the benchmark. A similar pattern may be observed for the LIST results shown in table IV. In the simplex case, 2.8% of faults were effect less; the remaining all caused failures. A total of 2.5% of injected faults were effect less in the duplex case, with all the remaining being detected. Finally in the triplex case 3.3% of faults were effect less, with the remaining being detected and corrected.

When compared to similar SPMD techniques such as [10][20], these results suggest that both the duplex and triplex mirror techniques are comparable in terms of memory size increase, and favorable in terms of both code segment increase and CPU overhead increase (although comparison between CPU overhead increases is difficult to judge between the single processor and multiprocessor cases). In the next section, these preliminary results are expanded upon, and a case study – based around a real-time cruise control system for a passenger vehicle - is presented that illustrates the effect of the methodology in a real-time control application.

6 Case Study

Whilst the results in the previous section favorably demonstrate the behavior of the technique, the simple applications that were considered may not – in themselves - be fully representative of the real world applications in which the technique is most likely to be applied. With this in mind, in this section a representative case study is described. The case study employed hardware-in-the-loop (HIL) testing techniques to gauge the effectiveness of the methodology in a vehicle cruise control system (CCS) application.

The principle of HIL simulation is shown in Fig. 9. The simulator is currently set up to represent the dynamics of a passenger vehicle in real-time, iterated at a rate of 1 kHz on a desktop PC. The nature of the testbed itself, and the dynamic models used to represent the vehicle have been described elsewhere [26][27]; we provide only a brief summary here.

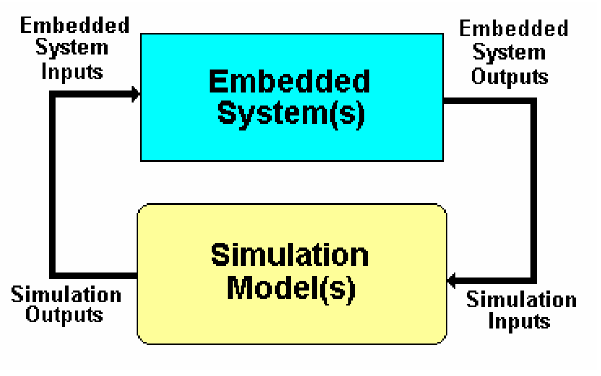

Fig. 9: HIL principle

6.1 Test Facility Description

Although the dynamic model of the vehicle is non-linear, it can be approximated in the operating range of the CCS by the following transfer function:

$$\frac{v(s)}{F(s)} = \frac{0.02}{15s+1} \tag{1}$$

… where v(s) is the velocity of the vehicle in meters per second, and F is the accelerating force (which is dependant on the accelerator setting, engine RPM and wheel slip conditions).

The main requirement of the CCS, which is implemented by the embedded system under test, is to provide the vehicle driver with an option of maintaining the vehicle at a desired speed without further intervention, by automatically controlling the vehicle throttle setting. It performs this function by measuring the current vehicle speed from a sensor and performing a PID calculation to determine the throttle setting. The classical form of the PID algorithm employed in this study is as follows:

$$u(t) = K_c e(t) + \frac{K_c}{T_i}\int_0^t e(t)dt + K_c T_d \frac{de(t)}{dt} \tag{2}$$

… where u(t) is the commanded throttle setting, e(t) is the error between reference (desired) speed and actual (measured) speed, and Kc, Ki and Kd are the system gains, chosen to give the desired closed loop performance [1][2][3]. Additionally, the module is required to indicate the current speed of the vehicle and the status of the control system to the driver, via a serial interface to an LCD. It also must interface to a number of switches to receive commands from the driver ("CCS enable", "CCS disable", "Speed up", etc).

Again, the C167 microprocessor was employed in this study to implement the CCS. Fault injection was implemented using similar techniques to those outlined in Section 5. Overall, the CCS testbed summarized here was chosen as a representative system as it can be considered to be a critical application [28], and previous studies have shown that (amongst other things) transient effects and processor faults – such as data errors - can be a major contributory cause to potential dangerous system failures [29].

6.2 CCS Design

The CCS was implemented as a set of six software tasks for the C167 microprocessor, which were then scheduled non-preemptively using the scheduler described by Pont [11]. The tasks, along with their periods (P) and execution times (E) are summarized in table V.

Table V: Task parameters

Name	Task ID	P (ms)	E (ms)
Sensor	T1	20	0.5
Control	T2	20	1.2
Actuator	T3	20	0.5
Status	T4	100	9
Display	T5	20	5
Safety	T6	100	0.1

A scheduler tick interval of 20 ms was chosen for this implementation. The sensor task (T1) sampled the vehicle speed (as a voltage) through an analog-to-digital (ADC) port, re-scaled the voltage to the required speed range and performed range and range-rate sanity checks. The control task (T2) performed the PID calculation; a digitized version of (2):

$$u_j = K_P e_j + K_i \sum_{m=0}^{j} e_m + K_d (e_j - e_{j-1}) \tag{3}$$

… where uj is the commanded throttle setting at sample j, and ej is the error at sample j. The three system gains were chosen (manually) to give the desired closed loop performance; a critically damped 95% settling-time of approximately 6 seconds. The actuation task (T3) performed a further sanity check on the resulting throttle command and translated this output to a parallel port of the C167, to control a throttle servo actuator. These three tasks were scheduled to execute sequentially with a period of 20 ms, giving an overall sampling rate of 50 Hz.

A status-update task (T4) was executed once every 100 ms. This task monitored and "de-bounced" the "cruise enable" and "resume" switches. It also monitored switches on both the accelerator and brake pedals: if either of the pedal switches was depressed, the cruise control was disengaged. In addition, a display update task (T5) was also executed every 20 ms to control the LCD.

The final system task was the safety task (T6), also executed every 100 ms. This task serviced the watchdog timer, and also verified the system configuration (such as timer reload value, I/O configuration and interrupt settings). Additionally, the general approach to software fault tolerance, as depicted in Fig. 7, was build into the application. Throughout the software development process, good programming techniques (such as those discussed in [5][30][31]) were observed at all times.

6.3 Experimental Methodology

A total of two experiments were performed using this test facility. In the first, a basic version of the CCS implementation as described above was tested under fault injection conditions. In the second, the software was additionally augmented with duplex datatypes, and the experiment repeated. The following critical variables were hardened:

- Sensor / setpoint data: 16-bit representations of actual and desired vehicle speed;
- Control data: three internal 16-bit dynamic states of the PID controller;
- Output data: two 16-bit variables used to apply final throttle control;
- Status data: four 8-bit status flags used to store system information;
- Scheduler data areas: the internal data structures of the task scheduler (32 bits / task).

In each of the two experiments performed in this case study, the vehicle cruise control was initially enabled at 50 MPH (80.5 KPH). The 'driver' then commanded periodic speed changes from 50 MPH to 40 MPH (64.4 KPH), and vice versa, every 10 seconds. One second prior to this commanded speed change, a fault was injected into the XRAM of the system under test by the simulation PC. It should be noted that although the memory requirements of the two implementations was slightly different, as described in the following section, faults were injected into an equally large area of memory to enable a fair comparison.

After the injection of each fault, the resulting system behavior was automatically classified using a simple model-based performance monitor to gauge the operation of the system in real-time. The performance monitor compares the real system behavior with the desired system behavior to detect deviations from the specification that are indicative of a system failure (e.g. sluggish / oscillatory performance, out of range or 'stuck at' errors). After each resulting fault had been classified, the results were logged into a text file by the PC for later analysis. The overall testing strategy that was employed in this study is as shown in Fig. 10.

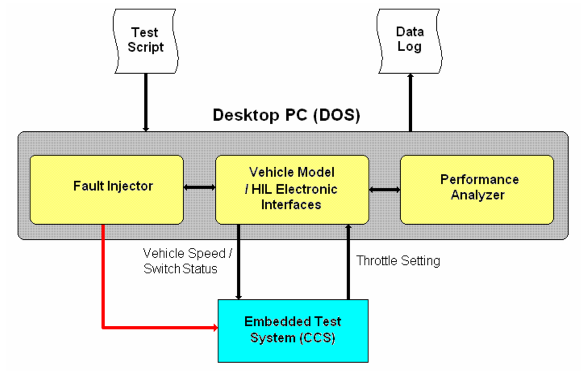
Test Script
Data Log
Desktop PC (DOS)
Fault Injector
Vehicle Model / HIL Electronic Interfaces
Performance Analyzer
Vehicle Speed / Switch Status
Throttle Setting
Embedded Test System (CCS)

Fig. 10: Experimental methodology

6.4 Experimental Results

We begin this section with a comparative analysis of the code size, RAM usage and CPU utilization of the simplex and duplex CCS designs, as shown in table VI. From this table it can be seen that the increases in code size and CPU utilization measures are effectively negligible (≅ 2.5% and 1.6% respectively) , and the RAM usage has – as expected – increased directly in proportion to the number of hardened variables, a total of 42 bytes.

Table VI: Required system resources for CCS

Resource	Simplex	Duplex
Code Size (b)	1956	2004
Memory Size (b)	204	246
CPU Utilization (%)	45.4	46.1

Considering now the recorded failure behavior, it can be seen from table VII that the method has a large positive impact on the number of recorded failures in the system. It must be noted that in this case, the effects of the faults were classified purely on their *functional* effects on the system; as such, many of the effect-less failures in the duplex system were classified as such purely because they were detected *before* they could cause a functional failure, allowing the system to return to a known (healthy) state.

It can be seen that while a small number of failures occurred in the duplex system, the simplex system sees an almost 19-fold increase in the number of recorded dangerous failures. This can be directly attributed to the fact that critical control variables (such as the speed setpoint) and scheduler data areas (such as storage of task periods) have now been protected from disturbances. Such disturbances, if left undetected, can obviously have a hugely detrimental impact on the operation of the system.

Table VII: Recorded failure rates for CCS

	Simplex	Duplex
Faults Injected	10000	10000
Effect Less	7729	9880
System Failures	2271	120

Additionally, this fault injection information may be used to estimate the failure rates of the two systems due to transient effects. As mentioned in Section 1, the failure rate for SEU's in a ground based installation such as the CCS may be estimated to be in the region of 10^{-8} / bit / hr. The transient failure rate λ_T may therefore be calculated as follows:

$$\lambda_T = 10^{-8} \cdot B \cdot P_f \tag{4}$$

Where B is the numbers of bits in RAM that may be affected by a transient and p_f is the probability that such an upset will lead to failure. Using the information from tables VI and VII, the values of pf for the two implementations may be calculated as 0.2271 and 0.012 for the simplex and duplex case respectively. Applying (4) yields the following result:

$$\begin{aligned} \lambda_T(Simplex) &= 3.71 \times 10^{-6} \\ \lambda_T(Duplex) &= 2.36 \times 10^{-7} \end{aligned} \tag{5}$$

As can be clearly seen from this result, although the methodology results in a (slightly increased) usage of system RAM, the resulting reduction in the transient failure rate is significant, and in this case is enough to encompass an entire safety integrity level (SIL) [4][5].

7 Conclusions And Further Work

In this paper, a novel approach to software implemented fault-tolerance has been presented. The approach, based on an SPMD architecture, can be used to compliment on-chip error detection mechanisms and existing SBST techniques for COTS processors used in embedded system designs.

The approach we have described relies on both data and instruction duplication. It has been described how the required data types can be implemented as C++ classes and exported into C programs. It has been shown that the method is easily applied, results in readable code, and is able to tolerate 100% of the injected faults in both of the preliminary benchmarks described. In addition, the methodology was shown to perform extremely well in a real-time control application, reducing the observer number of system failures by a factor of 19. Since such a factor can be enough to encompass an entire SIL, such a result should be of interest to prospective developers of safety critical embedded systems.

Whilst the application of the techniques clearly provides high levels of fault detection and tolerance, there is obviously a trade-off with increases in the code and data size and task execution time. Prospective designers must obviously also take these factors into account when considering the techniques. The investigations that have been carried out in this paper reveal that, whilst the impact of the techniques is mostly application dependant, in most cases they can be tolerated with ease.

Additionally, the techniques we have described in this paper can be applied without the need for automatic code generators, and the impact on source code readability and maintainability is negligible. It is also noted that the technique is portable and can be applied with ease to any program structure. Again these points are of note from the perspective of the safety critical system developer.

With the availability of low-cost, high-performance 32-bit microprocessors, the impact of increased CPU overheads may be somewhat diminished over the results described in this study. Further work in this area can explore this possibility, along with possible techniques aimed at providing similar levels of redundancy for the CPU registers.

8 Acknowledgements

The work described in this paper was supported by the Leverhulme Trust (Grant F/00 212/D). The first author is also grateful to Michael J. Pont of the embedded systems laboratory for his suggestions and advice during the preparation of the paper.

An early version of this paper was presented at a Special Session on Safety-Critical Systems at the 6th International Conference on Computational Intelligence, Man-Machine Systems and Cybernetics (CIMMACS), Tenerife, Spain, December, 2007.

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, in 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, in Proc. IEEE Latin American Test Workshop, 2002.

[11] M.J. Pont. Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers, ACM Press / Addison-Wesley Publishing, 2001.

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

[13] Cascaval, P., Silion, R. “March Test for 3-Coupling Faults in Random-Access Memories: A Built-in Self-Testing Logic Design,” WSEAS Trans. Computers, 2(6), pp. 215-221, February 2007.

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

[15] J. Sosnowski, Software-based self-testing of microprocessors, Journal of Systems Architecture, Vol. 52, pp. 257-271, 2006.

[16] MIL-HDBK-217F, Military Handbook of Reliability Prediction of Electronic Equipment, December 1991.

[17] A. Rajabzadeh, S.G. Miremadi, Transient detection in COTS processors using software approach, Microelectronics Reliability, Vol. 46, pp. 124-133, 2006.

[18] S.A. Jankovic, D.M. Maksimovic, Power saving modes in modern microcontroller design, diagnostics and reliability, Microelectronics Reliability, Vol. 43, pp. 319-326, 2003.

[19] J.S. Lee, M.C. Kim, P.H. Seong, H.G. Kang, S.C. Jang, Evaluation of error detection coverage and fault tolerance of digital plant protection system in nuclear power plants, Annals of Nuclear Energy, Vol. 33, pp. 544-554, 2006.

[20] C. Gong, R. Melhem, R. Gupta, On-line error detection through data duplication in

distributed memory systems, Microprocessors and Microsystems, Vol. 21, pp. 197-209, 1997.

[21] Ong, H.L.R and Pont, M.J. (2002) "The impact of instruction pointer corruption on program flow: a computational modelling study", Microprocessors and Microsystems, 25: 409-419.

[22] Ong, H.L.R, Pont, M.J. and Peasgood, W. (2001) "A comparison of software-based techniques intended to increase the reliability of embedded applications in the presence of EMI" Microprocessors and Microsystems, 24 (10): 481-491.

[23] Pont, M.J. and Ong, H.L.R. (2003) "Using watchdog timers to improve the reliability of TTCS embedded systems", in Hruby, P. and Soressen, K. E. [Eds.] Proceedings of the First Nordic Conference on Pattern Languages of Programs, September, 2002 ("VikingPloP 2002"), pp.159-200. Published by Microsoft Business Solutions. ISBN: 87-7849-769-8.

[24] MISRA, "Development guidelines for vehicle-based software," Motor Industry Software Reliability Report, October 2004.

[25] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner and J. Reisinger, "Tolerating transient faults in MARS", in Proc 20th Int. Symp. on Fault Tolerant Computing, pp. 466-473, Newcastle upon Tyne, UK, June 1990.

[26] Ayavoo, D., Pont, M. J., Fang, J., Short, M., and Parker, S. A 'Hardware-in-the Loop' testbed representing the operation of a cruise-control system in a passenger car. In: Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), pp. 60-90, 2005. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

[27] Short, M. J. and Pont, M. J. Hardware in the loop simulation of embedded automotive control systems. In: Proceedings of the 8th IEEE International Conference on Intelligent Transportation Systems (IEEE ITSC 2005) held in Vienna, Austria, pp. 226-231, 2005.

[28] Castelli, J., Nash, C., Ditlow, C. and Pecht, M. Sudden acceleration – the myth of driver error. University of Maryland, CALCE EPSC Press, ISBN 0-9707174-5-8.

[29] Mauser, H. and Thurner, E. Electronic Throttle Control – A Dependability Case Study. Journal of Universal Computer Science, Vol. 5, No. 10, pp. 730 – 741, 1999.

[30] de Almeida Jr, G.,J., Melnikov, S.S.S., de Camargo Jr, J.B., Jose de Sousa, B. "Defensive Programming for Safety-Critical Systems," WSEAS Trans. Systems, 2(2), pp. 307-312, April 2002.

[31] Short, M. "Development Guidelines for Dependable Real-Time Embedded Systems," In: Proceedings of the 6th IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2008), Doha, Qatar, April 2008.