

A Comparison of Software-Based Techniques Intended to Increase the Reliability of Embedded Applications in the Presence of EMI

Royan H. L. Ong, Michael J. Pont and William Peasgood

Control & Instrumentation Research Group

Department of Engineering

University of Leicester

University Road

LEICESTER

LE1 7RH

United Kingdom.

Pre-print of: Ong, H.L.R., Pont, M.J. and Peasgood, W. (2001) "Do software-based techniques increase the reliability of embedded applications in the presence of EMI?" *Microprocessors and Microsystems*, **24** (10): 481-491.

Abstract

Corruption of the instruction pointer in an embedded computer system has been shown to be a common failure mode in the presence of electromagnetic interference, and previous investigators have suggested that the use of techniques such as “Function Tokens” and “NOP Fills” can reduce the impact of such failures. In this paper, both a statistical analysis and empirical tests of code from an embedded application are used to assess and compare these techniques. Two main results are presented. First, it is demonstrated that claims about the effectiveness of Function Tokens may be neither well founded nor generally applicable: specifically, it is concluded that, rather than increasing system reliability, the use of Function Tokens will have the opposite effect. Second, it is demonstrated that NOP Fills may be easily applied in most embedded applications, and that the use of this approach can have a positive impact on system reliability.

Keywords:

Software Prevention, Instruction Pointer Corruption, Electromagnetic Interference, Function Token, NOP Fill

1. Introduction

Recent economic, legislative and technological developments in the automotive sector mean that an increasing number of road vehicles contain sophisticated Distributed Embedded Systems (DESs), consisting of a number of microcontrollers and/or microprocessors linked by one or more computer networks. In many cases, a key motivation for the introduction of DESs has been a desire to improve safety. Here, DESs such as air bags and anti-lock brakes [15] have already proved effective. Forthcoming developments are thought likely to offer further safety improvements. For example, while the human driver is very good at steering the car under normal conditions, where decisions can wait for around a second, “steer-by-wire” applications are thought likely to prove more effective where the vehicle begins to skid or roll over and more rapid responses are required [1]. In a similar vein it has been estimated that adaptive cruise control systems have the potential to reduce the number of fatal motorway accidents worldwide by between 3% and 10% [2, 8].

Despite these current (and potential) safety improvements, the increasing reliance of modern road vehicles on DESs also introduces new risks, not least because the distributed nature of such applications means that a hardware or software failure anywhere in the system has the potential to impact greatly on the overall safety integrity level (SIL) of the vehicle. Many factors may influence the reliability of the software in such applications, including the programming language used and the overall design methodology [6]. Within the present paper, we are concerned with the influence of electromagnetic interference (EMI) on embedded automotive applications constructed from microcontrollers.

The passenger car is a harsh environment for embedded systems. The typical car contains numerous electromechanical devices such as contact breakers, alternators, relays and ignition coils that are excellent sources of high-energy, wideband, electromagnetic noise that is capable of corrupting many electronic circuits [12]. To deal with such problems, hardware solutions, including device shielding, wiring screening and input/output filtering is widely used [9]. However, hardware solutions are expensive, can suffer physical damage, and – in applications

such as Hall-effect sensors – can interfere with normal device operation. As a result, various studies [3, 4, 5, 7, 11, 13, 14] have been conducted to explore the possibility of using software-based EMI detection and recovery techniques as an alternative to or, more commonly, as an adjunct to current hardware-based techniques.

Arguably, the most serious form of EMI-induced error in an embedded microcontroller is corruption of the instruction pointer (IP), also known as the program counter. Two software techniques, “Function Tokens” (FT) and what we refer to here as “NOP Fills” (NF) have been proposed [3, 5, 10, 11] to deal with IP corruption. In this paper, we carry out a detailed empirical and theoretical assessment of each of these techniques. The results are carried out using members of the widely-used 8051 family of microcontrollers, but the findings are applicable to any embedded microcontroller-, microprocessor- or DSP-based system.

The paper is organised as follows. Section 2 of this paper describes the impact of IP corruption and its consequences. In Section 3, both Function Tokens and NOP Fills are described in detail and their usage illustrated. A model to estimate the percentage of memory locations where Function Tokens and NOP Fills will detect IP errors is shown in Section 4. Section 5 compares the model with simulation results for a set of programs employing different error detection and recovery methods. The effectiveness of NOP Fills and Function Tokens are also compared in this section together with some general comments on both techniques. Future work and conclusions are presented in Section 6 and Section 7 respectively.

2. The Impact of IP Corruption on Program Execution

The Instruction Pointer is only one of many registers in an embedded processor and there is no evidence to suggest that this particular register is any more or less susceptible to EMI than the others. However, the impact of corruption to the IP is arguably the most serious result of EMI, as it can result in disruption to the program flow.

As mentioned in the introduction, we are concerned with the 8051 microcontroller in this paper. Since the IP of the 8051 is a 16-bit wide register, we can make the reasonable assumption that – in response to IP corruption – the IP may take on any value in the range 0 to 65535. In these circumstances, the 8051 processor would fetch and execute the next instruction from the code memory location pointed to by the corrupted IP register. This code memory location may contain program code (a situation we discuss in Section 2.1), data constants (Section 2.2) or may be empty (Section 2.3). To complicate matters further, the amount of physical code memory is usually less than that addressable by the IP, resulting in an effect known as memory aliasing; we postpone consideration of this issue until Section 2.4.

2.1 Vectoring to unprogrammed memory locations

We consider first the possible impact of the program flow being diverted to unprogrammed memory locations. Such locations will usually, by default, have the contents 0xFF, which corresponds to the “MOV R7,A” in the 8051 instruction set (“Copy the contents of the accumulator to register R7”).

In many applications, the program code will occupy the lower code memory addresses, and the remainder of the memory will be unprogrammed. In these circumstances the processor will execute “MOV R7,A” instructions until the IP reaches the end of the physical code memory. The processor will then continue executing program code at location 0x0000. This can have an impact similar to a processor reset.

In other applications there may be unprogrammed “gaps” in the memory maps, followed by constant data or program code. Execution of this code (or data, treated as code) is likely to have less predictable side effects, as we discuss below.

2.2 Vectoring to programmed (code) locations

Clearly, corruption of the instruction pointer that causes the program flow to be diverted to a “random” address is likely to cause severe side effects. However, the precise impact of such diversions can be very difficult to predict.

A particular problem arises because in the 8051 (and many other processors), more than half of the instructions are “multibyte instructions”, such as “POP” and “ACALL” that occupy two or three memory locations, respectively. IP corruption may cause the program flow to be diverted to any of these locations.

To illustrate the nature of the resulting “multibyte instruction trap” (MIT), consider the assembly code shown in *Listing 1*.

<<<*Listing 1*>>>

If the IP is corrupted and takes on the value 0x0101, then the code in *Listing 1* will be interpreted as shown in *Listing 2*.

<<<*Listing 2*>>>

In this example, the first three instructions of the program code have been misinterpreted while the rest remain unchanged. Of course, the number of instructions that will be misinterpreted depends on the instruction sequence, the corrupted IP value and the state of the processor at the time of IP corruption. In short, the precise impact is impossible to predict in most practical situations.

2.3 Vectoring to programmed (data) locations

The problems with misinterpretation of instructions also apply to data values stored in the code area since, to the processor, data constants - such as digital filter coefficients stored in the code

area - are indistinguishable from program code. Again, the results of this are very difficult to predict.

2.4 Memory aliasing

Many 8-bit microcontroller applications do not require the full 64kB of available code memory and, as a result, most microcontrollers have on-chip ROM memory of between 1kB and 12kB. This can result in an effect known as memory aliasing, illustrated in *Figure 1*.

<<<*Figure 1*>>>

Figure 1 shows a 2kB program on microcontrollers with 16kB and 64kB of physical code memory. In this example, the IP is addressing location 0xA552 on the 64kB device. Note that, in the 16kB device, the lack of the upper two address lines means that addresses 0x2552, 0x6552, 0xA552 and 0xE552 will all address the same physical code location.

3. Software-based IP Corruption Detection and Correction Techniques

As noted in the introduction, two software-based IP error detection and correction techniques have been proposed and discussed in previous studies [3, 4, 5, 10, 13].

We consider both of these techniques here.

3.1 NOP Fills

The technique we refer to here as “NOP Fills” [3, 4, 10] has two components. First, we fill unused locations at the end of the program code memory with single-byte “No Operation” (NOP), or equivalent, instructions. Second, a small amount of program code, in the form of an

“IP Error Handler” (IPEH), is placed at the end of code memory: at its simplest, where more sophisticated forms of error recovery are not possible, the IPEH will simply reset the processor. Note that the IPEH is unreachable except in the event of IP corruption. A code memory map view of this implementation is show in *Figure 2*.

<<<*Figure 2*>>>

The operation of NOP Fills may be easily predicted. When an IP error occurs and the IP points to a memory location within the NOP Fill area, the processor will repeatedly execute NOP instructions until the IP points to the start of the IPEH. The error handler then carries out its intended recovery function.

3.2 Function Tokens

The second software-based technique used to reduce susceptibility to EMI is the Function Token [3, 4, 5].

The implementation of a simple Function Token is superficially straightforward:

- Before we call a function, we set the value of one or more (global) variables to a known value.
- During the function execution, we check that the global variables have the required values: if they do, we are reassured: if they do not, we assume that (through EMI or some other fault) we have been “thrown” into the function, and we take “appropriate action”.
- Depending on the system, and the function call, this appropriate action may involve (for example) re-starting the system, or returning the system to an appropriate recovery state.

Variations on this basic scheme are possible. For example, Niaussat [10] uses data memory locations as flags, but the technique is essentially the same.

Figure 3 shows a simple implementation of Function Tokens with two functions and the main process. The square boxes with its value represent the tokens.

<<<*Figure 3*>>>

Implementing Function Tokens will inevitably increase the program code size: the precise increase depends on the number of checks carried out and its complexity. On average, the simplest check takes approximately 20 bytes of program code memory, rising rapidly as complex token comparisons are carried out. Coulson [5] estimates a code swell of between 10% and 20% when Function Tokens are implemented.

4. Statistical Analysis of the Problem

A model for predicting the safety of a program in the event of IP corruption is described in this section. This model, based on the compiled program code, calculates the probability that IP errors will be detected and suitable recovery strategies carried out before any unwanted processor state changes (only IP increments should be allowed).

In this model, the locations in code memory locations are each classified as Safe Locations (SL), Unsafe Locations (UL) and Dangerous Locations (DL). The following list defines each location type:

Safe Locations (SL) – In the event of IP corruption, vectoring to a Safe Location means that the detection and recovery (that is, execution of the IPEH) will be guaranteed to happen before other non-error checking instructions are executed (except NOP instructions).

Unsafe Locations (UL) – In the event of IP corruption, vectoring to an Unsafe Location means that immediate error detection / recovery cannot be guaranteed. However, the error may not be fatal and program execution may recover, or the IPEH may be invoked.

Dangerous Locations (DL) – In the event of IP corruption, vectoring to a Dangerous Location will mean that an MIT error will occur, followed by unpredictable program flow.

4.1 Classifying program instructions

All program instructions are classified as either UL or DL apart from the first byte of the FT and IPEH routines. *Table 1* shows the number UL and DL for each 8051 instruction arrangement based on the instruction value.

<<<*Table 1*>>>

In *Table 1* the instructions are classified and named based on their size (prefix “S”, “D” and “T” for single, double and triple byte instructions respectively) as well as the location of zero-valued (“NOP”) bytes (indicated by the suffix). Instructions with no zero-valued bytes are suffixed “N”.

Note that in the table “XX” stands for any non-zero byte value.

We can summarise the contents of *Table 1* as follows: the first byte of every instruction is considered UL since that instruction will not be misinterpreted when the corrupted IP takes on this value. However, any subsequent non-zero bytes (in multibyte instructions) will be classified as DL since they will cause instruction misinterpretation.

Note that instructions of type T2 are unique in the sense that even though the second location is zero in value, these instructions are considered to be “DL”. This is due to the fact that if the corrupted IP takes on the address of the second byte of a T2 instruction, instruction misinterpretation will still occur (since the third byte is non-zero). Since this type of instruction is

essentially the same as an instruction having all non-zero valued bytes, it is can be given the alternative label “TN”.

4.2 A model for predicting the impact of IP corruption

Figure 4 shows a typical memory layout with a physical code memory of size “M” employing both NOP Fills and Function Tokens.

<<<*Figure 4*>>>

In *Figure 4* and the discussions that follow, the following definitions (all in bytes) apply:

- M** = The total size of the code memory (ROM) available in the microcontroller.
- P** = The number of code memory locations occupied by the program code and constants (code memory locations occupied by the error-checking routines are excluded).
- P*** = The number of code memory locations occupied by the program code and constants (that is, **P**) plus all error checking routines, but excluding memory locations occupied by NOP Fills (if any).
- U** = The number of unprogrammed (“empty”) code memory locations.
- T** = The number of code memory locations occupied by Function Token checking routines.
- NF** = The number of code memory locations filled with NOP instructions.
- C** = The number of code memory locations occupied by the IP Error Handler (IPEH) routine.

All calculations and discussions are based on the assumption that, in the event of IP corruption, the contents of the IP may take on any value with equal probability.

Referring to *Figure 4*, **P*** is calculated according to *Equation 1¹*:

<<<Equation 1 >>>

- where:
- = Average size of each Function Token checking routine (bytes)
 - = Average size of each IPEH routine (bytes).
 - n** = Number of Function Token checks
 - m** = Number of IPEHs

The size of the NOP Fill area, NF, may be calculated from *Equation 2*.

<<<Equation 2>>>

Since all bytes in the NOP Fill guarantee error detection and do not change the state of the processor (apart from IP increments), they are classified as Safe Locations. The first byte of every Function Token check and the IPEH are also Safe Locations since error detection and recovery is also guaranteed. Therefore the number of Safe Locations may be defined in *Equation 3*.

<<<Equation 3>>>

Unsafe Locations would be calculated based on the program's instruction type. Thus by calculating a program's instruction breakdown, UL (in bytes) is given by *Equation 4*.

<<<Equation 4 >>>

- where:
- B_S** = Amount of single byte instructions in **P*** (bytes).
 - B_{DN}** = Amount of DN instructions in **P*** (bytes).
 - B_{D2}** = Amount of D2 instructions in **P*** (bytes)
 - B_{TN}** = Amount of TN instructions in **P*** (bytes).
 - B_{T23}** = Amount of T23 instructions in **P*** (bytes).
 - B_{T3}** = Amount of T3 instructions in **P*** (bytes).

As shown in *Table 1*, not all bytes of double and triple byte instructions are considered unsafe. Hence DN, TN and T3 instructions are multiplied by the ratio of the number of UL byte(s) to the length of the instruction. The number of Function Token checks and IPEH are also subtracted from UL since they have already been accounted for in SL. *Table 1* is also used to determine DL, which is given by *Equation 5*.

<<<Equation 5 >>>

Equation 5 may in turn be simplified to *Equation 6*

<<<Equation 6>>>

since there is no overlap between SL, UL and DL. *Equation 3*, *Equation 4* and *Equation 6* for SL, UL and DL respectively can be restated in percentages of physical code memory by dividing them by **M**. This gives *Equation 7* to *Equation 9*.

<<<Equation 7>>>

<<<Equation 8>>>

<<<Equation 9>>>

Note that *Equation 7* to *Equation 9* are valid irrespective of the amount of physical code memory available due to the impact of memory aliasing as discussed in Section 2.4.

5. Analysis and Simulation of NOP Fill and Function Token

Four identical programs, each implementing different IP error detection and correction technique, were written in C and compiled with the Keil C51 compiler version 1.52. These programs were simulated on dScope, an 8051-based simulator developed by Keil GmbH. A simulation script

written in dScope script, was used to automatically control and log results of the entire simulation.

The base program itself is a counter, which increments one of its I/O ports once a minute while running an FIR filter at 1kHz. Filter samples are derived from reading the values of random ROM locations and the output written to an I²C E²PROM memory. The I²C protocol is software-based and controlled by the microcontroller itself. The following list describes the differences between the four programs.

Program A – Original version without any form of error checking.

Program B – Implements the “NOP Fill” error detection technique.

Program C – Implements “Function Tokens”.

Program D – Implements both “NOP Fill” and “Function Tokens”.

Figure 5 shows a graphical representation of the code memory segments with the same symbols used to denote different code memory segments. (not proportional in size with only a few Function Tokens represented in Program C and D).

<<<*Figure 5*>>>

where: **E** = Empty code memory locations (when NF is not implemented)

In all, 32 Function Token checks were implemented in programs C and D at function entry and exit points and just before a critical line of code is executed. On detecting an error by any of the FT checks, the FT checking routine vectors program execution to the IPEH, which is situated at the end of the physical code memory. The IPEH, like most other functions, can be located anywhere within the physical code memory with the help of compiler switches and/or assembly directives as the Keil C51 compiler assigns each function its own relocatable memory segment.

Locating the IPEH at the end of the physical code memory allows both Function Tokens and NOP Fill to use the same routine for handling IP errors. Two methods are available to create the NOP Fill, the simpler involves filling the buffer of the PROM/EPROM programmer with 0x00 before downloading the program code. The slightly harder method involves making the NOP Fill part of the downloaded code using assembly-language directives. For embedded systems with In-circuit Serial Programming (ISP) capabilities, the latter method is preferred as there is not a need to physically remove the microcontroller. Doing the "NOP Filling" in assembly also gives much better control over fill regions for different NOP Fill topologies and cuts down on PROM/EPROM programmer configuration steps.

5.1 Calculating SL, UL and DL for test programs

Before applying the relevant equations to calculate the percentage of Safe, Unsafe and Dangerous Locations, the programs' instructions need to be categorised as S, DN, D2, TN, T23 and T3. Such a task was left to an in-house program specifically written to derive the required statistics by processing the downloadable (Intel H86) code. *Table 2* shows the statistics of each program.

<<<*Table 2*>>>

The first thing to note is the difference in program sizes between those implementing Function Tokens (programs C and D) and the rest. The former is roughly 30% larger due to the extra code needed to implement Function Tokens. By taking the percentage of each instruction type, programs C and D are seen to have a higher ratio of multibyte instructions, and thus more MIT locations. An extra piece of code to aid the simulation script detect error is added to the end of programs A and C, which explains their larger code sizes as compared to programs B and D respectively.

The total program size shown in *Table 2* excludes unprogrammed locations, U, and NOP Fills. NOP Fill sizes for programs B and D are 2157 and 1563 bytes respectively. Since programs A and C do not employ NOP Fills, their SL value is 0 and 33 respectively.

Table 3 shows the statistics given in bytes and percentage of physical code memory for each program calculated from the equations given in the previous section.

<<<*Table 3*>>>

5.2 The simulation procedure

As noted in Section 5, this simulation was carried on dScope version 1.51. The simulation procedure is described in this section.

During each simulation cycle, one IP error is introduced, on a pseudo-random basis, between instruction cycle 1 and instruction cycle 10,000,000. The instruction cycle in which the error occurs (EIC), together with the erroneous IP value (EIP), are determined in advance using the simulators' random number generator.

Breakpoints are used to stop program execution when the current instruction cycle is equal or greater than the generated error cycle value, EIC. At the breakpoint, the simulation script automatically changes the IP value to the previously calculated error value, EIP.

If EIP points to an empty code memory location or into a NOP-Fill location, the respective error is logged and a new simulation cycle commences.

If EIP points to a code memory location within the program, the simulator compares this value with a list of all MIT locations. If there is a match then an MIT hit is logged and the simulation script starts a new simulation cycle.

If EIP does not point to any of the above-mentioned categories, the simulator single-steps for up to another 50,000 instruction cycles while checking a number of important program variables. If, during this period, an error is detected by an FT checking routine, then a Function Token hit is

logged and a new simulation cycle is started. When the 50,000 cycle limit is reached – which implies that no errors have been detected – the situation is classified as “Continued” and another simulation cycle is begun.

“E Jumps” happen when the IP vectors to empty code memory locations.

In all, 1000 simulation cycles are carried out in this way, representing 1000 EMI-induced IP errors for each program.

5.3 Simulation results

Simulation was carried out on four Pentium 200MHz PCs running Windows NT 4.0 with the results shown in *Table 4*.

<<<*Table 4*>>>

Program A shows the importance of employing at least some form of error detection. When both techniques are compared, NOP Fills has the higher error detection rate than Function Tokens.

The error detection rate is further increased to nearly 67% $((263 + 402) / 1000 * 100\%)$ when both techniques are implemented. The difference between the “Continued” values for programs A, B and C, D is primarily due to the fact that programs A and B are simpler than programs C and D. In addition, programs C and D also show a rise in MIT since they contain more multibyte instructions.

5.4 Classifying simulation results as SL, UL and DL

In order to compare the simulation and calculated results, the categories shown in *Table 4* has to be classified as in the same manner as that of *Table 3*. Though *Table 4* deals with the hits recorded in each category, they can be classified as SL, UL and DL since the IP has been assumed to have equal probability to hold any addressable value when a corruption occurs.

Therefore, when Program B has 528 hits for NOP Fills, it can be stated as 52.8% of IP errors landed in Safe Locations (NOP Fill locations are SL, for reasons discussed in Section 4.2).

Jumps to empty code memory locations (E Jumps) and MIT hits are classified as DL as they would cause a system reset in the former case, and would cause misinterpretation of instructions in the latter. “Continued” is classified as UL since the processor still manages to continue correct program execution.

The classification of Function Tokens is not as straightforward as it seems (refer to *Figure 6*).

<<<*Figure 6*>>>

In *Figure 6*, the best performance from Function Tokens is obtained at IP entry point (ii), where the IP falls at the first byte of the Function Token check. However, this is rarely the case and scenarios (i) and (iii) predominate. Scenario (iii) is particularly dangerous as execution of the critical code could spell disaster for safety-critical systems. Note that this is despite the fact that the error situation in scenario (iii) would be detected by the next Function Token check. In view of this, it may be concluded that only one FT location (the first byte) is a Safe Location. As a result, with Function Token checks requiring some 20 or more bytes of code, every FT adds (say) 19 UL or DL locations and only one SL. Overall therefore, Function Tokens may detect errors, but do not increase a system’s safety.

With these criteria in mind, *Table 5* shows the simulated results in terms of SL, UL and DL.

<<<*Table 5*>>>

6. Discussion

Table 6 combines the simulated and calculated results shown in *Table 3* and *Table 5* respectively.

<<<*Table 6*>>>

The first thing to note in *Table 6* is the good correlation between the calculated and simulated results for Safe Locations. In the case of UL and DL, the difference between simulated and calculated results is slightly larger, mainly due to the classification method employed.

Please note also that Program B has the greatest number of Safe Locations, though the percentage of detected IP errors is greater for Program D. This apparent anomaly stems from the fact that locations identified by Function Token checks are classified as UL, as discussed in Section 5.4.

<<<*Figure 6*>>>

The findings are summarised in *Equation 7* to *Equation 9*.

Equation 7 shows that the number of Safe Locations is largely governed by the size of the NOP Fill area. Thus, by increasing the size of this area in relation to the program size, the percentage of SLs will increase. *Figure 7* illustrates this relationship by increasing the physical memory size while keeping all other factors unchanged.

<<<*Figure 7*>>>

Figure 7 supports the claims by Coulson and Campbell [3, 4, 5] regarding the effectiveness of NOP Fill techniques.

Turning now to consider Function Tokens, *Figure 8a* to *Figure 8c* show a family of graphs in which the number of Function Token checks are varied.

<<<*Figure 8a*>>>

<<<*Figure 8b*>>>

<<<*Figure 8c*>>>

Figure 8a to *Figure 8c* summarise an important result: the number of Safe Locations in the program *decreases* as the number of FT checks *increases*. In short, the more Function Token checks included in a program, the less safe the program becomes. Note that this result contradicts earlier findings in this area [3, 4, 5, 10, 11].

7. Future Work

The results presented in this paper mark the completion of the first stage of a longer project which is developing techniques intended to minimise the impact of EMI on embedded applications. We briefly describe two of the areas of this future work here.

7.1 Impact on different processor architectures

As discussed in this paper, one way of accessing the likely impact of EMI on a programmable electronics system is to examine the number of dangerous locations (DLs) in the code. The number of DLs is, in turn, linked to the number of multi-byte instructions.

On the basis of these findings, it is tempting to conclude that computer architectures which employ only single-byte (or single-word) instructions will be inherently safer than those employing multi-byte (or multi-word) instructions. However, it is premature to draw such a

general conclusion. Many modern computer architectures employ single-word instructions. They also typically include complex architectural features, such as pipelining, out-of-order execution and branch prediction: such features may, directly or indirectly, have an impact similar to the “Multibyte Instruction Trap” discussed in this paper. Further work is required to clarify these issues.

7.2 Refinements to the current model

The results presented in this paper are based, primarily, on an analysis of processor operations at the processor level. A more detailed analysis of the problem identified here will involve the development of models at the sub-cycle level. Verification of such models is, however, only possible with the aid of sub-cycle level simulators.

8. Conclusions

The results from the studies presented in this paper suggest that the use of NOP Fills should be standard practice in environments where EMI may pose a threat to embedded programmable systems. By contrast, the use of Function Tokens cannot, on the basis of the results presented here, be recommended.

9. References

[1] **Ackermann J**, “*Active steering for better safety, handling and comfort*”, Proceedings of Advances in Vehicle Control and Safety, Amiens, France, pp.1-10, July 1-3, 1998.

[2] **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, pp. 2141-2148, Nov. 1994.

- [3] **Campbell D**, “*Designing for Electromagnetic compatibility with Single-Chip Microcontrollers*”, Motorola Application Note AN1263
- [4] **Campbell D**, “*Defensive Software Programming with Embedded Microcontrollers*”, IEE Colloquium on Electromagnetic Compatibility of Software, Birmingham, UK, Nov 1998 (Conference code 98/471).
- [5] **Coulson DR**, “*EMC Techniques for Microprocessor Software*”, IEE Colloquium on Electromagnetic Compatibility of Software, Birmingham, UK, Nov 98 (Conference code 98/471).
- [6] **Falla M**, “*Advances in Safety-Critical Systems*”, University of Lancaster Press, 1997.
- [7] **Glenewinkel M**, “*System design and layout techniques for noise reduction in MCU-based systems*”, Microprocessor and Microsystems, Vol. 20, No. 5, pp303-309, Sept 1996.
- [8] **Haney PR, Richardson MJ, Clarke NJ, Barber PA**, “*Development of Adaptive Cruise Control Systems for Motor Vehicles*”, Proceedings of Control '98, Swansea, UK, pp. 25-31, 1998.
- [9] **Klemmer G**, “*Shielding approaches*”, Proceedings of the 1996 Regional Technical Conference on Decorating in the Year 2000, Chicago, USA, Sep 30-Oct 1 1996 (Conference code 46649).
- [10] **Niaussat A**, “*Software techniques for improving ST6 EMC performance*”, ST Application Note AN1015/0398
- [11] **Pont MJ, Kureemun R, Ong HLR, Peasgood W**, “*Increasing The Reliability Of embedded Automotive Applications In The Presence Of EMI: A Pilot Study*”, IEE Colloquium on Electromagnetic Compatibility of Software, Birmingham, UK, Nov 98 (Conference code 98/471).

[12] **Wu Q**, “*A study of automobile electromagnetic disturbance or noise source*”, Proceedings of the 1997 International Symposium on Electromagnetic Compatibility, Beijing, China, May 21-23 1997 (Conference code 47002).

[13] “*Design with Microcontroller in noisy environments*”, ST Application Note AN435/0894

[14] “*MISRA Report 3: Noise, EMC and Real-Time*”, The Motor Industry Research Association, UK, February 1995.

[15] “*NHTSA Light Vehicle Antilock Brake System Research Program Task 4*”, National Highway Traffic Safety Administration (NHTSA) Report, US, Jan. 1999.

Figure 1: A schematic representation of “Memory Aliasing”

Figure 2: Example memory allocation with NOP Fills

Figure 3: A schematic representation of a program with Function Tokens

Figure 4: A schematic representation of a program implementing both Function Tokens and NOP Fills

Figure 5: A schematic representation of the code memory organisation for Program A, B, C and D. See text for details.

Figure 6: A schematic representation of error handling with Function Tokens. See text for details.

Figure 7: The results of changes to physical code memory size (NOP Fills).

Figure 8: The results of changes to physical code memory size and the number of Function Token checks.

Table 1: Instruction classification. See text for details.

Table 2: Instruction breakdown for programs A, B, C and D.

Table 3: Calculated SL, UL and DL values for programs A, B, C and D.

Table 4: Results of 1000 error simulation cycles.

Table 5: Classified simulation results.

Table 6: Comparison between simulated and calculated results.

Listing 1: Original assembly-language instructions.

Listing 2: The effects of IP corruption on Listing 1.

Equation 1

Equation 2

Equation 3

Equation 4

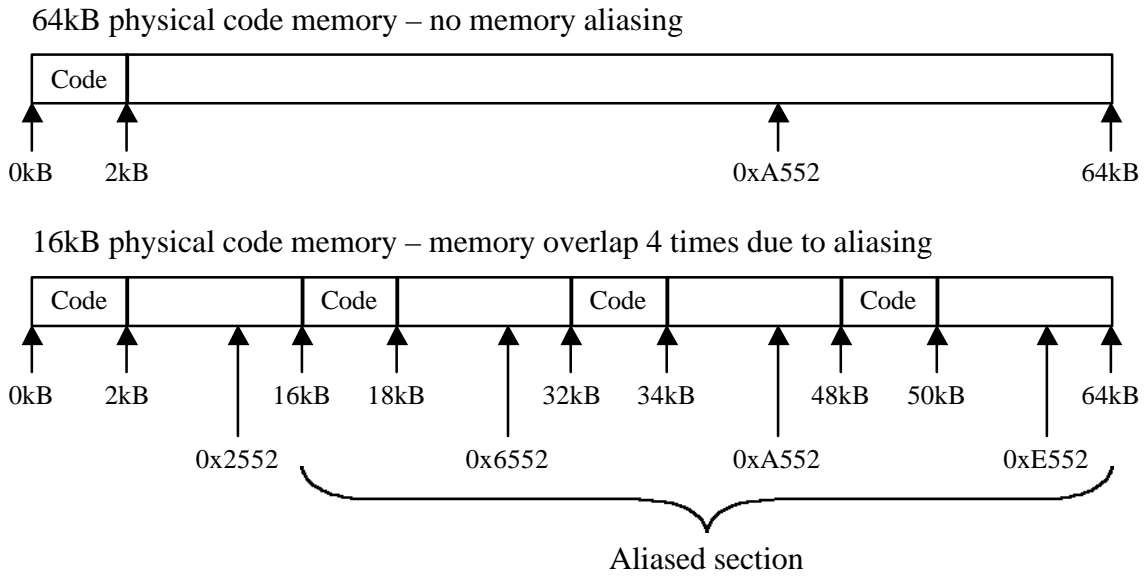
Equation 5

Equation 6

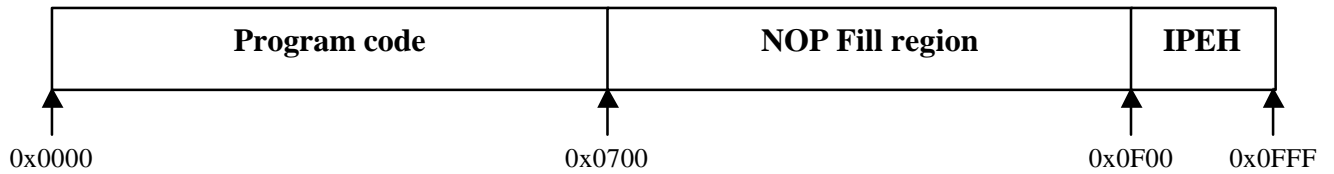
Equation 7

Equation 8

Equation 9

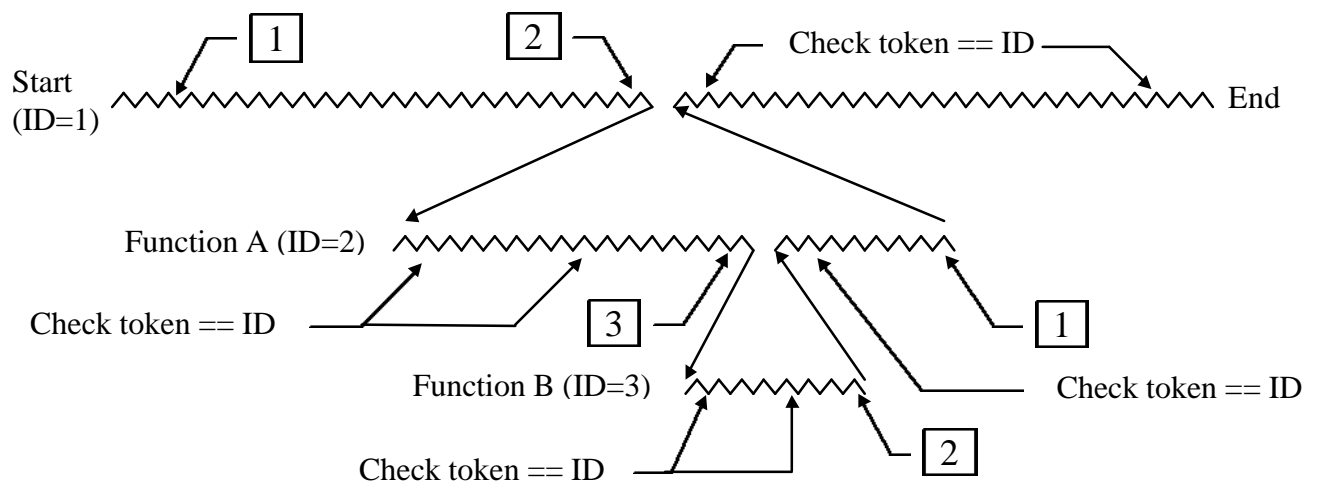


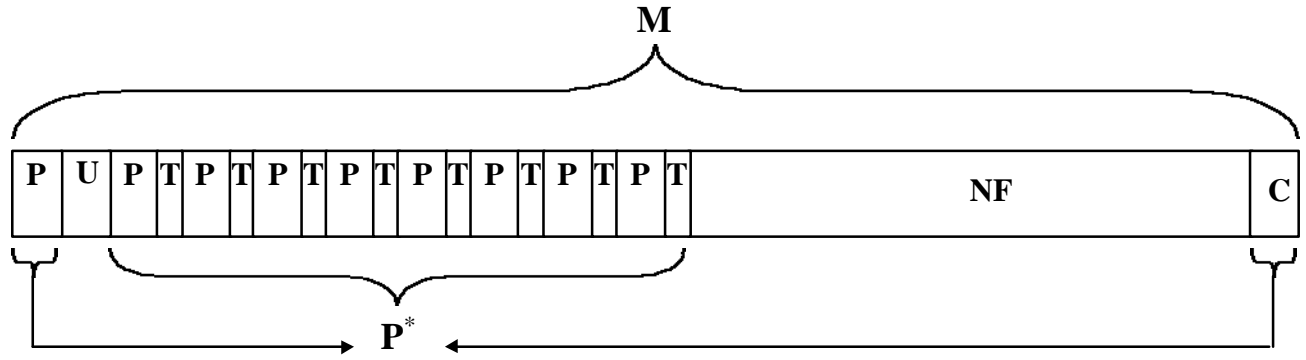
<<<Figure 1>>>



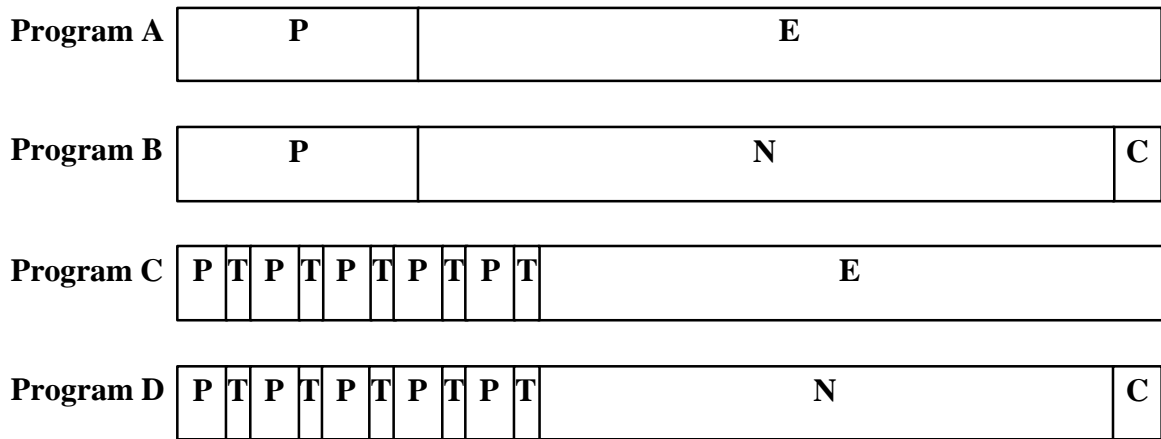
<<<Figure 2>>>

<<<Figure 3>>>



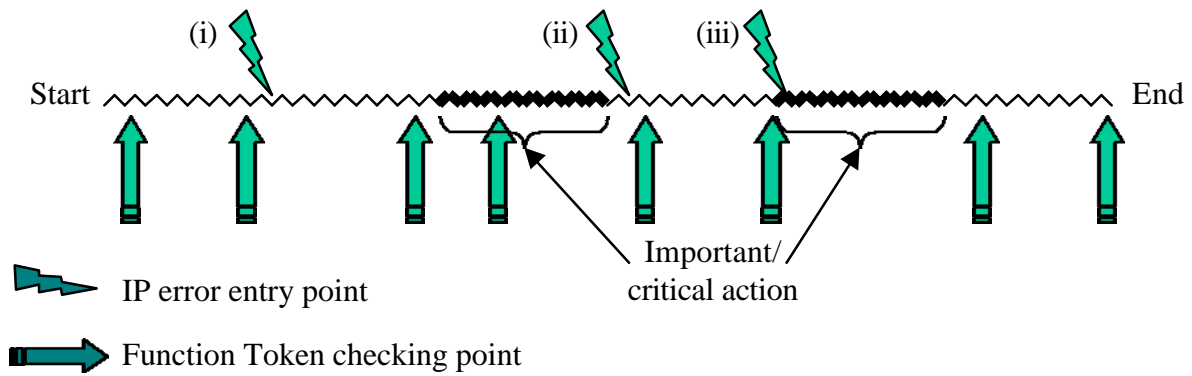


<<<Figure 4>>>

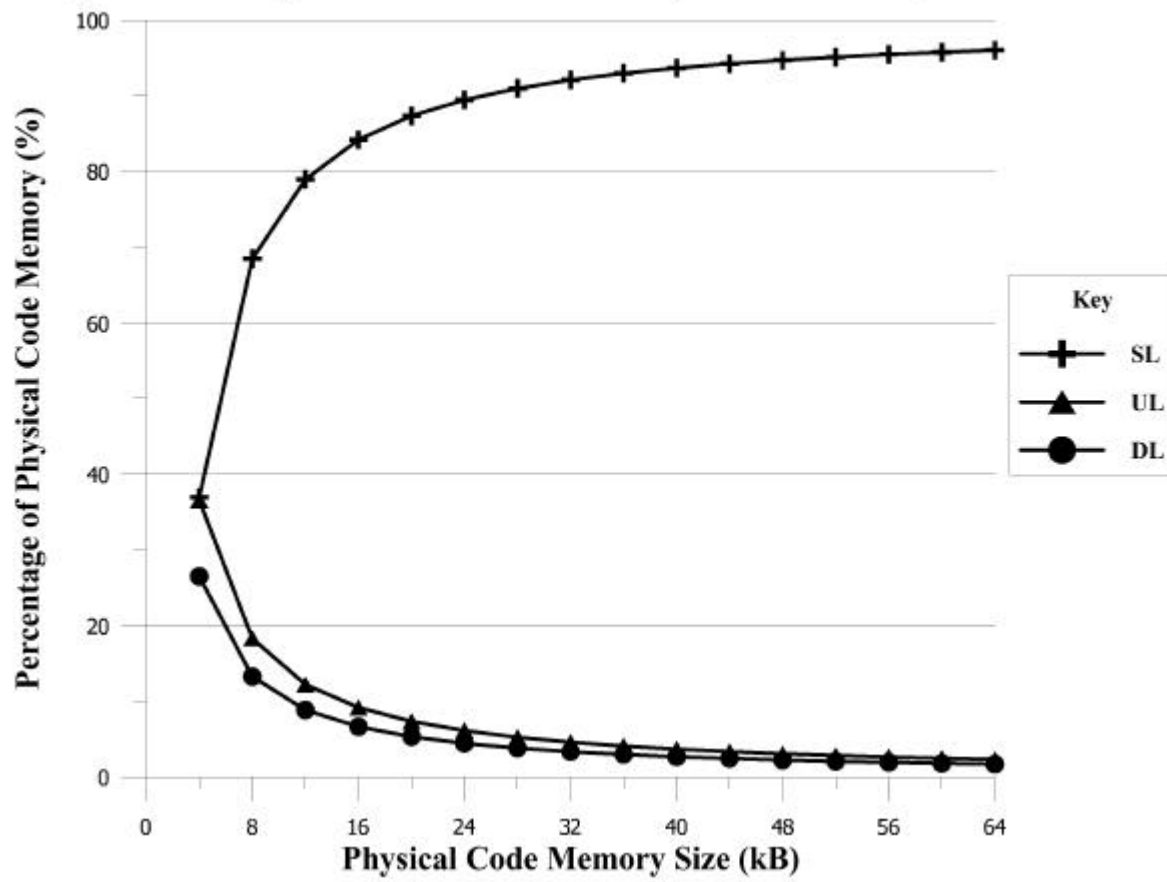


<<<Figure 5>>>

<<<Figure 6>>>

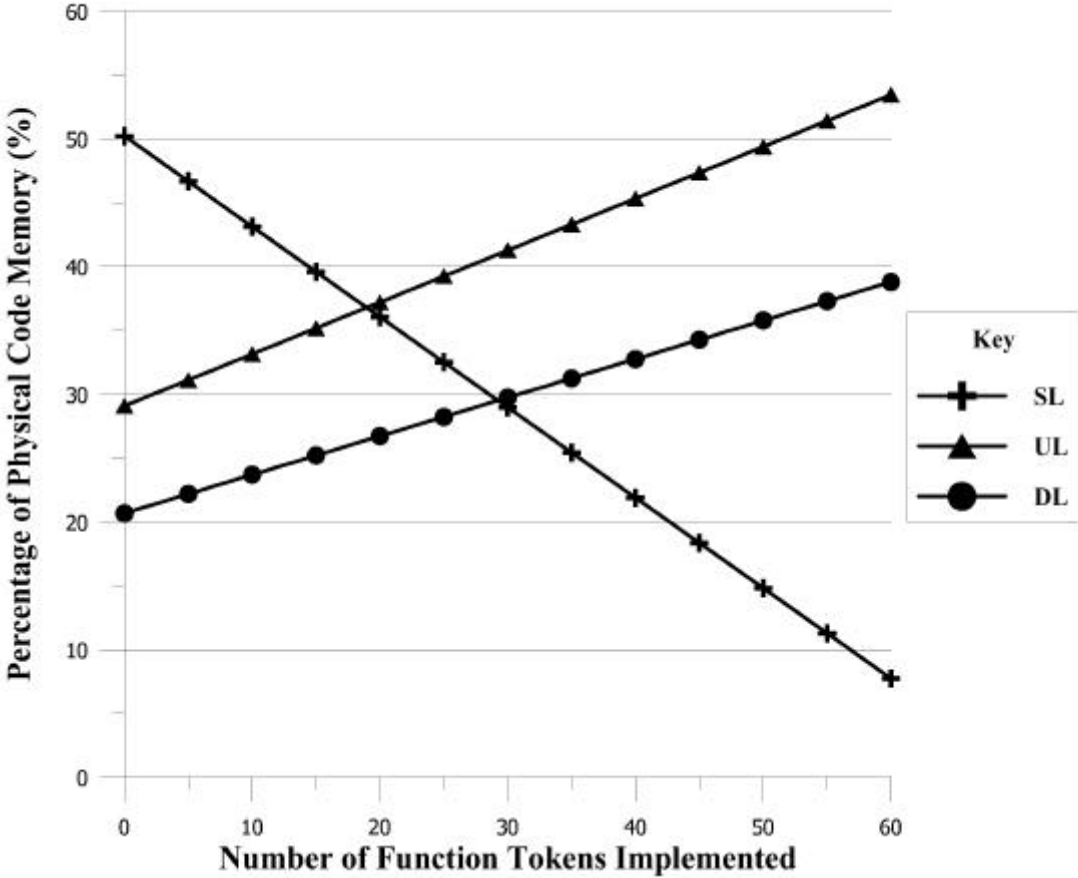


Impact of Physical Code Memory Size on SL, UL and DL



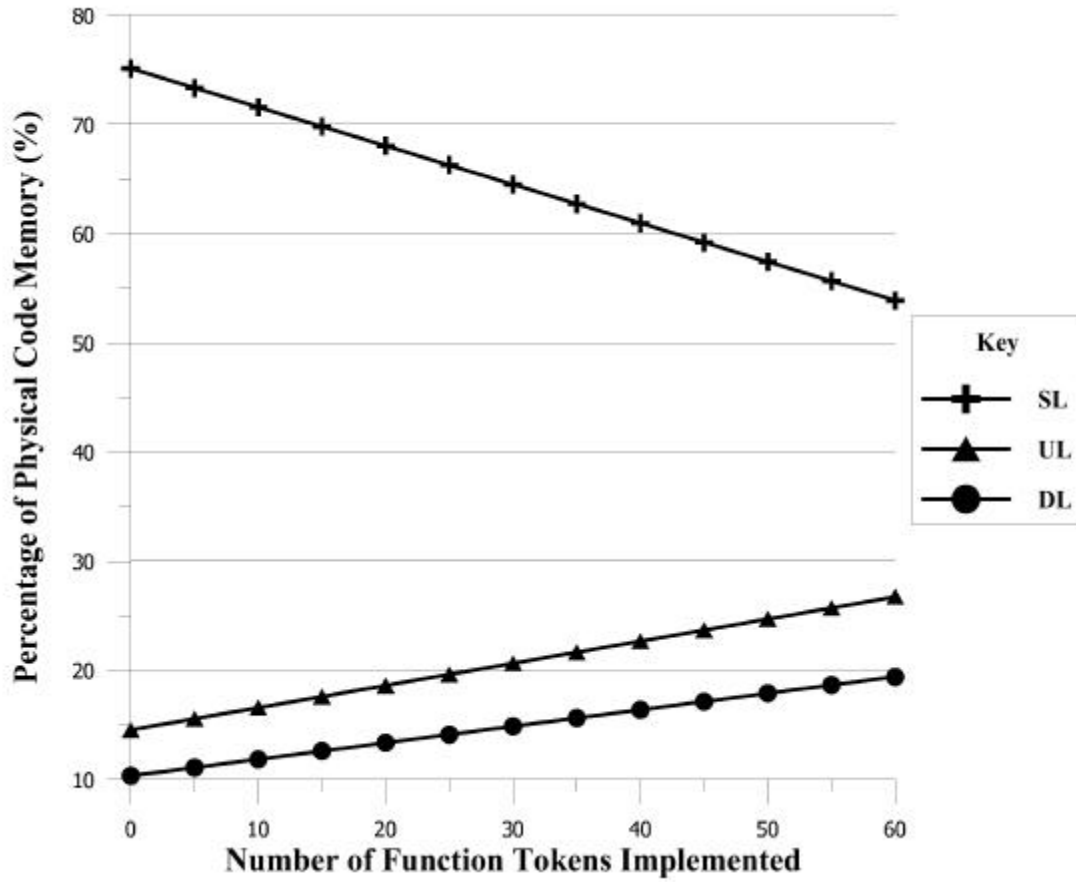
<<<Figure 7>>>

Impact of Function Token Checks on SL, UL and DL (4kB)



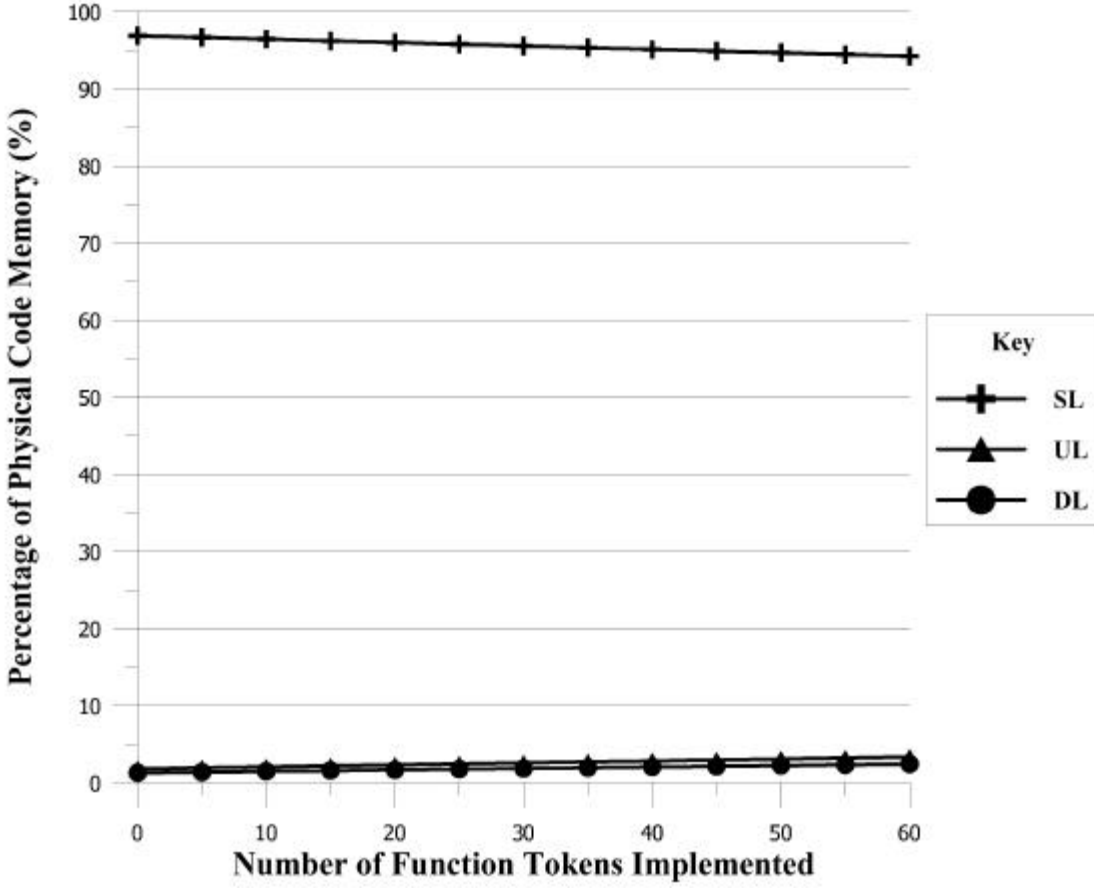
<<<Figure 8a>>>

Impact of Function Token Checks on SL, UL and DL (8kB)



<<<Figure 8b>>>

Impact of Function Token Checks on SL, UL and DL (64kB)



<<<Figure 8c>>>

```
0100 759850 MOV    98H,#050H
0103 438920 ORL    89H,#020H
0106 758DFD MOV    8DH,#0FDH
```

<<<Listing 1>>>

```
0101 98      SUBB  A,R0
0102 5043    JNC   #43H
0104 8920    MOV   20H,R1
0106 758DFD MOV   8DH,#0FDH
```

<<<Listing 2>>>

$$P^* = P + (\quad \times n) + (\quad \times m)$$

<<<Equation 1>>>

$$NF = M - P^* - U$$

<<<Equation 2>>>

$$SL = NF + m + n$$

<<<Equation 3>>>

$$UL = B_S + (B_{DN}/2) + B_{D2} + (B_{TN}/3) + B_{T23} + (B_{T3} \times 2/3) - m - n$$

<<<Equation 4>>>

$$DL = (B_{DN}/2) + (B_{TN} \times 2/3) + (B_{T3}/3)$$

<<<Equation 5>>>

$$DL = M - SL - UL$$

<<<Equation 6>>>

$$SL_{(\text{percentage})} = SL_{(\text{bytes})} / M$$

<<<Equation 7>>>

$$UL_{(\text{percentage})} = UL_{(\text{bytes})} / M$$

<<<Equation 8>>>

$$DL_{(\text{percentage})} = DL_{(\text{bytes})} / M$$

<<<Equation 9>>>

Instruction value (Hex)			Classification			Instruction	
Byte 1	Byte 2	Byte 3	Byte 1	Byte 2	Byte 3	Size	Type
XX	-	-	UL	-	-	1	S
XX	00	-	UL	UL	-	2	D2
XX	XX	-	UL	DL	-	2	DN
XX	00	00	UL	UL	UL	3	T23
XX	00	XX	UL	DL	DL	3	T2 (TN)
XX	XX	00	UL	DL	UL	3	T3
XX	XX	XX	UL	DL	DL	3	TN

<<<Table 1>>>

Program	Total Size	Single (bytes)	Double (bytes)		Triple (bytes)		
			DN	D2	TN	T23	T3
A	2017	840	818	80	258	0	21
B	1931	825	790	64	228	0	24
C	2635	941	1156	130	366	0	42
D	2520	938	1112	110	321	0	39

<<<Table 2>>>

Program	SL		UL		DL	
	(bytes)	(%)	(bytes)	(%)	(bytes)	(%)
A	0	0.00	1429	34.89	2667	65.11

B	2158	52.69	1375	33.57	563	13.75
C	33	0.81	1766	43.11	2297	56.08
D	1596	38.96	1704	41.60	796	19.43

<<<Table 3>>>

Program	Recovered Errors		Undetected Error			Total 10. E rr or s
	FT	NF	E Jump	MIT	Continued	
A	0	0	474	165	361	1000
B	0	528	0	157	315	1000
C	313	0	346	237	104	1000
D	263	402	0	236	99	1000

<<<Table 4>>>

Program	SL		UL		DL	
	(hit)	(%)	(hit)	(%)	(hit)	(%)
A	0	0.0	361	36.1	639	63.9
B	528	52.8	315	31.5	157	15.7
C	0	0.0	417	41.7	583	58.3
D	402	40.2	362	36.2	236	23.6

<<<Table 5>>>

Program	SL (% of code)		UL (% of code)		DL (% of code)	
	Cal.	Sim.	Cal.	Sim.	Cal.	Sim.
A	0.00	0.00	34.89	36.10	65.11	63.90
B	52.69	52.80	33.57	31.50	13.75	15.70
C	0.81	0.00	43.11	41.70	56.08	58.30
D	38.96	40.20	41.60	36.20	19.43	23.60

<<<Table 6>>>

¹ An alternative means of determining P^* is to note the size of the downloaded program either from the compiler's summary log, or from the feedback generated when the downloadable program is read into the buffer of a PROM/EPROM programmer.