Paper Preprint – To Appear In:
Proceedings of the 6th IEEE AICSSA Conf. 2008

# Development Guidelines for Dependable Real-Time Embedded Systems

Michael Short
*Embedded Systems Laboratory, University of Leicester, Leicester, UK.*
*mjs61@leicester.ac.uk*

## Abstract

*Embedded control systems play an increasing role in many safety critical system designs. The correct and dependable implementation of such systems depends on many factors, including the design of system hardware, software and fault tolerance mechanisms, the choice of programming language, and also the testing, verification and validation techniques employed. In this paper, a set of guidelines for the development of dependable embedded systems is presented. Although the paper is primarily concerned with single-processor applications, extensions to multi-processor systems are discussed where appropriate. Although the creation of dependable embedded systems cannot simply rely on the enforcement of several such rules or guidelines, experience gained from several years' experience of teaching, research and development in these areas indicates that adherence to a small, but workable, set of rules and guidelines can avoid many of the traps and pitfalls commonly encountered in the creation of dependable embedded systems.*

## 1. Introduction

Modern control systems are almost invariably implemented using some form of embedded digital computer system. 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 dependable, in that it is both reliable, timely and functionally safe [1-7]. Special measures must therefore be taken at all stages of the design process to ensure that the required Safety Integrity Level (SIL) has been achieved. The SIL of a system depends on the consequences of system failures, which can be determined using risk assessment; a required dangerous failure rate $\lambda_d$ is then assigned for a system based on this risk. Demonstrating that the dangerous failure rate for a system is at a specific level requires many factors to be taken into consideration; a major element in this process is the determination of reliability, safety, security and availability measures for each sub-system and component as part of a safety case.

Prospective designers of embedded systems have many factors to consider – for example the choice of hardware / software architecture, programming language and communications network - and many of these decisions are known (or thought) to influence both the performance and dependability of the resulting system (e.g. [3][4][5][6]). Some of the main factors to be considered are illustrated in Figure 1. This paper is specifically concerned with general design guidelines and programming / testing techniques for use in creating dependable real-time embedded systems, programmed using a sub-set of the ANSI C language; embedded C has become the choice of many developers in these areas in recent years [4][6][9].

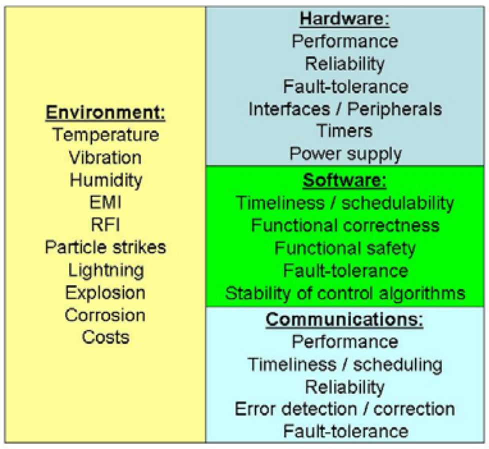

**Figure 1. Dependable system design factors**

Most serous system developers use some form of design and coding guidelines, which define a set of ground rules for the overall system design and the software to be written. In this paper, a small – but highly workable – set of rules and guidelines is presented. Although the creation of such systems cannot simply rely on the enforcement of several such rules or guidelines, experience gained from several years' experience of teaching, research and development in these areas indicates that adherence to a small, but workable, set of rules can avoid many of the traps and pitfalls commonly encountered in the creation of dependable embedded systems.

The rules and guidelines presented are divided in four broad categories, and cover hardware design, software design, programming guidelines and testing techniques. In all cases the rules have been formulated to be consistent with much previous work in this area. For example, the rules are consistent with the MISRA coding guidelines [3][4] and IEC 61508 [11], and also many previous works in this area – for example [1][2][5][6][8]. However, the discussion presented here is intended to compliment these existing works, identify the common best practice within them, and focus their content from a real-time embedded perspective.

The remainder of the paper is organized as follows. In Section II, general hardware considerations for real-time embedded systems are discussed. In Section III, recommendations are made for the software architecture of the system. Section IV presents a summary of what is considered to be the main recommendations and rules applicable to the software design process. In Section V, recommendations regarding testing, verification and validation are presented. Finally in Section VI, the paper is concluded.

## 2. Hardware considerations

From the hardware perspective, designers have the choice of a number of different key system elements:

- Microprocessor;
- Memory;
- Oscillator circuit;
- Watchdog timer;
- Off-chip peripherals;
- Communication devices.

A wealth of useful information regarding these choices is discussed by [9]. In summary, the choice hardware elements is dictated by several factors, including the complexity and timing constraints of the software functionality, required number of I/O lines, costs, and target environment. A sensible choice of processor would be to select a device with several years' proven deployment, which is not likely to become discontinued in the immediate future. For systems with a high SIL, then the use of specialized processor architectures is recommended. When using general purpose processors, designers must be aware that special measures must be implemented to protect against transient effects such as EMI and particle strikes. Additionally, the processor may have undocumented test modes that may be inadvertently triggered during normal operation. At the present time, the 8-bit 8051 architecture, 16-bit (X)C167 architecture and the 32-bit ARM-7 architectures (and close variants) are good choices of general-purpose processors in a range of different performance levels.

Choice of RAM memory will typically be dictated by the nature of software, with data-intensive systems obviously requiring larger memory requirements. However it should be remembered that when coding a system, the smaller the memory footprint, the less likely the system is to suffer a data error. However, there is a trade-off; in many cases sacrificing RAM usage will be at the expense of code execution time and ROM usage, which may affect the timeliness of the system, to be discussed in the next section.

For a real-time system, the performance – and hence timeliness - of the system is highly dependant on the availability of an accurate clock (hardware timer). Since such timers are almost totally reliant upon the oscillator, then care must be taken in this aspect. The use of a crystal oscillator over alternates (e.g. ceramic resonator) is recommended for stability reasons. For systems that are to be subjected to high temperature fluctuations or excessive vibration, then a temperature and/or vibration compensated oscillator circuit should be considered. An oscillator watchdog circuit, in conjunction with a brownout/reset circuit, should be considered mandatory [1][2][3][7][9].

The use of a watchdog timer as a primary guard against transient faults should also be considered mandatory. When a general-purpose processor is to be employed, then an external watchdog should be considered; as mentioned, such devices may have undocumented test modes that may be inadvertently triggered during normal operation, in which the watchdog timer is also disabled. If an external device is used, this condition is prevented by design.

Where possible, the use of on-chip peripherals for performing functionality such as ADC conversion is recommended, mainly to reduce the complexity of the system; and also to reduce the number of components and IC's in the design, which in general will decrease the overall hardware failure rate. However, when cost or performance constraints dictate that additional components must be employed, then the use of field-proven, high-reliability devices is recommended.

In multiprocessor systems, some form of shared communications network is employed to allow the devices to communicate. In these systems, care must be taken to provide fault tolerance in this medium, as it provides a single point of failure. In many safety-related protocols this fault-tolerance is pre-designed into the network in the form of bus-guardians, redundant cabling media, and fault-tolerant clock synchronization. However, if a more general purpose network is to be employed – such as the CAN protocol [10] or the RS-485 standard – then special considerations need to be taken to ensure fault-tolerance. Further details may be found in [9][11][12].

In all cases, determination of the overall reliability of the components in the system should be performed - using suitable analysis techniques [1][13] - to ensure that the failure rate is commensurate with the SIL of the system.

## 3. Software considerations

### 3.1 Design paradigm

From the software perspective, designers may choose to implement a system based around four (highly inter-related) categories [14]:

- Time-triggered;
- Event-triggered;
- Preemptive;
- Non-preemptive (co-operative).

From the perspective of the dependable systems designer, a wealth of previous research has argued that the use of event-triggered systems should, in the main, be avoided where possible. Instead, the use of time-triggered techniques for both communications and task scheduling should be considered, as this is known to improve the system predictability considerably [9][15][16][17][18].

From the time-triggered perspective, it is common to represent a real-time system as a number of communicating tasks, where each task *t* in the task set $\tau$ is represented by a four-tuple:

$$t_i = \left(p_i, c_i, d_i, r_i\right) \tag{1}$$

In which $p_i$ is the task period, $c_i$ is the worst case computation time of the task, $d_i$ is the task deadline and $r_i$ is the task release time, represented as whole integers in suitable processor time units, for example microseconds. With such a representation, it is normal to assume the following:

$$p_i = d_i > r_i \geq 0, 0 < c_i < p_i \tag{2}$$

i.e. with the task deadline equal to its period and release time and computation time less than the period. Determination of the worst case computation time *c* of each task has been well covered in the literature, and can be obtained by a variety of methods loosely termed 'code profiling'. When the task computation times have been determined, appropriate scheduling analysis can be performed. This analysis is essentially a decision procedure giving a yes/no answer as to whether the schedule is feasible or not. In the case of the co-operative system, tasks must run to completion once dispatched. When the release times in the task set $\tau$ are all equal, schedulability is decided by the following equation [14]:

$$\sum_{i=1}^{i=n} c_i \leq \mathrm{gcd}(p_1, \ldots, p_n) \tag{3}$$

Where *gcd* is the greatest common divisor of the task set periods, i.e. the 'tick' interval of the scheduler. In the pre-emptive case, tasks may interrupt each other's execution based on their assigned priorities. When the priorities are assigned inversely proportional to the task periods (which is known to be an optimal priority assignment), schedulability is decided as follows [14]:

$$\sum_{i=1}^{i=n} \frac{c_i}{p_i} \leq n(\sqrt[n]{2} - 1) \tag{4}$$

Again, with all release times assumed equal. At first glance it would seem then that a greater CPU utilization may be achieved when implementing a system pre-emptively. However, these analyses do not include scheduler overhead – which are vastly increased in the latter scheduler – and do not include blocking terms introduced by resource conflicts and deadlock avoidance mechanisms associated with the latter. A fuller analysis that includes the influence of these mechanisms is given by Katcher et al. [19].

In both cases, when the release times of the tasks are free to be chosen by the designer (a so-called ‘offset-free’ task set), then the achievable CPU utilization may be increased (or decreased) by a large factor, depending on the choice of release times. However, the complexity problems of deciding schedulability and assigning effective release times to tasks increases dramatically with such systems, and is known to be an NP-Complete problem [14][20]. Details of recent advances in this area, producing (in most cases) tractable solutions, may be found in [21].

In addition, it has also been argued that co-operative systems are both easier to inspect and verify [22] and exhibit greater tolerance to transient disturbances [23] than equivalent preemptive systems. Based on these discussions, it is therefore recommended to use co-operative scheduling when developing safety-related systems. A suitable design for a portable, efficient scheduler coded in embedded C is given in [9]. Although this paper is primarily concerned with single-processor designs, this basic scheduler methodology can be extended to multiprocessor designs (a ‘shared-clock’ scheduler), and can be implemented over multiple, redundant communication networks to great effect [12].

## 3.2 Runtime behavior

In addition to the avocation of time-triggered, co-operative scheduling, it is recommended that the system possesses a minimum number of run-time mechanisms to mitigate the effects of transient errors. These mechanisms should include the use of a watchdog timer, duplex duplication of critical data with comparison, sanity checks of control signals, mechanisms to detect task overrun, and the enabling of all on-chip exception traps in the target processor. Such traps will normally consist of most of the following elements:

- Stack overflow;
- Stack underflow;
- Illegal operand;
- Illegal word access;
- Protected instruction fault;
- Divide by zero;
- Illegal bus access.

In addition, all unused areas of ROM and RAM memory should be filled with (or initialized to) illegal operands to provide added control flow error detection. On activation of any of these traps, a full system reset of the microcontroller should be forced. On system boot-up/reset, the microcontroller should perform – at minimum - the following software-based self-tests [24]:

- Internal RAM/register/stack validation;
- External RAM validation;
- ROM checksum;
- Peripheral test (e.g. ports, timer).

If any of these tests are failed, the microcontroller must activate any appropriate warning signals and then enter a safe state. The overall recommended approach to software fault-tolerance is illustrated in Figure 2. The simple techniques presented here, although somewhat dependant on the implementation hardware, typically allow for transient error coverage in excess of 95% [9][23][25]. Although some researchers have advocated the use of specialized (software-based) transient error detection mechanisms, in general it is recommended that such techniques are avoided. This is because such techniques increase the system complexity considerably and oftentimes require the use of automatic code generators for their implementation, which may sometimes be problematic from a safety perspective and may cause problems with certification. Although the use of such code generators has many potential benefits, to date few such generators have been certified to the appropriate levels, and – at the present time - their use should proceed with caution [1][3][4].

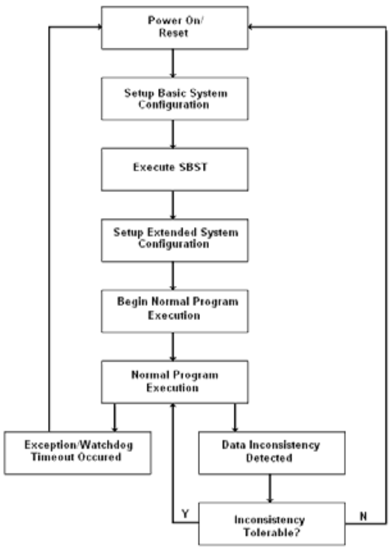
Power On/ Reset
Setup Basic System Configuration
Execute SBST
Setup Extended System Configuration
Begin Normal Program Execution
Normal Program Execution
Exception/Watchdog Timeout Occured
Data Inconsistency Detected
Inconsistency Tolerable?
Y
N

**Figure 2. Software fault-tolerance**

## 4. Code development considerations

In this section, attention is now focused on guidelines and recommendations for the use of the C language in such systems, and a basic set of guidelines for software development in embedded C is presented. Although defining an exhaustive list of such guidelines is beyond the scope of this paper, the interested reader is referred to [1-8]. In essence, the list that is presented aims to employ only the well defined features of the C language in order to create bounded, predicable, readable and maintainable code.

### 4.1 Development Tools and Processes

The choice of compiler may have a great influence on the quality of the resulting code. Typically a well-supported compiler with a favorable, documented performance level should be chosen. A development team should have an ‘in-house’ coding style guide which is strictly adhered to, and when appropriate, version control software should be employed in addition to (and integrated with) the in-house documentation and release control procedures.

Code must always be compiled with all warnings set to the maximum levels, and production code must compile without a single such warning. The use of the preprocessor should primarily be limited to simple *#include* statements, *#define* constants, syntactically meaningful macro definitions, and header guards – which - aside from the closing *#endif* of a header guard - are to be placed at the top of a file only. Conditional compilation use must be limited and never span multiple files. Language extensions – e.g. inline assembly – must be fully encapsulated and isolated.

The use of ‘lightweight’ static checking and source code metrics tools should be a regular part of the development process; the use of ‘heavyweight’ formal methods tools should be considered mandatory for systems with a high SIL. This will be discussed further in Section V.

### 4.2 Program Flow and Looping

The call graph of an embedded control system program should ideally be acyclic, and the potential number of loops and iterations performed by any *single* execution of any given system task should always be bounded. This is required to prevent runaway code and task overruns. In practice this means that several main points should be adhered to in terms of control flow and looping.

Only the simple, compound control flow statements provided by the C language should be used – the use of *goto, setjmp* or *longjmp* constructs is prohibited. The use of either direct or indirect recursion (functions or chains of functions which can call themselves) should be avoided - use iterative functions instead.

Inside each task, all loops that rely directly on non-deterministic input - or the results of complex data manipulation - for termination must have a fixed upper bound (loop timeout). The three expressions in a *for* loop must only be used for loop control, and numeric variables used within a *for* loop for iteration control must not be modified in the body of the loop. At most a single *break* statement may be used in any single loop. Finally, the controlling expressions used in *for* or *while* loops should not, either directly or indirectly, use any floating point types.

### 4.3 Functions

Obviously, functions are an integral part of all embedded C programs, and their correct and consistent use is required to produce reliable code. Correct use of functions will typically include the following elements.

All functions should have a prototype. Ideally the identifier names as well as types of any arguments should be provided in this prototype. Functions should automatically be declared at file scope (*static*), unless they require external linkage. Functions shall not use variable (*variadic*) argument lists.

Where possible, parameters passed to a function should be tested for validity, and if a function returns error information, this information should be tested. The use of the standard C *errno* facility should be avoided. Where the return type of a function is non-void, all possible return paths should include an explicit non-void return statement. Where possible, there should only be a single point of return at the end of the function.

### 4.4 Memory and Variables

As with functions, variables and data - and their management in memory - are all integral parts of an embedded C program. However, their correct and consistent usage is required to produce reliable, readable code, and to ensure that testing can proceed efficiently. In particular, the following elements should be observed.

All data objects must be declared at the lowest possible level of scope, with appropriate linkage (where required). All variables must be initialized before usage. Variables must not be reused for multiple, incompatible purposes, and variables in a local scope shall not use the same identifier as variables with a global or file scope.

Programs must avoid dynamic data allocation, and avoid the use of (built-in) library functions that make use of dynamic memory allocation (e.g. *string.h*).

Additionally, the use of pointers is riddled with potential traps and pitfalls - even experienced programmers can become easily confused. Therefore, limit the use of pointer indirection to a maximum of two levels; ideally one in most circumstances. Pointer arithmetic must be limited to array indexing only. Uninitialised pointers must never be used, and where appropriate checked that they are not NULL before use. The address of an object should never be assigned to a pointer with a larger scope. And finally, restrict the use of function pointers to scheduler implementations only.

### 4.5 Libraries

The creation, use and re-use of code libraries are an essential part of most software engineering processes. It makes no sense to re-engineer solutions to problems that have been well tackled in the past. However, there are a number of key points to adhere to in this respect. These include the following main elements.

All libraries - with available source - shall be coded in accordance with the 'in-house' coding style, and shall adhere (where applicable) to strict ANSI guidelines. For libraries without available source code (e.g., from third party vendors), they must have appropriately documented, tested and verified behavior. When libraries are reused from previous projects, appropriate testing and verification should be performed to confirm that the code performs as expected *in the new context*.

Reserved macro's, functions and identifiers contained in standard libraries must not be redefined, undefined or reused in any way. Arguments to, and return values from, library functions must be checked for validity. Care must be taken when using functions from the standard libraries: a list of unsuitable features is as follows: *stdio.h, time.h, setjmp.h, errno.h*: do not use these libraries at all. The following functions and macro's in stdlib.h and stddef.h must not be used: *offsetof, abort, exit, getenv, setenv, system, atof, atoi, atol, calloc, malloc, realloc* and *free*. In the following section, attention is now turned to testing, verification and validation.

## 5. Testing, verification and validation

The role of testing, in any software development process, is to find and remove software defects ('bugs') through comparisons of the expected system behavior (specification) with its actual behavior. When developing dependable systems it is often inappropriate, unethical or even impossible to fully test the system within its natural operational environment [1][2][25]. In such cases, HIL simulation of the system's environment can enable developers to make assessments of performance without compromising safety. The principle of HIL simulation of an embedded control system is illustrated in Figure 3.

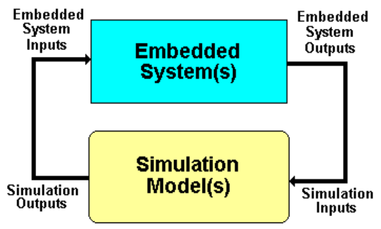

**Figure 3. HIL simulation principle**

The embedded system outputs are fed directly to the simulation, where they are sampled and used as input variables. Previous papers have discussed applications of HIL simulation to dependable system testing (e.g. [23][25]); its use is therefore highly recommended. Typically, software verification conditions – in the form of pre and post conditions implemented as sanity checks - should be the primary means to detect anomalous conditions during testing. The form of a typical C sanity check is shown in Figure 4, where __FILE__ and __LINE__ are macro's inserted by the preprocessor at compile time. The sanity check density should be a minimum of two per function [1][2][4][6].

```
#define sanity_check(c) ((c) ? (true) : \
debug(“%s, %d: sanity check ‘%s’ failed”, \
__FILE__, __LINE__, #c), false)
```

**Figure 4. Sanity check C macro**

However in software-based systems designed to have a failure rate less than $10^{-4}$, such as those considered in this paper, testing alone is impractical (on any reasonable timescale) and alternate means for complete verification must be considered [26]. This generally implies that the use of formal methods should be introduced. Such techniques employ mathematical methods to determine whether software meets its specifications, and provide documentary evidence that it is free from defects. Of the available formal methods that may be employed in a specific situation, formal model checking has proven to be a highly popular and successful technique which is well suited to many critical embedded applications, and integrates well to existing development and testing processes [27].

### 5.1 Formal Model Checking

The majority of model checking algorithms use Kripke structures to formally represent the computations of a finite-state system. Kripke structures are essentially graphs in which nodes represent system states and edges represent possible transitions between states. A model checking algorithm is an automated algorithm that decides whether a given Kripke structure $P$ is a model of (satisfies) the specification $\psi$. In practice a number of extremely efficient bounded model checkers for the C programming language have been developed (for example [28]); for this reason, formal verification – in conjunction to exhaustive testing – is recommended.

If these techniques are correctly employed, then – in conjunction with exhaustive testing - it is possible to prove that *all* sanity checks will *always* be passed under normal operating conditions. In this case, the sanity check macro given in Figure 4 may then be easily modified to reset the processor (as opposed to outputting a debug message) upon detection of an anomaly, as shown in Figure 5. Reset is simply a function (or macro) to reset the processor, either by issuing a software reset signal or, for example, by disabling interrupts and entering an infinite loop to trigger the watchdog. This provides an additional, low cost and highly effective method to detect and recover from transient failures in production code.

```
#define sanity_check(c) ((c) ? (true) : \
        Reset(), false)
```

**Figure 5. Run-time sanity check**

### 5.2 Fault Injection Testing

In most cases, dependable systems will be implemented using some form of fault tolerance [9][11]. In such cases, fault injection is the preferred means for extracting dependability information as part of system validation exercises [29]. Additionally, one extremely promising area of research in this area has concentrated on the use of fault-injection, in combination with rare events techniques and automated performance monitoring, to extract fault coverage information for the calibration of dependability models [25]. This technique, although experimental, is also recommended.

## 6. Conclusions

In this paper, a set of development guidelines has been proposed for prospective designers of dependable embedded systems, covering basic hardware and software foundations, programming guidelines and recommendations for software testing and verification. Embedded systems developed around such principles are typically capable of achieving an extremely high level of safety integrity. In conclusion, although the guidelines are not meant to be strictly enforced, it is hoped that they provide a sensible set of working rules to assist in future development of safety-related embedded systems.

## 7. References

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

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

[3] MISRA, “Development guidelines for vehicle-based software,” Motor Industry Software Reliability Report, November 1994.

[4] MISRA. (2004). Guidelines for the use of the C language in vehicle based software. Motor Industry Software Reliability Report, Released October 2004.

[5] SAE, "Class C Application Requirement Considerations," SAE Recommended Practice J2056/1, SAE, June 1993.

[6] Holzmann, G.J. (2006). The Power of Ten: Rules for Developing Safety Critical Code. IEEE Computer, Vol. 39, No. 6, pp. 93-95, June 2006.

[7] IEC 61508 - Functional safety of electrical/electronic/ programmable electronic safety-related systems, Part 3. (2000).

[8] Maguire, S. Writing solid code. Microsoft Press, 1993.

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

[10] Bosch, CAN Specification Version 2.0, Robert Bosch GmbH, 1991.

[11] Iserman, R., Schwarz, R. and Stoltz, S. (2002). Fault-tolerant drive-By-Wire Systems, IEEE Control Systems Magazine, Vol. 22, No. 5, pp. 64-81, 2002.

[12] Short, M. and Pont, M.J. (2007). Fault-tolerant, time-triggered communication using CAN. IEEE Transactions on Industrial Informatics, Vol. 3, No. 2, pp. 131-142.

[13] MIL-HDBK-217F. Military Handbook - Reliability Prediction of Electronic Equipment. Department of Defence, Washington DC, 1990.

[14] Buttazo, G. (1997). Hard real-time computing systems: predictable scheduling algorithms and applications. Kluwer Publishers, Norwell, MA., 1997.

[15] A. Albert, “Comparison of event-triggered and time-triggered concepts with regard to distributed control systems,” in Proceedings of Embedded World, Nurnberg, Germany, 17-19 Feb, pp. 235-252, 2004.

[16] I.J. Bate, “Scheduling and timing analysis for safety critical real-time systems,” PhD. dissertation, Department of Computer Science, University of York, November 1998.

[17] Kopetz, H. (1991). Event-Triggered Versus Time-Triggered Real-Time Systems. Lecture Notes in Computer Science, Vol 563, pp 87-101. Springer-Verlag, Berlin/New York, 1991.

[18] Lonn, H., and Axelsson, J. (1999). A comparison of fixed-priority and static cyclic scheduling for distributed automotive control applications. In: Proc. 11th Euromicro Conf. on Real-Time Systems, York, UK, 9–11 June 1999, pp. 142–149.

[19] Katcher, D. I., Arakawa, H., and Strosnider, J. K. Engineering and analysis of fixed-priority schedulers. IEEE Trans. on Software Engineering, Vol. 19, No. 9, pp. 920-934, 1993.

[20] Jeffay, K., Stanat, D.F. and Martel, C.U. On non-preemptive scheduling of periodic and sporadic tasks. In Proceedings of the 12th IEEE Symposium on Real-Time Systems, pp. 129-139, 1991.

[21] Gendy, A.K. and Pont, M.J. "Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems", IEEE Transactions on Industrial Informatics, Article in press, 2007.

[22] Xu, J. and Parnas, D.L. (2000). Fixed Priority Scheduling versus Pre-Run-Time Scheduling. Real-Time Systems, Vol. 18, pp. 7-23, 2000.

[23] Short, M, Pont, M.J and Fang, J. Exploring the Influence of Preemption on Dependability in Time-Triggered Embedded Systems: a Preliminary Study. Paper to be presented at the 20th Euromicro Conference on Real-time Systems (ECRTS 2008).

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

[25] Short, M and Pont, M.J. Assessment of high-integrity embedded automotive control systems using hardware-in-the-loop techniques. Journal of Systems and Software, Article in press, 2007.

[26] Butler, R.W. and Finelli, G.B. (1993). The infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. IEEE Transactions on Software Engineering, Vol. 19, No. 1, pp. 3-12.

[27] Biere, A. Cimatti, A., Clarke, E., Strichman, O. and Zhu. Y. Bounded Model Checking. In Advances in Computers, Vol. 58, Academic press, 2003.

[28] Clarke, E., Kroening, D. and Lerda, F. A Tool for Checking ANSI C Programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), pp. 168-176, 2004.

[29] Arlat, J., Costas, A., Crouzet, Y., Laprie, J-C and Powell, D. (1993). Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. IEEE Trans. Computers, Vol. 42, No. 8, pp. 913-923.