# Empirical Comparison of Software-Based Error Detection and Correction Techniques for Embedded Systems

Royan H. L. Ong
Student member IEE
Department of Engineering,
University of Leicester,
Leicester, United Kingdom.
+44 (0)116 2522873
hlro1@le.ac.uk

Michael J. Pont
MBCS
Department of Engineering,
University of Leicester,
Leicester, United Kingdom.
+44 (0)116 2522559
m.pont@le.ac.uk

## ABSTRACT

"Function Tokens" and "NOP Fills" are two methods proposed by various authors to deal with Instruction Pointer corruption in microcontrollers, especially in the presence of high electromagnetic interference levels. An empirical analysis to assess and compare these two techniques is presented in this paper.

Two main conclusions are drawn: [1] NOP Fills are a powerful technique for improving the reliability of embedded applications in the presence of EMI, and [2] the use of Function Tokens can lead to a reduction in overall system reliability.

## Keywords

Instruction Pointer Corruption, Electromagnetic Interference, EMI, Function Token, NOP Fill, Software-based Error Detection Techniques, Embedded systems

## 1. INTRODUCTION

The use of microcontrollers in embedded systems has increased tremendously with the rise in consumer demands for safer, cheaper and more versatile electronics devices. As for other sectors, the greater sophistication demanded by modern-day applications coupled with the miniaturisation trend makes microcontrollers the ideal electronics device in many applications.

The automotive industry forms a key part of the microcontroller market, not least because of a desire to improve passenger safety levels. In this regard, embedded electronics systems, such as air bags and anti-lock brakes [**9**], have already proved effective while adaptive cruise control systems are thought to have the potential to reduce the number of accidents between 3% and 10% [**3**]. However, the increasing reliance on embedded systems for safety-related automotive systems also introduces new risks. Many factors, from the hardware, software and design point of view, may influence the reliability of embedded systems: within the present paper, we are concerned with the influence of electromagnetic interference (EMI) on the microcontroller itself (see [**1**]), with specific reference to the corruption of the instruction pointer (IP).

This paper is organised as follows: the problem associated with IP errors and software-based techniques proposed by other authors to detect and correct such errors are described Section 2. Section 3 introduces the experimental procedure carried out to statistically evaluate these techniques. Both experimental results and discussion are found in Section 4 while Section 5 concludes this paper.

Although the experiment and results were based on the 8051 family of microcontrollers, most topics discussed here are applicable to other microcontroller families.

## 2. SOFTWARE TECHNIQUES TO DEAL WITH IP ERRORS

Arguably, the most serious form of EMI-induced error in a microcontroller is corruption of the instruction pointer, the register that stores the address of the next instruction to be executed. Though similar in construction to other registers, such as the accumulator, corruption of the IP can lead to unpredictable program branches, and – consequently – dramatic changes in system behaviour.

The impact of IP corruption is particularly difficult to predict because most microcontroller instruction sets include some instructions longer than one byte in size ("multibyte instructions") [**10**, **11**]. During program execution it is possible that IP corruption may occur before all the bytes of a multibyte instruction have been read. Under such circumstances, the correct instruction is executed but with the wrong operands – a phenomenon we term the "Early Multibyte Instruction Trap" (EMIT) [**11**].

Further problems arise when the corrupted IP points to a memory location that contains the second or third byte (or greater) of a multibyte instruction. The processor will misinterpret the value found at this location (and, often, at subsequent locations too). We refer to this phenomenon as the "Late Multibyte Instruction Trap" (LMIT) [**11**]. Note that it is also possible that **both** EMIT and LMIT will be evident, in succession, when an IP error occurs.

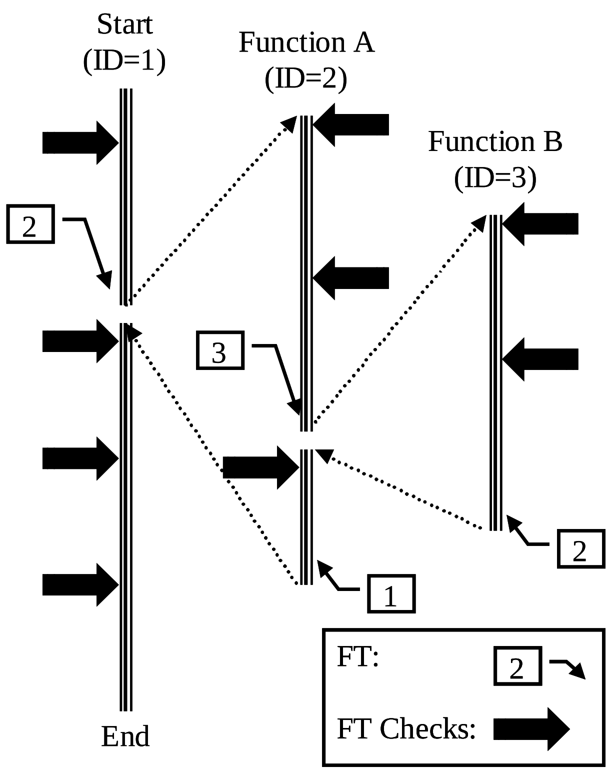

**Figure 1: Schematic of Function Token Implementation**

Function Tokens (FT) and NOP Fills (NF) are two software-based techniques proposed by various authors such as Banyai & Gerke [**2**], Campbell [**4**, **5**], Coulson [**6**] and Niaussat [**8**] to deal with program flow errors brought on by IP corruption.

We describe both of these techniques below.

## 2.1 Function Tokens

The idea behind Function Tokens [**2**, **4**, **5**, **6**, **8**, **10**, **11**] is to use one or more free data memory locations (known as the "token") to store the unique ID of each function (see Figure 1). Just before a function is called, the token is assigned the value of the function ID. The token is then compared with the function's ID within the function itself. If the ID comparison fails, the program will execute the IP error handler (IPEH); this is diiscussed further below.

The above schema describes the simplest implementation of FT, assuming a one-to-one relationship between caller and callee functions. However, normal programs usually have functions with many-to-many relationship, which makes implementation more complicated. For effective FT implementation, the calls between various functions must be fully mapped.

FTs are implemented as part of the program itself, and result in an increase in the final code size. Coulson [**6**] estimates a code swell between 10-20% in one FT implementation.

## 2.2 NOP Fills

NOP Fills [**4**, **5**, **6**, **8**, **10**, **11**] are carried out on unused program locations. Typically, program code is contiguously programmed into physical code memory locations from address location 0x0000, which usually leaves some free memory locations at the end of the physical memory space.

What NOP Fill (NF) does is to fill these locations with the value of 0x00, equivalent to the "NOP" (No Operation) 8051 instruction. As a result, an erroneous program branch into this region will not significantly alter the processor state. An IPEH would be located at the end of the physical code memory to deal with runaway program execution caught by the NOP Fill.

Note that, under normal program execution, both the NOP Fill region and the IPEH are unreachable.

## 2.3 Instruction Pointer Error Handler

The IPEH is a function that carries out the appropriate error handling when an IP error is detected. The type of error handling varies greatly between applications. For example, a "warm" reset may be appropriate for non-critical systems. Alternatively, a shutdown may be more appropriate for systems with multiple redundancies. For some safety-critical applications, such as engine-management units, placing the system into a 'safe' mode in the event of an EP error may be particularly effective.

Note that by locating the IPEH at the end of the physical code memory, it is possible to "share" this function between NOP Fills and Function Tokens, when both techniques are implemented. Figure 2 shows a generic physical memory layout when both techniques are implemented.

## 3. SIMULATION

The simulation discussed in this paper was carried out on dScope; an 8051 simulator developed by Keil GmbH. This simulator is capable of simulating at the instruction level as well as the C-syntax level, which ideally suits our requirement. In addition, the simulators' C-like scripting language allows us to carry out simulations almost autonomously. Unfortunately, its instruction-level resolution means we could not simulate and detect EMIT errors, and only LMIT errors are considered in this paper.

## 3.1 Simulated Programs

Three sets of programs, AlarmClock [**12**], GreenHouse [**7**] and CentralHeating [**13**], were used for the simulations discussed here. Each program set has four different versions employing different IP error detection and correction techniques. The four combinations of each program are suffixed _A, _B, _C and _D, and employ the error detection and correction techniques as so:

| | | |
|---|---|---|
| **Prog_A** | – | Without any form of error checking. |
| **Prog_B** | – | Implements NOP Fills. |
| **Prog_C** | – | Implements Function Tokens. |
| **Prog_D** | – | Implements Function Tokens and NOP Fills |

Figure 2 shows the physical code memory layout schematic of each program version as it would appear when programmed onto microcontrollers with internal code memory. Note that this figure is not to scale.

All programs were written in C and compiled with the C51 Compiler by Keil GmbH. These programs were chosen for the simulation as they are representative of real-world applications. Though lacking computationally intensive algorithms apart from a software-based I$^2$C protocol, these programs had complex control structures, the most complex of which, the AlarmClock project, was built on a cooperative scheduler [12]. These programs are available via their respective references.

We refer to each program as Alarm_C, Heat_A, etc., or collectively as, program "A" for Alarm_A, Green_A and Heat_A. Referring to "Alarm" itself would mean all four versions of that program.

## 3.2 Implementing NFs and FTs

Two methods are generally used when creating the NOP Fill region. The simpler method involves filling the EPROM programmer's code buffer with the value of 0x00, followed by downloading the program code destined for the microcontroller. In doing so, all empty locations would have the value of 0x00, thus the NOP Fill is created.

The second method is to create the NOP Fill in assembly language with the help of assembler directives. Although harder to implement, this method allows greater flexibility and control especially when different NOP Fill topologies are intended. The former method also tends to be more time consuming in the long run due to the overhead in configuring the EPROM programmer on every programming cycle. Apart from that, modern microcontrollers with In-circuit Serial Programming (ISP) programming capabilities can also be programmed without physically removing the microcontroller. The second "NOP Filling" method was used in the studies reported here.

Since the test programs were taken from various sources, all the code had to be meticulously scrutinised to map out all the callers and callees of every function, including those called from the interrupt service routine (ISR). Once the mapping was complete, each function was assigned a unique ID. Function Token checks were implemented at the start of each function and just after program execution is returned to the caller routine.

With ISRs, no Function Token checks were carried out since they could be called from practically any part of the program. Though it is possible to create a "local" token, this approach was deemed unnecessary, as the ISR routines were short and non-critical.

The IPEH, as mentioned earlier, resides at the end of the physical code memory location. For the simulation programs "B", "C" and "D", the IPEH was written in assembler and located at a predefined memory location with the help of assembler directives.

Table 1 shows the program and NOP Fill size of each program as well as the number of Function Token checks implemented.

Due to the increase in code size for Alarm_C and Alarm_D, a microcontroller with 8kB of internal code memory is assumed, though the other programs could fit into 4kB devices. This is done to ensure a standard basis of comparison.

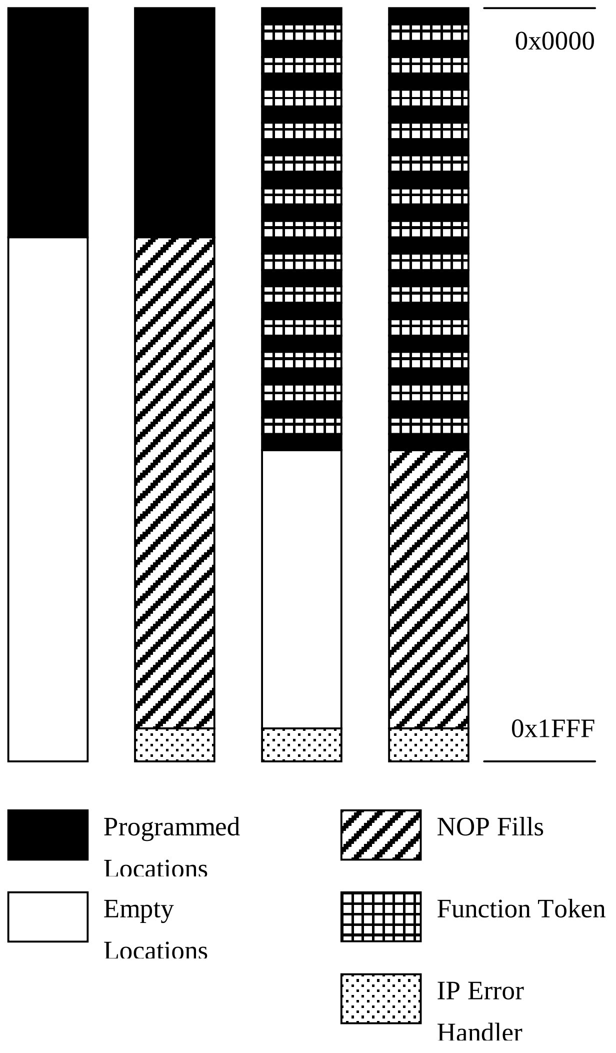

**Figure 2: Generic Memory Map when Implementing NOP Fills and Function Tokens**

## 3.3 Simulation procedure

For the results to be statistically convincing, a large number of simulation cycles have to be carried out. dScopes' C-like scripting language helped to automate the simulation procedure.

The simulation script included all LMIT locations for each program. This was generated by an "in-house" program scanning the microcontroller-downloadable file (in Intel H86 format). This script controlled the entire simulation process, classified the results and generated the appropriate log files.

One thousand simulation cycles were carried out on each program with an error injected anywhere between the 1st and 10000000th microcontroller instruction cycle. Before commencing each simulation cycle, the instruction cycle in which the error would occur (ECY), and the corresponding erroneous IP value (EIP), is generated randomly. Each simulation cycle also starts from the beginning, equivalent to a microcontroller reset.

**Table 1: Program statistics**

| Alarm Clock | Total size (bytes) | NOP Fill (bytes) | FT checks (amount) |
|---|---|---|---|
| **A** | 3283 | 0 | 0 |
| **B** | 3331 | 4861 | 0 |
| **C** | 6535 | 0 | 120 |
| **D** | 6535 | 1645 | 120 |

| Green House | Total size (bytes) | NOP Fill (bytes) | FT checks (amount) |
|---|---|---|---|
| **A** | 2286 | 0 | 0 |
| **B** | 2329 | 5862 | 0 |
| **C** | 3940 | 0 | 98 |
| **D** | 3940 | 4245 | 98 |

| Central Heating | Total size (bytes) | NOP Fill (bytes) | FT checks (amount) |
|---|---|---|---|
| **A** | 1954 | 0 | 0 |
| **B** | 1997 | 6195 | 0 |
| **C** | 3799 | 0 | 120 |
| **D** | 3799 | 4387 | 120 |

The simulated errors are classified into five categories as follows:

| | |
|---|---|
| **EL** – | Empty Location. The simulated error points the IP to an empty code memory location. |
| **NF** – | NOP Fill. The simulated error is detected and corrected by the NOP Fills. |
| **FT** – | Function Token. The simulated error is detected and corrected by an FT check. |
| **LMIT**– | Late Multibyte Instruction Trap. The simulated error points the IP at the second or third byte of a multibyte instruction. |
| **UE** – | Undetected Error. The simulated error goes undetected by NF or FT. |

By evaluating EIP alone, EL, NF and LMIT are immediately identifiable. The simulation script would first determine if EIP points to a location outside the programmed code memory, or, within the NF region for programs implementing NF. If it did, the error was immediately logged as an EL or NF hit and a new simulation cycle was started.

If EIP pointed to a code memory location within the program itself (including the IPEH, if any), the simulation script looked up the LMIT table to determine if that particular address location is part of the second or third byte of a multibyte instruction. A match in that table caused an LMIT hit to be recorded and a new simulation cycle was started.

Only when none of the above-mentioned conditions was met would actual simulation of the microcontroller from the first to the ECY$^{th}$ instruction cycle begin. Simulation time is further shortened by simulating at the fastest possible speed up to the ECY$^{th}$ cycle, instead of stepping through each instruction. This was done by setting breakpoints as no errors were injected before the ECY$^{th}$ cycle.

Upon the breakpoint triggering, the value of the IP is changed to that of EIP by the simulation script. Up to a further 50000 instruction cycles would then be executed in single step mode to determine if an FT check has detected the error. Only if the FT check fails to detect an error would it be recorded as a UE hit.

## 4. RESULTS AND DISCUSSION

The simulation script was written to produce a raw text summary of each cycle containing ECY and EIP values. Total hits for EL, NF, FT, LMIT and UE for each program were also displayed at the end of the summary, as shown in Table 2.

**Table 2: Simulation results**

| Alarm Clock | EL | NF | FT | LMIT | UE |
|---|---|---|---|---|---|
| **A** | 587 | 0 | 0 | 146 | 267 |
| **B** | 0 | 598 | 0 | 164 | 238 |
| **C** | 169 | 0 | 460 | 276 | 95 |
| **D** | 0 | 224 | 405 | 270 | 101 |

| Green House | EL | NF | FT | LMIT | UE |
|---|---|---|---|---|---|
| **A** | 718 | 0 | 0 | 130 | 152 |
| **B** | 0 | 711 | 0 | 136 | 153 |
| **C** | 524 | 0 | 224 | 218 | 34 |
| **D** | 0 | 526 | 209 | 219 | 46 |

| Central Heating | EL | NF | FT | LMIT | UE |
|---|---|---|---|---|---|
| **A** | 748 | 0 | 0 | 128 | 124 |
| **B** | 0 | 772 | 0 | 107 | 121 |
| **C** | 538 | 0 | 145 | 272 | 45 |
| **D** | 0 | 547 | 163 | 237 | 53 |

### 4.1 Impact of various errors

Before discussing the variations between different programs, it is necessary to discuss the impact of each error category on the overall program reliability.

Jumps to empty code memory locations (EL) should not be tolerated since program code would be executed at location 0x0000 after the IP points to the address above the highest physical code memory. This phenomenon is due to memory aliasing and though it is similar to resetting the microcontroller, it does not reset internal flags and registers to known states. For embedded systems, this scenario should be avoided, more so when safety is of prime concern.

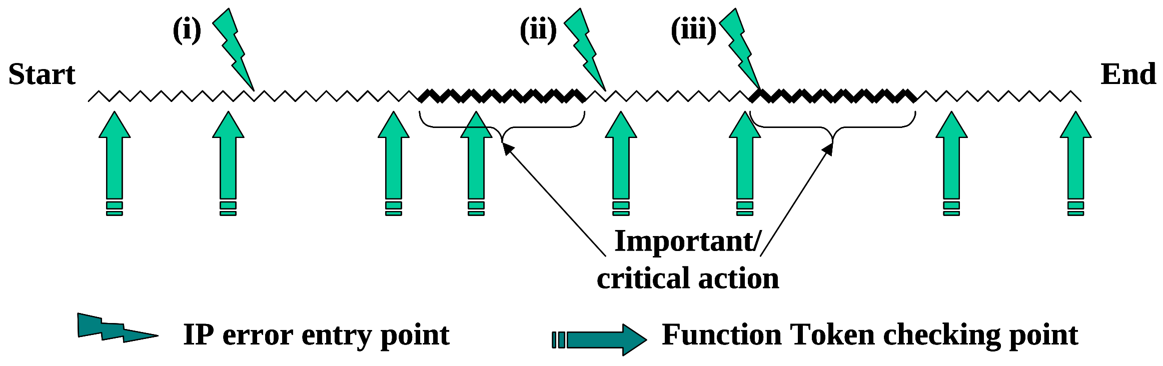

**Figure 3: Function Token Error Detection Scenarios**

On the other hand, any IP errors “falling” into the NOP Fill region will be detected, and the suitable recovery strategy applied. Moreover, only the IP and timing-associated registers change before the recovery strategy is executed. As a result, this is probably the best condition for a microcontroller to be in as a result of IP errors. The main problem with NOP Fills is it is possible that – in systems with large NOP Fill areas – there will be a relatively long error-recovery latency. A slight variation on the basic NOP Fill technique which is designed to reduce the recovery latency is discussed in [**11**].

Function Tokens do detect IP errors and execute the IP Error Handler, as shown in programs C and D. However, it must be stressed that the errors are generally detected some instruction cycles after they occur, as Figure 3 illustrates. Scenario (i) is the typical situation where the IP erroneously “fall” between FT checks and is detected within a reasonable time span. However, scenario (ii) is much preferred as the FT check detects the error on the next instruction cycle: unfortunately, this will very rarely happen. In scenario (iii), the FT check did detect the IP error, but only after critical instructions were executed.

Arguably, the worst error situation happens when the IP causes an (L)MIT error. In such circumstances, software-based error detection and correction techniques may prove ineffective. Hence, the state of the processor and program flow may be unpredictable.

Undetectable Errors (UEs) may also, clearly, prove dangerous, since the system may still continue to function without any apparent problems. In our experiment, only a few critical variables were checked to determine system integrity. However, it is possible that more subtle problems could arise, such as incorrect timer duration, and input readings, which would be very hard to detect even with expensive equipment and increased software complexity.

Note that our simulation actively detects all but “Undetectable Errors”, which are inferred by a process of elimination.

## 4.2 Program variations

Quite a few points may be made regarding the differences between the various programs.

The first thing to note is the variation in code size. As predicted, the code sizes for programs employing error-checking routines are larger than those without (see Table 1). However, “B” program sizes are only marginally larger than “A” programs, as a result of the additional IPEH. By contrast, programs “C” and “D” are between 69% and 96% larger than programs “A” and “B”. The huge difference is solely attributed to the implementation of Function Tokens, which in turn is proportional to the complexity and number of Function Token checks used. Coulson’s [**6**] estimates of 10% to 20% code swell are only obtainable if the FT checks are implemented at the start and end of each function – resulting in less than 100% branch coverage.

By observing Table 2 alone, we could see that with no error checking, a large percentage of IP errors will lead to the execution of “empty instructions” (specifically “MOV R7,A”). Implementing NOP Fills alone traps around the same number of errors that would have been destined for EL. We conclude that NOP Fills, which are easily implemented, should be used in all situations.

As mentioned earlier, implementing Function Tokens increases program size. As a consequence, an increase of LMIT hits was recorded for programs “C” and “D”. This leads to the conclusion that though FT detects and corrects errors, they would also increase the probability of encountering an MIT error, which is one of the trade-offs of using this technique. The complexity involved in implementing Function Tokens also opens a new avenue for software bugs and design errors.

The conclusion drawn by comparing programs “B”, “C” and “D” is most important in this paper. As observed from Table 2, implementing both techniques simultaneously increases the number of detected IP errors. However, based on previous discussion, it would be desirable to have NF detect the errors instead of FT. When both techniques are implemented, the size of the NOP Fill region decreases due to an increase in program size.

## 4.3 Effectiveness of NOP Fills

The relationship between the NOP Fill region size and its effectiveness may not be immediately clear. The key lies with the assumption that when an IP error occurs, the probability of the IP taking on any addressable value (0 to 65535) is the same. Hence, the size of the NOP Fill region gives a good estimate of its probability to detect and correct IP errors. Figure 4 strengthens this point by showing the good correlation between the percentage of physical code memory taken up by the NF (black), and the percentage of errors detected by NF (grey). It also shows the relationship between NF size and the percentage of errors detected by it to be directly proportional. Based on this finding, Figure 5 shows the interpolated results for programs “B” and “D” when the amount of physical code memory increases.

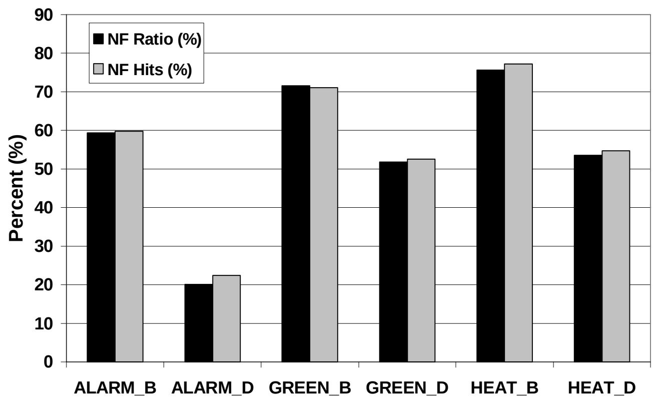

**Figure 4: Correlation between NF hits and NF size**

A mathematical relationship between code size, instruction types and other parameters with the statistical effectiveness of FT and NF can be found in [**10**].

Though it would be good to have as large a NOP Fill region as possible, this may not an economically-viable solution for all systems. Hence, a compromise has to be made between physical code memory size and the NF hit probability. There is also a threshold beyond which increased memory size has little impact on NF hit probability, as can be seen in Figure 5.

## 5. CONCLUSION

Function Tokens and NOP Fills do detect and correct errors associated with IP corruption. However, the effectiveness of Function Tokens is questionable and implementation is an error prone affair. Overall, FTs may increase a system’s IP error detection rate, but it do not necessarily increase its reliability. On the other hand, NOP Fills are simple and effective, and hence should be used in all embedded systems.

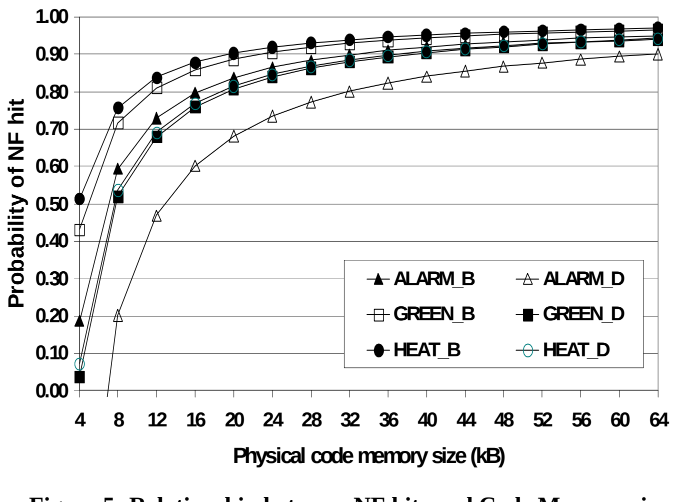

**Figure 5: Relationship between NF hits and Code Memory size**

## 6. REFERENCES

[1] Armstrong, K. What EMC is, and Some Examples of EMC Problems caused by Software. IEE Colloquium on Electromagnetic Compatibility of Software (98/471), Birmingham, UK, 1998.

[2] Banyai, C. and Gerke, D. EMI Design Techniques for Microcontrollers in Automotive Applications. Intel Application Note AP-711.

[3] Broughton, J. Assessing the Safety of New Vehicle Control Systems. Proceedings of the First World Congress on Applications of Transport Telematics and Intelligent Vehicle-Highway Systems, Paris, France, 1994, 2141-2148.

[4] Campbell, D. Designing for Electromagnetic compatibility with Single-Chip Microcontrollers. Motorola Application Note AN1263.

[5] Campbell, D. Defensive Software Programming with Embedded Microcontrollers. IEE Colloquium on Electromagnetic Compatibility of Software (98/471), Birmingham, UK, 1998.

[6] Coulson, D.R. EMC Techniques for Microprocessor Software, IEE Colloquium on Electromagnetic Compatibility of Software (98/471), Birmingham, UK, 1998.

[7] Meikle, Colin. Green House Computer. Everyday Practical Electronics Magazine, Wimborne Publishing, (Jul 98) 492; (Aug 98) 610.

[8] Niaussat, A. Software techniques for improving ST6 EMC performance. ST Application Note AN1015/0398.

[9] NHTSA Light Vehicle Antilock Brake System Research Program Task 4. National Highway Traffic Safety Administration (NHTSA) Report, 1999.

[10] Ong, R. H. L., Pont, M. J., Peasgood, W. A comparison of software-based techniques intended to increase the reliability of embedded applications in the presence of EMI. Microprocessor and Microsystems, Elsevier Science (in-press).

[11] Ong, R. H. L., Pont, M. J., Peasgood, W. Hardware-Software Tradeoffs when Developing Microcontroller -Based Applications for High-EMI Environments, IEE Colloquium on Hardware-Software Co-Design (00/111), London, UK, 2000.

[12] Pont, M.J. Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers (in press). Addison Wesley, 2001. (ISBN 0-201-33138-1).

[13] Stone, R. Central Heating Controller. Everyday Practical Electronics Magazine, Wimborne Publishing, (Nov 96) 831.