UTILIZING MULTICORE ARCHITECTURES TO ENHANCE SOFTWARE VERIFICATION IN REAL-TIME EMBEDDED SYSTEMS

Author
Padraig Justin Fogarty
BEng; MEng

Supervisor(s)
Dr. Donal Heffernan, University of Limerick

Submitted for Degree of Doctor of Philosophy

University of Limerick, April 2013
ABSTRACT

UTILIZING MULTICORE ARCHITECTURES TO ENHANCE SOFTWARE VERIFICATION IN REAL-TIME EMBEDDED SYSTEMS

Padraig Justin Fogarty

The hypothesis of this research is that new techniques are required to facilitate software verification on the highly-integrated, but resource constrained, real-time embedded systems; which are widely used in safety-critical applications. Software verification is an essential but expensive undertaking which often consumes as much or more resources than design activities; this is particularly the case in embedded systems that require functional safety. This research explores the existing techniques for software verification on these systems and the verification challenges posed by modern highly-integrated devices. The author then proposes a novel target-level verification approach which addresses some of these challenges.

Advances in semiconductor manufacturing processes have fuelled the relentless shrinking of IC design geometries. This has dramatically reduced the area required for each functional block, reduced costs, and allowed more complex circuits to be realised; which has led to the System-on-Chip (SoC) designs which now include multiple processors within a single die. Undoubtedly many benefits result from this increased integration, but one significant drawback is the loss of access to the many signals indicating the internal operational state. Visibility of these signals is essential for many embedded software verification purposes.

In parallel with increasing SoC complexity, verification technology has transformed from using full in-circuit emulation, to bond-out devices, to on-chip instrumentation (OCI), each providing less visibility to the execution state of the processor. A key benefit of OCI approaches is the associated reduced physical interface requirements; unfortunately this also limits the real-time data that can be captured and transferred to external analysis tools. The author proposes the alternative of using this OCI in conjunction with a co-processor to perform monitoring and verification tasks on-chip; thus overcoming the interface limitations and enhancing visibility.

The experimental platform used to explore the feasibility of using a co-processor and OCI for software verification activities is described; and several case studies are examined. The results demonstrate that this approach does offer a means of addressing several software verification challenges and provides some unique capabilities, but also has some limitations. These benefits and limitations are discussed and suggestions for future work to advance this research topic are provided.
DECLARATION

I hereby declare that this thesis is entirely my own work and has not been submitted to any other University or higher education institution, or for any other academic award in this University. Where the work of other people has been used, it has been fully acknowledged and fully referenced.

Signature: ___________________________   Date:______________

Padraig Justin Fogarty   (8907331)

ACKNOWLEDGMENTS

The author wishes to thank the Irish Research Council for Science, Engineering and Technology (IRCSET) funded International Centre for Graduate Education in Micro and Nano Engineering (ICGEE) for its generous financial support for this research work.

I also want to express my deepest gratitude to Dr. Donal Heffernan, my supervisor, for his unfailing encouragement, support, and guidance, throughout the course of this research.

Finally, I want to thank Adam, Kellie, and Clodagh, for affording me the time and space to complete this journey.
# TABLE OF CONTENTS

Chapter 1. Introduction ........................................................................................................... 1  
  1.1 Rationale for this research .......................................................................................... 4  
  1.2 Research objectives ................................................................................................. 5  
  1.3 Novelty of the research ............................................................................................ 5  
  1.4 Publications ............................................................................................................. 6  
  1.5 Thesis layout ............................................................................................................ 6  

Chapter 2. Verification of embedded software ................................................................. 8  
  2.1 Introduction ............................................................................................................... 8  
  2.2 Software Verification ............................................................................................... 8  
  2.2.1 Level of verification required ............................................................................... 9  
  2.2.2 Functional safety ............................................................................................... 10  
  2.2.3 Implementation language .................................................................................. 13  
  2.3 Dynamic software verification activities .................................................................... 13  
  2.3.1 White-box testing ............................................................................................... 14  
  2.3.2 Code coverage .................................................................................................... 15  
  2.3.3 Black-box / Functional testing .......................................................................... 15  
  2.3.4 Performance profiling / Statistical testing ....................................................... 16  
  2.3.5 Load / Stress analysis ....................................................................................... 16  
  2.3.6 Fault injection .................................................................................................... 16  
  2.3.7 Integration testing ............................................................................................. 17  
  2.3.8 Regression testing ............................................................................................. 17  
  2.3.9 Requirements tracing ....................................................................................... 18  
  2.3.10 Security testing ................................................................................................. 18  
  2.3.11 System-level testing ......................................................................................... 18  
  2.4 Emerging verification challenges ........................................................................... 18  
  2.4.1 Multicore architectures ...................................................................................... 19  
  2.4.2 Non-deterministic execution ............................................................................ 23  
  2.4.3 Reliability of highly-integrated devices ............................................................ 25  
  2.4.4 Lack of visibility ............................................................................................... 26  
  2.4.5 Security ............................................................................................................ 27  
  2.4.6 Debugging ......................................................................................................... 27  
  2.5 Verification by simulation and emulation .............................................................. 29  
  2.6 Summary ................................................................................................................. 33  

Chapter 3. Correct by construction .................................................................................. 34  
  3.1 Introduction .............................................................................................................. 34  
  3.2 Test driven development ......................................................................................... 34  
  3.3 Component based development ............................................................................. 35  
  3.4 Design by contract .................................................................................................. 35  
  3.5 Model based development ...................................................................................... 37  
  3.6 Formal methods for verification ............................................................................. 39  
  3.7 Runtime monitors / Runtime verification ............................................................ 43  
  3.8 Summary ................................................................................................................. 45
## Chapter 4. Support for verification at the chip level

4.1 Introduction

4.2 Reuse of silicon test interfaces

4.3 I/O interfaces

4.3.1 IEEE 1149.1 (JTAG)

4.3.2 IEEE 1149.7

4.3.3 IEEE-ISTO 5001 – 2003 (Nexus)

4.4 On-chip core interfaces

4.4.1 IEEE P1687

4.4.2 IEEE 1500

4.5 On-chip instrumentation (OCI)

4.5.1 Instrument trace

4.5.2 OCP-IP

4.5.3 FS2

4.5.4 MIPS

4.5.5 ARM

4.5.6 Infineon MCDS

4.5.7 UltraSOC

4.5.8 Combined instrumentation

4.6 Built-in self-test and Software based self-test

4.7 Software instrumentation

4.8 Summary

## Chapter 5. Alternative approach

5.1 Introduction

5.2 Alternative approach

5.2.1 Key benefits of the proposed alternative

5.3 Related research

5.4 Summary

## Chapter 6. Experimental Platform

6.1 Introduction

6.2 CPU12X

6.3 XGATE

6.4 Background debug module (BDM)

6.5 Debug module (S12XDBG)

6.6 Development platform

6.7 Summary

## Chapter 7. Verifying runtime behaviour

7.1 Introduction

7.2 Experimental setup

7.3 Monitoring execution sequences

7.4 Measuring execution timing in real-time systems
7.5 Measuring periodic events in real-time systems ........................................ 92
7.6 Results ........................................................................................................ 96
7.7 Summary ..................................................................................................... 99
Chapter 8. Runtime verification of requirements ............................................. 100
  8.1 Introduction................................................................................................ 100
  8.2 CPAP device and requirements for adjustment of settings ...................... 100
  8.3 Experimental application platform ........................................................... 102
  8.4 Verifying setting change timeouts ............................................................ 104
  8.5 Verifying application checks on data-structure integrity ......................... 105
  8.6 Verifying application checks on setting range ....................................... 107
  8.7 Results ..................................................................................................... 109
  8.8 Summary .................................................................................................. 110
Chapter 9. Conclusions and future work ...................................................... 111
  9.1 Introduction............................................................................................... 111
  9.2 Review of background material and resulting alternative approach........ 111
  9.3 Results from experimental work ............................................................. 116
  9.3.1 Limitations of this alternative approach ............................................. 119
  9.4 Conclusions ............................................................................................ 120
  9.5 Future work ............................................................................................ 122
Bibliography .................................................................................................... 126
LIST OF FIGURES

Figure 1: Software development lifecycle V-Model ........................................................... 9
Figure 2: Generic system-level model of an embedded system ................................. 17
Figure 3: Hardware architectures for multicore devices ............................................... 20
Figure 4: Software architectures for multicore devices ............................................... 20
Figure 5: Technologies for on-chip verification in modern SoC designs ....................... 47
Figure 6: IEEE 1149.1 interface signals ........................................................................ 49
Figure 7: IEEE 1149.7 interface signals ........................................................................ 50
Figure 8: Nexus development interface ........................................................................ 51
Figure 9: P1687 zones .................................................................................................... 52
Figure 10: IEEE 1500 core test wrapper ......................................................................... 53
Figure 11: OCP-IP inter-core debug fabric .................................................................... 58
Figure 12: Comparison of relative merits of test and debug technologies ..................... 69
Figure 13: Proposed architecture ................................................................................... 71
Figure 14: Using OCI as a monitor and trigger for software instrumentation ............... 71
Figure 15: Relative positioning of alternative against existing technologies ............... 72
Figure 16: Distributed debug architecture with dedicated ASIC .................................. 73
Figure 17: Block diagram of MC9S12XE ....................................................................... 76
Figure 18: Programming model for CPU12X ................................................................. 77
Figure 19: Programming model for XGATE ................................................................. 78
Figure 20: Block diagram of S12XDBG module ............................................................. 79
Figure 21: S12XDBG comparators ................................................................................. 80
Figure 22: State sequencer transition diagram ............................................................... 81
Figure 23: MC9S12XE development platform ............................................................... 82
Figure 24: Development platform using co-processor .................................................... 83
Figure 25: Experimental setup ....................................................................................... 85
Figure 26: Application and instrumentation software isolation and interaction ............ 86
Figure 27: Cyclic task execution ..................................................................................... 86
Figure 28: Flowcharts for monitoring execution sequences .......................................... 87
Figure 29: Example of embedded system generating a real-time signal ....................... 89
Figure 30: Flowcharts for monitoring real-time execution timing .................................. 90
Figure 31: Execution timing waveforms captured by oscilloscope ............................... 91
Figure 32: Flowchart for XGATE instrumentation to verify periodic events ................ 94
Figure 33: Plot of captured PWM register values and the period between updates ....... 95
Figure 34: Timing diagram for cyclic application example .......................................... 96
Figure 35: Simplified system-level block-diagram of a CPAP device ......................... 100
Figure 36: State diagram for CPAP setting adjustment ............................................... 102
Figure 37: Flowcharts for XGATE monitoring of CPAP application ......................... 103
Figure 38: Flowchart of instrumentation code for verifying timeout requirement .... 104
Figure 39: Flowchart of instrumentation code to verify data structure integrity ....... 105
Figure 40: Flowchart of instrumentation code to verify setting range checks ...... 108
Figure 41: MC9S12XE100 core current consumption, when operating at 50 MHz. 109
Figure 42: Alternative architecture supporting multiple instrumentation functions .. 120
Figure 43: Proposed approach applied to a quad-core SoC architecture ............... 122
Figure 44: The Monitoring and Checking (MaC) architectural framework .......... 124
Figure 45: Runtime architecture for the MaC framework ................................. 125
Figure 46: Development board for experimental work ...................................... A-1
Figure 47: MC9S12XE100 memory map .......................................................... B-1
Figure 48: Close-up of waveform showing timing of OutputPWM function......... C-4

LIST OF TABLES
Table 1: Classification of ASIL as per ISO 26262-3.............................................. 11
Table 2: Methods for the verification of software unit design and implementation .... 11
Table 3: Software V&V activities proposed in IEEE Std 1012-2012 ...................... 14
Table 4: Examples of low-level software verification activities .............................. 14
Table 5: Status data outputted after deadline exceeded ..................................... 91
Table 6: Setting change timeout values recorded from application .................... 105
Table 7: Allowable values for CPAP settings.................................................. 107
Table 8: Key verification challenges posed by highly-integrated SoCs.................. 112
Table 9: Advantages and disadvantages of emulation and simulation solutions..... 112
Table 10: Correct by construction software development methodologies........... 113
Table 11: Summary of on-chip interface and instrumentation solutions.............. 114
## ACRONYMS AND ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABV</td>
<td>Assertion-Based Verification</td>
</tr>
<tr>
<td>ADC</td>
<td>Analog-to-Digital Converter</td>
</tr>
<tr>
<td>AMP</td>
<td>Asymmetric Multi-Processing</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-Specific Integrated Circuit</td>
</tr>
<tr>
<td>ATE</td>
<td>Automated Test Equipment</td>
</tr>
<tr>
<td>BDM</td>
<td>Background Debug Module</td>
</tr>
<tr>
<td>BIST</td>
<td>Built-In Self-Test</td>
</tr>
<tr>
<td>BMC</td>
<td>Bounded Model Checking</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial off-the-shelf</td>
</tr>
<tr>
<td>CPAP</td>
<td>Continuous Positive Airway Pressure</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital-to-Analog Converter</td>
</tr>
<tr>
<td>DAP</td>
<td>Debug Access Port,</td>
</tr>
<tr>
<td>DbC</td>
<td>Design by Contract</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random-Access Memory</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>DVS</td>
<td>Dynamic Voltage Scaling</td>
</tr>
<tr>
<td>ECC</td>
<td>Error Correcting Code</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>ESD</td>
<td>Electrostatic Discharge</td>
</tr>
<tr>
<td>FDA</td>
<td>Food and Drug Administration</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HIL</td>
<td>Hardware-in-the-loop</td>
</tr>
<tr>
<td>I/O</td>
<td>Input / Output</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>ICE</td>
<td>In-Circuit Emulation</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IMA</td>
<td>Integrated Modular Avionics</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group (the common name for IEEE 1149.1 interface)</td>
</tr>
<tr>
<td>LED</td>
<td>Light Emitting Diode</td>
</tr>
<tr>
<td>MaC</td>
<td>Monitoring and Checking</td>
</tr>
<tr>
<td>MBD</td>
<td>Model Based Development</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>MCDS</td>
<td>Multi-Core Debug Support</td>
</tr>
<tr>
<td>MISRA</td>
<td>Motor Industry Software Reliability Association</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multiprocessor System-on-Chip</td>
</tr>
<tr>
<td>NoC</td>
<td>Network-on-Chip</td>
</tr>
<tr>
<td>OCD</td>
<td>On-Chip Debugger</td>
</tr>
<tr>
<td>OCI</td>
<td>On-Chip Instrumentation</td>
</tr>
<tr>
<td>OCP</td>
<td>Open Core Protocol</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>PIT</td>
<td>Periodic Interval Timer</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse Width Modulation</td>
</tr>
<tr>
<td>RAM</td>
<td>Random-Access Memory</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RTC</td>
<td>Run-to-Completion</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>SBST</td>
<td>Software Based Self-Test</td>
</tr>
<tr>
<td>SEU</td>
<td>Single-Event Upset</td>
</tr>
<tr>
<td>SIL</td>
<td>Safety Integrity Level</td>
</tr>
<tr>
<td>SIP</td>
<td>System-in-Package</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multi-Processing</td>
</tr>
<tr>
<td>SMT</td>
<td>Satisfiability Modulo Theory</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-Chip</td>
</tr>
<tr>
<td>SWI</td>
<td>Software Interrupt</td>
</tr>
<tr>
<td>TAM</td>
<td>Test Access Mechanism</td>
</tr>
<tr>
<td>TAP</td>
<td>Test Access Port</td>
</tr>
<tr>
<td>TBB</td>
<td>Thread Building Blocks</td>
</tr>
<tr>
<td>TDD</td>
<td>Test Driven Development</td>
</tr>
<tr>
<td>TLM</td>
<td>Transaction-Level Model</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>WCET</td>
<td>Worst-Case Execution Time</td>
</tr>
<tr>
<td>WDT</td>
<td>Watch-dog Timer</td>
</tr>
</tbody>
</table>
CHAPTER 1. INTRODUCTION

In recent years embedded systems have become ubiquitous and support many of the everyday tasks and services which people living in technologically advanced societies take for granted [1]. In many cases the presence of these embedded systems goes unnoticed by the user. In part, this is due to the inherent nature of embedded systems, which means that the user may not consciously interact with the software or electronics and therefore may not even be aware of their presence [2]; another factor is the fact that embedded systems are often expected to always operate reliably, without the need for any user intervention [3]. However, the cost of verifying software on embedded systems can consume up to 70% of the development budget [4], [5]; and the confidence placed in these embedded systems is only justified if they can be robustly designed and thoroughly verified.

The dramatic reduction in the cost of processors and microcontrollers has also enabled the development of smart components, often replacing or augmenting traditional mechanical or electromechanical solutions [6]; which offer the benefits of greater configurability and higher reliability. In the main, this cost reduction has been driven by advances in silicon manufacturing processes; the shrinking of silicon device sizes has enabled a continuous reduction in the cost of manufacturing existing devices and the realisation of more highly-integrated and more complex devices [5]. In many cases, current microcontrollers for embedded systems now contain more than one processor core in addition to circuitry which would once have been peripheral to the device; in what are referred to as multicore System-on-Chip (SoC) devices [7], [8], [9], [10], [11], [12]. The term multicore can be applied to a wide variety of different architectures [13], for the purposes of this research it refers to SoC architectures which contain two or more processor cores; in some literature the term multiprocessor SoC (MPSoC) [14] is used in reference to such devices.

Availability of these sophisticated SoCs has undoubtedly enabled embedded systems to be produced at much lower cost, but cost reduction is also one of the key driving forces behind their existence; this is particularly the case in high-volume applications. Consequently, although highly complex SoC devices are realisable and available, many embedded systems utilise SoC devices to minimise cost and therefore the on-chip and I/O resources used are still kept to a minimum [1], [15]. An additional consideration is the desire to keep power consumption to a minimum [15]; executing an application using multiple processors operating at lower frequencies and voltages can potentially lower power consumption [16]. However, as geometries shrink, power density and associated thermal dissipation problems become more significant [17], [18]. It is therefore still desirable to minimise all on-chip resources, including those required for verification purposes. This thesis is primarily concerned with the target-level verification of software operating on these resource
constrained embedded systems (typically operating at frequencies of less than 100 MHz, utilizing 8-bit, 16-bit or 32-bit processors, ≤ 64K bytes of data memory, and ≤ 1M bytes of program memory); and in particular those embedded systems which are designed to operate in high-reliability or safety-critical applications.

Although embedded systems are often intended to provide a more reliable alternative to mechanical solutions, using complex SoC devices actually makes the task of ensuring the reliable operation of the system even more difficult [19]. For embedded systems, increased integration often means vital information relating to the software execution on the target platform becomes even more difficult, if not impossible, to extract [20], [21], [22]. Efficient and reliable utilisation of multiple processor cores on a single device also presents new challenges; in the main these relate to managing concurrent execution and access to shared resources. However, the author shows that the availability of multiple processors on a single integrated device may also present the opportunity to consider alternative target-level verification approaches.

For desktop systems, the more complex architectures enabled by increased integration undoubtedly bring new software design and verification challenges too [23], but it also increases the available resources; enabling desktop applications to continue to be verified within a rich development ecosystem [24]. Of course, many embedded software verification activities can be carried out on a desktop platform. Embedded design activities including system-level architecting and modelling are generally performed on the desktop. Software can be written, compiled, and tested on desktop based integrated development environments; before porting to the embedded target [25]. The development environment may also include simulation or emulation capabilities [15]; and many static code analysis tools which run on desktop platforms are equally applicable to desktop application code and embedded systems code.

Despite the ability to design and develop embedded software on a desktop platform it is also necessary to perform software verification at runtime on the physical system-level target; this is particularly important for real-time safety-critical or high-reliability systems [26], [27], [28]. In contrast to desktop systems, embedded systems operate in a much more constrained environment, and with distinct requirements as described by Koopman [29] and Lee [3]. It is therefore imperative that engineers gain access to new techniques and tools to enhance visibility and access to the interaction and functioning of software on embedded systems in their end-use application platform, to facilitate its systematic testing and verification. It is the challenges associated with software verification on that embedded platform which is the subject of this research.

One conventional approach to provide greater visibility into the software execution on
highly-integrated embedded targets is to add software instrumentation to the application code [30], [31], [32], [33], [34]. This has the dual benefits of requiring little or no additional hardware resources and of being easily adapted for various verification tests. However, it has the one significant disadvantage, which is that software instrumentation is intrusive to the application. Executing this additional instrumentation code could introduce new errors or mask existing errors, and may alter the runtime temporal behaviour of the application. Simply increasing the operational speed of the application processor to compensate for the increased processing burden may seem a reasonable solution, but this would create a verification platform which is no longer equivalent to the target, and would unnecessarily increases power consumption; efficient operation and minimising power dissipation is often a key design requirement for embedded systems [15], [35], [17], [36]. The alternative of adding dedicated on-chip instrumentation hardware and I/O resources to assist with verification also increases power consumption, and can be prohibitively expensive; particularly if these resources cannot be utilized for other purposes.

Clearly, this presents a dilemma for verification of the embedded software used in safety-critical applications; verification requires greater visibility, at reasonable cost, but this cannot impact upon the application being verified. The author proposes an alternative approach which exploits the increasing availability of multicore SoCs. By placing the instrumentation code on one CPU, or co-processor, and utilizing the existing on-chip instrumentation hardware to synchronise its execution with the application code running on another CPU, the author demonstrates that the benefits of software instrumentation can be obtained without impacting upon the application.

Using a commercially available microcontroller based SoC device [12] as the target platform, the author conducts a number of experiments to evaluate the feasibility of this alternative approach. The initial experiments are performed on a contrived real-time embedded system. These experiments examine the capability to unobtrusively: monitor the runtime execution sequences; measure the execution timing to ensure that timing constraints are met; measure the timing of periodic events; and capture runtime data for on-chip analysis and off-line recording/display. Using a case-study from an actual medical device, the author then examines the ability to perform runtime verification of safety-related software requirements.

The experiments demonstrate that this alternative approach is feasible. In all cases it was possible to configure the on-chip instrumentation to monitor the application and trigger the software instrumentation on the co-processor. This provides a non-intrusive, or minimally intrusive, means of monitoring the application and enables the co-processor to capture and analyse data relevant to the runtime execution behaviour. This approach also offers additional
benefits, such as the ability of the co-processor to measure the timing between events, and the ability to monitor every update to a peripheral or memory location; without the need to add additional instrumentation for each.

The experiments conducted also highlight some limitations to the proposed approach. In some case these are due to constraints imposed by the experimental platform used; for example, the number of events which can be monitored simultaneous is dictated by the on-chip instrumentation hardware, and the rate at which status or captured data can be exported off-chip is dictated by the I/O interface used. A more fundamental limitation is that the time required to execute the instrumentation code on the co-processor dictates the minimum response time between successive instrumentation activations. This same limitation does not exist when instrumentation code is added directly into the application, since such code would execute in an inherently sequential manner; but, this disadvantage needs to be balanced against the fact that the inline instrumentation code impacts upon the execution timing of the application, whereas instrumentation code residing on the co-processor does not.

Nonetheless, the author shows that this alternative approach offers a novel way to address some of the verification challenges posed by today’s highly-integrated embedded platforms. This approach may become even more necessary, and at the same time more practical, as the number of processor cores integrated into SoC devices continues to increase.

1.1 Rationale for this research

The software engineering research community offers formal theories, based in sound semantic models that lead to verifiable designs or correct by construction designs. Program verification determines whether a program satisfies a specification. However, program verification is unsolvable in general, and theorem prover solutions do not in practice scale to large embedded software systems. In spite of shortcomings, formal methods research has huge potential and continues to receive much attention with an aim to prove a design to be correct; so that, verifiable commercial products might be built without the need for complex development and exhaustive testing. With increasing complexities, new approaches are needed. Burns and Hayes [37] state that new scientific foundations are now required for specifying, designing, and implementing complex real-time systems.

For this field, with the exception of a few well-publicised examples, formal methods have not yet met expectations for the development of commercial products. Some major research initiatives are showing potential; for example DEPLOY [38], a European Commission FP7 project, aims to make major advances in engineering methods for dependable systems. The DEPLOY project has several industrial partners using Event-B and Rodin [39] on deployment projects for real embedded products. ADVANCE [40] is another FP7 project where the objective is to develop a unified tool-based framework for automated formal verification and
simulation-based validation.

While such developments are encouraging, research towards solutions for correct software designs needs to be complemented with solutions that can monitor the exact program behaviour, non-intrusively, for highly-integrated, multi-core embedded systems. Even if a complex software system can be ‘believed’ to be correct, confidence in the total hardware/software product combination is paramount; therefore, accurate monitoring and testing of its behaviour in its target environment is still necessary. The execution of real-time application software can be affected by system issues such as unpredicted hardware performance, real-time operating system (RTOS) performance, I/O interrupt behaviour, and interference from other software applications on the system.

To perform in-situ software verification, validation, development, and debug, it is imperative that engineers have adequate tools. There is significant research into on-chip instrumentation (OCI) architectures. OCI solutions exist for on-chip trigger and trace infrastructures, using high-bandwidth interfaces and on-chip trace buffers to capture data at very high rates. However, current and emerging solutions have limitations due to available bandwidths and available on-chip hardware, which restrict their capabilities for software verification and debug. The fundamental research question is to ask if there is a better architectural solution for on-chip software verification and debug instrumentation.

1.2 Research objectives

The purpose of this research is to review the emerging solutions for embedded software development and verification, in particular on-chip monitoring and debug instrumentation, and to propose an enhanced OCI solution which is based on the use of a co-processor. The main objectives of this research are to:

- Examine the current state-of-the-art with regard to embedded software verification techniques and emerging verification challenges.
- Examine the capability of current embedded software design and development methodologies to produce software which is correct by construction.
- Research the capabilities of existing silicon test and OCI solutions.
- Propose an alternative technique using OCI and a co-processor which can assist in addressing target-level software verification challenges.
- Examine the feasibility of using this alternative technique in a number of case studies.

1.3 Novelty of the research

Significant research has been devoted to the study and development of techniques which might enable software to be developed without errors. Although significant advances have
been achieved towards this goal, the current situation remains that extensive software verification is still required to ensure the correct and safe functioning of software in embedded targets. Despite advances in silicon manufacturing processes, which have provided faster and more complex SoC devices at economical prices, the continuing need to restrain costs has meant that on-chip circuits to support software verification have not kept pace with architectural complexity or speed. The hypothesis of this research is that new techniques are required to facilitate software verification on the highly-integrated, but resource constrained, real-time embedded systems, which are widely used in safety-critical applications.

The author proposes the use of existing OCI capabilities coupled with a co-processor to perform software verification tasks on-chip. This proposed approach has the potential to analyse the captured data on-chip and to optimise the use of on-chip resources in a non-intrusive, or in a minimally-invasive fashion. In this author’s opinion, on-chip data analysis represents an effective way of addressing the I/O bandwidth limitations of SoC designs and enabling new verification and debugging tools.

1.4 Publications

The research outputs include the following generated, and planned, publications:

**Journal papers**


**Paper in draft form**


**Conference papers**


**Other**


1.5 Thesis layout

This thesis is organised as follows:

- Chapter 2 examines the general topic of software verification for embedded system. In particular, it is focused upon a number of sectors that involve the development of
embedded system which have safety-critical software elements, or where functional safety of the entire system is required. The chapter provides a summary of some of the main standards and regulatory requirements applying to these sectors, and the principal dynamic software verification activities applied. Emerging verification challenges resulting from greater complexity and higher levels of integration are then considered. Finally, the chapter examines the practicalities of using simulation and emulation platforms to perform verification of embedded software.

- Chapter 3 explores the possibility of creating software which is designed to be correct and therefore requires less verification, or ideally none. The chapter covers topics including test driven development techniques, component and model based development, model checking and runtime monitoring and verification.

- Chapter 4 surveys the current state-of-the-art with regard to on-chip support from software verification. Both standard-based and proprietary solutions are examined for I/O interfaces, on-chip core interfaces, and OCI. The chapter also considers the suitability of reusing silicon test interfaces and test techniques for software verification purposes.

- Chapter 5 describes the alternative approach advocated by the author, of using OCI and the processing capabilities of a co-processor to assist with software verification. The key potential benefits of this approach are also outlined.

- Chapter 6 provides a technical description of the experimental platform used in the subsequent experiments. This consists of a commercial off the shelf SoC platform which contains many of the key hardware components required to examine the feasibility of the proposed alternative approach.

- Chapter 7 describes three experiments carried out to investigate the feasibility of unobtrusively monitoring the target-level execution behaviour and measuring execution timing on a hypothetical real-time system; using OCI and a co-processor.

- Chapter 8 provides a case study of verifying safety-related software requirements on a medical device. The software architecture of the original medical device was replicated on the experimental platform, and the proposed approach was used to perform the runtime verification on a sample of the software requirements.

- Chapter 9 discusses the results of the experiments and the background research conducted. The author then provides the principal conclusion arrived at and describes ways in which this work might be improved upon; including making suggestions for relevant areas of further study.
CHAPTER 2. VERIFICATION OF EMBEDDED SOFTWARE

2.1 Introduction

There are many authoritative books related to the subject of software verification [29], [41], [42], [43], [44], [25]. The topics covered span a wide range including organisational dynamics, quality system procedures and practice, software development approaches, software analysis techniques, testing activities, debugging methods and tools, etc.; which reflect the broad scope of the subject. In this chapter the author outlines the background to those topics with which this thesis is primarily concerned: the techniques and technologies currently used for dynamic software verification on resource constrained real-time embedded systems; the challenges when considering emerging highly-integrated devices; and target-level software debugging challenges.

2.2 Software Verification

Software verification is defined in IEEE Std 610.12-1990 [45] as:

“verification. (1) The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.”

The more recent IEEE Std 1012-2012 [46] and IEEE Std 829-2008 [47] standards expand upon IEEE 610.12 to include a more comprehensive definition:

“(B) The process of providing objective evidence that the software and its associated products conform to requirements (e.g., for correctness, completeness, consistency, accuracy) for all life cycle activities during each life cycle process (acquisition, supply, development, operation, and maintenance); satisfy standards, practices, and conventions during life cycle processes; and successfully complete each life cycle activity and satisfy all the criteria for initiating succeeding life cycle activities”.

In contrast the same standards, [46], [47], define validation as:

“3.1.53 validation: (A) The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. (B) The process of providing evidence that the software and its associated products satisfy system requirements allocated to software at the end of each life cycle activity, solve the right problem (e.g., correctly model physical laws, implement business rules, or use the proper system assumptions), and satisfy intended use and user needs.”

The distinction between verification and validation is often simply stated as follows: ‘Verification, asks if we built the product correctly. Whereas validation, asks if we built the correct product’ Unfortunately this distinction is often a matter of perspective as to what the ‘product’ is; in SoC design the deliverable is typically the working silicon device, so the tools and techniques used to check the silicon are generally considered to be for validation purposes. However, these same tools and techniques may be used when the SoC is embedded...
into the system-level design, where they are used for software verification purposes.

This ambiguity is compounded by the fact that software verification spans a wide range of activities from high-level documentation reviews to low-level integration tests. Wallace and Fujii [48] provide an overview of the typical software verification and validation task and how these can be linked into standard software development models; they also emphasise that verification and validation are best considered as interrelated activities. IEEE Std 1012-2012 [46] also highlights the interrelated nature of the verification and validation processes. In contrast, guidance documents from the Food and Drug Administration (FDA) [49] make a distinction between verification and validation. The FDA focuses primarily upon validation of the finished medical device, with verification being described as the range of activities which support the conclusion that the software is validated. Consequently, within the medical device sector ‘software validation’ is commonly used as a phrase to encompass all verification and validation activities throughout the development lifecycle.

This distinction is also often reflected in the software development lifecycle model employed. Figure 1 illustrates the classic V-Model development lifecycle with validation being represented at the top-level, and verification as the activities at the lower level levels [27], [50], [51]. This V-Model is widely applied for safety-critical embedded systems where verification and validation of hardware and software are essential activities; therefore, for the purpose of this research the development process is understood to reflect this model.

![Software development lifecycle V-Model](image)

**Figure 1: Software development lifecycle V-Model**

### 2.2.1 Level of verification required

The level of verification required for any particular piece of software is dependent upon its use case and the regulatory requirement for the sector for which it is intended. Unfortunately, this creates a scenario where, even for safety-critical software, there is no single set of
criteria. Instead different sectors have different criteria, some sectors require software certification, while others do not [52]. In avionics one presumes that the pilot will be highly trained to safely operate the aircraft; in the automotive sector the driver is considered to be capable (hazards are classified according to controllability) [53]; whereas for certain medical devices the patient cannot be considered to be capable. For many medical devices a clinician is required to operate the device, but hazards analysis on medical devices must recognise that a clinician may not be present either.

The level of verification required may also be influenced by application sector considerations and the system-level architecture employed. Cooling [54] gives an overview of several real-time embedded operating systems where the system-level composition is noticeably different, because the operating systems are aimed at different sectors. Grossmann et al. [55] show how test techniques developed for telecommunications applications need to be modified to fit real-time automotive application requirements.

2.2.2 Functional safety

In many industries ensuring the functional safety of the entire system is paramount; where software is used in such systems it can serve an equally important role to hardware or mechanical components. Unfortunately, faults in such safety-critical systems often have their root-cause in software [56]. Leveson [57] lists notable errors in avionics and automotive applications and highlights the inherent complexity and flexibility of software, and the difficulties in reliably combining elements at the system level, as contributory factors. Mayer et al. [19] reference a study which suggests 77% of electronic failures in cars were due to software; and that the US National Institute of Standards and Technology estimates the cost of software defects, in the US alone, to be in the region of $59.9billion/year. As the cost of fixing a defect found early in the design process is lower [58], [59], [60], [61], and can be many orders of magnitude less than the cost of product recall or compensation for injury, it is expedient and financially beneficial to adopt appropriate methods for software verification.

One common approach to defining the appropriate level of safety required for a design is to assign a Safety Integrity Level (SIL), which is based upon the criticality of a fault and its likelihood of occurrence; although this approach is used in several industries, many have differing numbers of SILs, severity classes, and classification criteria [56]; and there is not always a direct correlation between these SILs [62]. For medical devices, EN 62304 [63] categorises software into Class A, B or C according to the severity of the associated risk; IEC 61508 defines four Safety Integrity Levels 1, 2, 3, 4 [64]; whereas the more recent automotive specific standard ISO 26262 defines four Automotive Safety Integrity Levels (ASIL) A, B, C, D [53].

Table 1 shows how classification of the component using severity, probability, and
controllability criteria determines the appropriate ASIL in accordance with ISO 26262-3 [53] (Section 7.4.4). The designation ‘quality management’ (QM) signifies that the hazard or risk posed can be addressed by standard quality management procedures and there is no specific requirement for that class to comply with ISO 26262.

<table>
<thead>
<tr>
<th>Severity class</th>
<th>Probability class</th>
<th>Simply controllable</th>
<th>Normally controllable</th>
<th>Difficult or uncontrollable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Light and moderate injuries</td>
<td>Very low</td>
<td>QM</td>
<td>QM</td>
<td>QM</td>
</tr>
<tr>
<td>Low</td>
<td>QM</td>
<td>QM</td>
<td>QM</td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>QM</td>
<td>QM</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>QM</td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>Severe and life-threatening injuries (survival probable)</td>
<td>Very low</td>
<td>QM</td>
<td>QM</td>
<td>QM</td>
</tr>
<tr>
<td>Low</td>
<td>QM</td>
<td>QM</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>QM</td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>Life-threatening injuries (survival uncertain), fatal injuries</td>
<td>Very low</td>
<td>QM</td>
<td>QM</td>
<td>A</td>
</tr>
<tr>
<td>Low</td>
<td>QM</td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>High</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Classification of ASIL as per ISO 26262-3

Having established the appropriate SIL level for a software component, the verification techniques recommended for that level are generally given in corresponding software development standards. For example, Table 2 shows the ISO 26262-6 [27] verification methods for software unit design; depending upon the ASIL these methods are signified as being recommended (+), highly recommended (++), or having no recommendation (o).

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Walk-through</td>
<td>++ + o o</td>
</tr>
<tr>
<td>1b Inspection</td>
<td>+ ++ ++ ++</td>
</tr>
<tr>
<td>1c Semi-formal verification</td>
<td>+ + ++ ++</td>
</tr>
<tr>
<td>1d Formal verification</td>
<td>o o + +</td>
</tr>
<tr>
<td>1e Control flow analysis</td>
<td>+ + ++ ++</td>
</tr>
<tr>
<td>1f Data flow analysis</td>
<td>+ + ++ ++</td>
</tr>
<tr>
<td>1g Static code analysis</td>
<td>+ ++ ++ ++</td>
</tr>
<tr>
<td>1h Semantic code analysis</td>
<td>+ + + +</td>
</tr>
</tbody>
</table>

Table 2: Methods for the verification of software unit design and implementation

Standards such as ISO 26262 consider functional safety at a system level; therefore, aspects which impact upon software verification are not confined to the just one sub-section of the standard. Instead this standard, and similar ones, encompass the entire product life-cycle and software development life-cycle. These standards must therefore be considered as a whole; with potential implications for software verification techniques spread throughout the
documents. For example, the hardware level sub-section of ISO 26262 [65] includes software based diagnostics methods which may be used to achieve necessary coverage of hardware faults. ISO 26262-4 [66] (Section 8.4.2.2) also references several appropriate hardware-software, system and vehicle integration test methods including: requirements-based, fault injection, error guessing, back-to-back (comparing actual object to simulation model), performance, resource usage, stress test, internal and external interface tests, including user tests under real-life conditions.

When considering software verification in isolation, aspects of these standards can sometimes be seen as conflicting; ISO 26262-4 [66] (Section 7.4.3.7) lists system design properties for ASIL C and D such as avoidance of unnecessary complexity in hardware and software components as desirable, and testability during development and operation is considered highly desirable. Unfortunately, testability often requires the introduction of otherwise unnecessary hardware and software features.

Standards enable component suppliers to better understand the needs of their customers; manufacturers of key components for safety-critical embedded systems, such as microcontrollers, often include features specifically designed to aid conformance to these standards. A Freescale ‘SafeAssure’ white paper [62] describes IEC 61508 and ISO 26262 in general terms, and defines the sources of failure as: random hardware failures, systematic hardware failures, and/or systematic software failures. The white paper also describes features such as dual-core lockstep operation, built-in self-test (BIST), error-correction code (ECC) memory, monitors, and redundant functions in hardware, available in that range of products to help in meeting the requirements of these standards. Fritsch et al. [51] describe how the Altera FPGA design tools, IP blocks, and supporting data, can assist in compliance with the requirements of IEC 61508; but also highlight the significant role that the customers quality management systems plays.

Standards relating to functional safety provide useful guidance and recommendations, which reflect best-practice for management of software development in the industry concerned, and the expected capabilities and methods for those producing products within that industry. As highlighted by Meyer [67], there is often a considerable distinction between process and product focused solutions to improve software quality. Such standards inevitably focus upon the development process and do not prescribe technical implementation details, nor how the detailed design or verification activities should be performed, what development tools, or what programming language should be used; these decisions are left to the individual company or industry to dictate.
2.2.3 Implementation language

Software verification techniques and tools are often interconnected with the selected programming language; the primary focus of this research is upon verification techniques suitable for embedded software written in C. Havelund [68] advises that the most widely used language for embedded systems is C, which suggests that software verification tools and techniques should support that language, or be easily adopted by engineers familiar with it. The C language is also widely used by the related development tools, automatic code generation or software synthesis tools used in SoC design flows often produce C or C++ code [69]. Kim and Bond [70] survey the languages and extensions offered to enable parallel processing on multicore architectures for signal and image processing applications, the majority of which are based upon C or C++.

Companies may be willing to adopt new hardware architectures, which provide additional functionality, to which their engineers can apply their existing skills, but the huge base of legacy code makes companies reluctant to adopt new languages, which in many cases would also necessitate retraining. And given the significant base of existing embedded systems software written in C and the exorbitant cost for redesigning safety-critical software, leveraging legacy code is often an economic necessity [71]. Considering that functional safety standards also allow for the software reuse by ‘Proven in use argument’ [72], companies conforming to those standards have an additional motivation to retain existing software and incorporate this into new designs.

The use of the C language in safety-related design is not without difficulties. Barry [73] highlights the difficulty in meeting the requirements of IEC 61508 when using the C language, and how these difficulties are exacerbated by the necessity to use compilers which are uncertified. Nonetheless the C language is widely used in safety-critical embedded applications and its deficiencies are often alleviated by use of additional standards and guidelines; for example the Motor Industry Software Reliability Association (MISRA) has produced guidelines [74] to promote use of only unambiguous C language constructs for embedded automotive applications.

2.3 Dynamic software verification activities

Many software development standards and quality system guidelines [46], [47], [75], [76], [77], [78], give only an outline of expected or recommended high-level processes, procedures, or methods, and related verification activities; Table 3 lists 11 software V&V activities proposed in IEEE Std. 1012-2012 [46]. These high-level methods must therefore be translated into domain specific low-level verification activities; which for the purposes of this research are those dynamic or functional verification activities, or tests, performed upon safety-critical or high-reliability real-time embedded system platforms to ensure that the
software meets its requirements. Table 4 lists more detailed examples of the dynamic software low-level verification activities required; compiled by examining literature and guidance documents related to such systems [27], [29], [41], [49], [50], [63], [79], [80], [81].

<table>
<thead>
<tr>
<th>IEEE 1012-2012 Software V&amp;V Activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Software Concept V&amp;V</td>
</tr>
<tr>
<td>2 Software Requirements V&amp;V</td>
</tr>
<tr>
<td>3 Software Design V&amp;V</td>
</tr>
<tr>
<td>4 Software Construction V&amp;V</td>
</tr>
<tr>
<td>5 Software Integration Test V&amp;V</td>
</tr>
<tr>
<td>6 Software Qualification Test V&amp;V</td>
</tr>
<tr>
<td>7 Software Acceptance Test V&amp;V</td>
</tr>
<tr>
<td>8 Software Installation and Checkout V&amp;V</td>
</tr>
<tr>
<td>9 Software Operation V&amp;V</td>
</tr>
<tr>
<td>10 Software Maintenance V&amp;V</td>
</tr>
<tr>
<td>11 Software Disposal V&amp;V</td>
</tr>
</tbody>
</table>

Table 3: Software V&V activities proposed in IEEE Std 1012-2012

| White-Box testing                     |
|                                        |
| Code coverage                         |
| Black-Box / Functional testing        |
| Performance profiling/ Statistical testing |
| Load / stress analysis                |
| Fault injection                       |
| Integration testing                   |
| Regression testing                    |
| Requirements tracing                  |
| Security testing                      |
| System-level testing                  |

Table 4: Examples of low-level software verification activities

Testing for presence or absence of errors?

Myers [86] makes the very valid argument that testing would be better considered as ‘the process of executing a program with the intent of finding errors’, rather than the definition provided by ITEA ‘The aim of testing is to gain confidence that a test object possesses the required properties’ [81]; similar contradictory viewpoints exist elsewhere [43], [82]. For the purposes of clarity, the descriptions here are given from the perspective of verification/testing being an attempt to show that the software is correct; this in no way attempts to detract from the view that tests are better if devised to find errors.

2.3.1 White-box testing

White-Box testing implies knowledge of the internal structure of the software being tested; using this information the engineer can devise tests to exercise the code to ensure that
the execution follows expected paths and that the expected results are obtained [27], [29], [49], [50], [63], [79], [80], [81], [83]. White-Box testing on an embedded target requires visibility of the execution; or alternatively, test stubs and drivers, which are verified to perform the expected tests and output the necessary verification results. Gross et al. [83] identify challenges when applying white-box test techniques to UML designs, if a sufficiently low-level of implementation detail is not available.

Algorithm analysis is a white-box activity concerned with checking that any computational or control algorithms not only function as intended, but also meet specific performance criteria such as execution time or precision. This is particularly important where algorithms are developed and tested on a desktop development environment, and then targeted at embedded systems; which may not have the same resolution or supporting hardware acceleration features.

2.3.2 Code coverage

Code coverage could be considered as falling under the category of white-box testing, but it is often listed as a separate requirement or metric to indicate how comprehensively testing has been conducted, it also requires specific data to be made available to support the conclusion that the required level of coverage has been achieved. The level of detailed employed in code coverage is dependent upon application and certification requirements and may include: statement coverage where every code statement (or line of code) is executed; decision/branch coverage where decision point or branch in the code is executed; condition coverage where each independent condition in the code is executed; or multiple-condition coverage where all possible combinations of conditions are executed [27], [28], [29], [49], [50], [63], [79], [80], [84], [85], [86]. The FDA software validation guidance document [49] identifies branching as one of the key aspects of software which contributes to its complexity and which may lead to undetected defects. Dupuy and Leveson [87] showed that modified condition/decision coverage (MC/DC) can detect errors not found with functional testing. Brosol [84] outlines some challenges in achieving code coverage and describes the use of execution trace data to verify MC/DC as required by DO-178B. Code coverage aims to show that all paths through the software are stimulated, and can also be used to drive hardware verification of SoC and FPGA platforms [88], [89].

2.3.3 Black-box / Functional testing

In contrast to White-Box testing, Black-Box testing implies no knowledge of the internal structures of the software code. Instead the software is tested so as to ensure that each component (unit, object, function, module, task, etc.) meets its stated requirements. Black-Box testing generally involves treating the component as an encapsulated unit with specified inputs and outputs, and devising a sufficient subset of test-cases to exercise the component
Since exhaustive testing is impractical in all but trivial cases, techniques such as equivalence partitioning, boundary-value analysis, cause-effect graphing, and error guessing are used to identify the subset of test-cases to be performed.

### 2.3.4 Performance profiling / Statistical testing

Performance profiling can be used to check that specified performance requirements for the software are met. Depending upon the application, embedded systems may have real-time response requirements, throughput requirements, or resource usage requirements; which are particularly important in embedded systems where resources are less abundant than in desktop environments. In addition, performance profiling can be used to drive optimizations by identifying fragments of the software which, although functionally correct, cause execution bottlenecks or consume excessive resources. Of course performance is also dependent upon the underlying hardware, and as architectures become more complex so does the challenge of verification.

Profiling information is typically obtained by instrumenting the software at points of interest, or by sampling of the key processor registers at regular intervals and performing statistical analysis on the data obtained. Patel and Rajawat provide a detailed survey of current profiling techniques targeted at embedded systems, particularly those intended for SoC or FPGA platforms.

### 2.3.5 Load / Stress analysis

Load or stress analysis is closely related to performance profiling, but involves testing the system at and/or beyond its expected normal operational conditions to ensure that there is sufficient margin for error; and that any system degradation is handled in an acceptable manner. In some cases the additional margin in a design is used to provide fault tolerance; when erroneous execution behaviour is detected the operations which failed are repeated.

### 2.3.6 Fault injection

Fault injection testing is necessary for high-reliability embedded systems designed to be fault-tolerant. The degree of fault tolerance required is dependent upon the application but the minimum requirement for many applications is that the system should continue to operate correctly under single fault conditions (or single upset conditions). For hardware fault scenarios which cannot ordinarily be created without causing physical damage to the system, it is preferable to use software simulations or dedicated fault injection features to mimic the fault and thereby evaluate the system response. In embedded systems, fault conditions may be hardware or software based; but both can be
equally difficult to investigate. Fidalgo et al. [99] describe the use of a modified On-Chip Debugger (OCD) module to insert fault behaviour into memory cells and registers so that performance of the fault-tolerance features can be tested. Portela-García et al. [100] describe a means of reusing OCD infrastructure in combination with a hardware based (FPGA) fault generation platform to enable fault injection and analysis on embedded processors.

2.3.7 Integration testing

Integration testing involves checking the behaviour of software components when they are combined to form larger elements in accordance with the architectural design of the final system. Individual software components may behave as expected when testing in isolation, but when combined, the interaction between components and the wider system may expose undetected errors or untested dependencies [27], [29], [49], [50], [63], [79], [80], [81], [83].

Figure 2 illustrates a generic embedded system which controls some form of plant. As shown, the unit level software components can have dependencies upon: other units, external inputs, system-level inputs, and may drive system-level outputs, or external outputs. Integration may be implemented in a non-incremental or, more usually, an incremental manner; and may follow a top-down or bottom-up approach, with the testing required reflecting the approach adopted.

![Figure 2: Generic system-level model of an embedded system](image)

2.3.8 Regression testing

Regression testing is required when a software modification is made to a previously tested system. The objective is to ensure that the modification, which may be due to a requirement change or bug fix, has not introduced a new bug nor caused other elements of the software or system to regress [29], [49], [50], [63], [79], [80], [101]. Depending upon the development methodology employed, modifications made to software during the early unit-level design phases may or may not require regression testing; in general, if the modified component has been integrated with the other software components, regression testing is required.
2.3.9 Requirements tracing

Requirements must be traced throughout the development process to ensure that high-level user/system requirements are correctly distilled into corresponding low-level requirements and implemented in the final software [49], [63], [80], [102]. When considering verification of any embedded system functional requirement, if the corresponding test doesn’t exist or if it can’t be shown to exist, then it is not possible to state that the requirement has been met. Requirements tracing is often considered to be a desk based activity, but it is an essential element of most quality systems.

2.3.10 Security testing

Security testing ensures that no unintended access mechanisms are left exposed in the final embedded system [29], [49], [50], [63], [79], [80]. For some systems, security concerns relate to unauthorised access to information within the system, in others the primary concern is alterations to the operation of the system; in safety-critical systems, such unauthorised access to the system could pose a life-threatening risk. For those systems where communications interfaces are integral to the design, the interfaces must be tested to ensure that no unexpected methods exist to interfere with the system. For those systems without communications interfaces it is necessary to ensure that any interfaces used during system development are fully secured or disabled [15].

2.3.11 System-level testing

System-level testing is concerned with ensuring that the entire system performs its intended function [28]; the consideration that the system can be used safely and effectively is also integral to this [49], [80], [81], [103]. EN 60601-1 [79] and EN 62304 [63] refer to the intended function of medical equipment as the ‘essential performance’ requirements. Testing performed at the system level includes validation testing, acceptance testing, and subjective tests, generally falling under the heading of usability testing. These user level factors are not the primary subject of this research, instead the system-level tests which are of concern here are those which much be performed to verify that the software requirements are met.

2.4 Emerging verification challenges

As microcontroller architectures become more complex so too do the challenges in verifying the software which executes on them. In addition to dedicated hardware acceleration functional blocks, modern MPSoCs include an increasing number of computational engines, and this trend is likely to continue [104]. Of course it is impossible to definitively say how many cores will be integrated into tomorrow’s ICs. However, it is clear that simply increasing the complexity of a single processor core, or increasing its operating frequency, are no longer the obvious choices when seeking greater performance [105], [106], [107], [108], [36], [109]. Agarwal and Levy [35] proposed the "Kill rule for multicore", which
states that for a percentage increase in core area there should be a corresponding increase in performance or else the trade-off is not worthwhile. This would seem to be a sensible metric to adopt, and as their paper suggests this would indicate that many processors of today are overly complex and that future IC designs are likely to comprise of multiple processor cores but with simpler architectures [110].

If the target platform contains only a single processor core, only a single thread of execution can be active at any instant [36]. That fact allows extrapolation of behaviour between captured execution points, typically branch locations, which is a usual method of dealing with the increasing volume of data generated as operating frequencies increase [111]. Where multiple processor cores are present there can be multiple threads of execution active simultaneously; thus, the volume of data which must be captured increases accordingly. Defining what constitutes a multi-processor architecture is not straightforward, even expert options on an unambiguous definition differ [14], [13]. However, the fact is that integrated circuits continue to become more complex. This trend not only complicates IC design activities but also the task of effectively using this additional computational capability [112].

2.4.1 Multicore architectures

For many years, discrete multi-processor designs have been deployed in high-performance embedded systems [113] and safety-critical embedded systems [114], often with two processors executing in lockstep [36], where the additional expense and complexity has been unavoidable. In more recent years, multicore SoCs and general purpose multicore microcontrollers have gained momentum [114], [6], [12], [7], [8], [9], [11], [10], [115]. In part this is due to technological improvements enabling more complex SoC designs and the filtering down of design and development advances made with multicore desktop processors; but multicore architectures also offer distinct benefits over discrete multi-processor designs such as lower system cost, greater processing efficiency, and simplified inter-processor communication [116]. Some indirect but highly desirable benefits may also accrue; for example in the automotive arena lower electrical power consumption facilitates the use of lighter wiring-looms, which not only reduces material costs but can in turn improve fuel-efficiency.

Today’s multicore SoC architectures are generally highly customised to meet the design requirements of the application [69], [117], [118], [119]. This is reflected in a multitude of devices, comprised of different CPU cores or IP blocks, each with unique features and capabilities. Due to the wide spectrum of architectures available it is not possible to precisely categorise each; however, as illustrated in Figure 3 it is possible to divide multicore devices into two broad high-level categories, homogenous and heterogeneous [70], [108], [120], [121]. Homogenous multicore refers to devices where several identical CPU cores are used;
this architecture is generally found in desktop processors or embedded processors designed to serve a wide range of general purpose applications. Heterogeneous multicore refers to devices where a number of different processors are integrated into a single device; the processors are chosen to meet the specific application requirements, which could for example include a digital signal processor (DSP) core, a graphics processing core (GPU), and/or a custom processing engine for other computationally intensive tasks.

![Homogenous multicore](image1)

![Heterogeneous multicore](image2)

*Figure 3: Hardware architectures for multicore devices*

From a software perspective, multicore architectures are generally presented as either: symmetric multi-processing (SMP), where the CPUs are treated as shared computing resources available to a single operating system (OS); or asymmetric multi-processing (AMP), where each processor runs a different OS, a separate instance of the OS, or without an OS (bare-metal) [36], [120], [122]. In some cases the architecture is complicated further by being a combination of AMP and SMP [123]. Heterogeneous architectures obviously lend themselves to an asymmetric multiprocessing paradigm, but as illustrated in Figure 4 homogenous architectures can also be organised with distinct OS instances or different operating systems.

![Symmetric multiprocessiong (SMP)](image3)

![Asymmetric multiprocessiong (AMP)](image4)

*Figure 4: Software architectures for multicore devices*

Clearly more processor cores present the potential for greater software performance, but engineers need to have the tools and skills to exploit this [124], [125]. And the performance benefits of multicore are not always easily quantified. The selection and benchmarking of various multicore architectures is dependent upon application requirements. Gal-On and Levy [126] provide an overview of the various architectural features and how these can impact
upon the performance of traditional benchmark suites. Their paper also outlines why new application specific benchmarks may be more applicable when comparing current and future multicore devices. Paul and Meyer [104] suggest that the widely used Amdahl’s Law is based upon homogenous architectures and needs to be revised to take account of heterogeneous architectures. In [127], this topic is also examined and Gunther’s Conjecture is presented, which highlights the need to account for the additional cost of synchronisation between processors.

In an ideal scenario it may be desirable to abstract software from hardware, thereby eliminating the need to consider the type or number of processor cores. Unfortunately this is not practical, and writing efficient software on multicore architectures is inherently more complex [23], [36], [125], [128]. Shared memory access on multiprocessor architectures is much more complex than with single core multi-threaded architectures [128]. Traditional locking mechanisms to provide exclusive access to shared data are difficult to design and verify when multiple processors must be considered [129], [130], and correct locking schemes may be computationally expensive. Local cache memory is often added to processors to minimise the impact of memory latency, in multicore designs the use of multiple caches introduces additional sharing and coherency challenges [36]. For some applications these challenges have led to the adoption of programming techniques such as transactional memory, whereby access to shared data is treated as a series of discrete transactions upon completion of which the update to memory is either committed or aborted [131], [132], [133], [134]. While multicore does bring complexity to multi-threaded software, Stewart [135] illustrates benefits such as the ability to allocate hard real-time tasks to a dedicated processor.

Although parallel programming techniques, using MPI [136], OpenMP [137], or Pthreads [138], have been successfully used multi-processor architectures for many years, these can be computationally expensive and may not fit naturally into embedded environments [70], [108], [109], [124], [139]. Gropp and Thakur [140] illustrate the overhead required for thread-safe MPI implementations, and difficulties encountered when multithreaded processes are considered. Hsiung et al. [125] present a model-driven development approach using the VERTAF/Multi-Core framework, which enables automatic thread/task code generation (utilizing Intel’s thread building blocks (TBB) library [141] and Quantum Framework middleware [142]) for embedded systems running Linux. Holt et al. [108] examine the difficulties in developing and debugging software for multicore architectures, and the lack of a standard programming model for asymmetric platforms. They highlight the fact that heterogeneous architectures pose particular challenges and propose the multicore communications API (MC API) as a more suitable inter-core communications mechanism for
embedded systems. However, the communications transport layer is not included within the MCAPI specification; instead, this is left for the implementer to define [143]. McCool [128] identifies the need to insulate programmers from creating race conditions and deadlocks when using multicore processors, and examines parallel programming models which are efficient and scalable. However the deadlock-free data-parallel models proposed are most applicable for scientific and graphics intensive applications rather than general purpose applications.

For the vast array of traditional embedded systems, which were not designed with multicore SoC architectures in mind, the transition to the multicore could be precarious. Cooling [144] outlines how embedded software which has been written with threading in mind may operate correctly in on a SMP platform; however, he also points out that it is unlikely that many existing designs fall into this fortuitous category; and even where such applications do exist the performance benefits might not be realised without making modifications. Cooling also shows how Hypervisor technology can enable AMP, whereby multiple operating systems can coexist on a multicore device. For software written in an OS which supports Hypervisor this may provide a route to exploit the benefits of multicore processors, but again this is likely to be just a small subset of embedded applications.

Ward-Foxton [145] suggests that the majority of embedded system designers will naturally adopt the AMP route first; but that in many cases the transition is made without hypervisor technology. Instead the design is simply manually partitioned and the multicore processor treated as several independent processors. However, while this approach may be workable for dual cores she does not see this as practical where more cores are available. Kopetz et al. [146] propose multicore as a suitable platform for the integration of automotive ECUs (similar to the IMA approach [36]), whereby existing distributed automotive control functions can be hosted on a single SoC with multiple distinct cores. They describe a model-based software design process which enables the distributed system and the communication between nodes to be modelled at a high level of abstraction; with the design then distilled down to software on the individual cores. However, rather than a shared memory architecture, Kopetz et al. advocate an architecture where each core operates from local memory only, which they suggest improves fault containment.

Multicore architectures introduce new challenges for verification of real-time embedded systems with increased potential for contention using shared bus [90], [136] or network-on-chip [109], [136], [147], [148], [149], [150] based interconnect. Such contention not only has the potential to cause bugs which are hard to detect and test for, but it also leads to unpredictable execution timing as discussed in the next section.
2.4.2 Non-deterministic execution

Lee [3] gives a clear description of why embedded systems are different from desktop applications; highlighting properties of embedded systems such as interaction with the physical world and the need for timeliness. Lee discusses various programming models (frameworks) used and the benefits and flaws of each; and emphasises the fact that the concept of time is often lost when considering programming in a general sense, and that modern processor architectures often hamper attempts to create deterministic systems. Younis et al. [71] also examine the need for temporal correctness in real-time embedded control systems and emphasise the challenge that increased integration poses for verification. Littlefield-Lawwill and Kinnan [151] stress the need for a strictly deterministic schedule in partitioned IMA systems, and examine how access to shared resources and managing asynchronous interrupts can impact upon this.

Multi-threaded software applications running on parallel architectures may not execute in a deterministic manner [23], [128], [130], [139], [152]. For some applications (such as data searches or sorts) the deterministic execution order of threads may not be of concern, once the correct result is achieved; whereas for other application scenarios such as mathematical calculations (e.g. summations) the execution order may determine the final result, which is obviously of concern. As suggested by Park et al. [130], recreating a software bug may not necessitate replaying one exact deterministic execution path leading to the error; and their ‘probabilistic replay via execution sketching’ (PRES) technique exploits this to speed the isolation and re-creation of concurrency related bugs on desktop multi-processor platforms. However, for real-time embedded systems deterministic execution is important for a variety of reasons.

Predictable execution timing is essential to enable feasible scheduling in real-time embedded systems. Determining the worst case execution time (WCET) for embedded code is a very complex task [153], [154], and unfortunately temporal properties of complex embedded processors cannot be verified by static analysis alone. Advanced features of modern processors (such as multi-level caches, speculative execution, out-of-order execution, branch prediction and complex pipelines) make the challenge of estimating a realistic WCET extremely difficult if not impossible [155], [156], [157]; and if possible, the resulting WCET timing is only valid for that processor [158]. The general solution to this dilemma is to create a conservative estimate for the WCET, which is then used in scheduling analysis.

An overly conservative estimate for WCET leads to poor processor utilization. Liu and Layland [159] examined the utilization achievable using fixed, dynamic, and mixed, scheduling algorithms. They show that high utilisation can be achieved when using dynamic or mixed scheduling algorithms; but when using fixed scheduling, with rate-monotonic
priority assignment, the utilization bound provided for a feasible schedule is only approximately 70% (unless the task request periods are exact multiples). Much research has been conducted into improving upon this utilization bound whilst ensuring schedule feasibility. Min-Allah et al. [160] highlight the challenges in trying to obtain an optimal task schedule. They propose a nonlinear constrained optimization technique which results in better utilisation and maintains feasibility.

Timing is not only influenced by hardware and software architectural features; it may also be impacted by system-level considerations such as thermal limitations or power consumption [161]. Dynamic voltage scaling (DVS) enables on-the-fly alteration of the operating voltage, and corresponding frequency, in modern processors. Min-Allah et al. [16] describe a technique to schedule tasks at the lowest possible feasible operational frequency rather than simply scheduling at the first feasible frequency; operating at a lower frequency, and corresponding lower voltage, enables a reduction in power consumption [17]. Min-Allah et al. first consider the application of this technique to single core architectures, but also show how it can be applied to optimise system performance of multicore architectures, when using a common frequency. Phatrapornnant and Pont [162] show that DVS can cause significant jitter in embedded systems; and propose an algorithm which reduces jitter for tasks with real-time deadlines whilst facilitating DVS. Tavares et al. [163] examine the formal modelling and verification of task schedules on an embedded platform which includes DVS capabilities. Santos et al. [94] consider a similar DVS scenario but include timeslots for fault recovery tasks.

Burns and Hayes [37] describe the complexity in defining temporal properties, and/or requirements, of complex systems precisely, and propose a timeband framework to better describe such temporal details. Devietti et al. [164] propose methods to make the execution of software on multi-processing systems deterministic; these decompose the software into quanta which can execute in parallel, and discrete points at which the quanta must be synchronised. Such a scheme can assist in verification and debug since snap-shots of the application/system state at the synchronisation points will be deterministic. Olszewski et al [152] propose a modified POSIX-based deterministic locking scheme for parallel/multithreaded applications, Kendo; by enforcing repeatable acquisition of locks the program output can be made repeatable, which greatly simplifies verification and debugging.

Verification and debugging of software which does not execute in a deterministic manner poses new and significant challenges. For example, it may not be possible to recreate an error if the exact execution order is not maintained (indeed the error may only exist during one particular execution sequence); and verification of such software may be problematic if data captured during executions runs is not consistent [152]. The difficulty in not having
inherently predictable execution timing, and challenges in accurately determining WCET
time, have led some authors to propose new architectures such as the precision-timed (PRET)
architecture [158], [165], [166]. Schoeberl [167] also examines the problem and proposes a
multiprocessor architecture based upon multiple JOPs.

2.4.3 Reliability of highly-integrated devices

Borkar [17] explains how technology scaling can enable lower power consumption and
higher operational frequencies. However, he also highlights the fact that the lower voltage
thresholds and gate capacitance of newer technologies increases the likelihood of soft-errors,
or single-event upset (SEU), occurring. And as Borkar explains, error correction or detection
hardware is usually confined to memory components; which overlooks the significant
potential for undetected errors in the CPU core and other logic components. Fuchsen [36]
explains how avionic platforms are made more resistant to SEU by the operation of multiple
processors in lock-step and including ECC for caches and registers; but suggests additional
redundancy may be needed for multicore devices which are even more sensitive to SEU.
Kopetz et al. [146] suggest that for embedded automotive applications using multicore
devices, the software could periodically store the state information for a core, the state could
then be restored in the event of a SEU.

Saha [168] describes how various disturbances can cause computational errors in
equipment without error-correction capabilities, and cites a 16-Mb DRAM as having a
possible failure rate of one per week. Saha proposes software-based fault detection schemes,
in preference to more expensive hardware methods. This approach relies upon
instrumentation of the application code and detection of corruption within this added code;
however, this is of questionable merit. Firstly, the probability of detecting a transient error is
proportional to the amount of instrumentation code added. Therefore applications where this
approach might be applicable would have both the spare processing time and code space
required, both of which are unlikely for inexpensive microprocessor based equipment.
Secondly, the likelihood of the instrumented code being active when a transient error occurs
is proportional to the frequency of the events and the amount of instrumented code. But for
frequent transient events it is as likely that the application code will be active when an error is
caused, and unless it fortuitously triggers the instrumented code, this error will go undetected

Soft errors can also be caused by subtle design flaws which may be very hard or
impractical to detect. Foster et al. [169] describe a corner case design flaw which lead to a
dead lock situation, but which was not triggered whenever the debug features were used.
Weiss and Hochberger [170] describe how reading an SoC memory cell while the timer had
an underflow and switched from 0x0000 to 0xFFFF led to a wrong value due to a local power
supply undershot. Vermeulen and Bakker [171] explain that the lower noise margin in low-
voltage processes makes the design more susceptible to: local voltage spikes and dips, temperature variations, crosstalk, and substrate noise.

Embedded systems, whether battery powered or operated from a generated power source, are typically designed to operate within strict power budgets [94], [15]. Although reducing the operating voltage and frequency has the potential to reduce power consumption [17], as previously stated lower voltage thresholds increase the likelihood of soft errors. In the main, this reduction in operating voltage is enabled by the use of smaller silicon device geometries, but using smaller geometries introduces the additional difficulty of managing in the resulting higher power densities.

Although ICs are generally considered to be reliable, once manufacturing tests have been successfully completed, ICs do suffer from wear-out effects such as electromigration [60]. Li et al. [18] suggest that “electromigration increases exponentially with temperature and reduces the life of products by four times”; and while IC designers do account for thermal effects and electromigration, these problems pose a much greater risk on modern ICs with smaller geometries and higher power densities. Software for safety-critical devices must therefore be designed robustly to mitigate these additional risks, which again increases the complexity of software verification.

2.4.4 Lack of visibility

Although integration enables designers to realise more sophisticated hardware architectures, these architectures do not necessarily facilitate easier functional silicon or software verification. Vermeulen and Goossens [172] identify the limited amount of real-time data that can be streamed off-chip or captured in on-chip buffers as a key factor limiting the debug in SoC designs; they also highlight the disadvantages of halting the device to aid debug. Their paper focuses on SoCs with multiple clock domains, which presents the additional challenge of capturing a globally consistent state.

More complex hardware supports more complex software applications; unfortunately access to the signals which are essential for tracing and monitoring software execution also become more deeply embedded as features are migrated on-chip [173]. This creates a significant dilemma for software verification because as the complexity of applications increases, visibility of real-time execution diminishes.

The option of simply making the necessary signals visible on I/O pins of the IC is not feasible because the number of I/O pins which a design can support does not scale in linearly with the number of core logic cells. I/O pads are necessarily larger than core logic cells, due to the larger driver circuits, power rails, ESD protection devices, and bonding pads; these features also make I/O pins relatively more expensive. Therefore IC designers aim to
minimise the number of pins. Increasing operational frequency of the IC core is an additional factor which opposes the concept of simply using I/O pins to extract real-time execution data; in many case it is no longer practical or economical to design I/O pins to support the bandwidth required [173]. The current technologies available to support functional verification at the chip level are examined in more detail in Chapter 4.

2.4.5 Security

In contrast to the challenges that lack of visibility poses for verification, it may appear that less visibility enhances security. Indeed providing access for verification activities and ensuring the security of a device are often considered to be conflicting requirements. However, the majority of embedded systems have some form of interface to the real world; therefore the challenge becomes one of ensuring that those interfaces can be verified to provide the required level of security.

Security concerns are not only relevant to safety-critical embedded systems; it is also a key requirement for embedded systems used in the financial sector. Karsai et al. [174] examine the challenges associated in verification of software on smart cards, where there is a desire to allow the card functionality to evolve, or be updated, whilst maintaining security of existing sensitive data. For business software applications, security is often accessed against three interrelated but often conflicting requirements - Confidentiality, Integrity and Accessibility (CIA); these requirements might equally be used for embedded systems.

Integration also enables new system-level platforms to be constructed. In the avionics sector the concept of Integrated Modular Avionics (IMA) has emerged as a means of sharing computing resources whilst achieving saving in terms of Space, Weight and Power (SWaP) [151]. In theory, if strict temporal and physical portioning can be achieved, disparate software modules, often with different safety requirements, can execute on a shared processing platform [71]. However, such shared platforms increase the risk that an errant malicious software application in one partition could compromise the security of a robust application in another; therefore more extensive verification is required to militate against such occurrences.

2.4.6 Debugging

If the aim of testing is to identify software errors or ‘bugs’ [175], then the aim of debugging is to identify the root-cause of the error so that the necessary corrective action can then be taken. Although both are closely related, debugging is a distinct activity from testing; not least because debugging begins with an apparent error, or bug, whereas testing does not. As noted by Grier [176], debugging is an empirical rather than a systematic activity, which though often taking up more than 50% of a project time-scale, is not a well-defined discipline commanding its own literature. Even the most authoritative publications tend to be sets of guidelines and check-lists [41], [44]. This is due to the nature of debugging: from observing
the machine's actual behaviour and comparing it to its hypothetical behaviour, debugging engineers must assess a model and adjust their ideas and code accordingly [176], [177].

Vermeulen [178] provides a comprehensive overview of current SoC 'functional' debug techniques including examples from a number of designs. This paper clearly identifies the primary challenges in designing on-chip debug support, not least the fact that it is not possible to extract the volume of data required to observe all activity on the chip in real time. Vermeulen also points to the additional challenges posed by multicore designs and the need to better define the criteria which provide the best debug support for these and future architectures. The expansion of multicore design techniques contributes greatly to this development problem, not only in terms of the increasing volume of data to analysed, but also the additional challenge of visualising software execution on parallel architectures in a comprehensible way, as long since described by Pancake [179]. Park et al. [130] examine the difficulty in simply recreating a concurrency related bug on multi-processor platforms without the need for an exhaustive number of reruns.

These comments apply to the debugging problem in general, but the process is even more difficult for real-time systems in which the system being observed cannot be halted to allow the debugger to check all details of the system's internal state [111], [180], [181]. If the system cannot be halted, this means that the debugger must attempt to find or reconstruct the problem behaviour from traced program execution and data transfer information captured from the processor while a system is running [92], [171], [182], [183], [184]; a similar real-time data capture process must be applied to debug first-silicon desktop processors [169]. This in turn means that huge quantities of data must be captured and stored at full system execution speed [181], [185]; but as Vermeulen and Goel [186] illustrate, the volume of on-chip data trace can easily become impossible to extract. Efforts to reduce the volumes of data transmitted off-chip have led to the use of various data compression techniques [111], [187], [188], [189], [190], [191]; taken along with the trace filtering capabilities provide by ARM [192] and similar solutions from other manufacturers, this represent the state-of-the-art in system tracing today.

When debugging on multi-processor SoC (MPSoc) targets, additional issues arise including questions such as: if one core has been halted, should all others be halted to preserve the relative state of the entire system [172]? When tracing program execution and data accesses from more than one core, how should the traced data be correlated and how can the relative timings of each core traced data be preserved [180]? How can data tracing across multiple clock domains be best accomplished [188]? How can communication channels between processors be debugged [193]? These questions and more, greatly complicate the landscape for developing debugging tools.
SoCs with multiple clock domains, or globally asynchronous locally synchronous (GALS) designs, introduce additional complexity when trying to capture the global state of the device. Vermeulen and Goossens [172], [194] examine the problem of capturing a globally consistent state in such designs and propose the CSAR (communication-centric, scan-based, abstraction-based, run/stop-based) debug approach. This approach halts execution at defined communication events rather than local or global clock events, which means that the global and local state captured is more consistent since the sampled data is not dependent upon the timing between asynchronous interfaces and local clocks. However, this approach requires debug to be performed in a run-stop manner with data being extracted over a standard IEEE 1149.1 interface.

Unfortunately, debugging at the system level is potentially even more intractable, given the less well-defined nature of system and software debugging tasks themselves [176]. Difficulties with system-level debug are compounded by the emergence of multicore designs and the inherent complexity of real-time systems [195], and unfortunately, many bugs do not appear when the software is executed in a simulator [171], this applies in particular to concurrency and timing related issues [165].

2.5 Verification by simulation and emulation

Simulation and emulation are closely related concepts, and the terms are sometimes used interchangeably. Even the distinction between the terms given in IEEE Std 610.12 1990 is at best subtle [45]; with the key concept being that emulation produces the same outputs as the system, whereas simulation produces results like the system.

“emulation. (1) A model that accepts the same inputs and produces the same outputs as a given system. See also: simulation.”

“simulation. (1) A model that behaves or operates like a given system when provided a set of controlled inputs. See also: emulation.
(2). The process of developing or using a model as in (1).”

In the past it may have been possible, and economically viable, to produce an emulation system which produced the same outputs as the target processor. However, as devices become more complex and IC design and fabrication costs escalate, this option is becoming increasingly impractical [19], [188], [196]; and as Zorian [197] explains, modern emulation solutions cannot replicate the exact behaviour of the final silicon. Instead, present day emulation systems have largely transformed into [22]: a) programmable hardware platforms which provide similar performance to the target, but with the benefit of greater adaptability, as described in this section; b) system-in-package or die extension solutions which can be removed from production parts to save cost, as described in Chapter 4; or c) low-cost OCI solutions which provide limited access to runtime behaviour, as described in Chapter 4. At the
same time simulation systems have progressed from purely software focused instruction set simulators to accurate system-level simulation tools which provide results that are now much closer to that of emulation systems. Therefore the distinction between the two is becoming even less clear.

When available, simulation environments can provide a useful platform to perform initial software verification and debugging [69], [22], [198], where the much greater visibility of the internal state of the simulation models is a significant benefit. Da Silva and Sanchez [96] show how fault injection and analysis can be performed using a SystemC transaction level model (TLM) of the LEON3 SPARC CPU. Hedde and Pétrot [199] use a simulation environment to gather trace data from multiple processors which can later be analysed. Yang et al. [200] describe their processor exception verification tool (PEVT), which automatically generates the hardware and software modules needed to enable verification of complex interrupt handling on microprocessors using RTL simulation. De Schultz et al. [201] show how the ArchC tool, which creates SystemC simulation models of processors, can be used to create disassembler and debugger tools. Kao et al. [191] highlight several deficiencies when tracing in a simulation environment including the disparities between the simulation and physical target, the relatively slow speed of simulation, and the fact that many simulators only trace program execution data. Patel and Rajawat [92] describe the benefits of using simulation for profiling but they also highlight the key disadvantage, which is the slow operation of accurate simulation. This problem of slow simulation is also identified by many other authors [97], [99], [150], [188], [198], [202], [203], [204], [205].

Where cycle accurate simulation is too slow, FPGA based prototypes or emulation systems are often used to accelerate the process [188]; and these platforms can also provide greater visibility than the final target. Zorian [197] outlines the benefits of using FPGAs to emulate SoC soft-cores, but highlights the fact that in most cases they cannot be used for hard-cores. Chuang et al. [204] describe a novel method of seeding a simulation environment with snapshot data acquired from an FPGA platform, thereby improving speed and visibility. Broscol [84] advocates an approach which combines an emulation platform (virtual machine) and hardware trace on a non-instrumented platform to achieve DO-178B test coverage requirements.

Gong and Lu [89] describe how verification of system-level hardware functions on a FPGA platform can be difficult when using a general-purpose operating system (GPOS). They propose a new verification purpose operating system (VPOS) which provides greater control and access when using software based test cases to test these system-level functions. As described, the test cases run on the FPGA include a self-checking facility whereby a signature value representing the system status at the end of the test is computed and
compared against a pre-computed signature from simulation data. For tests which fail, the test case is rerun in a simulation environment to debug the failure. In this way the FPGA is used to accelerate the running of long boundary-case tests, whereas the simulation is used to provide more detailed access when required.

The tools which enable debugging of designs on FPGA architectures often require the remapping of the design as target nodes change. This of course causes the behaviour to change which can either mask prior faults or introduce new fault conditions [206]. This probe problem is described by Schwalb et al. [207] who propose a method of selecting and accessing data from a node of interest which does not require reconfiguration of the FPGA. The approach described uses an on-chip processor to access the FPGA architectural fabric which enables readout of the flip-flop states. Unfortunately the implementation as presented is not sufficient for real-time systems; nonetheless it does illustrate the use of an on-chip processor to aid debug and diagnostics. Tombs et al. [206] describe how the UNSHADES system can be used to extract the internal state information from a FPGA; by using the Xilinx Capture macros their solution avoids the need to remap the design, but in order to extract information the design must be halted.

Engblom [208], [209] argues that virtual platforms address many of the challenges encountered when developing software on multicore architectures, or porting code to multicore. Such virtual platforms have many noteworthy features [210], including the ability to capture more complete snapshots and perform reverse debugging [211]. In [212], Engblom describes the uses of Windriver's 'Simics' system to capture snapshots of a running system. These snapshots can then be used to transfer bugs between developers and bug reporters, with the developer being able to replay the sequence of events which leads to the bug occurrence. However, Engblom's claim that software development does not require timing nor pin level accuracy in the simulation environment is at best a flawed generalisation. Of course there are systems where this may be the case, particularly where an OS completely insulates the software from the hardware, but in the context of real-time embedded systems where timing and asynchronously triggered I/O events are critical, this statement is not valid.

Alternatively, as explained by Sandmann et al. [213] simulation platforms can be used in model-based designs to generate an early abstract model of the system upon which various design solutions are explored; this model is then refined as the design evolves with increasing levels of detail. Emulation can also be used to evaluate the system-level behaviour of real-time control techniques. Ben Salem et al. [214] show how a real-time motor control application can be evaluated on an FPGA platform where the motor performance is emulated. This enables the control algorithms to be evaluated with different motor characteristics and optimised before connection to a live system.
A simulation technique, referred to as hardware-in-the-loop (HIL), can also be useful to verify or evaluate safety-critical embedded control modules when a full system test-bed is unavailable or impractical to construct. Not only can HIL simulation enable design exploration and mitigate the need for live-testing in the early development stages, but in some situations it would be hazardous or unethical to conduct the verification on a live system before having established a high-degree of confidence in the performance of the control module. Krákora and Hanzálek [215] describe a FPGA based HIL platform for testing embedded control units. Short and Pont [98] suggest that many HIL simulation platforms do not include sufficiently complete models, to include factors such as traffic flow, and propose an enhanced HIL simulator to evaluate safety-critical automotive control modules.

Kao et al. [20] show that in-circuit emulation (ICE) or debug need not be an exclusively hardware concept. They divide emulation into foreground and background elements and further subdivide these into software and hardware implementations. By applying these four combinations to an ARM processor based platform they are able to contrast the area and code-space overhead associated with each, against the relative performance of each approach. Their findings demonstrate that each solution could be applied successfully depending upon the application requirements. No performance data in terms of data throughput is provided, but reference is made to the fact that the JTAG interface used is slow relative to a parallel I/O interface. The two strategies suggested by the authors to improve ICE performance are: using a higher bandwidth interface with an efficient communications protocol; and storage of ICE operations on the IC thereby avoiding the overhead associated with sending these over the interface.

Hochberger and Weiss [183] describe the hidICE (hidden ICE) system which extracts only the embedded data and signals from peripheral devices on a SoC platform, which cannot be predicted, and use this to synchronise execution with a CPU emulator platform upon which all relevant signals and data can be monitored. The primary benefit of this arrangement is that there is no requirement to transfer vast volumes of data over a restricted I/O; however, it does presume that the peripheral data to be transferred occurs at much lower frequency than the CPU program data accesses. In addition a hash value is computed on the SoC platform and transferred to the emulator where it is compared to ensure both are operating in tandem; a difference in hash values indicates a difference in execution which then requires further investigation. Related publications by the authors [170], [85] show its applicability to a range of SoC platforms, but do highlight the synchronisation interface as a potential bottleneck. Backasch et al. [216] use the hidICE emulation platform to demonstrate its use in runtime verification, by adding novel hardware blocks which allow dynamic selection / reduction of the signals to be monitored.
2.6 Summary

Whether one considers software verification and validation as being interrelated or dissimilar activities, the fact remains that rigorous testing of software is a necessary step for any product where reliability or safety is desired. Existing standards and guidelines which consider safety-critical embedded systems tend to be focused on development processes and on the needs of a particular industry, often with differing recommendations. Alignment of best-practices and recommendations across industries might appear desirable; however, the recent introduction of ISO 26262 which is an industry specific adaption of the more general IEC 61508 standard, suggests that the opposite is the trend.

This means that no single set of methods exist for designers of ‘embedded-systems’ to follow. Some cohesion does exist in the higher-level concept of identifying the appropriate level of verification based upon the SIL of the device; and filtering the different standards down to the low-level dynamic software verification activities also shows some commonality. Although standards are undoubtedly of benefit and may be even necessary, or mandated, in certain industries, they do not specify in detail how to achieve the required level of software verification. Instead the primary focus of such standards is on best-practices for software design and construction; and as suggested by Meyer et al. [101], there appears to be a general disconnection between software construction and verification activities.

Unfortunately, modern multicore SoC architectures do not simplify the situation. Instead, as outlined, increased integration brings new challenges and compounds many existing ones. Of course integration enables new and more sophisticated devices, but this leads to greater complexity; and as highlighted by Leveson [57], this complexity, and our inability to deal with it, plays a significant role in the cause of errors. A key issue which magnifies the challenge of dealing with the complexity is the lack of visibility, which hampers both verification and subsequent debugging activities.

Verification of software using simulation or emulation platforms can certainly alleviate some of the difficulties created by these new architectures. This can be particularly useful in early development stages when the target hardware is not available or is not fully operational. However cycle-level simulation can be too slow and may not accurately reflect the subtle timing of the final target; plus the cost of building emulation systems has in many cases become prohibitive. In any case, for safety-critical embedded systems the software must also be verified on the target platform.

Of course, the ideal alternative is to construct software that is free from errors, which is the topic that Chapter 3 examines.
CHAPTER 3. CORRECT BY CONSTRUCTION

3.1 Introduction

This chapter examines some of the software development methodologies which are intended to enable software to be designed and implemented correctly therefore easing verification challenges; and techniques which are intended to facilitate runtime verification or monitoring.

Meyer [217] promotes the concept of Quality First, where the aim is to focus upon the quality of software as it is written, and to build functionality into the application incrementally. Ganssle [218] suggests that software developers should aim for pretty darn perfect code, if not perfect code. Havelund and Holzmann [219] accept that perfection in software is not feasible, and instead advocate in-house certification of code and software engineers to meet an ‘adequate’ standard. To help software engineers avoid problematic design practices and common coding errors use of coding guidelines are encouraged [27], [49], [219]; these can be in-house documents or recognised best-practice documents from external groups such as those produced by MISRA [74]. In addition, broad programming rule sets have been devised, for example those of Holzmann [220].

In recent years significant emphasis has been placed on the concept of developing software which is correct by construction. The objective is to minimise or ultimately eliminate the need for software verification, by eliminating errors which can be made during the software design phases. In the classic V-Model, as illustrated in Figure 1, design and implementation activities are considered to precede testing. Planning for verification should of course occur during the design phases but execution of tests is often left until implementation is complete or nearing completion. This approach is referred to by Grenning [25] as ‘Debug Later Programming’ and by Koopman [29] as debug-oriented testing, which presumes that errors are introduced during development which the developer or testers must later find and remove.

3.2 Test driven development

Grenning [25] advocates the use of the alternative test driven development (TDD), which proposes that the developer write test code for each software requirement first and only then should the software to implement that requirement be written. Adopting a philosophy akin to Myers [86], a central idea behind this approach is that the developer’s objective is to ensure that the test initially fails; and only passes when the software is functioning correctly. This TDD approach blurs the distinction between white-box and black-box testing as outlined earlier, since it requires the developer to have intimate knowledge of the internal working of the software, which is incrementally developed in parallel with the test cases, whilst the test cases, which are requirement driven, can be written from a black-box perspective.
A key benefit of the traditional V-Model approach is that the test and development are independent tasks and therefore can be performed by independent personnel; which is generally recommended as best practice; TDD makes separation of the test and development roles more difficult. Meyer [221] cautions that TDD should not be seen as a substitute for specification. However, the TDD approach does have the distinct advantage that testability of all code is inherent, whereas the V-Model approach gives no such confidence.

3.3 Component based development

PCB level hardware design involves combining components and ICs from various suppliers, each with distinct capabilities to achieve a working and hopefully reliable circuit; SoC design often takes a comparable approach with cores or IP blocks from various sources being combined to produce a more complex circuit. In a similar way the idea behind component based software development is that software applications could be developed more effectively by combining existing proven software elements, or trusted components. Meyer [67] describes a software ‘component’ as satisfying three conditions: 1) being usable by other software; 2) having a defined interface; and 3) not being restricted to specific clients.

Although this concept is compelling for many reasons, the biggest difficulty is achieving the required level of trust in these software components [67], [83], [222]. Although component based design has gained acceptance in general purpose business programming environments, most embedded software modules are not developed to be used as standalone or reusable components [217], [223]; embedded software interfaces may not be well defined, and software reuse is often limited to the applications using the same programming language, development tools, or even target. In [224], the authors identify unique characteristics of real-time systems, including temporal and resource constraints, which are not present in business processing systems. Although strict adherence to the component based development philosophy may not naturally fit resources constrained embedded systems, nonetheless the general concept of building trusted and reusable components has benefits for all software. An overview of several component based models, including those successfully used in embedded systems, is given in [225]; and many development methodologies seek not only to improve software quality but also its composability.

3.4 Design by contract

The design by contract (DbC) methodology for software development attempts to formalise the relationship between software components. This approach was proposed by Meyer, the designer of the Eiffel object-oriented programming language, as a means of improving reliability and supporting reuse which is central to object-oriented programming rational; but the methodology is not exclusively for object-oriented programming. Meyer [226] argues that defensive programming, which aims to handle all exceptional cases, results
in more complex, and often redundant, code which is therefore more difficult to test and verify and less reliable. Instead, the DbC approach is similar to standard business contracts where the software can be viewed as comprising of customer and supplier components; customers are obliged to satisfy pre-conditions before invoking the supplier component, and presuming the customer meets its obligations, suppliers are obliged to provide the service or function, satisfying post-conditions and invariants [226], [227]. In addition to improving reliability and reusability, DbC can be utilized to enhance testing [101], [228], [229], [230]. Hakonen et al. [61] show that the exploiting the presence of contracts can help to reduce the number of unit-level test cases required. In [231], Pei et al. demonstrate how source code which includes contracts can support automatic debugging and correction.

The benefit of contracts is of course dependent upon their completeness and effectiveness; Wei et al. [230] describe how simple contracts added by programmers can be automatically analysed and enhanced versions generated. Zhang et al. [232] show how source code assertions can be automatically added to model based designs, which include legacy code. Brunel et al. [233] describe how expressing properties, using assertions and assumptions, can enable implementers, working on different levels of a design, to adopt a DbC approach; with the checking of properties being consistent between modelling, simulation, and on the final target. Firesmith [227] compares design by contract and defensive development, and argues that defensive development, using assertions, is equally useful but more robust. Samek [142] advocates the use of assertions to detect those conditions which the software is unable to handle correctly, and that such assertions should be included in the released code. Assertions are undoubtedly suitable for detecting conditions where the software is used beyond its design specifications and may, as Samek contends, in some instances be more suitable than defensive programming approaches.

However, because the functionality of many embedded systems is ‘frozen’ at release time, the benefit of including those assertions which check static design properties is questionable. Inclusion of any assertions in released code is a matter of some debate; even Meyer [226] accepts that runtime monitoring of assertions may not always be necessary. Most C language implementations include a \#define to conditionally exclude assertion code fragments. Given the tight resource constraints of many embedded systems, the incentive to include assertions for any properties that cannot realistically be generated at runtime is greatly diminished. Gross et al. [83] emphasise the distinction between assertions and built in self-test capabilities. They contend that assertions are useful, but since software does not suffer from wear-out, once functional, the software component will continue to perform as expected; instead, they suggest that software self-test should focus on the dynamic environmental aspects which can influence the behaviour of the software component.
Software reuse is obviously desirable; it may therefore seem appropriate that all source code should include assertions for critical properties or capabilities, and if required, a mechanism to disable these assertions [220]. However, the use, or usefulness, of assertions at runtime hinges upon the runtime characteristics of the software. If the software in question is to be treated as a component within a larger software application, then assertions can provide a convenient and reliable means of ensuring that a violation of the capabilities of the component is detected, without requiring knowledge of the component internals. Alternatively if assertions are used to check properties which are dynamic, such as the checking of event queues in the QF [142], then this is an equally valid runtime use. However, the use of assertions to check static design properties at runtime is not as compelling; clearly checking static properties can be a convenient means of uncovering design flaws during early development phases but inclusion of such assertions in released embedded software is still of debatable benefit [220].

3.5 Model based development

The concept of model based development (MBD) (or model driven development) is to first capture the essential design requirements of a software application in a high-level model which can be executed or simulated to prove the design, but yet is often independent of the target platform [234]. This enables the application to be architected and iterated as required at a high-level of abstraction and only refined into a concrete implementation once the design is suitable. Not only does the high-level model provide a rapid development platform upon which alternative solutions can be evaluated, but it can also serve as a reference model, or oracle, against which the more detailed low-level implementations can be checked [93].

As described by Gross et al. [83] one benefit of a model based approach, using UML, for test and development is that the system can be described at high level of abstraction or low level, including detailed design information, using a single notation. This greatly eases the challenges traditionally associated with translating a design from requirements to implementation and the inherent risk that important features may be lost or misinterpreted in the process. MBD is also of particular benefit where different teams, or companies, work independently to develop aspects or elements of larger systems; as is typical for automotive electronic control modules. Di Guglielmo et al. [234] combine both a UML based model-driven design environment and model-driven validation tools to create an integrated and highly-automated design flow. Dubois et al. [102] show how requirements traceability can be achieved across different phases of a model-based design including verification and validation phases.

A number of commercial tool suites are available which support MBD for embedded systems such as: Stateflow & Simulink from MathWorks, VisualState from IAR, and Rational
Rhapsody from IBM. Samek [142] presents the Quantum Platform (QP), which is a platform for implementing real-time event-driven embedded systems based upon UML active object concepts, and which can be designed using UML statecharts. The platform includes a real-time framework and event processor which manage the developer’s event-driven application; these can be executed within the co-operative or pre-emptive kernels provided with the platform, or upon another developer selected RTOS or kernel; the platform also includes optional software tracing capabilities. A modelling tool [235] is also available, which can automatically generate framework compatible source code from graphical statecharts.

Samek distinguishes between providing a real-time framework and a traditional real-time toolkit or library, contending that the Quantum Framework (QF) represents an inversion of control whereby the user application is executed, or controlled, by the framework; as distinct from the toolkit approach where the user application calls functions and ultimately controls the execution. Samek suggests that this ‘inversion of control’ leads to a key benefit i.e. it provides a proven and reliable execution platform and the user can therefore focus on their application code; and advises that concerns regarding race conditions, deadlocks, non-determinism, priority inversion, and starvation can be eliminated by following two simple, but quite significant, rules: 1) sharing of resources and memory is not allowed; and 2) blocking and waiting are not allowed, instead all functions must run-to-completion (RTC). For applications where sharing is required (for example when using the pre-emptive kernel) support is provided for priority-ceiling mutexes and context storage in a similar manner to a traditional RTOS.

Hsiung et al. [125] present a model-driven development framework which augments QF with the ability to automatically generate code using Intel’s TBB library [141] for elements which can be executed in parallel on a multicore architecture. Bhatt et al. [236] suggest that model based development for embedded systems is more focused on the creation of high-level simulations of control models and the auto-generation of the corresponding target-level code and test cases. They compare the traditional approach to software development and MBD, highlighting areas where MBD may be less robust than traditional developer driven review processes. In particular they highlight the need for accurate models and simulation tools, and potential hazards posed by automatic code generation, not least the fact that it distances the developer from the actual target code.

Kopetz et al. [146] describe the construction of a high-level executable platform independent model (PIM) upon which the independent sub-systems typically found in automotive applications can be created. Using this approach, high-level behavioural models of the components (expressed using C, Java, or Stateflow) are created and can be executed. These high-level models focus on the communications and corresponding interfaces between
components and allow the sub-system design to be checked without consideration of the target platform. These models are then translated into platform specific models (PSM) which include target specific design details.

3.6 Formal methods for verification

In a manner akin to mathematical proofs, the concept of formal verification is that if a piece of software can be proven to be correct by design then it must meet its requirements/specifications for all use cases. Formal verification techniques fall into two broad categories: theorem proving; and model checking. A detailed description of these techniques is beyond the scope of this research; overviews of these topics and background information can be found in [237], [238], [4], [239], [2], [240], [241], [242], [243]. In practice, formal verification by theorem proving is applied only in limited scenarios as it requires considerable human intervention and expertise to arrive at a definitive proof; model checking, which involves testing of a model against a formal specification, can be more easily automated and is more widely adopted [243].

Formal and semiformal verification methods have been applied in hardware design flows for many years [202], [233], [237]. Clarke et al. [244] describe a tool which uses Bounded Model Checking (BMC) to verify an ANSI-C model of a circuit and enables comparison against the HDL implementation. Bhadra et al. [245] provide a comprehensive review of verification techniques combining formal and informal methods. A key challenge in using model checking is state-space explosion [59], [233], [243], [246]; where the number of possible software states to be checked expands to unmanageable levels, typically resulting in memory and/or time limits being exceeded. Holzmann et al. [247] show how dividing the model checking task into a number of parallel executions on separate CPUs can help to address the time limit issues which exhaustive checks can encounter. Kühne et al. [248] used BMC to verify a simple shared-memory locking algorithm and parallel multiplication algorithm on a SystemC implementation of a dual RISC architecture.

In [249], Bhadra et al. outline some of the challenges faced in validating multi-processor SoCs, not least the fact that some bugs can only be found by dynamic testing. They argue that formal proofs can be used for smaller blocks but do not scale to the system level, and propose a scheme whereby trace data is extracted and compared against expected results generated from an abstract model. In general, hardware verification is applied to simulation models at varying degrees of abstraction, with formal methods being applied to small discrete elements or very abstract models, and bounded semiformal methods being applied to larger components or more detailed models [250], [251].

This approach fits naturally with current SoC design methodologies; Cai and Gajski [252] describe how transaction-level models (TLMs) are used to refine a SoC design from
specification to implementation. At the specification stage high-level abstract models enable the rapid exploration of alternative architectural design options. As the design process advances, the models for the various system modules are replaced by models with increasing accuracy, until the final cycle-accurate implementation model is in place. The models used during the implementation phase are obviously slower to simulate as they must accurately represent the final design behaviour in both functional and timing aspects. Nonetheless the ability to iteratively model an SoC in this manner, speeds the development process and reduces the volume of data which needs to be captured and analysed in the early design phases, and enables the designer to examine the system with the appropriate level of abstraction at each design phase. Of course the ideal solution is to have TLMs which are both accurate and fast to simulate, which is what van Moll et al. [203] aim to provide.

Leveson [57] identifies invalid or incomplete specifications as a key source of software flaws in safety-critical systems, and proposed the Safeware methodology for software; where the specification of the system is modelled as an executable state-machine which can then be formally analysed. Ball and Rajamani [34] describe the use of the SLAM toolkit to check properties of a C program. The C program is first automatically instrumented to enable checking of the property of interest (properties are written in SLIC [253]) and the instrumented code is then abstracted into a boolean program and checked using a model checker. If an error path is found the result is first checked to ensure that it is a feasible execution path, if not the model is updated and the process is repeated. Chaki et al. [254] present the MAGIC tool for verifying safety specifications of C programs, but highlight the need to derive a suitable small set of predicates to facilitate verification. Ivancic et al. [255] provide a description of model checking C programs; including detailed examples of how C code must be refined and translated into abstract model formats to enable checking with various tools. The challenge in applying model checking to large programs is also highlighted, with one example containing 1652 lines of code being described as non-trivial.

Di Guglielmo et al. [256] describe the use of assertion-based verification (ABV) in simulations of embedded software designs. An essential element of this approach it that the software is developed using model driven design techniques; this enables assertion checks to be performed when the model has reached a valid state, greatly reducing the number of false results, and the time taken. Straunstrup et al. [246] examined feasible methods of performing exhaustive verification of state reachability on designs created using Statecharts. Pajic et al. [257] also use a model driven design approach to develop C code for an implantable pacemaker. In this case the authors first used an UPPAAL model to verify the system-level design; this model is then translated into a model for Simulink/Stateflow, using a custom tool (UPP2SF); and from Simulink, C code for the embedded target is automatically generated.
While the final design does meet its intended requirements, the authors acknowledge that proof of correctness of the translation tool has yet to be completed.

Holzmann [258], [259] showed that automatic verification model extraction could be applied to significant portions of C code, with the remaining portions of the model being handwritten. Yeung and Schneider [260] provide a worked example using CSP to formally verify a DSP filter design. Their example highlights the fact that the design rather than the implementation is verified; and the limitation of using model checkers for designs involving complex functions such as floating-point maths. Holzmann and Joshi [261] show that the application C code can be included in SPIN verification models, but illustrate the continuing need to abstract as much detail as possible to make verification feasible. Cordeiro et al. [5] compare the use of three static verification tools for ANSI-C for an embedded medical device application. They describe the modifications needed for each tool and show that the results obtained from each tool varied. They also highlight the limitations of the tools in verifying system-level hardware-depandan code. In later work Corderio et al. [262] show how support for common embedded C data types and fixed-point arithmetic can be added to a SMT-based BMC tool. Behrend et al. [263] propose a hybrid simulation and formal verification approach for embedded software, which the authors suggest can produce more complete results than other approaches. In this instance the simulation is based upon a SystemC model of the software, and the formal verification is performed on a pre-processed copy of the code with inserted assert/assume statements.

Like programming, creating correct models is a difficult task and prone to human error [80]. Abrial et al. [39] detail the practical difficulties in using several existing modelling tools and the need for tools to support users in creating correct models. They describe Event-B as a system-level modelling language, and present Rodin as a tool to support creation and verification of Event-B models. Abrial et al. aim to make the model creation task more manageable by operating Rodin in the background while the models are incrementally developed; likening reasoning about models, or analysing models, to running a program, whereby the model must be stimulated to check correctness. Cousot and Cousot [264] identify many of the difficulties in applying such abstractions to embedded software. In later work [265], Cousot et al. present the ASTRÉE formal static analysis tool, the purpose of which is to prove the absence of runtime errors in C code. Visser et al. [266] also recognised the problems in applying verification to an abstract model; they present a detailed description of the Java PathFinder environment which enables verification of Java source code.

Feasible modelling of application software not only involves abstraction of the application implementation details but also abstraction from the operating environment. Alternatively certain assumptions may be made regarding the capabilities of the operating environment;
but, for real-time embedded systems these assumptions may not be well founded. Naeser and Lundqvist [267] address this concern by proposing a hardware based operating systems which is more feasible to formally verify. Xie et al. [268] present a co-verification approach for component based embedded systems. However, the vast majority of embedded systems are designed on commercial-off-the-shelf (COTS) hardware often using general purpose operating systems, requiring a more universal solution. Including accurate models of operating systems is not practical yet; although, Chaki and Gurfinkel [269] argue that the restrictions imposed by real-time embedded systems may help in developing formal verification tools for this domain.

Schlich and Kowalewski [270] highlight the potential for discrepancies between C code and executable, and the absence of platform dependant hardware information in many models; they presented the [mc]square model checker which instead operates on assembly language. Schlich [59] later details various abstraction techniques and their impact when model checking a number of applications. In [271], Schlich et al. demonstrate two simulation approaches, a simulator for their target microcontroller (ATMEL ATmega), and a virtual machine based platform. However, the suggestion that generation of suitable virtual machine models for other microcontrollers is possible using available documentation may be naive. Yang et al. [200] show that an academic implementation of an ARM7 core, involving over five years of development and successful use in a SoC, had latent bugs. And Reinbacher et al. [272] highlight the fact that data sources available to software engineers, when developing a simulator for [mc]square, may not be complete. Reinbacher et al. agree that model checking is needed to meet verification challenges as the complexity of embedded systems grows. Using the [mc]square model checker, they demonstrate how some hard to detect faults in embedded C code can be identified, and how some state-space issues can be addressed. However, they also concede that more complex systems involving real-time operating systems are impractical [273]. Mercer and Jones [274] present the Estes model checker which can check assembly code for the 68HC11 on the target hardware. Using the GNU debugger, the authors construct the state space of the program to be checked on the microcontroller and after execution the target for the required amount of time, using breakpoints or single stepping to halt execution, extract updated state information for their state model.

Great effort has also been put into investigating methods which would allow the formal verification of software. Although this approach has been shown to be possible for certain instances it has not yet been possible to develop a methodology for general purpose embedded software applications [52], [222]. The need to use models or abstractions of the software and hardware in verification or simulation environments limits the efficacy of formal verification in embedded applications. The reluctance of software engineers to adopt
formal techniques also relates to a lack of familiarity with the languages and tools used; as shown by Yoo et al. [275] greater attention must be paid to presenting these tools in a more acceptable format. Formal verification can undoubtedly help to minimise errors, but since many subtle errors only occur when executing the actual software upon the target hardware, functional testing is still a necessary step [93], [97], [217], [241], [275].

3.7 Runtime monitors / Runtime verification

Despite our best efforts to ensure that embedded systems are designed correctly, it is generally accepted that errors still occur at the system level which cannot be predicted, or detected by testing [52], [57], [272], [33]. Therefore, in high-reliability systems it is common to include hardware or software elements, referred to as monitors, to ensure the correct runtime operation of the system. Software based monitors are closely related to software instrumentation added for debug purposes, the key differentiation is that monitors are intended to remain active in the deployed system. Hardware devices include simple watchdog timers which reset the system if the software fails to generate a signal within a specified period of time (indicating erroneous execution); but more sophisticated techniques to verify correct behaviour, or monitor real-time resource usage [33], are increasingly used in embedded systems. Watterson and Heffernan [276] provide a comprehensive review of software, hardware, and hybrid monitors for embedded systems with an emphasis on runtime verification.

Goldberg and Havelund [277] propose using EAGLE to monitor and perform runtime verification on instrumented Java programs. In [278], Goldberg and Horvath propose the use of a model based monitor to enhance fault protection in avionics software designed to meet the requirements of the ARINC 653 specification [279]. Later Havelund [68] presents the RMOR (Requirement Monitoring and Recovery) framework which instruments C code to enable runtime verification using a software monitor. Lee et al. [280] described the Monitoring and Checking (MaC) framework which enabled formal specification of requirements and runtime monitoring of these using code instrumentation. In later work the authors propose the Java-MaC architecture which is focused upon applications using the Java language [281].

Watterson and Heffernan [282] demonstrate the use of the Java-Mac runtime verification system to provide a minimally invasive method of verifying a Java Optimised Processor (JOP) embedded within an FPGA. Despite using a simple application example this framework demonstrates the complexity in extracting the required data from an embedded system at runtime. However, this framework also points towards an interesting technique which could be applied to multicore devices whereby the core under examination could be instrumented to a minimal level thus maintaining determinism, whilst a secondary heavily
instrumented core could be made to replicate the execution behaviour.

As outlined by Zorian [197], embedded processors often include a software debug monitor to facilitate basic features such as firmware download and limited software debug, usually communicating with a host platform via a terminal-like serial interface. Scottow et al. [283] demonstrate how an invasive software monitor can be used in certain situations to extract performance related data with acceptable overhead. However the paper also highlights the problems associated with using software monitors particularly in real-time systems where the overhead can impact upon determinism. Šimša et al. [284] describe the dBug tool which is used to check safety properties of POSIX-compliant multi-threaded applications. By intercepting and controlling the start of threads the tool is able to create schedules which exercise the application threads in a methodical manner to identify potential runtime assertions, deadlocks, conflicting non-reentrant functions, or system abort.

For hybrid SoC designs (containing both hard and reconfigurable elements) Hopkins and McDonald-Maier [285] highlight how traditional techniques to monitor registers within the reconfigurable elements can impact upon the system behaviour and propose an enhancement to the reconfigurable circuitry fabric to enable the registers to be memory mapped. Larsson et al. [286] describe on-chip monitor hardware which is capable of detecting the occurrence of an incorrect read/write sequence due to the presence of a race condition when two CPUs access shared memories residing on-chip and off-chip. The monitor hardware described involves a bus interface, address/data matching circuits and a state machine which can be reprogrammed in-circuit to monitor different potential race conditions. Bartzoudis et al. [287] describe an error detection monitor unit (EDMU) intended to prevent erroneous, or malicious, PCI bus accesses on high reliability workstations. In [60], Burgess et al. present a software based approach, which requires no test infrastructure, to detecting faults that may occur in network-on-chip architectures long after post-production testing. This approach not only enables in-field diagnosis of faults but can also facilitate a degree of fault-tolerance if the on-chip data can be rerouted to account for the detected defects.

Backasch et al. [216] non-invasively extract trace data from a running system, to an emulation platform, where a reconfigurable verification engine then checks the propositions. Brunel et al. [233] use Logic Of Constraints (LOC) to monitor properties in simulation of models and to perform runtime checking; their approach also allows for off-line checking from simulation or runtime trace data, but as the authors explain, runtime trace is limited by available hardware resources. Krákora and Hanzálek [215] describe a FPGA based hardware-in-the-loop (HIL) platform for testing embedded control units. However, in this case, a simulation model is created for the system level, rather than the embedded target; and the test platform uses this system-level model to exercise the controller and verify the desired
Heffernan et al. [288] describe an on-chip SoC monitor which can be used for runtime verification purposes. Two example control applications demonstrate how the monitor can ensure that the state transitions of the applications occur within prescribed time limits. For the monitor solution proposed, the constraints to be verified are established prior to SoC synthesis and feed the design tools to automatically generate the appropriate circuitry. Reinbacher et al. [289] propose a similar runtime verification approach for microcontrollers. Such runtime verification solutions enable non-invasive, or minimally invasive, monitoring; but the fact that the hardware synthesis is specific to a set of constraints or assertions determined at design time, limits their applicability to more general software verification situations.

To address this difficulty Reinbacher et al. [290] present an integrated solution consisting of hardware assertion checkers and RISC engine; and later include a real-time clock to enable temporal properties to be checked [291]. The assertion checkers are configured to monitor data variables of interest which are evaluated against past-time linear temporal logic (ptLTL) claims. Using the RISC engine adds the flexibility to process different assertions without the need to resynthesize, but increasing the number of assertions changes the checking logic and does require synthesis. However, the architecture as described requires access to the data/program interface of the microcontroller; although this should not pose a difficulty in SoC designs into which this block might be integrated, it does somewhat limit its application. In [292], MacNamee and Heffernan had suggested a similar approach, but in that case, using debugging hardware and/or on-chip co-processor to implement requirements-based monitors in real-time embedded systems. Not only does use of a co-processor minimising the runtime impact upon the main processor, but if analysis can be performed on-chip this also reduces the I/O bandwidth requirements.

### 3.8 Summary

The desire to create software which is free from defects has driven significant advances in the manner in which software development is approached. Clearly software engineers must be adequately trained, and their striving for perfect or nearly perfect code is laudable; but the reality is that to-date no method has not been devised which enables software engineers to write code which is guaranteed to be free from errors. Use of coding guidelines and rules are of help, but their existence recognises the fact that implementation of software is at best imperfect. Therefore testing is a still a vital step in building confidence in correctness of our software.

The concept of that TDD is useful in that it elevates the importance test-case creation, and encourages engineers to first consider the conditions under which the software will be
deemed to have failed or be working correctly; but it does not negate the need for robust
design or system-level verification. Greening [25] shows how test-driven development
techniques can be applied to embedded C code, with the majority of coding activity taking
place in desktop environment. However, he also highlights the risk associated with ‘dual-
target testing’ due to the differences or incompatibilities between development tools and the
platforms.

Component based development and the related idea of design by contract may enable
engineers to build complex systems using trusted components; but they rely upon the
availability of these trusted components. The design by contract approach promotes the use of
pre-conditions, which provide a convenient means of identifying situations where the
component is incorrectly used, and post-conditions and invariants, which aim to identify
cases where the component fails to perform its expected function or causes unexpected side-
effects. The effectiveness of these checks is of course dependent upon the ability of software
engineers to write robust assertions, and inclusion of these assertions in the source code is of
course intrusive and in some runtime verification situations may be of questionable merit.
However, the inclusion of these assertions does provide a convenient mechanism to aid
automated test generation.

Model based development enables engineers to create an initial high-level abstract model
for a complex system and to verify the model, and refine it if necessary. The abstraction of
the design from the final implementation allows architectural design phases to be completed
more efficiently and provides flexibility between the design and implementation phases. This
approach can also provide a convenient means to share design information between disparate
design and implementation teams. Similarly formal methods for verification, and model
checking in particular, exploit the idea of creating an abstraction of the system and using this
to perform the verification. However, creating complete and accurate models is still a
significant challenge and these abstractions must ultimately be refined into a concrete
implementation and this too must be verified; preferably on the target platform.

Runtime monitoring and verification methods aim to address this need. By adding
monitoring hardware to the target or instrumenting the software, the runtime behaviour of the
system can be observed. However, adding additional logic may not be feasible, and as
discussed in the next chapter software instrumentation can be intrusive. Whether the
verification to be performed is traditional informal testing or verification based upon formal
methods, a critical factor limiting the feasibility of runtime verification is the lack of visibility
into execution on modern highly-integrated architectures.
CHAPTER 4. SUPPORT FOR VERIFICATION AT THE CHIP LEVEL

4.1 Introduction

As integrated circuits have evolved, so too have the tools and techniques used to test them, but not necessarily at the same pace or in the same manner. Many circuit level components have moved to become on-chip components, reducing costs and increasing functionality, but as discussed in Chapter 2, greater integration has the distinct disadvantage of less visibility. Consequently this dramatic increase in system integration and complexity driven by Moore's Law, and the proliferation SoC development techniques, have led to increasing silicon test, verification, and debug difficulties [169], [197], [293], [294].

Although this research is primarily focused on the challenges associated with software verification, because the same interfaces are often used for silicon and software purposes (verification, test, and debug), it is useful to consider the current state-of-the-art and expected advancements pertaining to both. While there are many commercial and research tools available to assist with target-level software development and debug, it is in fact the OCI that dictates what can be achieved with all tools.

This chapter provides an overview of these on-chip features and techniques, which as illustrated in Figure 5, include standard based and proprietary solutions, on-chip interfaces and I/O interfaces, plus software based and hardware based solutions. A more comprehensive technical description of the standard based interfaces and several proprietary OCI solutions can be found in a book by Stollon [198].

Figure 5: Technologies for on-chip verification in modern SoC designs
4.2 Reuse of silicon test interfaces

In a manner similar to software verification, silicon verification (often referred to as post-silicon validation) is an exhaustive activity which is performed on prototype or 'first silicon' parts when the design is first fabricated, to identify bugs in the design [175] and ensure that the IC functions in accordance with its specifications. Generally, the operation of the entire IC will be checked by comparing captured data from a physical device against simulation data gathered during the earlier design phases. This verification may also be repeated over a range of operating conditions and corner-cases.

Silicon debug is the process of identifying the root cause for any device failures found, which may be due to: design, manufacturing, or software / configuration issues. Vermeulen and Goel [186] describe the debugging process and a typical IEEE 1149.1 based on-chip debug solution. Ko and Nicolici [295] highlight the fact that the signals which can be monitored with on-chip resources are limited, and emphasise how critical the selection of traced signals is to the feasibility of identifying functional hardware bugs. Vermeulen et al. [296] define the principal requirements for breakpoint hardware when used for silicon debug. Although their paper is focused on developing a language and tool to automate hardware breakpoint generation for silicon debug, the requirements identified (signal observation, event combination, event sequencing, event counting, programmability) are also relevant in software debug.

Once the silicon verification and debug is completed, the fabricated silicon design is taken to be reproducible; therefore complex on-chip verification hardware is no longer needed. In some cases these circuits can be left unconnected or removed to reduce costs [22]; but the key point is that from a silicon verification and debug perspective, the motivation is to minimise the on-chip circuits required as these will not be essential in volume production.

In contrast to verification, silicon test is typically carried out on every IC manufactured to eliminate those parts which have manufacturing defects. For digital circuits this process involves stimulating the device with test patterns and checking the results against expected values. Problems in manufacturing testing which are well-documented [169], [197], [293], are being addressed through new Design for Test (DfT) techniques and SoC test disciplines, such as the IEEE 1500 Standard for Embedded Core Test [297], IEEE Standard 1149.7-2009 [298] and IEEE P1687 [299], [300]. Coordination between the activities of the various groups working on standardization of on-chip and external test interfaces is on-going [301].

Since silicon test must be carried out on each device manufactured, the on-chip silicon test hardware and test interfaces are generally accessible on the final packaged device. Consequently, it seems a very attractive proposition to reuse this ‘test interface’ for functional silicon debug, board-level testing, device programming [302], plus software verification and
debug. Therefore in addition to circuits intended for verification and debug the following sections also examine the use of standard, and proposed, silicon test interfaces.

### 4.3 I/O interfaces

The IC level I/O interface can be viewed as the first layer between the buried signals and the external debug environment. This interface is critical in determining the available data transmission bandwidth. The following sections give a brief overview of the principal interfaces currently used for test and debug purposes.

#### 4.3.1 IEEE 1149.1 (JTAG)

The IEEE 1149.1 “IEEE Standard Test Access Port and Boundary-Scan Architecture” [303] is the principal method for board level interconnection testing of complex digital circuits; Bennetts [304] provides a tutorial describing the use of JTAG and associated standards for board level testing. As illustrated in Figure 6 the standard defines four mandatory signals: Test Clock (TCK), Test Mode Select (TMS), Test Data Input (TDI) and Test Data Output (TDO), plus one optional signal Test Reset (TRST*). All mandatory signals are operated synchronous to the TCK signal. The TMS signal is used to control the on-chip Test Access Port (TAP) controller which consists of a simple state machine that determines the test mode for the IC. The standard also provides a certain degree of flexibility by facilitating proprietary test instructions.

TDI and TDO signals are arranged such that data can be serially shifted into and out of each IC (with TCK being the serial clock signal). Several ICs on a PCB can then be arranged such that they form scan-chains, thus minimising the number of signal tracks and I/O pins needed to transfer data. While this scan-chain arrangement minimises the number of pins required to test at board-level it can also become a limitation if these scan-chains become excessively long. A further significant limitation is the fact that the standard envisages only a single chip-level Test Access Port (TAP). For ICs containing multiple cores, the chip-level test architectures employed often result in test interfaces that are no longer fully compliant with the original standard [305].

![IEEE 1149.1 interface signals](image)

**Figure 6: IEEE 1149.1 interface signals**

Although intended primarily as a board level I/O test standard, IEEE 1149.1 has been habitually modified to include debug capabilities [100], [184], [187], [193], [196], [306],
Georgiev et al. [310] described how JTAG based access can enable debugging of LINUX kernel modules. This is seen as particularly useful in situations where drivers for other interfaces (serial, USB, or network based) have not been loaded and therefore traditional printk and KGDB debug approaches will not work.

However, since this standard was devised for performing I/O level test, with a minimal number of test pins being a primary motivation, the resulting interface provides limited I/O bandwidth [196]. Whetsel [311] describes a means of reducing the number of test pins further by using a parallel to serial conversion and bidirectional transceiver; and proposes making individual ICs addressable, avoiding the need to form long scan-chains. Vermeulen et al. [312] provide an overview of several approaches to accommodate multiple cores and propose an architecture which maintains compliance with the standard. However, despite providing a comprehensive overview of the issues when debugging multiple cores, and including details of the experimental setup used to demonstrate the viability of concurrent debugging of multiple cores, the paper does not address the issue of limited I/O bandwidth. But as noted by Stollon [198] this interface was “never designed to support any real-time analysis”.

### 4.3.2 IEEE 1149.7

The more recent IEEE 1149.7 “IEEE Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture” [298] aims to address some of the limitations posed by 1149.1 whilst maintaining compatibility with it. In addition to allowing serial connections of digital circuits as previously illustrated in Figure 6 (for ICs and on-chip cores), the 1149.7 standard also facilitates the interconnection of circuits in a two-wire or four-wire star configuration as illustrated by Figure 7; the Test Reset (nTRST) signal is again optional and therefore not generally counted.

![Figure 7: IEEE 1149.7 interface signals](image)

As described by Ley [313] this standard is of particular benefit where SoC or System-in-Package (SiP) designs are concerned, and many capabilities have been added to enhance debug. Undoubtedly, many of these enhancements do ease the integration challenges posed by SoC and SiP designs: the potential to reduce I/O requirements to just two pins; the ability
to use a star topology; the concept of 1149.7 being an 'adaptor' for existing 1149.1 TAPs; and the introduction of hierarchy.

Regrettably, the benefits for debug purposes are not as obvious. Clearly the simplification of the architectural complexity and the potential to have reduced scan-chain lengths eases access to each core, which must be of benefit. Provision for the use of idle states to transmit background data is certainly aimed at debug needs. However, when simultaneous access to multiple cores is needed, the critical limiting factor is I/O bandwidth. To facilitate a four-wire configuration the TDO pin must be capable of being controlled such that contention is avoided; and for two-wire configurations the TMS(C) must support bidirectional data transfers, the resulting interleaving of control and data signals onto a single pin is unlikely to increase bandwidth.

4.3.3 IEEE-ISTO 5001 – 2003 (Nexus)

The Nexus standard [189] was developed with the primary aim of addressing the challenges encountered when debugging real-time embedded control applications. The standard defines a development interface, as illustrated in Figure 8, a messaging protocol, and a comprehensive set of features (breakpoints/watchpoints; nexus recommended registers (NRR); standard API; program, data, and ownership trace; read/write access to mapped resources; time-stamping; data acquisition; port replacement and/or port sharing; memory substitution) many of which are optional.

![Figure 8: Nexus development interface](image)

The standard allows for four capability classes with each class supporting different features. Class 1 uses the IEEE 1149.1 interface (with optional addition of a RDY* signal), whereas Classes 2, 3 and 4 complement this by adding an optional auxiliary (AUX) port the size of which scales according to the capability required. With real-time control being a key consideration, the AUX port is the primary differentiator in Nexus as it directly targets the I/O bandwidth limitations of other solutions [184], [188].

Another important feature of Nexus is the option of providing real-time access to on-chip memory; this not only enables capabilities such as runtime calibration of control parameters as used in automotive applications [26], [19], [188], but also advanced verification tasks such as software based fault injection as proposed by Yuste et al. [314]. Peng et al. [315] provide a
more detailed description of using a Nexus debug platform to perform software based fault-injection to verify fault detection mechanisms. Fidalgo et al. [99] describe a similar approach of using a Nexus based debugger to insert fault behaviour into memory cells and registers so that performance of the fault-tolerance features can be tested. However, they found that it was also necessary to add customized OCI to make the fault injection process more deterministic.

The Nexus debug architecture provides more bandwidth for real-time embedded system verification, but of course this increased bandwidth is not free and the additional pins required are a finite resource in SoC designs. Unfortunately, with the exception of those few IC manufacturers that provide processors targeted at specific automotive applications requiring its capabilities, the Nexus standard has not been widely adopted [111]. However, as outlined by Stollon [316] work is on-going to update this standard to include 1149.7 and high speed serial ports (such as SerDes); the consequent reduction in I/O pins may be sufficient stimulus to facilitate its wider use.

4.4 On-chip core interfaces

For SoC designs containing multiple cores it is desirable to use a common on-chip interface between cores and many proprietary solutions exist for this [196]. The following sections give a brief overview of the two main on-chip core interfaces/wrappers currently used for test and debug purposes.

4.4.1 IEEE P1687

The proposed IEEE P1687 “Internal Joint Test Access Group (IJTAG) standard” is focused on test access at an on-chip level. Utilizing IEEE 1149.1 to access the on-chip instruments which accompany the multitude of cores in modern SoCs, this standard aims to address the challenges of test interface efficiency and adaptability whilst maintaining compatibility with existing tools [300], [317], [318]. Figure 9 illustrates the on-chip architecture which the standard proposes.

Figure 9: P1687 zones

The fundamental component is the addition of gateway logic to interface between the 1149.1 and P1687 on-chip zones, and the hierarchical connection of cores within the P1687
This scheme enables connection of numerous cores in a very flexible manner since multiple gateways and levels of hierarchy can be used as required. This hierarchical arrangement is critical to addressing the inefficiencies seen with existing daisy-chained or multiplexed 1149.1 access methods. During test the P1687 zone hierarchy can be configured such that only the cores of interest are included in the scan chain which may be a useful feature when considering targeted debug.

Of course this flexibility is not without some cost in terms of hardware, setup time, and additional scan time. Initial analysis by Zadegan et al. [319] on a range of benchmark designs suggests that the test application time overhead may be in the region of between 9% and 24%. These results may be somewhat overstated as the benchmark test schemes used did not include IP core with BIST capability. Nonetheless the analysis does highlight the overhead associated with the flexibility offered by P1687. Therefore even if IEEE P1687 does provide efficiencies in terms of test access time and utilisation of limited I/O bandwidth, it does not address the more general systems level debug challenges associated with the simultaneous runtime verification of multiple CPU cores.

4.4.2 IEEE 1500

The intent of the IEEE 1500 “Standard Testability Method for Embedded Core-based Integrated Circuits” [297] is to facilitate testing of independent IP blocks or cores as would typically be found within SoC designs. The standard defines the manner in which these cores can be tested using a wrapper architecture as illustrated in Figure 10, which in many ways is analogous to the use if IEEE 1149.1 for boundary scan testing of ICs. The standard is primarily focused on core wrappers and how these wrappers can be used with SoC level test access mechanisms.

![Figure 10: IEEE 1500 core test wrapper](image)

The allocation of test access mechanisms within a SoC is a non-trivial task and often requires considerable computational resources to arrive at an efficient solution. Iyengar et al.
examine some algorithms to improve the process of finding solutions in less time, which also results in better testing time. Yi et al. [321] propose reducing test time by using a simple JTAG interface to access IEEE 1500 thus enabling simpler automated test equipment (ATE) systems to test multiple devices in parallel. Higgins et al. [322] describe the use of the IEEE 1500 wrapper architecture to enable simultaneous testing of multiple cores; the paper also outlines the use of on-chip test controllers and reuse of the on-chip bus for test purposes.

Prior work of others such as Lee et al. [294], Huang et al. [323] and Krstic et al. [324], also highlight the benefits in using on-chip resources to implement test. The motivations for these approaches (namely the avoidance of the I/O bottleneck and optimization of test time) have relevance for debug applications too. Amory et al. [325] describe how an on-chip functional bus or Network-on-Chip (NoC) which has guaranteed throughput can be reused to transport test data. The examples presented require only a minor modification to a typical IEEE 1500 core wrapper cell and incur a small area overhead, but the main benefit is the elimination of a dedicated test access mechanism (TAM) and associated design effort. Van den Berg et al. [326] examine methods to optimize the bandwidth, in particular the elimination of idle bits, where functional interconnect is used instead of a TAM.

Goel et al. [327] describe flaws with the use of IEEE 1500 wrapper cells when considering SoCs with hierarchical core architectures. The paper illustrates how the IEEE wrapper cell itself is not fully testable and why the architecture is not sufficient where a child core exists within a parent core. New input and output wrapper cells are shown which overcome these difficulties and allow testing of this hierarchical arrangement of cores. Examples of the application of this new wrapper to several benchmark designs are provided to demonstrate the feasibility. Vermeulen et al. [328] showed how core based access could be used provide to enhanced functional silicon debug capabilities similar to those used for software debug, but IEEE 1500 does not provide any specific functionality to facilitate application debug.

4.5 On-chip instrumentation (OCI)

Aside from I/O interfaces and on-chip core access mechanism it is often necessary to provide additional on-chip circuits to implement the features required by modern debug environments; these circuits are typically referred to as OCI or on-chip debug (OCD). In [329], Stollon and Leatherman categorise typical on-chip instruments under the following five broad headings:

1. Logic analysis of fixed or configurable IP blocks
2. Run control and instruction and data trace of embedded processors
3. Trace and analysis of embedded buses
4. Embedded performance analysis tools for IP, processors, buses, or system level interactions
5. System level (Multi-Core) monitoring, synchronization, and cross triggering of all the above.
Run control generally refers to circuits which allow the embedded processor to be reset, started, halted, and in most case single-stepped. Monitoring, synchronization, and cross-triggering instrumentation, generally refers to circuits which support data capture, allowing the relevant signals to be monitored for interesting events and then triggering and synchronising capture or run control hardware. Monitoring instrumentation may also include: performance counters that are configured to record occurrences of interesting events; and circuits which allow the program counter to be unobtrusively sampled at regular intervals; both of which can be useful for software profiling. With the exception of designs with multiple clock domains these circuits do not pose significant challenges; and as these circuits are well understood and require modest bandwidth, they therefore do not warrant in-depth discussion.

Instrumentation for logic analysis provides circuits which enable monitoring and capturing of digital signals which have become deeply embedded in the chip fabric [190]. The aim is to insert circuits with capabilities similar to those which a desktop logic analyser would previously have provided when connected to external signals. Although the main purpose of these instruments is to support hardware verification tasks they can also be useful for some software verification activities. The primary challenge associated with logic analysis circuits is to identify during design, and provide access to, those signals which are sufficient to enable detection and diagnosis of faults; the secondary challenge is to provide a means of storing and extracting large volumes of data which may need to be captured while the system is executing. This issue overlaps with trace data extraction and is discussed further in the following sections. Daoud and Nicolici [190] examine several techniques for efficiently applying lossless compression, using hardware, to data captured from these on-chip logic analysis circuits. They highlight the need to balance the compression circuit area against achievable compression ratio, and propose a new metric with which this trade-off can be evaluated.

However, designing OCI for runtime tracing of instruction, data, or bus activity, is possibly the most complex issue from a software verification perspective. Not only must the trace circuits be designed to monitor buried signals, these signals are also multiple bits wide and the monitored data may update at the full cycle speed of the processor. Where these signals can be monitored, the challenge becomes one of how to capture and store the relevant data. The difficulty then, and perhaps the most challenging issue, is how to extract the data off-chip for analysis. Awkwardly, runtime trace data extraction is one of the most comprehensive and widely used means of supporting runtime verification of software.

The following sections examine further the challenges associated with using instrument trace for verification and give a brief overview of some of the principal OCI solutions
4.5.1 Instrument trace

Acquiring trace data for silicon debug poses similar problems to software debug. A key objective for both domains is to minimise the amount of data to be acquired whilst maintaining coverage. For functional silicon debug Ko and Nicolici [295] propose a method of reducing the number of signals to be monitored (hence reducing data) by using state restoration. This technique exploits the fact that the state data captured from digital circuit nodes can be used to extrapolate the state of other nodes which have not been captured, if the gate-level netlist is available. The paper also presents a set of algorithms for state restoration and selection of the optimal set of captured signal nodes.

However the design of software is not fixed like a netlist and a trace of software execution has the potential to produce vast amounts of data. Considering even a modest example of 16-bit microcontroller operating at a clock frequency of 20 MHz, full execution trace, of raw data, would produce data at a rate of 40 Mbytes/s (320 Mbits/s). Transferring this volume of data off-chip for analysis creates bandwidth problems [178], [181], which have been addressed by such techniques as branch-only-tracing, tracing of restricted areas of data memory, and complex triggering; all of which reduce the volume of data to be captured and sent to an off-chip data acquisition system [111]. This is the industry-standard means of overcoming the bandwidth problems encountered by the need to capture large volumes of software debug data associated with program information tracing

Panda et al. [205] examine the problem of efficiently extracting the large volumes of cache memory data needed during post-silicon debug. In this scenario the internal state of a processor must be captured at regular intervals to enable synchronisation with a much slower cycle-accurate simulator. They propose an on-chip hardware compression scheme where the cache data structure is taken into consideration, which results in a hybrid solution where data fields are compressed using LZW [330], whereas the cache ECC field is only sent if an error exists.

Pang et al. [187] describe their reconfigurable OCD module for SoC designs, and propose a three phase trace compression scheme (branch only trace, followed by address encoding, followed by Lempel-Ziv data compression). They report a resulting mean compression ratio in excess of 95:1 and suggest that a single output pin is sufficient for this real-time trace data. Hopkins and McDonald-Maier [188] proposed a powerful platform for debugging multiple core systems. The authors described a number of techniques to deliver data off-chip including an efficient trace packaging technique and differential compression for program and data trace. The proposed trace port in this case requires two control signals and a scalable data portion (four, eight, or sixteen bits wide); and the average size for program data after

Page 56
compression is given as 3% of the original. In later work, Hopkins and McDonald-Maier [111] performed a comprehensive review of trace requirements and current compression techniques, and concluded that a four-fold increase in compression or new approaches to capturing and exporting trace would be needed for the next generation SoC designs.

Trace compression is also proposed by Anis and Nicolici [331], but in this case, a signature is generated and stored at regular intervals from within an SoC (using a small number of signals which represent the execution state). The rate at which the signature must be stored into the trace buffer is much less than the data capture rates which would be required to capture the same signals in real-time thus significantly reducing the space required. Errors in execution are indicated where the signature captured does not match the expected value. The authors propose that multiple signature samples be taken over a long execution run to identify regions where errors occur and that subsequent execution runs can zoom-in on these execution regions. However, this approach requires that a suitable simulation environment for the SoC is available, and that the execution (including errors) can be predictably replayed.

Even on extremely complex CPU architectures, the size of the trace buffer is limited by practical and economic considerations [169]. Orme [332] advises that only modest on-chip program trace memory is needed, suggesting ‘a 4-Kbit RAM can hold over 30,000 lines of assembler code execution’. Unfortunately he does not detail the compression methods used! Such figures might be plausible for long code loops where only the branch data is recorded; however, he does concede that data trace is more difficult to compress and therefore more expensive to capture, store, and extract. Köhler et al. [185] examine how trace buffer size can be tailored to achieve 100% register content reconstruction; but their results also show that the optimal buffer size is application dependent. Mayer and Demi [333] suggest that a compact trace, where only function call and return data is traced and leaf functions are excluded, is sufficient in some applications. Kao et al. [191] describe an AMBA bus tracing module which can be configured to trace data at different levels of abstraction enabling the on-chip trace memory to be utilized for cycle by cycle tracing or transaction level tracing. They also propose different compression schemes for various data types (address, data and control) and provide experimental result with high mean compression rates. Although the module can be configured using a JTAG interface the authors propose using the processor to export the data off-chip. Hu et al. [334] examine the problem of sharing the limited trace port bandwidth between multiple on-chip trace sources.

As previously described, the hidICE solution proposed by Hochberger and Weiss [183] enables the synchronisation of the target with an emulation platform. The authors reason that extensive data, including trace, can be more easily extracted from the emulation platform.
While this approach does potentially simplify the problem of accessing the data and signals of interest, it requires an accurate emulation platform and the issue of dealing with the significant volume of trace data is merely moved to the emulation platform.

Where on-chip hardware does provide the necessary mechanism to extract instrumentation trace data, simply comprehending and analysing the volume of data which can be generated using trace capture tools poses a significant challenge [335]. Prada-Rojas et al. [336] describe a tool which enables the user to better visualise large dataset of captured trace information in a graphical format. A related difficulty is the lack of standards surrounding the use of trace data interfaces [32]; before the trace features can be used effectively, additional effort on behalf of software developers and tools providers is required to formalise how such interfaces will be exploited. Creation of custom solutions is of course an option, but this is prone to error and introduces another verification challenge and is therefore not the most desirable approach. Martin and Rohloff [32] describe how trace capabilities are typically used, and provide some simple examples of generic message formats which could be applied to numerous application scenarios. The Multicore Association also recognises the lack of standards for trace data format and their Tools Infrastructure Working Group (TIWG) is working towards defining a standard [337].

4.5.2 OCP-IP

As described by Stollon et al. [338] the OCP-IP Debug Working Group aims to define and standardise debug interface requirements for OCP compliant cores (or IP blocks) and proposes optional debug interface sockets for cores which require debug. Although the paper does outline many desirable debug capabilities such as run control, trace, and trigger, it makes clear the fact that the implementation details for debug within each core is specific to that IP block and not mandated by OCP.

![OCP-IP inter-core debug fabric](image)

*Figure 11: OCP-IP inter-core debug fabric*

Instead, as illustrated in Figure 11, OCP focuses on standardisation of the on-chip interconnection fabric between cores such that IP from various sources can be successfully
integrated in SoC designs. Similarly, the chip-level I/O interface is not regarded as falling within the scope for OCP but rather the objective is to define a solution which is compatible with many of the existing interfaces.

4.5.3 FS2

First Silicon Solutions (FS2) was formerly one of the leading companies providing OCI IP. Although proprietary in nature, publications related to these instruments [329], [339] do explain the rationale behind using OCI in general terms. Publication [329], focuses on how integrating such instruments into a typical EDA design flow can assist with the verification of designs in both prototype (FPGA) and the 'first silicon' phases, whereas [339] focuses more on background to OCI and its application.

Leatherman and Stollon [340] address specific multicore debug challenges by proposing a Multicore Embedded Debug (MED) architecture. This architecture adds several on-chip capabilities to deal with issues such as debug synchronisation, but the solution proposed for overcoming limited I/O resources is to multiplex the data streams from various cores using a scheme called HyperJTAG. While this may minimise and simplify the I/O interface it does not address the problem of limited I/O bandwidth.

4.5.4 MIPS

EJTAG [341] provides on-chip debug capabilities for SoCs using those MIPS processors cores which have the instructions required to support it. As the name suggests this debug architecture enhances the IEEE 1149.1 (JTAG) standard to provide access to a range of on-chip features including: single step execution, breakpoints, memory substitution and PC sampling. Debug of multi-threaded applications is supported within the specification but multicore debug (including virtual processing elements) requires duplication of the debug hardware, which can be daisy-chained to enable sharing of a single I/O interface. As such, despite supporting multicore SoC design, this architecture alone provides no mechanism to deal with the accompanying volume of data.

Trace information from a MIPS processor core can be obtained by connection of an additional on-chip Trace Control Block (TCB) to the processor's PDtrace interface [342]. The TCB compresses the trace data in a number of formats before storing it in on-chip or off-chip memory. External debugging tools can then access this trace data using the EJTAG interface, but for high-bandwidth an optional parallel interface (Probe IF) is also defined which requires additional I/O pins.

4.5.5 ARM

As one of the main providers of CPU cores and IP modules for SoC designs ARM also provides a number of 'CoreSight' modules tailored exclusively at debug [343]. This suite of IP
modules includes a debug interface 'Debug Access Port' (DAP), bus trace 'AHB Trace Macrocell' (HTM), hardware based processor trace 'Embedded Trace Macrossells' (ETM) and software driven 'Instrumentation Trace Macrocell' (ITM), and more recent Trace Memory Controller (TMC) and System Trace Macrocell (STM) versions [344]. Trace data can be transported on-chip via a dedicated bus interfaces and stored on-chip in a buffer or streamed off-chip with additional trace modules enabling data replication or combination using a 'Trace Funnel'. The 'Trace Port Interface Unit' (TPIU) provides the interface between on-chip trace data and external trace analysis tools. Support for triggers is also provided via 'Embedded Cross Trigger' (ECT), 'Cross Trigger Interface' (CTI), and 'Cross Trigger Matrix' (CTM). Orme [332], [345] provides an overview of the CoreSight modules available plus key design decisions and trade-offs required when considering a SoC debug solution. Su et al. [346] describe the application of CoreSight modules to a multicore SoC containing both ARM processors and a DSP core. Kyung et al. [90] describe an IP block which enables passive monitoring of performance and contention on AMBA AXI bus system.

For the purpose of this research, two key CoreSight elements which connect to external pins are highlighted. Firstly the DAP provides 'access ports' to on-chip resources such as the peripheral bus, high-performance bus and JTAG scan-chains, plus 'debug port' access to external tools via a serial wire or JTAG interface. The serial wire interface requires just two pins as compared to four pins for JTAG, and in cases where both interfaces are required it is possible to share the same pins. Secondly, the TPIU is used to format trace data into packets and provides an interface for the transfer of large amounts of trace data to external tools, the width of the interface is scalable from a single data pin up to 32 parallel data pins. As outlined, the I/O interfaces provided by these two modules are not dissimilar to other debug solutions nor are their bandwidth capabilities.

4.5.6 Infineon MCDS

Although core logic cells shrink in line with silicon process advancement, the core area required for any OCI still represents an added cost; the more sophisticated the OCI the greater the cost. For high volume parts this additional cost can be substantial and once the design has been deemed fully functional this cost is often considered unnecessary. The Infineon Multi-Core Debug Support (MCDS) solution promotes a novel approach to cater for OCI. Rather than adding any instrumentation or memory within the SoC core area these circuits are instead placed on the device perimeter or on a separate die, which is then bonded to the SoC [19], [22]. Any alternation to the layout of a digital circuit has the potential to alter its behaviour, and if the OCI is placed within the core area its removal requires a new layout with associated costs of redesign, verification, and manufacturing mask-sets. Therefore one key benefit of the Infineon MCDS approach is that this circuitry can be removed at a later
stage without impacting upon the SoC behaviour; therefore silicon and software verification activities need not be repeated. However, despite this distinct benefit Hopkins and McDonald-Maier [111] suggest that the prohibitive cost of manufacturing mask-sets alone could make the option of placing the OCI on the device perimeter impractical; of course this criticism does not apply to the option of using a separate die.

Mayer et al. [19] describe how MCDS also addresses the challenge of limited I/O bandwidth by focusing on trigger and trace qualification. By providing sophisticated data capture capabilities the hope is that only a small amount of relevant data need be stored in on-chip memory. This smaller amount of data can then be transferred off-chip using a conventional JTAG or other low-bandwidth interface. However, achieving highly focused on-chip data capture requires complex circuitry which can be difficult to configure correctly. Braunes and Spallek [347] describe the high level language and tools provided to make configuration of the MCDS practical for engineers; their paper also provides a good insight in the MCDS architecture.

In addition to being used on Infineon devices, the MCDS has also been made available as a licensable IP block for use in SoC designs [173].

4.5.7 UltraSOC

UltraSoC Technologies is a recently formed company focused on developing debug solutions for SoC designs, the promoters of which include McDonald-Maier and Hopkins amongst others. Although detailed information on the technical solution being pursued, called UltraDebug, is not readily available, the company has licensed patents from the University of Kent. Therefore some insight may be gained from relevant patents filed by the promoters and that University.

Two patents outline SiP device packaging solutions whereby the high-volume elements of a SoC design are implemented on the primary silicon, and secondary circuit functions are implemented on a separate die which can then be bonded to the primary part within the package [348], [349]. The patent [349] is more focused on debug and describes how such a secondary circuit could implement an interface conversion circuit to multiplex many low speed on-chip signals and transfer their data over a smaller number of fast I/O pins. This conversion circuit is described as being of electro-optic nature or of some other suitable fast communications type. In situations where the fast I/O interface requires expensive or exotic processing steps which do not fit within standard CMOS processes and for applications where the debug interface can be dispensed with once initial development is complete, this SiP approach offers obvious benefits.

A further patent [350] claims a novel detection circuit which can be reconfigured
dynamically to monitor different signals within a SoC design. The signals of interest can thereby be refined by adjusting the signals being monitored in response to the progression of the test or debug session. It is suggested that this approach offers the ability to monitor all signals but with the resources being focused only on the relevant signals at any given time. For SoC platforms that provide the necessary reconfigurable circuitry which can be adapted during runtime, this approach would appear to offer interesting possibilities.

An EU funded project (I3E) lists UltraSoC Technologies as one of the companies studied in examining successful transformation of research to innovation; the company is included as one of 30 best practice reports [351] which briefly describes their technology as follows:

"The speed of electrical device pins, the chips' external debug connections, is increasing at a much slower rate than the speed of chip internal circuits. As a result, with each new chip generation, the requirement for testing quadruples, whilst the debugging capability remains virtually unchanged. The obvious solution of increasing the number of trace pins has very high cost implications, estimated at $4.6 million per 10M product units sold. Dr McDonald-Maier of the Department of Electronics, University of Kent, and his team developed a solution that uses optical interconnects combined with dedicated infrastructure circuits, to provide more than double the current debugging performance. As the performance of optical interconnects greatly exceeds that of conventional electronics, the solution is scalable with new chip generations. ... However, UST's products are not yet available to the market and the company has not revealed much about the characteristics since the project is still under development."

4.5.8 Combined instrumentation

Multi-processor SoCs can comprise of many different processor cores, often from different vendors. Murillo et al. [123] examine the problem posed by these complex architectures and identify the fact that while traditional debug tools can support individual processors, tools which support detection and resolution of concurrency related bugs are lacking. They propose a framework which abstracts the target-level details and focuses on the capturing event based data from the communications interfaces. However, the case study provided relies upon a simulation environment rather than a physical target platform. Vermeulen et al. [193] describe similar communications centric debug architecture from SoCs which use a NoC with distributed memory. Tang and Qiang [335] examine the challenge of designing OCI for NoC based designs and propose adding a debug probe to each network interface which not only monitors the network data but also uses it to transfer debug data. They also suggest using this probe to interface with the JTAG port of the core so that a single communications mechanism can be used for all debug data.

Vermeulen and Bakker [171] describe a debug architecture for an SoC containing both an ARM CPU and TriMedia DSP. To achieve the desired objective of providing a single debug
solution for both hardware and software purposes, the solution combines many existing technologies such IEEE 1149.1 interface, on-chip debug modules from ARM, TriMedia trace module and P1149.7 TAP architecture amongst others. For software debug in real-time, the solution provides an on-chip 8kB trace buffer and requires 16 I/O pins to transfer the data off-chip at a maximum possible rate of 4 Gbit/s (250 Mbit/s per pin). Interestingly the bus monitors used can calculate a checksum of the monitored values in addition to being used for breakpoint purposes.

Mobile phones are exemplary of the scenario where multiple processors and intelligent peripherals from unrelated vendors have been progressively integrated into SoC devices, enabling savings in both material costs and power consumption. This of course creates the challenge of debugging these various cores in some unified manner; to this end the Mobile Industry Processor Interface (MIPI) Alliance was formed. MIPI has developed specifications for common interfaces, including test and debug, used in mobile devices, aiming to ease integration and enhance interoperability between components.

Vermeulen [301] provides details of the areas upon which the MIPI Test and Debug (MIPI T&D) working group are focused, and references two relevant MIPI white papers [305], [352]; unfortunately, detailed specifications are only made available to alliance members. In [305], an overview of the test and debug specification is given including the use of IEEE 1149.7 and the System Trace Module (STM). The STM is described as being focused on software debug requirements and supports tracing from multiple threads (255) to a single STM; and multiple STMs can be used, either with individual I/O interfaces or sharing a single interface. In [352], the authors describe methods by which the debug interface can be shared between devices and how other interfaces such as memory card interfaces can be utilised to provide debug access.

4.6 Built-in self-test and Software based self-test

Built-in self-test (BIST) generally refers to the hardware features designed into a hardware IP block to enable it to perform self-diagnostics. BIST has become more common as the density of devices has grown and the cost of external test or software based test methods becomes prohibitive [197]. BIST enables the offloading of substantial test functions to the IP block, thereby reducing the I/O bandwidth required as only the control and results data need be transmitted over the interface [170], [294], [353].

Built-in test techniques are undoubtedly of benefit where the potential error(s) to be detected can be predicted in advance; the close coupling of component and test can overcome the lack of visibility otherwise inherent in integration and provides the ability to optimize the test-time resources for the component under test. However, the necessity to predict potential errors highlights a key differentiation between test and debug scenarios; due to the
unpredictable nature of bugs, the runtime monitoring of the IP block along with capturing significant volumes of data is hard to avoid. Weiss and Hochberger [170] recognise the benefits of using of BIST for detecting static faults, but also highlight its deficiencies in detecting subtle intermittent dynamic errors.

Software based self-test (SBST) involves the execution of software routines to stimulate the device and check its correct operation [353]. Paschalis and Gizopoulos [354] show how a periodic on-line SBST regime can achieve a high probability of detecting permanent and intermittent faults in embedded processors. Although similar to BIST, the use of software routines enables SBST to be more easily tailored to the application scenario and even modified to facilitate debug if necessary. The main drawback of SBST is that it requires execution time on a processor, and corresponding resources; and of course, the tests performed must not interfere with normal functions of the application. This means that SBST are generally scheduled to run at system startup or shutdown, but if the tests can be guaranteed to be free from any side-effects they may also be scheduled to run at regular intervals during idle time [354], [355], [356].

Bernardi et al. [357] proposed the addition of an infrastructure intellectual property (I-IP) block to assist in silicon debug and testing of SoCs. This I-IP block uses IEEE 1500 as an external interface but exploits the SoC processor to run a suite of SBST routines to test and debug the silicon. By restricting external data transfers to test commands and compressed test results, the bandwidth limitations of the interface can be overcome. However the execution of SBST in the manner proposed takes over the processor, which may be acceptable in a silicon test and debug scenario but is obviously not suitable for real-time software debug.

Gross et al. [83] draw a distinction between the motivations for hardware BIST and SBST, arguing that hardware components can deteriorate whereas software does not; therefore, they suggest that built-in software component tests should instead be focused upon the system-level integration or software environmental considerations that may influence the software behaviour. They also make a clear distinction between software assertions and built-in tests; even though assertions can be useful to detect erroneous behaviour, they suggest that a more holistic view is required when designing built-in tests for a software component.

4.7 Software instrumentation

As distinct from SBST, which generally aims to test the target platform or a hardware component, software instrumentation refers to the addition of software code which aims to provide visibility into the runtime execution of the software; and this software instrumentation may or may not include test functionality. At first glance software instrumentation is seen to be convenient for a number of reasons: 1) it can provide visibility into the software execution at a low-level which might not otherwise be possible; 2) the
instrumented code runs at the full execution speed of the target; 3) if sufficient code and data space are available, no additional hardware resources are required, therefore software instrumentation is often considered to be low-cost; 4) the software developer can use the same familiar tools and language for development and instrumentation; 5) software instrumentation can be added and removed as needed in a very flexible manner.

Software instrumentation is one of the most widely used techniques to enhance the visibility of runtime behaviour. Thane and Sandström [358] argue that since the most common method to make code execution more observable is to instrument it, then what is needed are tools to assist in the analysis of the data logs from the instrumentation. But software instrumentation is not without flaws. Kucharski and Zieliński [359] describe a typical process followed when trying to remove bugs in embedded code; they illustrate the need for a range of tools to effectively find and correct bugs and the need to ensure that the instrumentation code is executed and works correctly.

Wong [360] describes the typical use of printf() function calls to instrument and debug code, but highlights the subtle side-effects this can cause, which may introduce or mask bugs. Alternative less intrusive tools are outlined but their ability (or lack thereof) to capture data which may be changing quickly in real time is not discussed. Stewart [135] also highlights the challenges in using traditional print statement and symbolic debugging in multi-threaded real-time systems, and proposes using a logic analyser. Although this approach may be of benefit in some instances, it too involves two fundamental difficulties; firstly it presumes unused I/O pins are available, and secondly it requires instrumentation of the source code.

With the QP event-driven platform [142], software trace capability is provided by inclusion of optional ‘Quantum Spy’ (QS) features within the target application and a host based QSPY application. The approach adopted in QS is to include software instrumentation in the framework code, and if necessary the application code. The traced data is first stored in a shared memory buffer and then transmitted to the host during idle time; access to this shared memory requires protection, which is provided by means of critical section macros. To optimize trace buffer usage, QS also provides source level macros which allow the trace data records to be filtered. The data protocol is designed to be simple and efficient and any available communications interface can be used as the target to host data transport mechanism. No analysis of the traced data is performed on the target, in fact the approach adopted is to export data in a raw format and exploit the host application to parse the data into understandable information.

Havelund [68] points to the fact that engineers are reluctant to adopt unfamiliar methodologies, and presents RMOR as a usable approach for C programmers. Ball and Rajamani [253] present a Specification Language for Interface Checking (SLIC) which is
intended to facilitate automatic instrumentation of C libraries, using a pre-processor, to ensure that the specification of the API is adhered to. Hung et al. [31] propose a software instrumentation tool called Moduletracer, to extract execution trace and profiling data. The authors highlight the benefit of a pure software instrumentation based approach over existing solutions, which are limited to user space applications or use on-chip hardware features; and suggest that their software only approach is applicable to a wider range of situations. However, they also highlight two significant disadvantages with software instrumentation, the execution time overhead imposed, and the difficulty with storage and extraction of trace data.

Brouillette [361] describes how the software instrumentation provided by SVEN_TX combined with the hardware signal capture capability provided by OMAR, enables SoC designers to debug and monitor the behaviour of these complex devices. The approach taken to capture real-time events and minimise the overhead imposed by the instrumentation, is to capture to the system-memory rather than sending data over a slow interface. The captured data can be processed by background tasks at a later time. Brouillette argues that the overhead required by the software instrumentation in this case is so low that there is no need to remove it, and if left in-situ it can be a useful tool for field diagnostics. For situations where sufficient superfluous memory and processing resources are available these arguments may be valid, but this is rarely the case in resource constrained embedded systems, and leaving unnecessary functionality in safety-critical systems is generally not recommended.

Zhou et al. [362] propose iWatcher to address the impact of software instrumentation. Their approach uses software instrumentation to configure on-chip hardware to monitor data locations, or areas of the code, which are of interest; the hardware then triggers an appropriate monitor function when this region is accessed. Teodorescu and Torrellas [363] develop this idea further and propose a scheme whereby the section of code to be investigated is speculatively executed and if an error occurs it can be re-executed with enhanced instrumentation.

Source code instrumentation obviously requires access to the source, in the case of applications running on a RTOS or using third party libraries this may not be available. Even if this source is available, altering it may not be desirable as it may not be familiar to the software engineer and recompilation may introduce unexpected alterations in behaviour. Fagui et al. [91] address this problem by proposing a method of extracting profile data from embedded Linux platforms without instrumenting the source code; using an additional data collection thread they extract data from the running system and export this for off-line analysis.

Ekman and Thane [364], [365] propose a method of instrumenting embedded code at
runtime. The authors suggest that only modified code fragments be replaced (binary code
differences can be identified by comparing instrumented binary code against the original
target binary), and that this process can be performed during idle states, using a suitably
prepared target. While this approach may be faster and more flexible than replacing the entire
binary, it presumes that the application code executes from RAM, whereas many embedded
applications use FLASH memory for code. And although the authors emphasise the fact that
the instrumented code must be designed to not affect the original application flow or data, the
possible side-effects of altering execution time is not examined in detail.

Although convenient, the greatest weakness of software instrumentation is its invasive
nature [344]. Sundmark et al. [58] emphasise the fact that excessive instrumentation may
exacerbate non-deterministic behaviour when testing embedded systems. If software
instrumentation is added for verification purposes and later removed, the system-level
behaviour is altered [21]; such changes can mask or introduce bugs. The fact that such
instrumented software often doesn’t match production software limits its usefulness in code
coverage [80], [81], [84]. If software instrumentation is left in the released version then this
instrumentation code must meet the same standards of verification as the functional code; and
as previously pointed out many standards recommend that released software should include
no superfluous code. Ekman and Thane [30] go so far as to suggest that if the design must
comply with safety-related standards such as IEC 61508 and ISO 26262 no existing software
instrumentation techniques are suitable.

4.8 Summary

In many regards software verification is similar to silicon verification. The overlap
between silicon and software design activities also extends into literature with the terms
'debug' and 'verification' being used without clarification. Like software, hardware
verification and debugging of modern SoC devices requires access to the signals and data
which have become deeply embedded within the silicon. However, software verification does
differ significantly in that it may be necessary to perform software verification long after the
silicon has been fully tested, in some situations years later when the software is upgraded, for
example to add new functionality. Therefore software verification must be carried out using
the interfaces which are made available on the final packaged device.

BIST is very efficient for predetermined test; successful test case execution provides a
degree of confidence that the component continues to function as expected. However, when
tests indicate a failure, additional debug capabilities are required to analyse the root-cause.
Therefore all verification solutions must incorporate some degree of both test execution and
debug capability. To meet silicon test and debug requirements IC design engineers
incorporate standard access mechanisms which are primarily focused on minimising the costs
associated with test and using minimal I/O pins and silicon area. The bandwidth of these test access mechanisms varies and can support some limited software debug features, but these mechanisms are not designed for software verification and debug of real-time systems.

While the reuse of silicon test interfaces for software purposes has obvious economic benefits, they do not provide sufficient visibility or bandwidth for software verification. The duplicity of purpose between silicon and software test and debug often leaves software developers with a sub-optimal solution and blurs the line where these two distinct activities overlap. While there are efforts to standardise aspects of the debug interface [366], the current state-of-the-art is comprised of many distinct and often competing solutions. Hopkins and McDonald-Maier [21] provide an overview of debug infrastructure including how interfaces primarily designed for silicon test and verification have been used for software debug purposes, but also highlight the limitations of this approach when considering real-time system.

Nexus and proprietary OCI technologies targeted at software debug face the same bandwidth issue; the solution to which is usually a combination of: increasing the number of I/O pins, compressing the data, and/or limiting the debug capabilities to those which can be supported by the interface. The increased silicon area and in particular the increased I/O required by these solutions makes these technologies relatively expensive when compared to test technologies.

Proposed improvements to simply increase the available I/O bandwidth such as using high-speed [332] or optical [111], [181] interfaces are unlikely to fully address the problem in either the short or long term. Firstly, the use of such interfaces is likely to be linked to specific process technologies and/or require System-in-Package (SiP) assembly processes, which limits usage; Mayer and Demi [333] also highlight the signal integrity problems which high-speed interfaces can pose. Secondly, high-speed I/O require larger on-chip driver circuits and more power, both of which run counter to the objective in many embedded applications to reduce power and silicon costs. Thirdly, these solutions only shift the problem to the external debug and test equipment, and as highlighted by Rearick [367] existing high-speed interfaces already present challenges for ATE.

The alternative of improving bandwidth by simply widening the interface further is equally problematic. I/O cells and bond-pads on ICs are significantly larger than core digital cells; therefore, adding I/O pins is often not cost-effective. In I/O-bound designs not only would the I/O area increase but so too would the core area. As illustrated by Whetsel [311] the objective for many proposed interfaces is to reduce I/O thus enabling the die area to shrink. Plus, additional pins increase difficulties in making a reliable physical connection; this is particularly important in situations where rugged interfaces are required to debug a
real-time system while operating in harsh application environments.

Figure 12 provides an illustrative comparison of the various technologies discussed as a three-dimensional space where the axes represent three metrics: economy, data throughput, and suitability for real-time software verification. An ideal solution is represented by a star, which is located in the region of lowest cost, highest throughput, and greatest suitability for software verification and debug purposes.

**Figure 12: Comparison of relative merits of test and debug technologies**

Customised instrumentation added to either the hardware or software is useful where specific items need to be monitored; and can be relatively inexpensive, particularly if the solution devised can utilise an existing I/O interface. But excessive software instrumentation can impact upon normal execution and hardware instrumentation is not easily changed. The impact of software instrumentation on execution timing can be lessened by using buffer memory or instrumentation supporting hardware. However, as exemplified by the QS solution proposed by Samek [142] such buffers may themselves introduce shared memory difficulties, which the developer sought to avoid. And whether or not buffering is used, throughput is still limited by the ability of the software and hardware to export data to the external host in a timely manner.

This alternative approach of using a co-processor to carry out on-chip analysis of the data captured, which this author advocates, is examined in Chapter 5. This is a step towards the ideal solution as illustrated by the star in Figure 12 because: avoiding the I/O bottleneck greatly improves data throughput; the on-chip logic required for a simple co-processor is less expensive than increased I/O pins or exotic I/O technologies; and the ability to process data in real-time while the application is executing, serves software verification and debug needs better than existing solutions which are targeted at off-line test needs.
CHAPTER 5. ALTERNATIVE APPROACH

5.1 Introduction

Various solutions have been examined which aim to improve visibility into the runtime execution of software on highly integrated platforms. However, even where it is possible to capture the relevant data a key challenge remains in extracting this data. It is clear that despite the high level of integration possible in modern processes, minimising the number of I/O pins will continue to be a key design objective as the I/O cells and their associated area represent a proportionally larger cost; conversely the area required for core digital cells continues to shrink.

For runtime software verification purposes many existing OCI solutions already provide the necessary on-chip infrastructure to capture data at the fastest possible rate, which typically uses high-bandwidth interfaces to on-chip buffers. The MCDS described by Mayer et al. [19] represents one of the most sophisticated OCI solutions available; with the ability to create complex triggers, developers can focus the OCI upon the problem being analysed in a very precise manner, and minimise trace capture. However, they do not attempt to carry out any on-chip analysis of the data being acquired.

A common alternative practice is to add in-line software instrumentation to perform analysis of application behaviour; this requires minimal on-chip resources and is therefore often considered to be a simple, inexpensive, and flexible solution. Plus, reusing the development tools as used for the application requires no additional training or expense. However, using in-line software instrumentation has both advantages and disadvantages. The advantages centre upon economics and simplicity; whereas, the disadvantages centre upon the additional processing time required and the risk of unintended side-effects. These side-effects may result solely from the temporal alterations to the execution behaviour, or may be due to errors in the instrumentation code.

5.2 Alternative approach

On-chip analysis of data captured for verification purposes could potentially be carried out in SoC designs using spare processing capacity. In a manner similar to that proposed by MacNamee and Heffernan [292] for a requirements-based monitor, existing OCI hardware coupled with a simple co-processor could be used to perform on-chip analysis of the captured data for software verification purposes. As illustrated in Figure 13, with multiple CPUs the option exists to place all instrumentation software on one CPU while the application software resides upon the other. The existing OCI could then be utilised to monitor and capture relevant data for the instrumentation code. By isolating the software instrumentation within the second CPU the main disadvantages of software instrumentation could be minimised; no alterations to the application code would be required and the burden of executing the
instrumentation code is carried by the second CPU. This approach would be particularly attractive in multi-processor designs if the analysis task could be performed by a dedicated processor, which would have no impact upon the execution of the application software.

![Proposed architecture](image)

*Figure 13: Proposed architecture*

With traditional in-line software instrumentation, activation at the point of interest is inherent, i.e. when execution reaches the section of code where the instrumentation has been added, the instrumentation code will be executed sequentially. With the author’s proposed approach, the challenge then becomes one of executing the instrumentation code at the appropriate time. However, many existing OCI solutions include the capability to monitor the runtime execution and to generate a breakpoint or to trigger the external debug environment when notable events occur. As illustrated in Figure 14 the author proposes that these OCI capabilities could be used to trigger execution of the instrumentation software at the appropriate time.

![Using OCI as a monitor and trigger for software instrumentation](image)

*Figure 14: Using OCI as a monitor and trigger for software instrumentation*

With reference to Figure 14, a typical co-processor and OCI interaction scenario would progress as follows: 1) the instrumentation software configures the OCI to monitor the application for the execution of an instruction, or event, of interest; 2) the OCI hardware monitors the application execution and captures any trace data; 3) the OCI match detection logic hardware detects the event occurrence and notifies the co-processor; 4) the co-processor then analyses the captured data and determines if execution is proceeding correctly, and if necessary provides a status output. This process could then be repeated, or the OCI could be reconfigured for another event if necessary.

Ideally the OCI would be able to generate an interrupt on the co-processor. This would enable the co-processor to also be used for other unrelated tasks or to be placed in a low-power, or idle mode, until it is required to execute the relevant software instrumentation code.
Or alternatively, the co-processor could perform background processing on the previously captured trace data. Where an interrupt generation mechanism is not available the co-processor could use a polling mechanism which would require regularly reading the OCI registers, or another OCI generated signal, to determine its current state.

Figure 15 illustrates the relative merits of the alternative approach as compared to existing approaches. As shown the key benefits as obtained from traditional software instrumentation of economy and suitability for real-time software verification are retained. Although the alternative approach does not offer a faster I/O interface, the ability to perform on-chip analysis and filtering of captured data, without impacting upon the application execution, means that data throughput requirements can be lessened. Obviously analysing data on-chip using a processor is not as fast as using dedicated hardware monitors or complex OCI, but it offers a more flexible cost effective solution.

![Figure 15: Relative positioning of alternative against existing technologies](image-url)

### 5.2.1 Key benefits of the proposed alternative

In addition to providing the flexibility and benefits of software instrumentation without the associated drawbacks, the proposed alternative approach also offers a number of other benefits as follows:

- Isolating the instrumentation code in one location simplifies its removal, or alteration, and eases the challenges associated with demonstrating that it is free from side-effects.

- The ability to analyse trace or other captured data on-chip allows greater flexibility as to whether the data is stored, exported or discarded.

- The on-chip processing of captured data could reduce I/O data bandwidth requirements enabling the use of an interface with a lower pin-count. This also reduces cost and eases
challenges in making the interface rugged in applications where this is required.

- The ability to reprogram the co-processor to monitor different events, possibly while the system is operational, overcomes the limitations posed by many hardware only solutions which are restricted to the events selected at design time.

- The number of signals being monitored or captured on-chip is generally restricted to minimise I/O or trace buffer requirements. Using on-chip analysis, the number of potential signals could be greatly increased with a subset of these being selected at runtime for capture and analysis.

- In existing solutions, creating complex triggers (or cross-triggers) requires additional hardware which grows as the complexity increases. A co-processor could manage complex trigger conditions without additional hardware, thereby enabling extremely sophisticated trigger conditions to be created.

- Compression of captured data could be performed by the co-processor thereby possibly eliminating the need for dedicated hardware compression. Making the compression algorithm programmable allows flexibility to employ lossless or lossy compression as required depending upon application requirements.

5.3 Related research

Use of a dedicated processor to assist with the verification and debug of embedded systems has been proposed by other authors, but not in the form proposed here. Höller and Rössler [368] describe the addition of a micro-controller (8051) based ASIC to nodes within a distributed embedded system, as illustrated in Figure 16. Acting as a bridge between Ethernet and JTAG, the ASIC provides access to the integrated debug circuits on the embedded nodes, and can also provide additional capabilities such as time stamping and synchronisation between nodes. However, no analysis of the debug data on the ASIC is discussed, and since the platform uses the existing on-chip JTAG circuits its debug capabilities are limited to the features of this interface.

![Figure 16: Distributed debug architecture with dedicated ASIC](image-url)
Tang and Qiang [335] use an off-chip debug controller to assist with the coordination and translation of debug data to/from NoC packets which are transferred via the on-chip network infrastructure. In this case, the debug infrastructure enables connection to the JTAG port on the cores, and if available, their trace ports; but again no analysis of the data on the debug controller is discussed. Mouhoub and Hammami [150] used a dedicated Microblaze processor to monitor NoC performance. The Microblaze processor interfaces their NoC platform to an off-line data analysis host and allows the embedded network monitoring hardware to be reconfigured without the need to resynthesize their FPGA based platform; but their solution is limited to capture and monitoring of on-chip network traffic. Kopetz et al. [146] refer to the inclusion of a diagnostic core in their SoC architecture but do not provide details of its design or capabilities.

A related approach is to extract minimal observation data from a target processor and apply this to a more easily instrumented target emulation platform, as proposed by Weiss and Hochberger [170], or a virtual target running on a host PC as proposed by Watterson [369]. The primary difficulty with this approach is that the emulation platform may not accurately reflect the behaviour of the target. The runtime verification unit proposed by Reinbacher et al. [291] includes monitoring hardware and a programmable RISC core on-chip. This more closely reflects the architecture which this author advocates, but their solution is focused on run monitoring of propositions and involves the inclusion of custom monitoring hardware.

5.4 Summary

Multiple CPU architectures bring new verification and debug challenges; but these multicore architectures may also enable a new approach to software instrumentation. The author has outlined an alternative concept which isolates the instrumentation code in a secondary CPU core, which addressed many of the concerns with adding instrumentation to application code. Using software residing on the secondary core the author aims to demonstrate it is possible to then configure the OCI hardware to monitor execution of the software on the primary CPU. This solution retains the key desirable attributes of software instrumentation and removes many of its disadvantages.

A key benefit of this approach is that there is no need to capture, export, and analyse, vast amounts of redundant runtime execution data. Instead the co-processor can be programmed to output only the status data which is absolutely necessary; for example, execution is proceeding as expected, or an error has occurred. This greatly reduces the I/O bandwidth required to communicate with an external verification or debug tool, and in situations where an error occurs infrequently can save considerable time which would otherwise be wasted analysing vast amounts of execution trace data.

This alternative approach does require that the multicore device has sufficient on chip
debug hardware to monitor execution of the application code, but most devices already include some breakpoint or watchpoint capabilities, even if only at a minimal level. However, most on-chip debug hardware is designed to halt execution upon reaching a breakpoint, or capture data for extraction by external tools; therefore what is required, is that the debug hardware be able to trigger execution of the instrumentation code on the secondary on-chip CPU instead. This requirement should not be difficult to achieve and in fact some SoC designs include OCI which already support the ability to trigger on-chip execution.

The next chapter provides an overview of one such commercial-off-the-shelf (COTS) SoC platform which was used to evaluate the feasibility of this proposed alternative approach.
CHAPTER 6. EXPERIMENTAL PLATFORM

6.1 Introduction

To examine the feasibility of the proposed approach the Freescale MC9S12XE100 [12] was selected as a suitable example of a COTS microcontroller platform which is readily available and commonly used in the design of real-time embedded systems. The MC9S12XE100 is a SoC, which includes: a 16-bit CPU (CPU12X); a co-processor (XGATE); a debug module (S12XDBG); a background debug module (BDM); RAM, FLASH memory, and EEPROM; and peripheral modules, as illustrated in Figure 17. The device also includes: security features to disable access to the internal memories and debugging features; a memory protection unit to detect invalid address accesses at runtime; invalid instruction detection with exception handling; and ECC capability for FLASH memory to correct single and detecting double bit errors. McAslan [370] describes the memory protection features, and FLASH error detection and correction features.

![Figure 17: Block diagram of MC9S12XE](image)

This SoC is targeted at automotive applications and includes a wide-range of on-chip peripherals including: CAN modules, ADCs, PWMs, serial communications (I2C, SPI, LIN and UART), periodic and capture timers, PLL, and voltage regulators. However, for the purposes of this research the key modules are the two CPU cores and the supporting debug hardware; these are briefly described in the following sections.

6.2 CPU12X

As illustrated in Figure 18 the CPU core employs an efficient 16-bit register set which is capable of accessing 64 KB of local address space. However, the MC9S12XE100 SoC architecture utilises an 8 MB (global) address space. To enable the CPU12X to access this larger space two mechanisms are available: 1) special load and store instruction allow direct access to the global space; 2) bank switching (or page switching) enables various sections of FLASH memory, RAM, and EEPROM, to be moved into defined regions of the local memory map. Further details of CPU architecture can be found in the device reference manual [371], and [372] provides a detailed description of the memory page switching capabilities. Appendix B provides an illustration showing how the 8 MB global memory is
mapped to both the CPU12X and XGATE for the MC9S12XE100 part.

<table>
<thead>
<tr>
<th>8-bit Accumulators A &amp; B</th>
<th>7</th>
<th>A</th>
<th>0</th>
<th>7</th>
<th>B</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>or 16-bit Accumulator D</td>
<td>15</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index Register X</td>
<td>15</td>
<td>IX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index Register Y</td>
<td>15</td>
<td>IY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack Pointer</td>
<td>15</td>
<td>SP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Program Counter</td>
<td>15</td>
<td>PC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition Code Register</td>
<td>U</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>IPL[2:0]</td>
<td>S</td>
</tr>
</tbody>
</table>

*Figure 18: Programming model for CPU12X*

### 6.3 XGATE

A complete technical description of the XGATE can be found in [12], the following outlines some of the key features. The XGATE co-processor is primarily intended as an I/O processor to which real-time I/O intensive tasks such as interrupt handling can be delegated. In certain real-time embedded designs it is necessary or preferable to disable asynchronous I/O interrupts [151]. However, disabling interrupts obviously opposes the concept of having hardware capable of servicing interrupts in the first place. Therefore, routing interrupts to a co-processor such as the XGATE provides a convenient means of handling these asynchronous events. This provides the benefit of not altering the temporal properties of software executing on the primary processor, whilst also reducing the processing burden. Rather than executing continuously, the XGATE is in fact an interrupt driven processor which remains in a suspended state until activated. When an interrupt occurs, which the XGATE is configured to handle, the XGATE is activated and it executes the corresponding code ‘thread’ in a run to completion manner. Once the ‘thread’ is completed the XGATE is again suspended. The MC9S12XE family of devices includes version 3 of the XGATE, which supports a high and a low priority interrupt; the high priority interrupt can interrupt low priority one, but nested of high priority interrupts is not supported.

The XGATE is a 16-bit processor, but does not have the same instruction set, or architecture, as the CPU12X. The programming model for the XGATE is shown in Figure 19. Like the CPU12X, the XGATE can only access a limited 64 KB address space and the same global address space (8 MB) is used by both processors, but as illustrated in Appendix B the XGATE is restricted to accessing only a fixed subset of this space. Peripheral registers are common to both processors and are mapped to the same location on each. The XGATE core register set is mapped into this peripheral address space, enabling it to be accessed and configured by the CPU12X. The XGATE RAM and FLASH memory are mapped from fixed
global locations which are also accessible on the CPU12X. However, the XGATE cannot directly access other global memory locations and, unlike the CPU12X, has no support for paged memory blocks.

Although the XGATE can access FLASH memory, it is recommended that its program code reside in RAM; this allows the XGATE to operate at twice the speed of the CPU12X. The timing of accesses to the RAM area shared by both processors is interleaved so that conflicts are minimised. However, where access conflicts arise the CPU12X has priority over the XGATE. The XGATE can perform two accesses to RAM for each CPU12X access; but the timing is such that if both access the RAM at the same time the CPU12X access will succeed and one XGATE access will be allowed. The XGATE accesses FLASH memory at the same rate as the CPU12X, so simultaneous access to FLASH results in the XGATE being delayed. Obviously, to enable the reliable sharing of data between both processors additional protection is required; for this the XGATE includes eight hardware single-bit ‘semaphores’. The XGATE has instructions to set and clear these semaphores, whereas the CPU12X must access the semaphores through the XGATE registers which are mapped to both processors.

In addition to acting as an interrupt handler the XGATE can act as an advanced co-processor for numerous tasks. A Freescale application note [373] shows its use in various applications. Murvay and Groza [374] examine the benefits of using the XGATE to speed up the calculation of several cryptographic algorithms. McAslan [370] describes how the XGATE can be used to perform runtime checking on the integrity of RAM variables or buffers, or to compute alternative versions of important algorithms to ensure the result obtained on the CPU12X is correct. He also describes how the XGATE can check the timing of interrupts or could be used to offload real-time scheduling. Gong et al. [375] show how the XGATE can be used to offload the μC/OS-II time-tick interrupt task, thereby reducing the overhead on the S12X processor. Although these examples exploit some capabilities of the XGATE and the shared access to RAM, they do not make use of the on-chip debug features.
6.4 Background debug module (BDM)

The BDM is a low-cost debugging technology used in many Freescale devices. The BDM enables execution of hardware or firmware commands which are transmitted over a single-wire bi-directional interface. Hardware commands, which allow reading and writing of address locations on the target, are intended to cause minimal interruption to the CPU. Firmware commands require the halting of the active CPU application; the CPU is then used to execute the BDM firmware which is preprogrammed into FLASH memory. This mode is similar to many other embedded software-based monitors, which although low-cost and convenient can be very intrusive to the runtime behaviour of applications.

The BDM does enable a limited amount of on-chip debug to be implemented such as run-stop control, single stepping, plus reading and writing of memory locations. However, when active the BDM firmware is intrusive, and hardware commands are limited to reading and writing address locations with the corresponding data being transferred over a single-wire interface to the debug host. In terms of debug, the BDM capabilities are not dissimilar to those of many JTAG based debug solutions. For the purposes of this research the BDM is used only to download firmware to the target and is therefore not discussed in detail; further information can be found in the MC9S12XE reference manual [12].

6.5 Debug module (S12XDBG)

The S12XDBG is one of the key SoC modules of interest in this research, as the proposal is to use this module and the XGATE co-processor to facilitate on-chip software verification tasks. This module enables monitoring and capturing of XGATE and CPU12X bus activity on the SoC using breakpoint and trace capabilities which are similar to many other OCI solutions. The MC9S12XE reference manual [12] gives a detailed description of the module and its associated register set; the following sections briefly describe its most important features as shown in the simplified block diagram of Figure 20.

![Figure 20: Block diagram of S12XDBG module](image)

The tag and trigger control block manages the generation of breakpoints and can be configured to enable or disable XGATE software breakpoints being passed to the CPU12X. An important feature on CPU architectures which require multiple cycles, or use instruction pipelines, is the ability to discriminate between instructions which are fetched and executed,
and those which are fetched but not executed. The S12XDBG addressed this by providing the option of tagging instructions for which a comparator match has occurred. If an instruction is tagged in the comparator, the tag and trigger control block indicates this to the XGATE or CPU12X; then, if the instruction is executed a ‘taghit’ signal is generated and passed back to the control logic, which can then process the event.

The S12XDBG module includes four comparators to enable monitoring of instructions or data, on either the XGATE or CPU12X bus. As shown in Figure 21, all four comparators allow address matching. Comparators A and C also include the option of setting a data value for the comparator, which combined with a data mask allows the individual bits of the data value to be tested; whereas, comparators B and D do not have the capability to check the data bus value. The comparator control register allows additional configuration for each comparator including: configuring to monitor XGATE or CPU12X bus activity; optional match only on read or write; immediate breakpoint on match; instruction tagging on match; and comparator enabling or disabling.

![Comparators A and C](image1)

![Comparators B and D](image2)

*Figure 21: S12XDBG comparators*

In a default configuration each comparator generates a match signal when the corresponding comparator detects the address specified, and optional data, on the bus. The match control block allows the comparators to be paired (A&B / C&D) with their combined output generating a single match signal when the bus address is inside or outside the range of the two comparators. This is particularly useful, for example, when ensuring that execution or data access is staying within a particular software routine or data structure, or detecting invalid access outside a specified address range. The outputs from the match control block are fed to the tag and trigger control block which determines what action is taken, and as previously described may route the signal to the CPU if tagging is enabled. Register access is also provided to allow reading of match flag bits. However these bits are only set when a match first occurs, and once set, the register bits are only cleared when the module is re-armed. They can therefore give an indication of which comparators were triggered, but no indication of how often or in what order.

In addition to generating an immediate breakpoint on comparator matches, the S12XDBG module includes a state sequencer which can be used to create more complex trigger
conditions. Each state 1, 2 and 3 has a corresponding state control register which determines what transition will occur when a match is signalled. As illustrated in the transition diagram Figure 22, when armed (ARM register bit = 1) the default behaviour is to enter State 1. From this state a match can cause a transition to the final state which generates a trigger signal, or a transition to one of the other two states. Similarly states 2 and 3 can be configured to transition to either the final state or one of the other two states. If disarmed (ARM register bit = 0) the state sequencer returns to the default state 0.

![State sequencer transition diagram](image)

**Figure 22: State sequencer transition diagram**

The trace buffer includes a 64-bit wide, 64 line deep, RAM into which execution trace and data access information can be stored. The trace memory is used as a circular buffer which allows the trigger event to be aligned to the start, middle, or end, of the buffer. Depending upon the trace alignment selected, data capture may be triggered upon arming of the module, for middle or end alignment, or upon entering the final state. Once triggered, tracing continues until the required amount of capture data has been stored (0, 32 or 64 lines after the trigger signal); at this point a breakpoint signal can be generated. Although trace capture is generally used in conjunction with the state sequencer, software initiated capture can also be used; by writing to a register bit the state sequence is forced into the final state thereby triggering trace capture.

The type of data captured to the trace buffer can also be tailored by configuring the trace buffer mode to the situational needs. In normal mode, only program counter (PC) information indicating a change of execution flow is captured. In loop mode, redundant information which would otherwise be captured by the execution of tight software loops is excluded. In detail mode, both the address bus and data bus are captured for all memory and register accesses. In pure PC mode, the PC value for each executed instruction is stored. The organisation of the trace buffer data for each of these modes is described in detail in the MC9S12XE reference manual [12]. When tracing is not active the CPU12X, XGATE, or BDM, can read the captured trace data from the buffer. However, as the buffer data is only accessible through a single 16-bit register location, this requires reading the data in a first-in first-out manner.
6.6 Development platform

The Freescale CodeWarrior integrated development environment (IDE) was used for the
development of all software on the experimental platform. The IDE includes compiler,
debugger, and software simulator, for both the CPU12X and XGATE. Figure 23 shows the
typical configuration for a development platform using this IDE in conjunction with a
MC9S12XE target (Appendix A provides an image of the target development board used for
this research). This arrangement is typical of many development platforms for embedded
targets; for other architectures the BDM interface might be replaced by a JTAG interface with
similar run-stop control and trace data transfer capabilities, or if available, the trace data
might be transferred over an additional parallel interface.

![Image of development platform configuration]

Figure 23: MC9S12XE development platform

Software was written and compiled within the IDE, using the C language, and
downloaded to the target over a USB interface. USB simply acts as a convenient interface on
the development host, data is transferred to and from the MC9S12XE target via its BDM
interface; alternatively any other suitable BDM interface module could be used to connect
directly to the MC9S12XE. Once programmed, the software on the target is normally
dugged by monitoring execution using the S12XDBG module, with data to the
development environment also being transferred via the target BDM interface. However, as
previously stated BDM uses a single-wire bi-directional interface to the target, which has
limited bandwidth, and using the BDM can be intrusive to software executing on the
CPU12X target.

Using the software simulator provided within the IDE is one alternative way to verify the
software and avoid the limitations of the BDM interface. However, while this does allow the
software execution to be monitored in a more convenient manner, it also has limitations. The
most fundamental problem, as with many simulators, is that it does not precisely reflect the
target platform behaviour. This is primarily due to the difference in timing between
simulation and execution on the target, but the absence of simulation models for peripherals
or external components can also be a considerable problem. In fact, the simulator provided in
this case does not include a simulation model for the on-chip S12XDBG module; therefore,
for the experimental platform it could not be used to simulate the execution behaviour.
The alternative proposed and examined in this research, is to use the XGATE as an on-chip debug co-processor which exploits existing OCI to perform runtime monitoring of the embedded application. For the purposes of the ensuing evaluations, the CPU12X is considered to be the target processor upon which the application to be verified, or debugged, is executing. This arrangement is illustrated in Figure 24. The IDE is used for software development in the same manner as before, and the target is programmed using the USB interface. However, once the XGATE reconfigures the S12XDBG for its own use the standard debug features of the IDE are no longer available. Since the objective is to investigate verification and debugging of tasks on the target with reduced data transfer, this runtime data was sent to a status monitoring terminal using a serial communications interface (RS232). This terminal performs no processing on the data, it simply acts as a convenient means of displaying and recording status data received from the on-chip co-processor.

**Figure 24: Development platform using co-processor**

One drawback of using the S12XDBG module in this manner is that it cannot directly trigger execution of code on the XGATE (it is designed to only trigger BDM or the CPU12X) and although the primary purpose of the XGATE is to service interrupts, the relevant CPU12X software interrupt (SWI) cannot be routed to the XGATE. To circumvent this limitation a simple workaround was required. The debug module was configured to trigger the CPU12X SWI handler which contains the code fragment shown below. This code sets one of the XGATE software interrupt flags, thereby triggering the handler and enabling the XGATE to perform as required.

```c
interrupt void SWI_ISR(void) {
    /* Set bit to force execution of XGATE SW trigger 0 handler */
    /* The register mask bit must also be set in the same cycle*/
    XGSWT= BIT0|(BIT0<<8);
}
```
6.7 Summary

The MC9S12XE100 has been presented as a suitable platform to investigate the feasibility of using a debug co-processor to assist with target-level software verification tasks. Many low-cost microcontroller designs only provide one or two hardware breakpoints which are configured over a JTAG interface, whereas high-cost parts may include complex OCI with sophisticated trigger and trace qualification, and wide trace I/O ports. Although S12XDBG module is a proprietary solution, its capabilities are representative of the level of OCI found on many mid-range commercially available microcontrollers; the inclusion of the XGATE within the SoC makes this a very useful (but not ideal) candidate for this investigation.

The Freescale application note by Heisswolf [376] describes methods to debug XGATE code and also provides a concise introduction to the S12XDBG module features. However, the examples provided utilise the debug module in its intended configuration and therefore result in a breakpoint to either the CPU12X or BDM, both of which are intrusive. Although some examples use the state sequencer to check correct, or incorrect, thread execution sequences there is no mechanism to perform a status check on the sequence; this could be done by polling the state sequence flag bits, but this approach introduces overhead and added difficulties regarding sampling. An application note by McAslan [370] shows how the XGATE could be used to perform runtime checks on CPU12X memory locations for errors such as stack overflow or data corruption. However, this approach relies upon access to shared memory locations, and is limited to checking data within this region. Unfortunately, the examples provided also neglect the need for semaphores. Ensuring exclusive access is always advisable when using shared memory locations, whether for writing or reading.

A further significant limitation of using the S12XDBG module as classically recommend, is the lack of timing information which is often required for software verification purposes. However, by combining the capabilities of the XGATE and S12XDBG this author demonstrates that this limitation too can be overcome. The following chapters examine a number of case studies where the feasibility of this alternative arrangement is explored.
CHAPTER 7. VERIFYING RUNTIME BEHAVIOUR

7.1 Introduction

A typical scenario which is often encountered when verifying or debugging software, is where the engineer must verify that the runtime execution advances correctly through a series of expected states or tasks; and for real-time systems, that this execution sequence occurs within predefined time limits. Considering this scenario when using an MC9S12XE, traditional verification approaches would include: adding instrumentation code to generate status output; capturing and outputting of trace data using the BDM module; or as demonstrated by Heisswolf [376], using the S12XDBG state sequencer to enable several event triggers to be combined before a breakpoint is generated. By placing breakpoints at relevant code lines within the relevant application states, the execution can be traced and errant behaviour isolated. However, as previously discussed, for real-time systems halting at breakpoints is not always practical. In addition, all of these approaches are intrusive to varying degrees, and although the state sequencer is less intrusive, the number of states that it can monitor is limited by the available hardware resources.

In this chapter, the author describes a number of experiments which were carried out to evaluate the alternative of using the S12XDBG module to trigger the XGATE co-processor which then monitors the runtime execution. The primary objective of these is to monitor the application to determine if the intended execution behaviour is being followed and within expected time limits; if not, an indication that an error has occurred must be provided. The secondary objective is to examine the feasibility using the XGATE to reprogram the debug module to trigger on new events as application execution proceeds and thereby overcome the limitations of having finite OCI resources such as comparators and state sequencer states.

7.2 Experimental setup

Figure 25 shows the experimental setup used. Software development was performed on a PC host platform and downloaded to the target via USB, while the runtime execution status was monitored on-chip with status output to a terminal connected to the target via RS232. To enable verification of the timing values obtained an oscilloscope was used to monitor the state of LEDs on the development board. These could be turned on and off as required under software control and provided a secondary visual indication of the target execution state.

![Software Development Platform](MC9S12XE Development Board) ➔ USB ➔ Oscilloscope ➔ Status Monitoring Terminal

*Figure 25: Experimental setup*

Figure 26 provides a top-level illustration of the interaction between the XGATE resident
instrumentation code and the CPU12X resident application code as used in all evaluations. Instrumentation code on the XGATE configures the S12XDBG which then monitors the real-time execution of the application on the CPU12X. Any application trace data captured in the trace buffer can also be read by the XGATE. As previously described, due to the lack of a direct hardware interrupt signal it was necessary to use the breakpoint signal (BRK) from the S12XDBG to trigger the CPU12X software interrupt (SWI), which then triggered the XGATE handler. An important benefit of this arrangement is that the instrumentation code resides on the XGATE (with the exception of the SWI handler), which isolates the instrumentation from the application, minimising any undesirable temporal effects, and enables the instrumentation to be removed without altering the application behaviour.

**Figure 26: Application and instrumentation software isolation and interaction**

### 7.3 Monitoring execution sequences

To evaluate the ability of the XGATE to monitor the execution sequence of a number of functions, or tasks, a hypothetical application scenario was created. This consisted of a number of functions which were continuously executed in a sequential order as illustrated in Figure 27 (a). In this case the functions were executed within a super-loop, but they could equally be tasks executed by a RTOS; the aim was simply to monitor the execution sequence.

**Figure 27: Cyclical task execution**

(a) Unmonitored cyclical task execution sequence; (b) Arrangement using XGATE monitoring

Figure 27 (b) illustrates the application scenario with the addition of the XGATE, wherein the XGATE software interrupt handler is triggered on execution of each function. For the
purposes of this evaluation the number of functions in the application execution loop was ten. This could in theory be any number of functions; the important point is that it be more than the number of available S12XDBG comparators making it necessary to update the comparator(s) as execution proceeded.

Aside from limited comparators, another deficiency in S12XDBG module is the fact that is does not include a mechanism to generate a trigger in the event of no activity; for example, if the comparators are configured to trigger on the occurrence of an event but that event never happens, the debug hardware will never be triggered. To address this issue the XGATE code includes a periodic interval timer (PIT) interrupt handler which was configured to execute at intervals of 1 µs. This enabled the XGATE to detect the occurrence of a timeout event where no comparator triggered within a specified time-limit. Operating at 50 MHz this interrupt handler has a minimum execution time of 200 ns; if timeout events for shorter intervals were required a hardware timer could instead be used. However, using a software interrupt in this manner illustrates the capability of the XGATE to manage multiple threads of execution and is more representative of its typical operational configuration.

Figure 28 shows the flowcharts for the CPU12X application code (a), and the XGATE interrupt handler code (b) and (c).

![Flowcharts](image-url)
As shown in Figure 28 (a), functions 4 and 7 include a single line of code which caused the function to loop continuously while a pushbutton in the development board was depressed. This code was added to enable the timeout feature in flowchart (c) to be verified. When the XGATE timer interrupt handler detects that a timeout has occurred it outputs a single line to the serial port; consisting of a string indicating the timeout event, and the number of the function for which the expected trigger event has failed to occur. In the cases shown in the flowchart these failed trigger events will be for functions 5 and 8 (rather than currently executing, or stalled, functions 4 and 7); the resulting serial port output data is shown in Appendix C-1.

Under normal conditions, when execution proceeded as expected, no serial output was provided. Data could be outputted after every trigger event, a particular trigger event, or a number of events. However, as previously discussed, all execution trace output is subject to sufficient bandwidth being available on the I/O interface. In this situation serial transmission was limited to maximum baud rate of 256 kb/s, which although adequate for outputting status information is not sufficient for continuous real-time trace.

The instrumentation code requires an additional 370 bytes of code space and 34 bytes of data space on the XGATE. As previously stated the PIT interrupt handler requires 200 ns to execute, which repeating at regular 1 µs intervals represents 20% utilization of the XGATE. When operating continuously the XGATE increases the power consumption of the SoC by approximately 50%; therefore, the inclusion of the PIT handler alone increases the power consumption of the SoC by approximately 10%. This power consumption could obviously be reduced if a hardware timer was instead used to measure the timeout interval, as this would generate only a single interrupt on the occurrence of a timeout.

The other element of the instrumentation code, shown in Figure 28 (b), requires approximately 1.4 µs of execution time. However, since this is triggered by the application code execution, rather than at regular intervals, the incremental power consumption which this represents is dependent upon the application. Taking a scenario where the application code functions execute on average every 10 µs, then this instrumentation code would increase the utilization of the XGATE by 14% (representing a 7% increase in SoC power consumption); whereas, if the application functions executed at 50 µs intervals the XGATE utilization would increase by only 2.8%.

7.4 Measuring execution timing in real-time systems

In addition to monitoring correct execution sequences, verification that events occur within well-defined time limits is often necessary on real-time embedded systems. To evaluate this scenario the author created an example of a real-time application, which generates an analogue waveform using a PWM output from the microcontroller. Many
embedded control applications use PWM outputs to generate drive signals for power conversion or motor control applications, or as a substitute for an on-chip DAC [377]. In this instance, as illustrated in Figure 29, the PWM output was filtered to generate a simple analogue waveform; in this case a triangular waveform was chosen, but various waveform shapes could be created by suitable adjustment of the PWM duty cycle. The application software also includes an intensity setting which determines the maximum amplitude of the waveform; this can be adjusted using the pushbutton on the development board to increment or decrement the intensity setting. In this scenario, the timing of the application execution on the target is critical to ensure correct waveform generation and this timing must be verified.

Figure 29: Example of embedded system generating a real-time signal

Checking the timing of a function or entire application using debug hardware and breakpoints to halt execution may be possible in some limited situations, but for real-time systems, instrumentation software is generally added to the application. Execution timing data can then be obtained either using on-chip timers, or by outputting status signals on I/O pins which are then probed by external tools. Adding instrumentation to application code in this manner is intrusive; not only is there a possibility of introducing bugs into the code, but there is also the possibility that the instrumentation may alter the timing.

The experimental setup, as described before, was used to evaluate the alternative approach of using the XGATE as a less intrusive means of instrumenting the target platform to verify its timing. However, rather than having a general timeout limit as was the case in the last example, in this instance the instrumentation code was intended to measure the timing of the functions. As shown in Figure 30 (a) the application code consists of a timed loop which sequentially executes three functions: ReadKeys(), ComputeOutput(), and OutputPWM(). Each function is assigned an individual execution time deadline, which is intended to ensure that the total time for the three functions does not exceed the required loop time.

Figure 30 (b) and (c) show the flowcharts for the two XGATE co-processor interrupt handler functions. The PIT interrupt is configured to trigger at 1 μs intervals and its handler function simply increments a time counter value at each occurrence. The software interrupt is triggered when each application function is executed. As shown in Figure 30 (b), this acts as
the instrumentation handler function. Using the time counter value the software can measure
the time taken to execute each function. An execution time limit for each function included in
the instrumentation code enables the co-processor to determine whether the function
completed within the expected time, or if a timing violation has occurred.

As shown, the software resets the time counter after storing its value; the stored value is
used to record the minimum and maximum duration for each function and to check the
execution time against the deadline provided. To enable verification of the timing results, the
handler also outputs a status signal on the four development board LEDs. The debug module
is then updated with conditions for the next trigger event. If status information is due to be
sent to the monitoring host, it is then loaded into the serial buffer. The rate at which data is
sent to the monitoring host is configurable to the application needs; in this case the
instrumentation code was written to output the current status information after 100,000
iterations of the application loop (approximately every 5 seconds).

In this instance the instrumentation code requires an additional 670 bytes of code space
and 69 bytes of data space on the XGATE. As previously stated the PIT interrupt handler
requires 200 ns to execute, which repeating at regular 1 µs intervals represents 20%
utilization of the XGATE. The software interrupt handler which in this case implements the
bulk of the instrumentation code requires a maximum execution time of 3.5 μs. During each
loop of the application this handler is triggered four times, which represents a further 28%
(3.5 μs x 4 / 50 μs) utilization of the XGATE. Therefore, executing the entire instrumentation
code on the XGATE increases the SoC power consumption by approximately 24%.

The ReadKeys() function was written with an inherent flaw whereby each key press was
processed independently, introducing a delay of 10 μs for each key. Therefore, when both
keys were pressed simultaneously this function required an additional 20 μs to complete,
causing it to exceed its deadline of 20 μs. Appendix C-2 shows the data sent to the monitoring
terminal; showing the data when: no key was pressed, after one key was pressed, and after
two keys were pressed simultaneously. Table 5 summaries the output status data obtained
after both keys were pressed. This shows that the maximum execution time for ReadKeys
was measured as 29 μs, and that the instrumentation code detected this deadline violation.

<table>
<thead>
<tr>
<th>Function</th>
<th>Execution Time (μs)</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>ReadKeys</td>
<td>7</td>
<td>29</td>
</tr>
<tr>
<td>ComputeOutput</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>OutputPWM</td>
<td>9</td>
<td>12</td>
</tr>
</tbody>
</table>

*Table 5: Status data outputted after deadline exceeded*

The timing results obtained from the XGATE were verified by measurement of the LED
drive signals using a calibrated Agilent MSO6054A Oscilloscope. The instrumentation code
was written such that a different LED was turned on while each function was executing. The
execution timing of each function and its frequency of occurrence could therefore be verified
by measuring the associated output pulses using the oscilloscope. As shown in Figure 31 (a),
the timing values obtained are very close to the expected values, considering that the software
timing values have inherent discretisation errors.

![Figure 31: Execution timing waveforms captured by oscilloscope](image)

*(a) waveforms before any key was pressed; (b) waveforms when violation detected*

Whenever a timing violation was detected, the instrumentation code also toggled a fourth
LED, which served as an external error signal. By triggering the oscilloscope on this signal it was possible to verify the timing of the application functions the instant that a violation occurs. As shown in Figure 30 (b), as measured by the oscilloscope, the timing of the violating ReadKeys() function is 30.9 µs, which also corresponds well with the timing results obtained from the instrumentation code on the XGATE.

Having identified the function which caused the timing violation the application code error can be easily identified and resolved. In this instance, simply removing the flaw in the ReadKeys() function addressed the timing violation and enabled the application loop to complete within the expect time. If necessary the co-processor can also export the trace buffer contents to the monitoring terminal. Appendix C-3 shows the trace buffer contents outputted using modified instrumentation code; in this case the XGATE was configured to output data after the ReadKeys() execution timing exceeded 28 µs. It clearly shows the address of the branching instruction that the CPU12X repeatedly executed prior to completing ReadKeys().

However, the variation in minimum and maximum execution time for the functions points to another problem with the design of this example application. Because the application relies upon a simple sequential loop, any variations in the execution time for the first two functions will impact upon the starting time of the OutputPWM() function. This results in execution timing jitter which is evident on the oscilloscope waveforms in Figure 31 (a); a close-up of this waveform is given in Appendix C-4, Figure 48, which shows that the jitter is approximately 1 µs. This jitter was not artificially introduced; it was simply a consequence of using conditional code statements within the functions resulting in subtle timing variations between execution loops. Obviously elimination of conditional code from each function or an entire application is not practical. Nonetheless timing jitter, such as this, can introduce distortion into the generated output waveform which is clearly undesirable. Therefore, for applications which include events that are required to be strictly periodic, it is necessary to verify the periodic timing of these events and not just the application or function level execution timing. Ways in which a co-processor can assist this verification are examined in the following section.

### 7.5 Measuring periodic events in real-time systems

As shown in the previous scenario even when the application timing does meet expected deadlines, the timing of periodic events may still be subject to jitter. This jitter can introduce additional noise into the output waveform which may be difficult to observe on the output. In this particular scenario due to the filtering of the output and the quantisation noise inherent when using PWM hardware to generate a waveform, this distortion is not apparent on the output waveform.

Whether apparent or not, this jitter can nonetheless have a detrimental impact, particularly
in real-time control applications. The impact of jitter in distributed control systems has been studied for many years [378]; but it also impacts upon systems consisting of just a single controller. Albertos and Crespo [379] examine the scheduling of multiple control tasks on a single controller. Littlefield-Lawwill and Kinnan [151] describe how interrupts and access to shared resources can introduce jitter in IMA systems. Lincoln and Cervin [380] described a MATLAB toolbox which enables the impacts of jitter on the performance of control systems to be modelled. Martí et al. [381] examine the sources of jitter in control applications and describe how the control parameters can be adjusted to account for jitter. Later, Martí and Velasco [382] propose a scheme whereby the actuation period is fixed but irregular sampling is allowed, and the control value is computed based upon a prediction of the state at the actuation instant. In Lozoya et al. [383], they apply this control technique and show that allowing sampling to occur closer to the actuation instant has the additional benefit of being able to adjusting for more disturbances in the plant under control. Phatrapornnant and Pont [162] show that DVS can cause significant jitter in embedded systems; and propose an algorithm which reduces jitter for tasks with real-time deadlines whilst facilitating DVS.

To enable detection of this jitter, or verification of its absence, it is again necessary to monitor the execution on the target. In this case the instrumentation software residing on the XGATE was modified to measure the period between each update to the PWM module (the application code was not modified). This is similar to the approach adopted for the previous example where the XGATE software included one timer interrupt handler to measure the time, and a second handler to configure the debug module and respond to matching events. However, in this case since the timing of changes to the PWM output is the parameter which must be verified, the debug module was instead configured to monitor write accesses to the PWM hardware registers rather than execution of a particular instruction on the CPU12X.

This distinction is important and has two significant benefits: firstly, it enables timing of the actual update to the register, eliminating any delays which may occur after the execution of the software instruction; secondly, any writes to this register from any source including application code, errant code, or other peripheral devices, will be detected. The more traditional approach of adding software instrumentation into the application code is not only intrusive, but it is also unable to provide these two benefits. The pseudo-code which follows illustrates a typical application code instrumentation approach. Although it may appear that the instrumentation code will execute immediately after the PWM output has changed, this ignores the possibility that an interrupt could occur after the register has been updated but before the instrumentation code is executed. An interrupt may be a simple handler which quickly returns to the application, or if using an RTOS could trigger a context switch to another task.
//Write new output data to PWM register
PWM_register = new_value;

//Toggle output pin or send data to a port
Instrumentation_Output();

A flowchart for the XGATE instrumentation software is given in Figure 32. When the handler is activated it records the time since the last occurrence and if necessary updates the stored minimum and maximum. While these values provide the essential information required, in this example it was also considered useful to record the timing data for a number of consecutive occurrences and the corresponding value contained in the PWM output register; therefore, this additional information was captured and written to a circular buffer.

---

**Figure 32: Flowchart for XGATE instrumentation to verify periodic events**

The modified instrumentation code requires 608 bytes of code space and 1232 bytes of data space on the XGA TE. As before, the PIT interrupt handler utilizes of the 20% XGATE processing capacity, but the software interrupt handler has a reduced maximum execution time of 2.3 μs. Also, since this instrumentation is only triggered once during each application loop it only increased the XGATE utilization by 4.6%. Therefore, the entire instrumentation increases the SoC power consumption by approximately 12.3%.

In this instance the significant increase in data space is due to the addition of a buffer to store 400 entries for both the PWM output and timing period data; the maximum size of the buffer is only limited by available RAM. Having a large record of the timing values and PWM register contents enables the instrumentation code to export a trace of recent measured and captured data. Unfortunately, sending this data over a slow interface requires a significant
amount of processing time and can interfere with the measurement being performed. However, the time at which this data is sent can be tailored to the situational needs. In this example, a software trigger signal was generated each time a new maximum or minimum was detected, this trigger signal was used to synchronise the circular buffer such that the most recent minimum event would be captured at the midpoint of the buffer, and when the buffer was full, the data was exported.

Outputting the captured values to the monitoring terminal as comma separated ASCII strings enabled the data to be imported into a spreadsheet application and plotted. Figure 33 shows a plot of the data obtained after the application code processed a key press (which increased the waveform intensity value). There is no perceivable impact of jitter upon the values written to PWM register, and these values would appear to generate the desired triangular waveform. However, the captured periodic timing data shows two significant variations at the midpoint of the plot which give to a maximum time of 60 µs and a minimum of time 38 µs. This timing variation corresponds with the occurrence of an increment to the intensity setting, which causes the ReadKeys() function to execute for an additional 10 µs.

![Figure 33: Plot of captured PWM register values and the period between updates](image)

The oscillations which can be seen on the timing data, between 49 µs and 50 µs, after the midpoint and when no key press is occurring correlate well with the timing jitter measured previously; but, given the timing resolution of 1 µs this jitter could in reality be slightly larger. The larger minimum and maximum variations are obviously more significant, and likely to cause greater distortion. The occurrence of the maximum and minimum periods in consecutive measurements are interrelated and is more easily understood with reference to the illustration of the application timing in Figure 34. The three application functions are shown to always complete with the desired loop time of 50 µs with any slack being taken up by the loop timer. During the third iteration execution of the OutputPWM function is delayed, thereby extending the period from its last occurrence, and since the subsequent iteration does not exhibit the same the delay the next period measured is consequently shorter.
7.6 Results

The previous experiments were intended to evaluate the feasibility of using an on-chip co-processor and existing instrumentation hardware to observe and report on the runtime execution of an application on its target platform. All three experiments demonstrated that it is possible to monitor execution of an application in a minimally invasive manner. No in-line instrumentation of the application code was required, and with the exception of a simple interrupt handler no additional burden was placed on the application processor. While this handler obviously does have a slight impact (interrupt context switch and restoration, plus execution of a single instruction) it was only needed to circumvent a limitation of this particular architecture. This overhead could be eliminated if the OCI directly triggered the co-processor which would result in a completely non-invasive solution.

The first experiment showed that it was possible to use limited OCI resources (only single comparator was used) which could be reconfigured on-the-fly by the co-processor to trigger on sequential events. Using only a fraction of the OCI resources the co-processor could successfully monitor the execution, and correctly detected the occurrence of the timeouts when keys on the development board were pressed causing the selected functions to loop. Although the example application and error conditions were somewhat contrived, the experiment demonstrates the essential capabilities of the proposed approach. The ability to reconfigure the OCI from a co-processor facilitated the monitoring of more functions than would ordinarily be possible with the limited number of comparators available. The additional OCI resources including the three extra comparators and the state sequencer could have been used to generate triggers for more complex events, but many embedded processors have a limited number of comparators and do not include features such as a state sequencer.

Using a co-processor provided a number of additional benefits, including the ability to have a separate task monitoring the timing of these events, and when a timeout occurs sending an error indication to an external host; detection of such timeouts is important in real-time embedded systems, but is not possible using the existing OCI alone. A further limitation of solely using the existing OCI is that the state sequencer only triggers upon entering a final state. No trigger is generated if the events leading to the final state fail to occur; and if, for example it was necessary to record the number of times the particular state is entered, the
existing OCI cannot provide this. A co-processor triggered on each event occurrence can send regular status output data to indicate the current state of execution or a summary data on which states have been entered and how often. However, as previously highlighted the frequency of this output data is constrained by available interface bandwidth.

The second experiment demonstrated the ability to measure the execution timing for a hypothetical real-time waveform generation application consisting of a small number of functions executing in a cyclical fashion. In this case the co-processor was used to successfully measure the duration of each function, stored the minimum and maximum data and reported this condensed data at regular intervals. The data obtained was compared against timing data obtained using an oscilloscope and, given the resolution chosen for the timer, showed a close correlation to the co-processor results. An execution deadline for each function was also provided in the instrumentation code, which enabled the co-processor code to check and indicate if a timing violation had occurred.

By altering the co-processor instrumentation code it was also possible to output the contents of the trace buffer when a timing violation was detected. Although the contents of this buffer as shown in Appendix C-3 includes only three unique change of flow (COF) instruction address locations, the ability to selectively export trace data when notable events occur is nonetheless beneficial; it is also more practical than trying to continuously stream vast amounts of runtime data through bandwidth limited interfaces, much of which may not be relevant, but all of which must be interpreted off-line. The single repeated address which consumes the majority of the trace-buffer illustrates the deficiency in having a limited trace buffer when long execution loops are encountered. The debug module can be configured such that the repeated trace information from tight execution loops is not stored in the trace buffer; however, in the case of this experiment that would have eliminated the vital information as to where execution time was being expended.

The third experiment used the same hypothetical real-time application, but was focused on the detection of jitter in the timing between periodic updates to the output. In this case the co-processor software was designed to measure the timing of all writes to the peripheral output register of interest, using the OCI to monitor for the relevant on-chip bus activity. The approach of monitoring the output register, rather than the application code line which writes to the register, was shown to be feasible and provides the significant benefit of being able to detect any write (or read) access to the register, whether intended or not, using a single comparator.

In this instance the instrumentation code included a circular buffer (with 400 elements) into which the period between writes and the corresponding PWM output register values were stored. This data could then be sent to the monitoring terminal which allowed recording
and plotting of the data. This data can be sent at any convenient time or could for example only be sent after notable jitter was detected. The plots showed that the jitter can easily be detected from the timing data, but also showed that examining PWM output data alone would be insufficient. There are many software and hardware instrumentation solutions which aim to facilitate the capture of all changes to a particular variable or register, but without the corresponding timing information this data may be not sufficient for many real-time applications. Solutions which rely on regular sampling of a memory or register location may fail to detect transient alterations to the location, or would require sampling rates higher than the possible rate of change to guarantee detection. For situations where the alteration is caused by errant access to the location this rate may be impossible to predict.

Although the experiments demonstrated the feasibility of the alternative runtime monitoring approach proposed, they also highlighted some limitations. Perhaps the most significant issue is the execution time required for the co-processor to be activated, process the data, and reconfigure the OCI for the next trigger event; this sets a limit on the time between successive triggers. The instrumentation software must therefore be written such that this time is optimised. The accuracy of timing data can be maintained by ensuring that the timing value is stored and reset immediately, because the XGATE timer interrupt handler was configured to have a higher priority than the software interrupt it will continue to be updated while instrumentation routine is progressing. However, the analysis of the data must be optimised to enable the next trigger to occur in the shortest possible time.

The other significant limitation was the time required to send data via the serial port at a relatively slow baud rate (256 kb/s). Although the instrumentation code included functions to efficiently convert data values into ASCII strings suitable for transmission, and a serial buffer was provided to enable interrupt driven transmission, the maximum data rate is still limited. Of course, transmission of data values in a compressed format or simply in binary format would be more efficient, but this would require additional processing of the data at the receiver to parse it into an intelligible or readable format. Using ASCII output from the instrumentation code negates the need for this additional tool, is more convenient, and more useful for record keeping and auditing purposes.

The rate at which data must be outputted is also dependent upon the application. In the experiments described the maximum duration of the functions was in the order of tens of μs, in many real-world cases, the functions or tasks could execute for much longer durations or the instrumentation might be used to monitor functions which execute less frequently.
7.7 Summary

Three experiments were carried out to evaluate the concept of using existing OCI in conjunction with a co-processor to monitor the runtime execution of application software on a target platform. The experimental setup used was similar to the development environment used in many embedded systems development scenarios. All software was written using the IDE provided and downloaded to the target development board using the USB/BDM interface. The target was connected to a terminal via a standard serial interface which provided a convenient means of displaying status output data from the co-processor. An oscilloscope was used to verify the timing results obtained from the co-processor. Other than acting as calibrated reference for measurements this was not needed for the verification tasks.

The results obtained show that in each case the approach proposed is feasible and in some cases has distinct advantages. However, these examples were focused on verification of runtime execution sequences and in particular temporal properties, which although vitally important in real-time systems address only a subset of software verification activities. The next chapter examines a medical device case study where the requirements to be verified are driven by user level safety requirements, which translate into software design requirements. To ensure that the final software implementation includes the functionality needed to meet these essential device safety requirements, it is necessary to perform runtime verification.
CHAPTER 8. RUNTIME VERIFICATION OF REQUIREMENTS

8.1 Introduction

In this chapter the author describes a case study using a continuous positive airway pressure (CPAP) medical device, to evaluate the feasibility of using OCI and a co-processor to perform the runtime verification of safety-related software requirements. The two most fundamental requirements for a medical device are efficacy and safety. Efficacy can be established through rigorous clinical trial, or by demonstration of equivalence to a predicate device; however, before a device can be put in use, for either trials or therapy delivery, it must first be shown to be safe for use. Therefore, during the design of a medical device, considerable effort is expended upon verification of safety-related requirements.

These requirements can be broadly categorised into system-level design requirements which are intended to ensure the safe functioning of the device hardware and software, and user-level requirements which are intended to ensure its safe and effective use. Safety-related requirements may require risk mitigation strategies involving many diverse aspects of the device design, including mechanical, electronic, and software features. For the purposes of this research, the requirements of interest are those software requirements for which runtime verification is essential; a CPAP device serves as a worthwhile representative medical device.

8.2 CPAP device and requirements for adjustment of settings

The purpose of a CPAP device is to assist breathing by keeping the patient’s airways open. CPAP devices can be used in neonatal ICU settings where premature infants may have difficulty breathing unaided, or for older patients who suffer from medical conditions such as sleep apnoea. Figure 35 shows a simplified diagram of such a device, which is intended to provide a temperature controlled gas mixture to the patient at a continuous pressure.

Figure 35: Simplified system-level block-diagram of a CPAP device

As shown in Figure 35 the device includes: a microcontroller which manages the
functional behaviour, and to which various sensors are attached; a DAC to generate the necessary control signals; supervisory circuits such as reset control and watch-dog timer (WDT); and non-volatile storage. The basic operation of the device is briefly described as follows: Air and Oxygen (O$_2$) supplies, either from a main hospital supply or portable gas cylinders, are attached to the device. Using proportional solenoids under software control, the device adjusts the flow-rate of each gas supply such that desired O$_2$ mixture is obtained, as measured by the O$_2$ sensor. The temperature of the gas is adjusted to the desired setting by passing it through a temperature controlled humidification chamber. The air pressure at the patient nasal cannula is sensed using an independent air-line and the gas flow rates adjusted to maintain the desired pressure. Although this is a much simplified description, the three essential gas-delivery settings are seen to be: patient gas pressure, O$_2$ concentration, and gas temperature.

The device includes an uncomplicated user interface upon which these settings can be displayed and adjusted during operation; however it is also essential that these settings are not inadvertently altered. Risk mitigation measures against accidental alteration could include mechanical design requirements such as the mounting of the device on an IV pole, so that the user interface is vertical and objects cannot be left resting on the keypad, or electronic design requirements to include circuitry enabling detection of opens or shorts on the keypad. Even with these mitigations in place, it is still possible that the settings could be corrupted or that an untrained person may accidentally change the settings; the probability of this risk occurring and its severity are increased by the fact that a CPAP device may be operated in an environment without constant supervision, or where untrained persons such the patient’s family members may be present.

The user interface software, which manages the adjustment of these settings, is therefore designed to meet the following additional software requirements:

1) Any user initiated gas-delivery setting change shall be aborted if the accept key is not pressed within 10 seconds of the last adjustment (i.e. increment or decrement).

2) All gas-delivery settings are to be stored in a data-structure which includes a checksum of the setting values; this checksum shall be verified each time the data-structure is read.

3) When reading the data-structure each gas-delivery setting value shall be verified to be within defined upper and lower boundaries.

The first requirement imposes a two-step operation for user initiated changes to the gas-delivery settings, which reduces the likelihood of accidental adjustment. The other two requirements are intended to minimise the risk of the settings being altered by: errant code, a SEU, or a physical fault in the data storage location. As each of these requirements relates to
the safe use of the device, the associated software must be verified to ensure that these features are present and functional at runtime.

8.3 Experimental application platform

To examine whether a co-processor and OCI could be used to perform runtime verification of these software requirements, a suitable application platform is required. The original software design employed a proprietary state-based application kernel to manage the operation of the device. For confidentiality reasons, details of this software cannot be disclosed; instead, a simplified state-machine was created to replicate the behaviour of the key software functions relevant to adjustment of the three main gas-delivery setting. Figure 36 shows the statechart for this much simplified version of the application software, as was used in the subsequent experiments.

![State diagram for CPAP setting adjustment](image)

This statechart was created using a graphical modelling tool, which is freely available from Quantum Leaps [235]. In addition to enabling the design of statecharts, this tool can automatically generate source code for use with the event-driven frameworks and kernels presented by Samek [142]. However, in this case, the source code generated from the statechart was modified slightly to enable it to be used with a custom kernel, which more closely reflects the behaviour of the original CPAP device.

As can be seen from the statechart, while a setting is being displayed, in the `Display_settings` state, the user can elect to adjust this by pressing the ‘accept’ key; this causes a transition to the `Change_setting` state, where the setting can be incremented or decremented. While the setting is being adjusted, an out-of-range value will cause a transition to `Settings_data_error` state. If the user fails to accept the setting change within the required time, or presses the cancel key, the change is aborted and the state transitions back to `Display_settings`. If the setting change is accepted, the new value is stored and the copy used
by the gas delivery software is also updated. In each state where the settings data-structure is read, its checksum is verified and the settings are checked to ensure they are within defined ranges; any error will cause a transition to the *Settings_data_error* state. The actions performed within the *Settings_data_error* state include logging details of the error condition in an EEPROM and storing details of the previous stable operational state. The application does not leave this state, but instead waits for the WDT to generate a reset.

To enable the co-processor to verify the timing requirements, a PIT handler routine was again included in the XGATE code. As shown in Figure 37 (b), this simply increments a time counter value each time it is triggered; in this case the timer interrupts occur at 1 ms intervals, which is four orders of magnitude less than timing requirement to be measured and verified but does not significantly burden the XGATE. However, in the subsequent experiments, rather than using interrupts to synchronise the execution of the monitoring code on the co-processor as used in the previous examples, the alternative of using a polling technique was explored. Figure 37 (a) shows the flowcharts for this alternative scheme.

Since all execution on the XGATE must be initiated by interrupts, this code is once again written within an interrupt handler; however, in this case, the interrupt handler never terminates. Once triggered, by the CPU12X setting the appropriate software interrupt flag during start-up, the software configures the debug module comparators to monitor events of interest. The code then waits for a match to occur, performs the necessary analysis, outputs any status data to the serial buffer, and returns to waiting for the next event.

![Flowcharts for XGATE monitoring of CPAP application](image)

**Figure 37: Flowcharts for XGATE monitoring of CPAP application**

Appendix D-1 gives a transcript of the status output from the co-processor when this method was used to monitor each event signal occurrence on the experimental application. As shown, the co-processor can successfully monitor the events and can also provide an indication of state changes as they occur.
8.4 Verifying setting change timeouts

The first requirement to be verified is that ‘any user initiated gas-delivery setting change shall be aborted if the accept key is not pressed within 10 seconds of the last adjustment’. As shown in statechart of Figure 36 the application is designed such that all setting changes occur in the Change_setting state, and if the change is not accepted within the required time, the application is expected to return to the Display_settings state. It is therefore necessary to monitor the application to verify the timing of this transition.

As shown in Figure 38 the co-processor software was written to monitor the application and measure the timing of this transition. In the same manner as the application software, when the Change_setting state was entered and an increment or decrement event was signalled to that state, the co-processor software reset its local time counter value. On a subsequent transition to the Display_settings state the value of the time counter was outputted from the co-processor software to the status monitoring host, via the serial port. The transcript of this output data showing the timing values which were recorded, with a resolution of 1 ms, is provided in Appendix D-2.

![Flowchart of instrumentation code for verifying timeout requirement](image_url)

As can be seen from the output data, the transition due to the timeout does occur within the required 10 second limit; however, the results also show a variation of up to 8 ms in the times recorded. This variation is due to the fact that the increment and decrement events are not synchronised to the timer tick event of the application. Timer tick events are triggered at regular 10 ms intervals, whereas the decrement and increment events are generated by unsynchronised background scanning of the keypad inputs. Therefore, such timing variations are to be expected and the maximum deviation is dependent upon the timer tick resolution. To confirm this, the test was repeated with an increased application timer tick interval of 20 ms; the results obtained from both test are summarised in Table 6. As can be seen, the variation in
the measured values does indeed increase with larger timer tick intervals. This requirement should therefore include a measurement tolerance of one timer tick.

<table>
<thead>
<tr>
<th>Timer Tick interval</th>
<th>10 ms</th>
<th>20 ms</th>
<th>10 ms</th>
<th>20 ms</th>
<th>10 ms</th>
<th>20 ms</th>
<th>10 ms</th>
<th>20 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Captured data (ms)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 ms</td>
<td>9992</td>
<td>9995</td>
<td>10000</td>
<td>9994</td>
<td>9995</td>
<td>9987</td>
<td>9997</td>
<td>9990</td>
</tr>
<tr>
<td>20 ms</td>
<td>9993</td>
<td>9992</td>
<td>9997</td>
<td>9995</td>
<td>9992</td>
<td>9996</td>
<td>9992</td>
<td>9996</td>
</tr>
<tr>
<td>10 ms</td>
<td>9994</td>
<td>9987</td>
<td>9998</td>
<td>9993</td>
<td>9998</td>
<td>9985</td>
<td>10000</td>
<td>9991</td>
</tr>
<tr>
<td>20 ms</td>
<td>9995</td>
<td>9997</td>
<td>10000</td>
<td>9997</td>
<td>9998</td>
<td>9999</td>
<td>10000</td>
<td>9999</td>
</tr>
<tr>
<td>10 ms</td>
<td>9993</td>
<td>9984</td>
<td>9996</td>
<td>9988</td>
<td>9991</td>
<td>9999</td>
<td>9995</td>
<td>9984</td>
</tr>
<tr>
<td>20 ms</td>
<td>9992</td>
<td>9984</td>
<td>9996</td>
<td>9988</td>
<td>9991</td>
<td>9985</td>
<td>9992</td>
<td>9984</td>
</tr>
<tr>
<td>Min</td>
<td>9995</td>
<td>9997</td>
<td>10000</td>
<td>9997</td>
<td>9998</td>
<td>9999</td>
<td>10000</td>
<td>9999</td>
</tr>
<tr>
<td>Max</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max-Min</td>
<td>3</td>
<td>13</td>
<td>4</td>
<td>9</td>
<td>7</td>
<td>14</td>
<td>8</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 6: Setting change timeout values recorded from application

8.5 Verifying application checks on data-structure integrity

The second requirement to be verified is that ‘all gas-delivery settings are to be stored in a data-structure which includes a checksum of the setting values; this checksum shall be verified each time the data-structure is read’. As shown in statechart of Figure 36 the application is designed such that the data-structure is read on entry to each state, and if a checksum error is detected a transition to the Settings_data_error state should occur.

In this case the co-processor was used to inject invalid checksum values into the data-structure at specific times and monitor the subsequent application behaviour, as illustrated in Figure 39. As before, this was achieved using the co-processor to monitor event signals and the current execution state of the application. The instrumentation code then exploits the fact that data-structure and state variables can be accessed via shared-memory, enabling the co-processor to read and write these values as desired.

![Figure 39: Flowchart of instrumentation code to verify data structure integrity](image-url)
As shown in Figure 36, the default application behaviour in the \textit{Settings\_data\_error} state is to wait until an external watch-dog timer reset occurs; however, the co-processor instrumentation code was written such that the current state was stored before the checksum was altered, and upon entering the \textit{Settings\_data\_error} state the valid checksum and previous state were restored. This eliminates the need to reset the device, and also allows the software to proceed as if the fault had not occurred.

An unaltered transcript of the co-processor output data for the four test states is provided in Appendix D-3 and Appendix D-4, in these four cases an invalid checksum is inserted on entry to the state. Additional comments have been added to the transcript below to illustrate the behaviour of the instrumentation code.

-STARTUP-
\begin{itemize}
  \item \textbf{State: DISPLAY\_SETTINGS}
  \item \textbf{Event: ACCEPT\_SIG}
  \item \textbf{State: CHANGE\_SETTINGS}
  \item \textbf{Event: ACCEPT\_SIG}
  \item \textbf{State: STORE\_NEW\_SETTINGS}
  \item \textbf{Event: SETTINGS\_CHANGED\_SIG} \hspace{1cm} ** changed checksum **
  \item \textbf{State: UPDATE\_GAS}
  \item \textbf{Event: CHECKSUM\_ERROR\_SIG}
  \item \textbf{State: SETTINGS\_DATA\_ERROR} \hspace{1cm} ** restored checksum ** \hspace{1cm} ** forced state change **
  \item \textbf{State: UPDATE\_GAS}
  \item \textbf{Event: UPDATE\_COMPLETE\_SIG}
  \item \textbf{State: DISPLAY\_SETTINGS}
\end{itemize}

The results obtained show that the application software does indeed check the data-structure checksum in entry to each state and correctly transitions to the error handling state. However, the application can remain in the \textit{Display\_settings} or \textit{Change\_setting} state for an indefinite period of time. It is therefore useful to also consider the case where the checksum is altered after these states have been entered. The commented transcript below shows the results obtained when the instrumentation code was altered to examine this scenario; the unaltered transcript for the test conducted on both states is provided in Appendix D-5.

-STARTUP-
\begin{itemize}
  \item \textbf{State: DISPLAY\_SETTINGS} \hspace{1cm} ** changed checksum **
  \item \textbf{Event: INCREMENT\_SIG}
  \item \textbf{Event: DECREMENT\_SIG}
  \item \textbf{Event: CANCEL\_SIG}
  \item \textbf{Event: ACCEPT\_SIG}
  \item \textbf{State: CHANGE\_SETTINGS}
  \item \textbf{Event: CHECKSUM\_ERROR\_SIG}
\end{itemize}

Co-processor injects an invalid checksum value after the state under test is entered.

Application execution proceeds, but without detecting checksum error.

Checksum error is detected when state transition occurs; i.e. when data-structure is read again.
** State: SETTINGS_DATA_ERROR **
** restored checksum **
** forced state change **
** State: DISPLAY_SETTINGS **
Event: ACCEPT_SIG
** State: CHANGE_SETTINGS **
Event: INCREMENT_SIG
Event: DECREMENT_SIG
** State: DISPLAY_SETTINGS **

Although a strict interpretation of the requirement would suggest that the application meets the stated requirement because it does detect the invalid checksum, when the data-structure is next read; however, as can be seen from the transcript, this test highlights a potential flaw in the application design. The requirement as written provides poor protection against data corruption while the application remains in both the Display_settings and Change_setting state. Corruption of this data structure due to a SEU or errant code elsewhere in the application could occur at any time; therefore it would be preferable if the checksum were checked at regular intervals to reflect this possibility. This could be simply achieved by verifying the checksum on each timer tick event, or another suitable regular interval.

** 8.6 Verifying application checks on setting range **

The third requirement to be verified is that, 'when reading the data-structure each gas-delivery setting value shall be verified to be within defined upper and lower boundaries'. The statechart shows that a data_range_error event should be generated in each of the operational states of the application, and that this event should trigger a transition to the Setting_data_error state. To verify that the application does check these boundary values, and does transition to the error state as expected, it is necessary to inject invalid values into the settings data-structure at runtime. Table 7 gives the allowable values for each of the three main CPAP settings.

<table>
<thead>
<tr>
<th>Patient gas pressure</th>
<th>Minimum</th>
<th>2 cmH(_2)O</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum</td>
<td></td>
<td>10 cmH(_2)O</td>
</tr>
<tr>
<td>O2 concentration</td>
<td>Minimum</td>
<td>21%</td>
</tr>
<tr>
<td>Maximum</td>
<td></td>
<td>100%</td>
</tr>
<tr>
<td>Gas temperature</td>
<td>Minimum</td>
<td>33°C</td>
</tr>
<tr>
<td>Maximum</td>
<td></td>
<td>41°C</td>
</tr>
</tbody>
</table>

*Table 7: Allowable values for CPAP settings*

In this case, as shown in Figure 40 the co-processor resident instrumentation code was written such that when the application was entering the state under test, one setting value was set to a value beyond its limit. When the application subsequently entered the Setting_data_error state the co-processor would report the invalid value, restore the correct value, and restore the state variable to the test state. This would then cause the test state to be
entered once again and the next limit value to be tested to be written to the appropriate setting. This process repeats until each setting has been put at a value beyond its lower and upper limit.

![Flowchart of instrumentation code to verify setting range checks](image)

**Figure 40: Flowchart of instrumentation code to verify setting range checks**

The behaviour of the instrumentation code is illustrated in the commented transcript fragment below; the full output transcript for each state tested is given in Appendix D-6.10. As can be seen, the co-processor can stimulate and monitor the application to verify that the settings are checked against their boundary values as required; however, as in the previous case, this requirement is also weak as this checking is only performed on entry to the state.

-STARTUP-

!! Write out of range value

** State: DISPLAY_SETTINGS
** Event: DATA_RANGE_ERROR_SIG

** State: SETTINGS_DATA_ERROR
** Temperature = 32
** Restored setting
** Forced state change
!! Write out of range value

** State: DISPLAY_SETTINGS
** Event: DATA_RANGE_ERROR_SIG

** State: SETTINGS_DATA_ERROR
** Temperature = 42
** Restored setting
** Forced state change
!! Write out of range value

** State: DISPLAY_SETTINGS
** Event: DATA_RANGE_ERROR_SIG

** State: SETTINGS_DATA_ERROR
** Pressure = 1
** Restored setting
** Forced state change
...
8.7 Results

The results obtained show that runtime verification using the proposed approach is practical, and is achievable without the need to alter the application code. The ability to inject faulty setting values into the applications data-structure at strategic times enables the verification of the intended behaviour. This approach also enabled the testing of alternative scenarios such as the simulation of data corruption when the application was not performing its routine checks on the data. Although the application does eventually detect these errors, these tests show that data corruption could go undetected for prolonged periods, which suggest that more stringent requirements may be needed. The ability to time the state transitions, using a higher resolution timer than the application timer tick, again showed that while the requirement was met, a more precise requirement specification may be required.

For these experiments the alternative approach of polling the OCI debug module from the co-processor was used; and this was shown to be feasible. Although this approach offers the distinct benefit of not requiring an interrupt handler operating on the application processor it does have the drawback of monopolising one XGATE interrupt level; however, because the XGATE architecture supports two nested interrupts (or threads) it is still possible to use the higher interrupt level for background tasks such as timing. An additional drawback of polling the OCI is that keeping the co-processor continuously active unnecessarily increases power consumption.

The precise increase in power consumption is dependent upon the operational state of the application processor and SoC peripherals [12]; however, Figure 41 illustrates a typical scenario where the application processor (CPU12X) is continuously active whereas the XGATE is only intermittently activated to service interrupt handlers, or execute instrumentation code. As can be seen, the increase in current consumption while the XGATE is active, due to instrumentation code, is proportional to the execution time of the instrumentation code; continuous activation of the XGATE, as required by the polling technique, therefore increases the SoC power consumption by 50%.

![Figure 41: MC9S12XE100 core current consumption, when operating at 50 MHz](image-url)
This dramatic increased power consumption is entirely due to using the polling technique and not related to the instrumentation code size; the instrumentation code only requires 608 bytes of code space and 485 bytes of data space on the XGATE. This power consumption could be significantly reduced by using the previous approach where the OCI generates an interrupt to trigger the instrumentation. The maximum execution time for instrumentation within the software interrupt handler is approximately 9.4 μs; if triggered at each occurrence of the application 10 ms timer tick event (which is the fastest regular application event) this would utilize only 0.094% of the XGATE processing capabilities. Plus, in this case the PIT interrupt handler still requires 200 ns to execute but is triggered at a much lower 1 ms intervals, which represents only 0.02% utilization of the XGATE. The total increased power consumption would then be in the order of 0.05%.

Configuring the S12XDBG for use in this manner did however expose what appears to be a flaw in the state sequencer and comparator match logic. The state sequencer can normally be configured to generate a trigger signal after a user defined sequence of comparator match signals has occurred; this trigger is then used to synchronise the breakpoint logic and trace buffer. Unfortunately, when both breakpoint generation and trace buffer capture are disabled, as was the case in these experiments, the state-sequencer did not function as intended. In this case, this limitation did not pose a significant problem as polling a single comparator match-bit was sufficient, but it does limit the usefulness of this particular architecture, when polling the OCI, if more complex trigger conditions were required.

8.8 Summary

The objective of these experiments was to examine the feasibility of using OCI and a co-processor to perform runtime verification of software requirements. A case study of a medical device and a sample of three user-interface functional requirements were presented. These requirements are derived from essential safety-related features of the device; and since they form part of the risk mitigation strategy for the device in operation, these must be verified on the target platform at runtime. The results show that the proposed approach can facilitate the runtime verification of these requirements without impacting upon the application.
CHAPTER 9. CONCLUSIONS AND FUTURE WORK

9.1 Introduction

This chapter aims to demonstrate how the main objectives set out for this research have been achieved. Those objectives were to: examine the current state-of-the-art with regard to embedded software verification techniques and emerging verification challenges; examine the capability of current embedded software design and development methodologies to produce software which is correct by construction; research the capabilities of existing silicon test and OCI solutions; propose an alternative technique using OCI and a co-processor which can assist in addressing target-level software verification challenges; examine the feasibility of using this alternative technique in a number of case studies; arrive at conclusions on the capabilities of this approach.

The conclusions which the author arrived at following the examination of the background material are provided in the next section; resulting in the author’s proposed alternative approach. Section 9.3 discusses the result from the experiments conducted to examine the feasibility of this approach. Section 9.4 provides the main conclusions which the author suggests can be drawn from the work; and section 9.5 provides suggestions for future work.

9.2 Review of background material and resulting alternative approach

Whether one considers verification and validation to be distinct or inseparable activities, their common objective is to provide confidence that the end product meets its intended requirements. The standards associated with the development of embedded systems, with functional safety requirements, enable the designer to classify a product according to the risk posed to the end user; and provide guidance on the expected and recommended best practice design, verification, and validation techniques, to apply. Standards and guidelines generally encompass the entire product development life-cycle, and cover a wide variety of products falling under their scope, with varying SILs. Accordingly, these techniques are often described at a high-level in terms of general rather than specific activities.

It may seem desirable to verify all embedded systems using the most rigorous techniques. However, a balance must be struck between the resources required to perform verification and the risk posed, or the benefits to be gained. Consequently, verification techniques which require considerable design and development resources, such as semi-formal and formal methods, are only recommended for those systems where there is a higher probability of severe, life-threatening, or fatal injury. In practice, many systems are designed to minimise the probability of occurrence of errors which could result in serious injury, or to ensure that the system remains controllable if such an error occurs. Not only does this reduce the risk to the user, but it also enables the system designer to apply less arduous verification techniques.

There is also a significant motivation to minimise development costs through the reuse of
existing components, and in particular software. Reuse not only offers saving in development time and effort, it can also considerably reduce the verification effort if a justifiable ‘proven in use argument’ can be made. Therefore, in sectors where functional safety is a concern, there is a reluctance to make any alterations to existing ‘proven’ software; not only because an alteration could introduce errors, but also because it can erode the confidence which had been gained in the existing software implementation.

Whether designers seek to minimise or maximise verification activity, robust design still necessitates methodical testing of the product throughout the development process, including verification of the intended software behaviour on the target platform. However, it is clear from the background material that the ability to perform this target-level verification is hampered by the growing complexity of modern SoC devices and the verification challenges they pose, as summarised in Table 8. Nonetheless, the consensus from the background material suggests that exploiting highly-integrated complex multicore SoC architectures will continue to provide distinct benefits in terms of cost, power consumption, and performance.

<table>
<thead>
<tr>
<th>Lack of visibility into on-chip activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Difficulty in ensuring deterministic behaviour and determining the WCET</td>
</tr>
<tr>
<td>New programming challenges in managing inter-processor communication and data sharing</td>
</tr>
<tr>
<td>Greater susceptibility to SEU and electromigration</td>
</tr>
<tr>
<td>Increased volume of on-chip data to be captured and analysed</td>
</tr>
<tr>
<td>Inability to halt many real-time embedded systems for trace extraction</td>
</tr>
</tbody>
</table>

Table 8: Key verification challenges posed by highly-integrated SoCs

Simulation and emulation platforms aim to replicate the target-level behaviour and to provide enhanced visibility, but as summarized in Table 9 these solutions have advantages and disadvantages.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emulation (bond-out)</td>
<td>Greater visibility</td>
<td>Prohibitively expensive to produce</td>
</tr>
<tr>
<td></td>
<td>Accurately reflects target</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Operates at target-level speed</td>
<td></td>
</tr>
<tr>
<td>Emulation (FPGA)</td>
<td>Greater visibility</td>
<td>May not replicate exact behaviour</td>
</tr>
<tr>
<td></td>
<td>At, or close to, target-level speed</td>
<td></td>
</tr>
<tr>
<td>Bonded OCI</td>
<td>Greater visibility</td>
<td>Expensive mask-set cost</td>
</tr>
<tr>
<td></td>
<td>Can be removed to reduce cost</td>
<td></td>
</tr>
<tr>
<td>Simulation</td>
<td>Greater visibility</td>
<td>Slow execution speed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not always available</td>
</tr>
</tbody>
</table>

Table 9: Advantages and disadvantages of emulation and simulation solutions

Faced with the challenges posed by target-level verification the concept of creating software which is correct by construction becomes increasingly appealing. The advantages
and disadvantages of the several design methodologies, is summarised in Table 10. From the background material it is clear that the current tools and techniques which support the vision of creating software which is correct by construction, have not yet satisfied that ultimate goal. Therefore, runtime verification of the software at the target-level is still a necessary step. Runtime monitoring and verification approaches typically require the addition of software instrumentation, which can alter the runtime behaviour, or custom hardware monitors, which must be included when design is synthesised and are not easily changed. To minimise the impact upon the target, the runtime data is often captured on-chip and extracted for off-line analysis on a desktop system or used to synchronise the execution of a more easily instrumented emulation platform.

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test driven development</td>
<td>Test pass/fail criteria steer the software development process</td>
<td>Incompatibilities between desktop and target-level platforms</td>
</tr>
<tr>
<td>Component based development</td>
<td>Eases creation of complex software systems</td>
<td>Requires trusted/verified components</td>
</tr>
<tr>
<td></td>
<td>Suitable where components are dynamically linked</td>
<td>Assertions in code may not be suitable in embedded systems</td>
</tr>
<tr>
<td>Design by contract</td>
<td>Most suited to object-orientated software designs</td>
<td>Relies on runtime checking, often using assertions</td>
</tr>
<tr>
<td>Model based development</td>
<td>Enables design and simulation of systems at a high level of abstraction</td>
<td>Abstract models may not reflect final target-level behavior</td>
</tr>
<tr>
<td>Formal methods</td>
<td>Offers robust verification of design model</td>
<td>Models are difficult to create State space explosion</td>
</tr>
<tr>
<td></td>
<td>Even bounded model checking may help identify design flaws</td>
<td>Abstract models may not reflect precise target-level behavior</td>
</tr>
<tr>
<td>Runtime verification</td>
<td>Utilizes target-level platform</td>
<td>Instrumentation can be intrusive May require custom on-chip circuitry, or extraction of large amounts of trace data</td>
</tr>
</tbody>
</table>

Table 10: Correct by construction software development methodologies

As distinct from custom hardware monitors, SoC platforms typically include some level of on-chip test hardware or on-chip instrumentation. The key benefits and disadvantages of the various standard and proprietary on-chip solutions examined are summaries in Table 11. It is evident that standard-based silicon test access at the packaged device level has been accepted. Unfortunately, the principal limitation of these test interfaces is their low-bandwidth; which, in the main, is due to a strong motivation to minimise their cost. Nonetheless, the widespread availability of JTAG interfaces on the final packaged device has resulted in its common reuse to provide limited support for breakpoint and execution run-stop control. In some cases, captured execution trace data can also be extracted using these interfaces while the system is halted, but real-time trace or execution monitoring is not generally possible.
<table>
<thead>
<tr>
<th>Solution</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE 1149.1 (JTAG)</td>
<td>Standard based test interface</td>
<td>Limited I/O bandwidth</td>
</tr>
<tr>
<td></td>
<td>Widely used</td>
<td>Intended for board-level test</td>
</tr>
<tr>
<td></td>
<td>Low pin count</td>
<td>Standard does not support multicore architectures</td>
</tr>
<tr>
<td></td>
<td>Small state based on-chip TAP</td>
<td></td>
</tr>
<tr>
<td>IEEE 1149.7</td>
<td>Standard based interface</td>
<td>Limited I/O bandwidth</td>
</tr>
<tr>
<td></td>
<td>Reduced pin count</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Supports multicore architectures</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Some support for debug</td>
<td></td>
</tr>
<tr>
<td>IEEE-ISTO 5001 (Nexus)</td>
<td>Standard based debug solution</td>
<td>High-bandwidth features require more I/O pins</td>
</tr>
<tr>
<td></td>
<td>Scalable to meet application requirements</td>
<td>Advanced features are expensive</td>
</tr>
<tr>
<td></td>
<td>Designed for target-level verification</td>
<td>Not widely adopted</td>
</tr>
<tr>
<td></td>
<td>Support for multicore architectures</td>
<td></td>
</tr>
<tr>
<td>IEEE P1687</td>
<td>Hierarchical access for complex multicore core architectures</td>
<td>Limited I/O bandwidth</td>
</tr>
<tr>
<td>IEEE 1500</td>
<td>Standard based access mechanism</td>
<td>Limited I/O bandwidth</td>
</tr>
<tr>
<td></td>
<td>Provides core level access</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Can support on-chip test access</td>
<td></td>
</tr>
<tr>
<td>OCP-IP</td>
<td>Provides a framework for connection of SoC cores from multiple vendors</td>
<td>No specific OCI capabilities</td>
</tr>
<tr>
<td>FS2</td>
<td>Support for multicore architectures</td>
<td>Proprietary solution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HyperJTAG requires multiplexing of JTAG interface</td>
</tr>
<tr>
<td>MIPS</td>
<td>Support for multicore architectures</td>
<td>Proprietary solution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Probe IF requires additional I/O</td>
</tr>
<tr>
<td>ARM</td>
<td>Range of OCI modules which can be tailored to application needs</td>
<td>Proprietary solution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TPIU requires additional I/O for large volumes of data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Large OCI is expensive</td>
</tr>
<tr>
<td>Infineon MCDS</td>
<td>OCI located outside SoC core area</td>
<td>Proprietary solution</td>
</tr>
<tr>
<td></td>
<td>Can be removed to reduce cost</td>
<td>Large OCI is expensive</td>
</tr>
<tr>
<td></td>
<td>Supports filtering and complex triggers</td>
<td></td>
</tr>
<tr>
<td>UltraSoC</td>
<td>Aims to provide high speed optical I/O interface</td>
<td>Proprietary solution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Requires SiP construction</td>
</tr>
</tbody>
</table>

Table 11: Summary of on-chip interface and instrumentation solutions

Standard based solutions aimed at software verification and debug needs have not been as successful; despite being an industry standard, the Nexus interface has not been widely used. Although Nexus and proprietary OCI technologies targeted at software verification provide superior on-chip capabilities including execution trace capture, there is a desire to minimise the cost of the circuitry and in particular I/O resources needed by this OCI. Using faster I/O technologies necessitates larger I/O driver cells which can also be expensive and may create signal integrity challenges. Using optical interfaces may enable higher data rates and provide better electromagnetic immunity and reduced need for ESD protection [111], [181], but these
require specific process technologies and therefore usually necessitate SiP construction. The option of using a SiP solution may be attractive if one presumes that these OCI features will only be required on a relatively small number of devices; however, in this case, the designer must demonstrate equivalence between the verification and deployment targets.

Techniques to reduce the amount of trace data which must be sent off-chip for analysis include hardware funnels which enable the selective filtering of the trace data from multiple sources, and using complex trigger generation logic to capture only those events which are of interest. Compression techniques such as branch only trace and differential encoding are seen to provide good results for program execution trace but not for data memory; using standard data compression algorithms such as LZW have also been advocated. If the volume of data to be extracted can be reduced then this is obviously of benefit, but a fundamental question is whether this data needs to be extracted off-chip.

For silicon test, the use of BIST structures has long since been accepted as a means of addressing the challenge of performing tests which require vast amounts of data on devices with limited test I/O bandwidth. The disadvantage of BIST is that the test must be predefined and is generally implemented entirely in hardware. SBST provides a similar benefit of performing the test on-chip, but utilizes an on-chip processor to execute test routines. However, BIST and SBST generally relate to performing tests on hardware.

For software verification the addition of instrumentation code offers a low-cost solution, without the need for additional hardware; and the instrumentation code can be added to the target at any stage in the design process, even in-the-field if required. Plus target equivalence is inherent, because the same physical target can be used for verification and deployment. However, as explained in Section 4.7, the process of adding software instrumentation is not without problems. One difficulty is the overhead it can impose upon the application; for resources constrained safety-critical embedded control applications this additional overhead may not be acceptable. A further difficulty is that all inline instrumentation code must be debugged and verified to the same extent as the application code, adding greatly to the development effort. Plus, making any modification to software has the potential to exhibit the ‘Heisenbug’ phenomenon [44], whereby the instrumented code may mask the bug being observed, or may itself introduce a new bug.

Having unnecessary or unverified code included in an embedded system is clearly undesirable from a quality perspective; but it may also pose a serious threat to the reliability, safety, or security, of the system. For verification or test purposes, instrumentation code often intentionally includes mechanisms to alter the runtime behaviour of the system. If these mechanisms are not entirely removed or disabled, then the possibility exists that the system behaviour may be accidently or maliciously altered; at best, the consequences may be benign;
at worst, they could result in serious injury or loss of life. Like all other source code, software instrumentation requires careful management. At a minimum, thorough code reviews, reliable compilation tools, and robust software configuration management tools are required. Under ideal circumstances, instrumentation of software would not impact upon system resources nor pose any risks; using existing approaches this is clearly unrealistic. What is required then, is a solution leveraging the advantages of instrumentation whilst minimising the disadvantages.

The alternative approach proposed by the author, as described in Chapter 5, aims to provide this by utilising the increasing availability of multicore devices and existing OCI. This retains the benefits of software based instrumentation, which is more flexible than hardware instrumentation and is more easily altered to address various verification tasks, and can be written using the tools and language which the developer is familiar with. If the verification can be performed on-chip, then the data required to analyse or monitor the application behaviour need not be exported. By isolating the instrumentation code it can be more easily shown that the original application code has not been altered and that the instrumentation code can be completely removed if necessary; and executing the instrumentation code on a separate processor does not alter the original application code behaviour or performance.

9.3 Results from experimental work

Using a commercially available SoC device, the author conducted a number of experiments which showed that the alternative approach proposed is feasible. Although the OCI features of the experimental platform were limited, it was possible to use these to successfully activate the instrumentation code on a co-processor as intended; and none of the experiments required alteration of the application code. However, two different methods were used to trigger the co-processor at the appropriate application execution instant, each of which had benefits and drawbacks, and one method was slightly intrusive to execution on the application processor.

The first method used the OCI to generate an interrupt signal, which triggered an interrupt handler on the application processor, this handler then activated the instrumentation on the co-processor. The benefit of this method is that the co-processor instrumentation remained idle until needed, and the co-processor could execute other interrupt handler routines during this time. The drawback is that processing the required interrupt handler imposes a slight execution time penalty on the application. Although this method did introduce a small overhead on the application execution it should be noted that this was entirely due to a limitation of the experimental platform, and would not necessarily be present on other SoC architectures. Even using the same architecture, it should be obvious that if the XGATE were selected as the application processor and the CPU12X as the co-processor, then the OCI
interrupt would directly trigger the co-processor without impacting the application.

The second method used the co-processor to poll the OCI comparator status flags. This removes the need to trigger an interrupt handler on the application processor and eliminates the associated overhead. Although this method provides a non-intrusive solution, it has the disadvantage of requiring the co-processor to continuously execute in a loop, which wastefully increases power consumption. However, while polling the co-processor could still process interrupts at a higher interrupt priority level.

The first experiment was arranged to show that the co-processor could configure an OCI comparator at runtime and that when the trigger signal was generated it could activate the instrumentation code; this would then reconfigure the OCI for the next event, and so on. This arrangement is not dissimilar to the manner in which OCI comparators are typically used; except, rather than generating a breakpoint, which would halt execution, in this instance control is instead passed to the instrumentation code. The number of trigger events was also chosen to exceed the number of available comparators; this demonstrates that using a co-processor provides more flexibility than simply using the limited number of hardware comparators supported by the OCI. A second limitation of using the OCI alone is that there is no mechanism to detect the failure of an event to occur; by adding a timer interrupt routine to the co-processor it was possible to measure the time between events and provide an error indication when a timeout was detected.

The second experiment demonstrates that the same approach can be used to verify timing constraints of a hypothetical real-time embedded system. The results show that the instrumentation code residing on the co-processor could both measure the execution timing of the application functions and provide an indication when a timing deadline was violated. The timing results obtained were verified using a calibrated oscilloscope which showed that the results were valid. In this case the instrumentation code also computed the minimum and maximum execution timing and reported this status data at regular 5 second intervals; the rate at which this status data was provided could of course be altered to suit the situational needs. However, the purpose was to demonstrate that a co-processor can also perform on-chip analysis of the data rather than simply streaming vast amounts of data off-line for analysis.

The third experiment showed that the instrumentation code could then be changed, without altering the application code, to instead monitor the timing between updates to the output value, which from the previous experiment appeared to exhibit some jitter. If some form of compensation for jitter is not employed in control algorithms then it is important to verify that it is not required because jitter is not present, or it is sufficiently small. The approach proposed provides a means by which this information can be obtained without the need to instrument the application code and thus alter the algorithm. The result showed that
the co-processor could store the timing data and the captured output data value and output this buffered data as required. In this case the instrumentation was written such that a new minimum or maximum value would generate a trigger signal which was synchronised with the mid-point of the buffer. Many real-time kernels provide similar capabilities to monitor data variables, but these rely upon sampling variables at regular intervals or recording the variable each time the application code updates it. Both these approaches lack the ability to reliably detect spurious updates to the variable; unless the sampling rate is sufficiently high, which is generally impractical.

This experiment also illustrates the benefit of using the OCI to monitor events rather than simply relying upon in-line instrumentation of application code. The OCI is capable of monitoring all read and write events to a particular address location and triggering the instrumentation code on the co-processor at each occurrence. In-line instrumentation of application code only detects a change when instrumentation is executed; corruption of data at another time, for example by errant code or SEU, is not detected. This is of particular concern in small embedded systems if stack overflows occur, or when using the C language data or function pointers which can easily be corrupted. Plus, if a variable or peripheral address location is updated from several different parts of the application code, then in-line instrumentation must be added to each one, which greatly increases its impact.

The author then demonstrated how runtime verification of requirements could be performed using this approach; in this case a medical-device example was used and three safety-related software requirements were checked. The first requirement involved the timing of a state transition when a timeout was expected to occur. This required configuring the OCI and co-processor to unobtrusively monitor the application event signals and state transitions and to measure the appropriate transition timing. In this case the application was found to meet its requirement, but its precise timing behaviour was seen to be dependent upon its timer-tick resolution. Verification of the other requirements required the injection of an erroneous checksum and setting values while the application was executing and monitoring the application to ensure it correctly transitioned to an error handling state. The results showed that it was possible to selectively inject a fault at the appropriate time (on entering a new state) and that the application did detect the fault and transitioned as expected, thereby meeting the requirement. However, injecting the fault at another time showed that the application did not transition immediately, which highlighted the fact that the requirements were weak and could be improved upon.
9.3.1 Limitations of this alternative approach

The experiments conducted illustrate the concept of using a co-processor to execute instrumentation code and extract information relating to application code residing on another CPU; of course the MC9S12XE100 SoC was not designed for this role, and so this architecture does have limitations when used in this way.

Specific limitations of this architecture include the lack of a direct hardware interrupt from the debug module to the XGATE. However, since the XGATE is intended to function as a peripheral interrupt handler to which 108 other interrupt sources can already be routed, addition of the necessary hardware to enable direct triggering of the XGATE from the debug module should not be impractical. Using interrupts to trigger the co-processor necessitated writing a simple interrupt handler routine which resides on the application processor, the handling of the interrupt may interfere with the execution timing of the application; but this additional code on the application CPU is minimal and isolated to a single location. Polling the debug module was shown to be a feasible alternative, although it required the co-processor to remain active at all times. The other limitations of the SoC include: the windowing of registers into the memory map, which imposes considerable latency when accessing the debug module registers; the inability to access the trace buffer contents while the debug module is armed; and the apparent flaw in the state-sequencer which meant it could not be used when interrupts were disabled.

The more fundamental limitation of this approach is the time required to execute the instrumentation code on the co-processor. The time spent processing the instrumentation code following activation, dictates the minimum response time before the next activation. This required careful construction of the instrumentation code to ensure that trigger events were not missed. Where instrumentation code is added directly into the application this issue does not arise since the instrumentation code will execute in an inherently sequential manner, but inline instrumentation code directly impacts upon the execution timing of the application. However, given the impact instrumentation can have on the application behaviour, it is common practice to optimise this code or to only enable a small portion of the instrumentation code at any given time. When a particular property is being verified, or a specific bug is being isolated, it is preferable to only extract data related to that specific item. Therefore, in many cases the ability to only process one instrumentation trigger at a time may not be a significant limitation. If multiple or adjacent instrumentation triggers are required, the XGATE architecture can support a high priority ‘thread’ interrupting a lower priority one. Therefore, as illustrated in Figure 42 with suitable interrupt generation hardware, the XGATE or other multicore SoCs could potentially support more than one instrumentation trigger in the same manner as multiple interrupt sources are typically handled.
One aspect of the software instrumentation code which greatly impacted upon its execution timing was the limited baud rate at which the status output could be sent to the monitoring terminal. Although a buffer was used to allow the instrumentation code to operate more efficiently, this buffer had to be made sufficiently large to accommodate maximum amount of data which could be queued at any time. Using other architectures with higher speed interfaces, or modules specifically designed to provide high-speed output from software instrumentation would alleviate this difficulty.

Of course the output from the experimental examples is quite verbose and in more practical cases this could be significantly reduced, or sent in a more efficient format. However, outputting status information in a human readable format not only simplifies the process of checking the output data, but it is particularly important if a transcript of the output must be retained for quality systems audit purposes. A thorough external auditor will follow the entire trail from quality systems procedures through to, and including, output data records demonstrating compliance with these procedures. If an additional tool is required to interpret the output data, then the verification and validation of this tool will also be subject to rigorous audit; consequently minimising the number of tools required, reduces potential sources of error and also streamlines the audit process.

Despite the limitations encountered, the experimental platform served as a useful example of how the arrangement of: application processor, debug module, and co-processor, can be used to monitor and analyse target-level execution and greatly reduce the volume of data which must be extracted off-chip.

9.4 Conclusions

Despite advances in software design and development methodologies, target-level verification of embedded software remains an essential step in achieving the high level of confidence required for embedded systems used in safety-critical or high-reliability applications. Using SoC architectures with multiple processors has many potential benefits
for real-time embedded systems, but this potential is difficult to exploit if techniques to verify and debug these highly integrated devices are not readily available to software engineers. The author has described some of the difficulties faced in performing verification on these highly-integrated SoCs and the limitations of current solutions; and has proposed and demonstrated the alternative approach of using existing OCI and co-processor to alleviate these difficulties.

The experiments conducted on the demonstration platform show the feasibility of the approach proposed, but they also highlight some limitations. In some cases these are due to constraints imposed by the experimental platform used; for example, the number of events which can be monitored simultaneously is dictated by the on-chip instrumentation hardware, and the rate at which status or captured data can be exported off-chip is dictated by the I/O interface used. However, these limitations may not be present on other architectures, or indeed other architectural limitations may arise. The more fundamental limitation is that the time required to execute the instrumentation code on the co-processor dictates the minimum response time between successive instrumentation activations. However, executing software instrumentation will always require some processing resources, and if this burden is borne by the application processor then it impacts directly on the application; using a co-processor eliminates this issue at the expense of having to ensure that relevant event triggers are not missed.

Because the experiments conducted were of limited size and complexity it was somewhat easier to retain a global view on the execution behaviour and ensure that this limitation did not pose an issue. However, despite using a hypothetical scenario for the first three experiments, the need to measure timing to a resolution of 1 µs imposed constraints on the instrumentation which would not be untypical of a real-world scenario; and while the timing constraints on the subsequent experiments, using the case-study of a CPAP device, were considerably less stringent, these reflected the actual real-world device functional requirements. Nonetheless, the experiments do serve to demonstrate that the concept of using OCI and a co-processor in the manner proposed is viable.

In addition to being a viable alternative, this approach offers many benefits. In terms of cost, utilising a co-processor and the existing OCI does not require the addition of any circuitry. Using a co-processor for instrumentation purposes alone may appear to be prohibitive, but many multicore devices include at least one CPU which is underutilized, or used for non-critical functions, which could be assigned to the instrumentation task when needed. In terms of intrusiveness, the approach proposed is minimally invasive or non-invasive as it requires little or no modifications to the application software. Plus, the ability to isolate or eliminate the instrumentation from the application code is inherent within the approach proposed. In terms of performance, the impact upon the execution of the application
on the primary CPU is negligible. In terms of ease-of-use for developers, no new tools, languages, or methodologies are required; adding instrumentation software is no more difficult than writing application software.

In the author's opinion, the approach proposed represents an effective way of addressing many verification challenges on SoC platforms and could enable new target-level verification and debugging tools. The ability to process data in real-time while the application is executing, serves software verification and debug needs better than existing solutions which are targeted at off-line verification. However, the demonstration platform and the experiments conducted were intended to act merely as a proof-of-concept rather than as a final solution, and much work remains to be done.

9.5 Future work

In this section the author outlines some ways in which this work could be progressed in the future. The most obvious extension to this work would be to examine the feasibility of the proposed approach on alternative architectures. The experimental platform served as a useful archetype for a resource constrained embedded system since the OCI features available, and used, were quite limited. Still, it would be useful to examine this approach whilst exploiting the benefits of a more complex OCI solution. Using OCI which can provide more refined co-processor triggers (such as the Infineon MCDS solution), or a faster mechanism for exporting status data, would be particularly interesting.

The availability of SoC architectures with more than two cores could also be explored. For example, if a quad-core architecture were used as illustrated in Figure 43, the instrumentation code required to monitor the application code execution on all CPUs could be located on just one CPU and triggered by OCI which is capable of monitoring all on-chip activity. This retains the benefit if isolating the entire instrumentation code on a single processor and in a quad-core scenario utilizes only 25% of the processing resources; and this utilization decreases as the number of CPUs increase.

![Diagram of proposed approach applied to a quad-core SoC architecture](image-url)
Alternatively, if sufficient processing time were available the instrumentation code could be distributed across the four CPUs and executed as a background or idle time thread. This retains the benefit of performing the monitoring and analysis on-chip, but obviously the negates the ability to isolate the instrumentation code from the application.

A key benefit of the approach proposed is the ability to perform real-time on-chip monitoring and analysis of the execution behaviour, but in some cases it is also necessary to provide an off-line record of the execution trace data. For example, when performing verification, code coverage is widely used as a means of ensuring that each path in the code being tested has been fully exercised. Unfortunately, the limited trace buffer provided in many OCI solutions can quickly become full, often with repetitive data as illustrated by the trace buffer data shown in Appendix C-3, and exporting this data in real-time is generally impractical; existing solutions to reduce this data and their limitations have previously been discussed.

However, one technique to reduce captured data which is widely used for silicon-test methods, such as BIST, is to only provide a signature from test data. This signature is then compared against the expected value which indicates if the test was successful or not. In a similar way the author proposes that a signature, hash-value, or CRC, be computed for the execution trace data and only this value need be exported for off-line analysis. Where multiple execution paths exist, the off-line tools could compute the expected CRC for each execution path and determine which was followed. This could be achieved using a co-processor to read the trace-buffer memory, perform the computation on-chip, and then export this value. Unfortunately, it was not possible to examine the feasibility of this proposal due to the architectural limitations of the SoC used in the experimental work; the two key limitations were the inability to read the trace buffer while debug hardware was armed and the imposition of strict FIFO read access to the trace data. However, using a suitable alternative architecture this proposal could be investigated.

The approach proposed is not intended to be considered in isolation or to the exclusion of other verification techniques. It should instead be seen as providing a means of unobtrusively monitoring execution which could be complementary to other solutions. Considering the Monitoring and Checking (MaC) framework architecture [280] and the associated JavaMaC [281] which is based on formal specifications and provides a scheme to automatically link low-level observations of program execution, which can be emitted as events from a suitably instrumented target application program, to the relevant monitored properties. These properties can then be formally verified at runtime.

A simplified diagram of the overall MaC framework architecture is shown in Figure 44. The architectures include a static phase (before program execution), and a runtime phase
(during the execution). In the static phase, a mapping is established, from a formal requirements specification, between the following two entities: 1) high-level events, from the high-level requirements specification; and 2) low-level state information, to be extracted from the instrumented target program during execution. During the runtime phase, the instrumented program is monitored and checked with respect to its requirements specification.

**Figure 44: The Monitoring and Checking (MaC) architectural framework**

MaC automatically generates the runtime components: including a filter and event recogniser, generated from the low-level specification; and a runtime checker, generated from the high-level specification. The filter sends relevant state information to the event recogniser. The event recogniser detects an event from the state information received from the filter, according to a low-level specification. The recognised events are sent to the runtime checker. The runtime checker verifies, during execution, that the current execution history satisfies a high-level requirement specification.

The runtime architecture for the Monitoring and Checking (MaC) framework is shown in Figure 45 (a). The filter is implemented as a thread within the target application program, and communicates with a separate host computer using a dedicated communications link. The host computer executes the event recogniser and the runtime checker. Prior work by Watterson and Heffernan [282], [369], summarises the main disadvantages of the classical implementation architectures as follows: adding a thread to the target program is invasive and can make the program less deterministic; the communication link needs to strictly guarantee the timeliness and ordering of events; time synchronisation across host and target processors needs to be carefully managed; and use of a separate host computer is cumbersome and expensive.

The alternative approach proposed by this author offers an architectural solution which overcomes these drawbacks. Figure 45 (b) illustrates such a solution with the following key features: the filter can be implemented in the coprocessor and thus does not burden the target application; the communications link is implicit in the architecture; synchronised timing is
easily realised in the architecture; the single chip solution is streamlined and cost effective; and a single toolset could be devised for end-to-end automated development.

Figure 45: Runtime architecture for the MaC framework

As can be seen, this architectural solution is quite similar to the arrangement which the author used for the previous experimental work. Therefore, the author considers that with future work the proposed alternative could be applied to complement this and many other runtime software verification techniques.
BIBLIOGRAPHY


Page 131


Appendix A

Figure 46 below shows the development board which was used as the target for all experimental work carried out; with the key components and interfaces labelled.

![Development board for experimental work](image)

*Figure 46: Development board for experimental work*
Appendix B

Figure 47 below shows how the global memory map for the MC9S12XE100 device maps to the local memory address range of the XGATE and CPU12X cores.

Figure 47: MC9S12XE100 memory map
Appendix C

Monitoring execution sequences

The data below shows the serial output data sent from the experimental platform while monitoring the execution sequence of an example program which consisted of a cyclical loop of ten functions.

When the XGATE timer interrupt routine detected that a timeout had occurred it sent a string indicating the timeout error and the current function/task number to the serial port:

-STARTUP-
*TIMEOUT ERROR* ID:5
*TIMEOUT ERROR* ID:5
*TIMEOUT ERROR* ID:8
*TIMEOUT ERROR* ID:8
*TIMEOUT ERROR* ID:5
*TIMEOUT ERROR* ID:5
*TIMEOUT ERROR* ID:8
*TIMEOUT ERROR* ID:8
*TIMEOUT ERROR* ID:8
*TIMEOUT ERROR* ID:5
*TIMEOUT ERROR* ID:5
*TIMEOUT ERROR* ID:5
Measuring execution timing

The data below shows the serial output data sent from the experimental platform to the host capturing status data, while monitoring the execution timing of the real-time example program.

For ease of interpretation and verification the outputted data was sent as ASCII readable strings which are formatted as follow:

*C function name: minimum time measured, maximum time measured, deadline, error flag*

-STARTUP-

=======
ReadKeys: 7,13,20,N
ComputeOutput: 10,13,15,N
OutputPWM: 9,12,15,N
=======
ReadKeys: 7,13,20,N
ComputeOutput: 10,13,15,N
OutputPWM: 9,12,15,N
=======
ReadKeys: 7,13,20,N
ComputeOutput: 10,13,15,N
OutputPWM: 9,12,15,N
=======
ReadKeys: 7,19,20,N
ComputeOutput: 10,13,15,N
OutputPWM: 9,12,15,N
=======
ReadKeys: 7,19,20,N
ComputeOutput: 10,13,15,N
OutputPWM: 9,12,15,N
=======
ReadKeys: 7,19,20,N
ComputeOutput: 10,13,15,N
OutputPWM: 9,12,15,N
=======
ReadKeys: 7,29,20,Y
ComputeOutput: 10,13,15,N
OutputPWM: 9,12,15,N

Status outputs when neither key was pressed

Status outputs after each key was pressed individually

Status output when both keys were pressed simultaneously. Showing the measured time and error flag
Execution trace output data

The data below shows the execution trace data outputted from the experimental platform to the host capturing status data.

```
==TRACE== 78
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
0x00 7FC0C9 00 7FC0C9
```

0x80 7F82B5 80 7F80E3
Jitter in OutputPWM start time

Figure 48 show a close-up of the oscilloscope captured timing waveform for the OutputPWM function. The jitter on the start-time can be seen to be approximately half of the 2 µs oscilloscope time graduation, representing jitter of 1 µs.

![Figure 48: Close-up of waveform showing timing of OutputPWM function.](image-url)
Appendix D

Monitoring event signals and state changes

The data below shows the serial output data sent from the experimental CPAP application platform. In this instance the co-processors was simply configured to monitor event signals and state changes; for brevity, timer tick events have been excluded from the output. The data shows that the co-processor can correctly monitor application events and state changes, whether these are user initiated or generated by the program.

**State:** DISPLAY_SETTINGS
Event: QENTRY_SIG
Event: INCREMENT_SIG
Event: INCREMENT_SIG
Event: DECREMENT_SIG
Event: DECREMENT_SIG
Event: ACCEPT_SIG

**State:** CHANGE_SETTINGS
Event: QENTRY_SIG
Event: INCREMENT_SIG
Event: INCREMENT_SIG
Event: INCREMENT_SIG
Event: INCREMENT_SIG
Event: DECREMENT_SIG
Event: DECREMENT_SIG
Event: ACCEPT_SIG

**State:** STORE_NEW_SETTINGS
Event: QENTRY_SIG
Event: SETTINGS_CHANGED_SIG

**State:** UPDATE_GAS
Event: QENTRY_SIG
Event: UPDATE_COMPLETE_SIG

**State:** DISPLAY_SETTINGS
Event: QENTRY_SIG
Event: INCREMENT_SIG
Event: DECREMENT_SIG
Event: ACCEPT_SIG

**State:** CHANGE_SETTINGS
Event: QENTRY_SIG
Event: CANCEL_SIG

**State:** DISPLAY_SETTINGS
Event: QENTRY_SIG
Measuring setting change timeouts

The data below shows the serial output data sent from the experimental platform to the status capturing host, while monitoring the setting change timeouts on the CPAP application.

Transition to state: DISPLAY_SETTINGS
After time (ms): 9992
Transition to state: DISPLAY_SETTINGS
After time (ms): 9993
Transition to state: DISPLAY_SETTINGS
After time (ms): 9994
Transition to state: DISPLAY_SETTINGS
After time (ms): 9995
Transition to state: DISPLAY_SETTINGS
After time (ms): 9993
Transition to state: DISPLAY_SETTINGS
After time (ms): 9995
Transition to state: DISPLAY_SETTINGS
After time (ms): 9992
Transition to state: DISPLAY_SETTINGS
After time (ms): 9998
Transition to state: DISPLAY_SETTINGS
After time (ms): 9997
Transition to state: DISPLAY_SETTINGS
After time (ms): 9991
Transition to state: DISPLAY_SETTINGS
After time (ms): 10000
Transition to state: DISPLAY_SETTINGS
After time (ms): 9997
Transition to state: DISPLAY_SETTINGS
After time (ms): 9998
Transition to state: DISPLAY_SETTINGS
After time (ms): 10000
Transition to state: DISPLAY_SETTINGS
After time (ms): 9996
Transition to state: DISPLAY_SETTINGS
After time (ms): 9997
Transition to state: DISPLAY_SETTINGS
After time (ms): 9992
Transition to state: DISPLAY_SETTINGS
After time (ms): 10000
Transition to state: DISPLAY_SETTINGS
After time (ms): 9999
Transition to state: DISPLAY_SETTINGS
After time (ms): 9995
Injecting invalid checksum into data structure

The data below shows the transcript of the serial output data from the experimental platform to the status capturing host, while injecting invalid checksum values in various states of the CPAP application.

1) Invalid checksum inserted on entry to DISPLAY_SETTINGS state

-STARTUP-
** changed checksum **
*State: DISPLAY_SETTINGS*
*Event: CHECKSUM_ERROR_SIG*
*State: SETTINGS_DATA_ERROR*
** restored checksum **
** forced state change **
*State: DISPLAY_SETTINGS*

2) Invalid checksum inserted on entry to CHANGE_SETTINGS state

-STARTUP-
*State: DISPLAY_SETTINGS*
*Event: DECREMENT_SIG*
*Event: INCREMENT_SIG*
*Event: ACCEPT_SIG*
** changed checksum **
*State: CHANGE_SETTINGS*
*Event: CHECKSUM_ERROR_SIG*
*State: SETTINGS_DATA_ERROR*
** restored checksum **
** forced state change **
*State: CHANGE_SETTINGS*
*Event: ACCEPT_SIG*
*State: STORE_NEW_SETTINGS*
*Event: SETTINGS_CHANGED_SIG*
*State: UPDATE_GAS*
*Event: UPDATE_COMPLETE_SIG*
*State: DISPLAY_SETTINGS*
3) Invalid checksum inserted on entry to STORE_NEW_SETTINGS state

-STARTUP-

** State: DISPLAY_SETTINGS **
Event: INCREMENT_SIG
Event: DECREMENT_SIG
Event: ACCEPT_SIG

** State: CHANGE_SETTINGS **
Event: INCREMENT_SIG
Event: DECREMENT_SIG
Event: ACCEPT_SIG
** changed checksum **

** State: STORE_NEW_SETTINGS **
Event: CHECKSUM_ERROR_SIG

** State: SETTINGS_DATA_ERROR **
** restored checksum **
** forced state change **

** State: STORE_NEW_SETTINGS **
Event: SETTINGS_CHANGED_SIG

** State: UPDATE_GAS **
Event: UPDATE_COMPLETE_SIG

** State: DISPLAY_SETTINGS **

4) Invalid checksum inserted on entry to UPDATE_GAS state

-STARTUP-

** State: DISPLAY_SETTINGS **
Event: ACCEPT_SIG

** State: CHANGE_SETTINGS **
Event: ACCEPT_SIG

** State: STORE_NEW_SETTINGS **
Event: SETTINGS_CHANGED_SIG
** changed checksum **

** State: UPDATE_GAS **
Event: CHECKSUM_ERROR_SIG

** State: SETTINGS_DATA_ERROR **
** restored checksum **
** forced state change **

** State: UPDATE_GAS **
Event: UPDATE_COMPLETE_SIG

** State: DISPLAY_SETTINGS **
5) Invalid checksum inserted while application in DISPLAY_SETTINGS state

-STARTUP-
State: DISPLAY_SETTINGS
** changed checksum **
Event: INCREMENT_SIG
Event: DECREMENT_SIG
Event: CANCEL_SIG
Event: ACCEPT_SIG
State: CHANGE_SETTINGS
Event: CHECKSUM_ERROR_SIG
State: SETTINGS_DATA_ERROR
** restored checksum **
** forced state change **
State: DISPLAY_SETTINGS
Event: ACCEPT_SIG
State: CHANGE_SETTINGS
Event: INCREMENT_SIG
Event: DECREMENT_SIG
State: DISPLAY_SETTINGS

6) Invalid checksum inserted while application in CHANGE_SETTINGS state

-STARTUP-
State: DISPLAY_SETTINGS
Event: INCREMENT_SIG
Event: DECREMENT_SIG
Event: ACCEPT_SIG
State: CHANGE_SETTINGS
** changed checksum **
Event: DECREMENT_SIG
Event: INCREMENT_SIG
Event: ACCEPT_SIG
State: STORE_NEW_SETTINGS
Event: CHECKSUM_ERROR_SIG
State: SETTINGS_DATA_ERROR
** restored checksum **
** forced state change **
State: CHANGE_SETTINGS
Event: ACCEPT_SIG
State: STORE_NEW_SETTINGS
Event: SETTINGS_CHANGED_SIG
State: UPDATE_GAS
Event: UPDATE_COMPLETE_SIG
State: DISPLAY_SETTINGS
**Injecting invalid setting values into data structure**

The data below shows the transcript of the serial output data from the experimental platform to the status capturing host, while injecting invalid setting values in the various operational states of the CPAP application.

1) Invalid setting values inserted **on entry** to DISPLAY_SETTINGS state

-STARTUP-

!! Write out of range value

**State**: DISPLAY_SETTINGS  
**Event**: DATA_RANGE_ERROR_SIG

**State**: SETTINGS_DATA_ERROR  
** Temperature = 32  
** Restored setting  
** Forced state change  
!! Write out of range value

**State**: DISPLAY_SETTINGS  
**Event**: DATA_RANGE_ERROR_SIG

**State**: SETTINGS_DATA_ERROR  
** Temperature = 42  
** Restored setting  
** Forced state change  
!! Write out of range value

**State**: DISPLAY_SETTINGS  
**Event**: DATA_RANGE_ERROR_SIG

**State**: SETTINGS_DATA_ERROR  
** Pressure = 1  
** Restored setting  
** Forced state change  
!! Write out of range value

**State**: DISPLAY_SETTINGS  
**Event**: DATA_RANGE_ERROR_SIG

**State**: SETTINGS_DATA_ERROR  
** Pressure = 11  
** Restored setting  
** Forced state change  
!! Write out of range value

**State**: DISPLAY_SETTINGS  
**Event**: DATA_RANGE_ERROR_SIG

**State**: SETTINGS_DATA_ERROR  
** % O2 = 20  
** Restored setting  
** Forced state change  
!! Write out of range value

**State**: DISPLAY_SETTINGS
Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** % O2 = 101
** Restored setting
** Forced state change
State: DISPLAY_SETTINGS

2) Invalid setting values inserted on entry to CHANGE_SETTINGS state

-STARTUP-
State: DISPLAY_SETTINGS
Event: ACCEPT_SIG
!! Write out of range value
State: CHANGE_SETTINGS
Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** Temperature = 32
** Restored setting
** Forced state change
!! Write out of range value
State: CHANGE_SETTINGS
Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** Temperature = 42
** Restored setting
** Forced state change
!! Write out of range value
State: CHANGE_SETTINGS
Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** Pressure = 1
** Restored setting
** Forced state change
!! Write out of range value
State: CHANGE_SETTINGS
Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** Pressure = 11
** Restored setting
** Forced state change
!! Write out of range value
State: CHANGE_SETTINGS
Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** % O2 = 20
** Restored setting
** Forced state change  
!! Write out of range value  
State: CHANGE_SETTINGS  
Event: DATA_RANGE_ERROR_SIG  
State: SETTINGS_DATA_ERROR  
** % O$_2$ = 101  
** Restored setting  
** Forced state change  
State: CHANGE_SETTINGS  
State: DISPLAY_SETTINGS  

3) Invalid setting values inserted on entry to STORE_NEW_SETTINGS state  

-STARTUP-  
State: DISPLAY_SETTINGS  
Event: ACCEPT_SIG  
State: CHANGE_SETTINGS  
Event: ACCEPT_SIG  
!! Write out of range value  
State: STORE_NEW_SETTINGS  
Event: DATA_RANGE_ERROR_SIG  
State: SETTINGS_DATA_ERROR  
** Temperature = 32  
** Restored setting  
** Forced state change  
!! Write out of range value  
State: STORE_NEW_SETTINGS  
Event: DATA_RANGE_ERROR_SIG  
State: SETTINGS_DATA_ERROR  
** Temperature = 42  
** Restored setting  
** Forced state change  
!! Write out of range value  
State: STORE_NEW_SETTINGS  
Event: DATA_RANGE_ERROR_SIG  
State: SETTINGS_DATA_ERROR  
** Pressure = 1  
** Restored setting  
** Forced state change  
!! Write out of range value  
State: STORE_NEW_SETTINGS  
Event: DATA_RANGE_ERROR_SIG  
State: SETTINGS_DATA_ERROR  
** Pressure = 11  
** Restored setting  
** Forced state change
!! Write out of range value
State: STORE_NEW_SETTINGS
Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** % O2 = 20
** Restored setting
** Forced state change
!! Write out of range value
State: STORE_NEW_SETTINGS
Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** % O2 = 101
** Restored setting
** Forced state change
State: STORE_NEW_SETTINGS
Event: SETTINGS_CHANGED_SIG
State: UPDATE_GAS
Event: UPDATE_COMPLETE_SIG
State: DISPLAY_SETTINGS

4) Invalid setting values inserted on entry to UPDATE_GAS state

-STARTUP-
State: DISPLAY_SETTINGS
Event: ACCEPT_SIG
State: CHANGE_SETTINGS
Event: ACCEPT_SIG
State: STORE_NEW_SETTINGS
Event: SETTINGS_CHANGED_SIG
!! Write out of range value
State: UPDATE_GAS
Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** Temperature = 32
** Restored setting
** Forced state change
!! Write out of range value
State: UPDATE_GAS
Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** Temperature = 42
** Restored setting
** Forced state change
!! Write out of range value
State: UPDATE_GAS
Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** Pressure = 1
** Restored setting
** Forced state change
!! Write out of range value
State: UPDATE_GAS

Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** Pressure = 11
** Restored setting
** Forced state change
!! Write out of range value
State: UPDATE_GAS

Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** % O2 = 20
** Restored setting
** Forced state change
!! Write out of range value
State: UPDATE_GAS

Event: DATA_RANGE_ERROR_SIG
State: SETTINGS_DATA_ERROR
** % O2 = 101
** Restored setting
** Forced state change
State: UPDATE_GAS

Event: UPDATE_COMPLETE_SIG
State: DISPLAY_SETTINGS