Verification and Validation of complex vetronic systems with FlexRay

Daniel Summers

Submitted for the degree of Doctor of Philosophy

Department of Engineering and Design

School of Science and Technology

University of Sussex

September 2010

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification at this or any other university of other institute of learning.

The work in chapters 4, 5 and 6 is based on extending the MilCAN testbed, which was developed by P. Charchalakis and G. Valsamakis. In this thesis, MilCAN–related sections are present so that the design process leading to the development of the V&V testbed is coherent and complete. The FlexRay segment was developed by the author, and the TTP segment was developed by J.I. Melentis, another member of the research group in which this investigation was conducted. Therefore, information relevant to the TTP segment and its integration will not be shown in this thesis, unless it is deemed necessary for clarification or comparison. Additionally, the bridging and triggering systems detailed in this thesis are joint works conducted by I. Melentis, P. Charchalakis and the author.

This work was sponsored by the United Kingdom Ministry of Defence.

Abstract

Modern military vehicles have been retrofitted to contain a considerable number of embedded electronic subsystems of varying size and complexity over their operational lifespan, and many of these subsystems are interconnected by a variety of embedded networks. Upcoming next–generation vehicles will be designed with such networks and embedded subsystems built in from the day they are manufactured. When multiple subsystems and networks are present, it is often possible to benefit from integrating those networks and sharing information between systems.

Historically, however, integrating dissimilar embedded networks is a complex problem: it is not possible to certify a network composed of multiple different networking systems, and interactions between deterministic, safety–critical and non–real–time networks risk compromising the essential properties of one or more of them. Some form of integration testbed would likely simplify the process, along with an associated methodology.

This thesis details the development of such an integration testbed: its abstract design, mapping to implementing technologies and functional testing. The testbed is designed to be used in support of the V–Model (and similar testing–driven development methodologies) as a platform for development and verification testing of new and upgraded components and systems. As such, the testbed can be configured to simulate the target platform network and new components integrated and tested on it, then installed on the target platform with a minimum of additional effort.

Acknowledgements

I would like to thank my supervisor, Dr Elias Stipidis, for his guidance, patience and support during the research supporting this thesis.

I would also like to thank my second supervisor, Dr Falah Ali, and my colleagues and friends in the Vetronics Research Centre: in no particular order Dr Periklis Charchalakis, Dr Ian Colwill, Dr Obowoware Obi, Panos Oikonomidis, Dr Georgios Valsamakis, Dr Matthew Fowler, Dr Ireri Ibarra and Ioannis Melentis.

Finally, for acting as an anchor in the unpredictable tumult that has been the last five years of my life, I would like to thank my family and friends, in particular Yvonne and Christian Summers, Fouad Sethna, Rachael Acks, the Wayward Scholars and the denizens of #maelfroth and #afp.

Dedicated to the memory of Eileen Hewett and Frank Summers. Gone, but not forgotten.

Contents

List of Figures

List of Acronyms and Abbreviations

AESTPI Application Execution System and Target Platform Interface

API Application Programmer's Interface

ASSC Avionics Systems Standardisation Committee

ABS Anti–lock Braking System

CAN Controller–Area Network

CRC Cyclic Redundancy Check

DLL Data–Link Layer

DLL–P Data–Link Layer Packet

DIO Digital Input/Output

ECU Electronic Control Unit

EMI Electromagnetic Interference

FCR Fault–Containment Region

GVA Generic Vehicle Architecture

HUMS Health & Usage Monitoring Systems

HVAC Heating, Ventilation & Air Conditioning

IEEE Institute of Electrical and Electronic Engineering

ISTAR Intelligence, Surveillance, Target Acquisition and Reconnaissance

IP Internet Protocol

I/O Input/Output

kbps kilobits per second

LCD Liquid Crystal Display

LIN Local Interconnect Network

MAC Media Access Control

MID Message IDentifier

NEC Network Enabled Capability

NL Network Layer

NL–P Network Layer Packet

OBD–II On–Board Diagnostics, version 2

OSI Open Systems Interconnection

PTU Primary Time Unit

RADAR RAdio Detection And Ranging

RTR Remote Transmission Request

RTAI Real–Time Application Interface

RTC Real–Time Clock

TDMA Time–Driven Multiple Access

TCP Transmission Control Protocol

TTP Time–Triggered Protocol

uint16 Unsigned 16–bit integer

USAF United States Air Force

V&V Verification and Validation

VSI Vehicle Systems Integration

YRL Yaw–Rate Limiter

Chapter 1

Introduction

1.1 Modern Military Vetronics

1.1.1 Trends in Vetronic Complexity

Modern military vehicles contain an ever–increasing number of embedded electronic subsystems of various types and levels of complexity, and the trend shows no sign of slowing down. Current vehicles contain sensors, computers and communication equipment with capabilities far greater than those with which the vehicles were designed, some of which are linked with simple embedded networks to allow data exchange and information fusion. These electronic subsystems are commonly referred to collectively as "vetronics", a contraction of Vehicle Electronic Systems. Future vehicles will be developed with inbuilt control and sensing networks from the initial design, but are likely to undergo a similar process of upgrade and expansion over their service lifetime. Consequently, the process of designing new vehicles and components or subsystems is expected to continue to increase in complexity.

As combat doctrine continues to evolve towards a more information–centric view of the battlespace, and technologies such as ISTAR (Intelligence, Surveillance, Target Acquision and Re-

connaissance) and integrated command and control systems become major components of military operations, western armies are becoming more and more dependent on electronic vehicle systems (for battlespace information, surveillance, telemetry and positioning, as well as functions such as fire control and intra– and inter–unit communication). Since these systems are becoming key to military operations, it is self–evident that they should be reliable and robust. Equally, however, they should be maintainable and extensible, in order to incorporate new technologies and address new threats as they become apparent. Military vetronics, then, is a complex and ever–changing field, and the challenge of developing a cost–efficient, reliable vehicle architecture that is sufficiently modular and extensible to serve in a modern army is a difficult one.

1.1.2 Standards and Guidelines

The development process can, of course, be simplified by the provision of standardised processes and technologies. In the United Kingdom, Qinetiq have released a set of recommendations: the VSI (Vehicle Systems Integration) Standards and Guidelines [1] to guide the development process by offering advice on technology selection, recommended network architectures and device configuration. It is hoped that by following the Guidelines and implementing the Standards, device manufacturers and vehicle integrators will be able to increase the modularity, composability and maintainability of modern and next–generation military vehicles. The United Kingdom Ministry of Defence and Qinetiq are unusual in choosing to make such an effort: at present, no other armed force is attempting to specify open standard interfaces and architectures for military vehicles. The VSI programme is supported in part by the past work of my colleagues Dr Charchalakis [2] and Dr Valsamakis [3], among others.

The VSI Standards and Guidelines include recommendations on the use of multiple integrated networks, each one specialised to its task (safety–critical networks for vehicle drive– and fire–control, deterministic networks for internal management and sensor control, and high–bandwidth multimedia networks for video data, for example). A vehicle designed around these guidelines uses an appropriate network for each function, data being exchanged between them as necessary to report status or to manipulate properties of devices on other networks. Whether a network is "appropriate" for a given use is a function of its properties including available bandwidth, reliability, safety–criticality or determinism, complexity and (importantly) implementation cost.

Since the inception of the VSI project (in June 2000), a great deal of investigation and research has been performed on the subject of vehicle systems integration and vetronics. On the 20th of August 2010, the UK MoD published DEF–STAN 23–09 [4], the Generic Vehicle Architecture (GVA). The GVA is a set of mechanical, electronic, power and interface specifications that all new military vehicles acquired by the UK MoD are required to implement. It recommends the use of open standard technologies and systems where possible, giving advice on integration methods and collecting together a variety of recommendations and DEF–STANs for choices of implementing technology in different areas of vehicle integration. The GVA DEF–STAN was not published until this thesis was almost ready for submission: it is included here to indicate the way in which vetronics standards are continuing to develop.

1.1.3 Certification

Since the behaviour of the vehicle and its safe operation is strongly dependent on the correct operation of the vetronic systems, ensuring correct operation is of paramount importance. This can be done in simple systems by certification and testing, but as the number of protocols supported by a node and the number of network nodes in the system rises, it becomes increasingly impractical. Individual nodes on a given network can be certified as conforming to a given standard, as can simple segments in some cases, but the overall vetronics system cannot, as interactions between nodes tend to multiply with the square of the number of nodes. As the number of interactions between nodes (and within and across protocols and standards) increases, the certification process becomes intractable, as the number of variables that must be accounted for increases. To some extent, however, it is possible to certify the behaviour of a single network segment (composed as it is of a small number of certifiable nodes and a single communication standard), and then to connect certified networks together through carefully designed interconnectors. As a result, the production of an integration testbed on which individual subsystems can be tested and developed is of considerable interest. As well as allowing such testing and development, a testbed would also be useful for ongoing research into integration and complexity issues.

1.2 Embedded Systems

In this section, I will give a brief overview of the field of embedded systems, to ensure the reader is familiar with the terminology and concepts used later in this thesis.

1.2.1 Introduction to Embedded Systems

In its most basic form, an Embedded System is a small microcontroller that, together with a group of connected sensors and actuators, performs a function. Embedded Systems can be found in a variety of applications in all areas of modern life, from the mobile telephone and the microwave, up to and including the control systems of a modern automobile or, at the extreme end of the scale, a nuclear power plant. In each case, the control functions are implemented primarily in the form of a set of software instructions (commonly referred to as "firmware", since it bridges the gap between *soft*ware and *hard*ware) running on the microcontroller or group of microcontrollers, taking into account signals from the sensors and driving the actuators accordingly.

In the case of a mobile telephone, the microcontroller is called upon to perform a large number of relatively simple tasks: encoding and decoding voice data using an appropriate codec, communicating that data with the cellular telephone system in its encoded form, receiving input from a keypad and displaying output on a small screen. In the case of a nuclear power station, the control functions tend to be more complex, and are consequently often implemented on a group of interconnected microcontrollers, so that each microcontroller can be dedicated to a part of the task and hopefully reduce the running time of the control algorithm by executing unrelated parts in parallel. It is relatively common for control functions to be implemented by a group rather than a single microcontroller, particularly in automotive and industrial control applications, where the sensors and actuators connected to the control system may be distributed over a significant physical range rather than close together.

If the control system is to be composed of a group of co–operating microcontrollers rather than a single device, it will need a communications system to carry data between its component units. This communications system will be required to carry user commands, sensor readings, state broadcasts and remote dispatch requests, and will operate in one of several modes depending on the type of data it is carrying and the type of system of which it is a part.

1.2.2 A Proliferation of Platforms

A microprocessor is a general–purpose integrated circuit that reads instructions and data from a sequential memory structure, and manipulates the data according to the instructions. A typical microprocessor will be clocked in the hundreds to thousands of MHz range, have on–board memory management and floating–point acceleration, and require a set of support ICs such as both RAM and ROM, peripheral multiplexers and bus interface devices. A microcontroller is a simple microprocessor that reverses this arrangement: it will normally have memory and peripheral devices integrated in the same package, but tends not to support floating–point operations or complex memory management.

A microcontroller will typically include a range of integrated peripherals, commonly including analog–to–digital and digital–to–analog converters, pulse–width–modulation devices and communications modems for protocols such as CAN and LIN. Additionally, many will include several RS232 modems, both to connect to devices that use RS232 for control and to take commands from and give feedback to the user. A microcontroller will typically operate at a lower clock–speed than a microprocessor, commonly in the range from single unit MHz to a few hundred MHz. However, these speeds are quite sufficient for the embedded control applications in which microcontrollers are typically used, and result in a significantly lower power consumption than a microprocessor (as low as 200μA in the case of some 1MHz Atmel AVR devices). This broad range of integrated peripherals, coupled with their minimal power consumption, means that a microcontroller is often the only silicon device needed to implement a simple process controller.

A broad variety of embedded communications systems exist, ranging from the simple (RS–232, RS–485) through the frame–based (CAN, LIN, simple Ethernet) and ending in the safety–critical systems (TTP-C, FlexRay). The simple systems typically provide a way to transfer bytewise data between two linked microcontrollers. More complex frame–based systems introduce the concept of addressing, allowing a single message to be transferred to a single target device on a network containing many, or to be broadcast to a group of devices. A frame–based system will normally allow larger "chunks" of data to be transferred: where a standard RS–232 messages is typically small, containing a single octet of data, frame based systems can be used to transfer several octets in a single message (ranging from a maximum payload size of 8 octets in a CAN message to approximately 1500 octets in an Ethernet frame). Finally, safety–critical communications systems will include additional data alongside the payload that is used to keep all units in the cluster syn-

chronised to a common timebase, maintain the logical structure of the cluster, and allow the content of the payload to be error–checked to ensure that it has not been corrupted in transit. The subject of embedded communications is discussed in more detail in section 2.1.

1.2.3 Real–Time Devices

The great majority of microcontroller–based embedded systems (certainly those in process–control and automotive applications) can also be considered as "real–time" systems. A "real–time" system is one in which parameters such as communications latency, response time and worst–case execution time are tightly controlled (to take a simple example, an automotive steer–by–wire system requires a response time at or below the threshold of human perception, in the region of 10ms, to ensure safe and accurate operation).

Systems that can be considered "real–time" are normally classified as either *soft* or *hard* real–time. A hard real–time system is one in which continued correct function (and, potentially, operator safety) is strictly dependent upon the system performing within its designed response deadlines. Typically, hard–real–time constraints are found in areas where the system is interfaced at a low level with physical hardware of a timing–critical nature, such as industrial process control and in the automotive sector, as well as in medical technology such as nuclear–therapeutic and –diagnostic equipment. A soft real–time system, by contrast, will tolerate a certain amount of latency and deadline overrun, compensating where possible to stay close to the required timing. Soft–real–time systems can be found in areas such as video display systems (where it can be occasionally preferable to skip an entire frame than to slip progressively out of synchronisation with the audio track) and in operating system process management (where it may be more important that a process runs to completion than that other processes are serviced in a timely manner).

In order to provide real–time guarantees, the system should not be expected to operate at full capacity. A system loaded near its operational capacity may not be able to meet all its timing deadlines, due to resource contention or multi-tasking; it is common to see an 80% system load considered a safe maximum value.

1.2.4 Determinism

A given system is defined as deterministic in addition to being "Real–Time" if its behaviour can be accurately predicted from its current state and its current inputs. This is a desirable property in many embedded systems, as it means that the timing properties of a variety of operations within the system can be measured, then used to generate an efficient schedule of operations such that the system performs its function at the maximum possible speed. A non–deterministic system being scheduled in the same way risks the variation in operation timing properties causing one or more operations to overrun, which is likely to result in at best degraded performance, at worst outright system failure.

As a single embedded system can be deterministic, so can an embedded network. Deterministic networking protocols tend to require a small amount of additional overhead, both in terms of data required in frames travelling the network and in the nodes communicating on the network (a deterministic networking stack typically requires timing and interrupt resources from the host to maintain real–time behaviour). If a deterministic network is presented with the same input frames at the same points in the time cycle, it will always deliver the frames in the same order (not necessarily the input order) and with the same delivery latency. It should be noted that when determining whether one frame is "the same as" another frame for the purposes of studying a deterministic network, the decision can be made solely on the basis of the routing and priority information assigned to that frame: differences in the payload should not affect delivery parameters. MilCAN is an example of a deterministic embedded networking protocol; see section 5.3.1 for more details.

1.3 Research Goals

As has been noted previously, the level of complexity of deployed vetronic systems continues to rise. A complex deployed system implies a complex design and a complex development process. However, this need not be the case: a suitable developmental methodology (accompanied by an appropriate testbed) can substantially reduce the design and development workload for a given project by providing a base on which to build.

It is, therefore, the overall goal of this thesis to document the design of an abstract vetronic networking testbed, which was then mapped on to a set of implementing technologies and tested for

performance. It is hoped that this testbed and this example implementation will be a useful tool in the design and development of the next generation of vetronic networks and networked systems.

In order to be useful, the proposed testbed must exhibit certain features, at a minimum. It must contain at least one deterministic network segment, and at least one safety–critical network segment, since these networks are the most likely to present integration problems (integrating the majority of simple and frame–based networks is not difficult, as long as the network data is being transferred into has sufficient capacity to contain it, while deterministic and safety–critical networks have multiple other problems such as state maintenance, membership and timing). It must contain at least one programmable bridge device, capable of transferring messages from one network to another (ideally, every network segment will be connected to every other network segment, the connecting bridges being configured to block traffic by default). Finally, it must contain some form of measurement and monitoring system so that the behaviour of the segments composing the testbed can be manipulated to simulate one or more testing scenarios.

It is hoped that the methodology and testbed design resulting from this research will reduce the time necessary to develop vetronic systems, whether the system in question is a component to be added to an existing vehicle or a completely new vehicle architecture. This will be possible due to the way in which the testbed can be configured to simulate the behaviour of the other segments being developed: in the case of a new component or subsystem, it can simulate the dataflow to which that component or subsystem will be exposed in use, while in the case of a new network it can inject strategically chosen frames containing data to describe a testing scenario, and the response of the vetronic architecture to this scenario can be observed.

With an appropriately detailed specification, it will be possible to begin simulation testing comparatively early in the development process. This will result in a reduction in development risk (since flawed implementations should become apparent during early testing, before they become entrenched in the design). This reduction in risk will result in turn in a reduction in development time and development cost.

The existence of an integration testbed and methodology will simplify the development of segmented and service–oriented networks, as the segment under development can be connected to a simulation of the other segments in the vehicle. It is not necessary to develop the entire networked system monolithically, nor to maintain a complete hardware replica of the networked system for the purposes of testing the integration of new components.

It should be noted, of course, that military projects often find civil applications and vice versa. Although the Verification and Validation (V&V) Testbed detailed in this thesis was developed under the auspices of the UK MoD, there is no reason why it could not be used to aid in the development of a civilian vehicle.

1.4 Achievements

Modern vetronics networks continue to increase in complexity, as new devices and new features are added, both in the design stage and to finished vehicles. A vetronics network will often consist of multiple heterogeneous network segments, and the integration of these segments is desirable.

In the process of performing the research leading to this thesis, three main achievements were reached. They and their background will be explained in detail in the remainder of this thesis.

- Interfaces and procedure for integrating heterogeneous embedded networks were defined: in particular, for safety–critical and deterministic systems.
- An abstract vetronics testbed was produced, which can be configured to represent most vehicle networks, or act as a common reference during design and testing.
- An triggering and monitoring system was produced that allows performance and behavioural data to be gathered from the abstract testbed in a repeatable manner.

1.5 Thesis Outline

Chapter 2 provides an introduction to the technologies relevant to this thesis: fieldbuses and embedded networks, safety–critical systems and X–by–wire systems.

Chapter 3 provides an examination of the technical considerations that relate to the development of a network bridge, as well as presenting an introduction to the VSI Standards and Guidelines and to common development methodologies.

Chapter 4 is the first contribution chapter, and introduces the design of the V&V Testbed: the component network types, the triggering system and the essentials of the bridging system.

Chapter 5 is the second contribution chapter, giving the chosen technology mapping used to implement the abstract design in this research.

Chapter 6 is the third contribution chapter, presenting the results of a series of tests run, first on the testbed configuration described in chapters 4 and 5, and then on a slightly altered configuration that produces significantly better results.

Chapter 7 presents concluding remarks, as well as some suggestions for further work to investigate the properties of the testbed and to apply it to new technologies emerging in the field.

Chapter 2

Real Time Embedded Systems

Introduction

In this chapter, a brief overview of the field of embedded and safety–critical networking will be presented, touching on the key issues contained in each area and defining terms that will then be used in the rest of this thesis. There will be a discussion of fieldbus technology, including an example fieldbus (the Controller–Area Networking protocol), followed by a study of the concept and properties of a safety–critical system, including example applications.

2.1 Fieldbuses

2.1.1 Fieldbus Overview

Automotive electronics and aircraft electronics have followed a similar development path. In the case of aircraft, the mechanical pushrods and cables that once linked the pilot directly to the control surfaces and engines of the aircraft have been slowly replaced with more complex and featureful electronic links. In the case of cars, the mechanical links have been augmented over time with

electronic systems such as Anti-Lock Braking and Traction Control Systems, and it seems likely that electronic links will begin to replace the mechanical linkages in new vehicles in the near future. The presence of these systems, together with other electrical and electronic systems within the car such as HVAC, lighting, entertainment and navigation, means that the electrical and electronic complexity of a typical modern vehicle continues to increase (Albert, in 2004, noted that modern upper–class vehicles can contain up to 70 ECUs [5]

In 2002, Leen [6] notes that some modern high–end vehicles may have up to 4 kilometres of internal wiring. The reason for this is quite simple: in many cases, each electronic system uses dedicated cabling to carry data from input sensors and switches to output actuators, often via intermediate processors. This approach results in a considerable amount of effectively duplicated cable, running from adjacent switches on the control panel to adjacent actuators elsewhere in the vehicle. This duplication of cable adds a significant amount of weight and complexity to the vehicle. Additionally, it makes servicing and maintenance notably more difficult, as tracing and accessing a single cable in a multi–cable run becomes more challenging as the number of cables in the run increases.

Introducing a Fieldbus to the vehicle allows the complexity of cabling to be reduced without sacrificing the feature–set of the vehicle. In its simplest form, a fieldbus is a local–area network in which the local area is a vehicle rather than a building or a floor of a building. A traditional electronic control system would interconnect switches and actuators in a given subsystem with their controlling processor using dedicated cables. A fieldbus–based system, by contrast, will connect switches and actuators to physically local processors, which are then interconnected by a digital communications network. Control functions that would be based on a single processor may be handled on the processor responsible for controlling the relevant actuator, or by an intermediate processor. Instead of long sensor and actuator control cables being run to a central processor, a fieldbus system makes use of short sensor and actuator control cables and an increased number of processors, connected by a much smaller number of long data networking cables. As each processor will typically be responsible for multiple sensors and actuators, the complexity of the interconnect is decreased substantially.

It should be noted that the fieldbus concept is neither particularly new nor unique to the field of transport engineering: its original genesis was in the field of industrial process control, where fieldbuses such as Foundation Fieldbus and PROFIBUS were developed in the mid–1980s ([7],

[8]) to support production–line automation and error reporting. Functionally, industrial fieldbuses and vehicular buses are similar if not identical: in both cases they carry the data given to them according to a predefined schedule, it is merely the semantic meaning of the data that changes between applications.

If a metaphor is required, a fieldbus system can be seen as a public transport system for data. Instead of each subsystem of the vehicle having a dedicated transport system in the form of a cable link (or in this metaphor, a car), the fieldbus "picks up" data at the input point and carries it around the network, "dropping it off" at the output point. The analogy is imperfect because most fieldbuses are message–addressed rather than node–addressed (a message on the network has a message identifier, as opposed to source and destination addresses), so one message can be received by multiple nodes if its contents are of interest to them. In addition to simplifying cable routing, there is another potential benefit to the implementation of a fieldbus–based architecture: it becomes much simpler to add subsystems to the vehicle, as plugging new devices into the fieldbus and assigning them messages on the network is much simpler than finding a way to route yet another cable through congested cable-ways. In particular, the majority of fieldbus systems tend to implement some form of "composability", a property of the network that means that so long as two applications don't both write the same message IDs, they can coexist on the same network without interfering with each others' behaviour. When adding devices to a fieldbus in this manner, however, one must take care to ensure there is enough spare network capacity available to handle the new application.

The remainder of this section will discuss the properties of fieldbuses, in particular common Topologies used, Triggering methods (event triggering and time triggering) and the subject of Network Segmentation and Message Bridging.

2.1.2 Topology

The Topology of a communications network is the arrangement of its nodes, relative to one another. A diagram of the topology of a network represents the interconnections between nodes and approximate distances between them. It does not represent the positions of the nodes in physical space, or the precise route that connecting cables take between them: it is a logical map, not a physical one.

While the topology of a real–world network can be extremely complex, it will normally be possible to generalise it as one of the following examples, by aggregating nodes into subsystems and studying the system at a high level of abstraction (some less common topologies such as Ring and Daisy–Chain were not considered, since they are rarely seen in embedded networks).

Multiple Channels

A multiple–channel "topology" is not really a topology in and of itself: rather, most of the following network topologies can be built either single–channel (as shown in most of the diagrams) or multiple–channel (by replacing each single link with multiple parallel links). Introducing an additional channel to the network increases wiring complexity, but can result in significant benefits in terms of reliability or available bandwidth. It should, perhaps, be noted that the most common multiple–channel implementation is a dual–channel system.

If the two channels are redundant, carrying the same data, the reliability of the network is improved. The messages can be compared on arrival, and differences between messages that should be the same indicate a fault on one or other bus. In the simplest case, the receiver can request retransmission, or use the message checksum (if present) to select the undamaged message. If *both* messages are corrupted, a simple dual–channel system provides no benefit. However, it lays the ground for spatial and temporal redundancy (summarised below, and explained in more detail in section 2.2.1).

A multiple–channel redundant system in which the physical layer for each channel takes a different path through the vehicle or factory is spatially redundant. Damage to any one location is less likely to destroy the communications link, as the loss of one channel can be compensated for by the duplicate messages on the second channel. Alternatively, a multiple–channel system in which the same message is sent on different channels at different times is temporally redundant. Transient EMI and other faults that would otherwise damage all messages in the same way can be detected because the corruption occurs at different points in the different messages.

If the two channels are not redundant, a multiple–channel system allows the same topology to achieve a higher throughput rate, because messages can be sent on either channel. Assuming the nodes of the network are capable of producing and consuming messages at that rate, the multiple–channel system is capable of throughput up to the bandwidth of each channel multiplied by the number of channels. Equally, a second channel can be reserved for high–importance messages,

so that their transfer latency is not affected by the presence of other messages being transferred between nodes.

Passive Bus

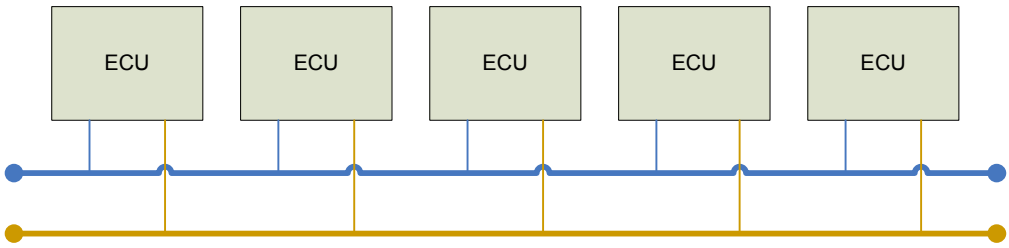

Figure 2.1: Passive Bus topology

The Passive Bus topology is simple and common. A communications bus is laid, and all nodes are connected to it directly. The communications bus will require termination resistors, but contains no further active components. Consequently, messages sent on the passive bus will be received by all connected nodes: it is necessarily a broadcast topology. The diagram in figure 2.1 depicts a dual–channel passive bus topology, as is often used in CAN networks.

Passive Bus systems are cheap and simple to build, requiring comparatively few components. Their broadcast nature, however, means that it is difficult to achieve high throughput when compared to topologies such as the active star. Controller–Area Networking development segments are commonly based on passive buses, as are FlexRay development networks, due to the comparative simplicity of wiring.

Tree

The Tree topology assumes that nodes within the network have multiple communications interfaces, and that messages can be forwarded over a succession of point–to–point links during their journey from source to destination. A network following the tree topology has a defined "root" node: nodes can then be classified into levels depending on how many links must be followed from them to reach the root node (the root node has level 0, nodes directly connected to it have level one, nodes connected to *them* have level 2, etc.). Additionally, a tree topology is said to have a "branching factor", defined as the average number of arcs connected to a node within the tree (by way of example, the topology depicted in figure 2.2 has a branching factor of 1.6).

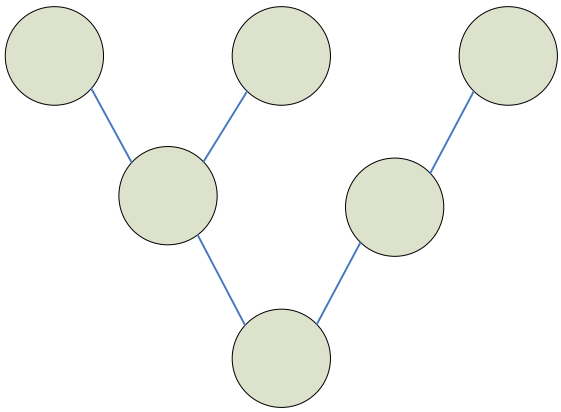

Figure 2.2: Tree topology

A tree topology is said to be a “spanning tree” if every node can trace a connection from itself to the root, but there are no cycles (closed loops in the graph). In the case of a weighted tree (link weightings being assigned according to the domain: in networking, latency or bandwidth are common weighting factors), the spanning tree for which every node has the minimum possible cost for the route linking it to the root is called the “minimum spanning tree” for that tree.

The minimum spanning tree topology is common in enterprise networking, where the connection from a desktop computer to the backbone router may pass through a series of switched subnets en route. It is somewhat less common in embedded networking, where the majority of devices will be on the same subnet as the devices to which they communicate.

Passive Star

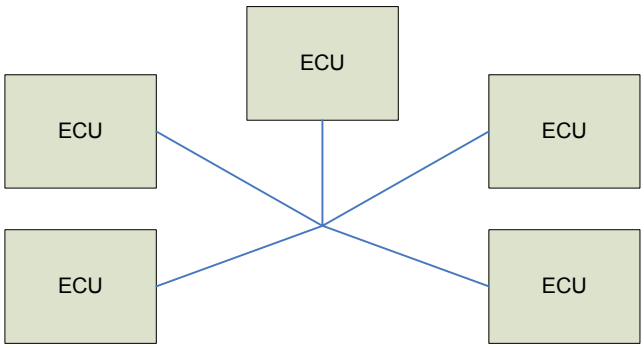

Figure 2.3: Passive Star topology

The Passive Star topology is more complex to construct than the Passive Bus, since it requires a central cable splice between individual node links. Like the passive bus, it contains no active

components, being nothing more than a wiring loom. Again, like the passive bus, the passive star is necessarily a broadcast topology, since there is nothing to prevent the signal from propagating along all branches of the star.

Ignoring the relatively minor complexity involved in building the central cable splice, a passive star topology has the same advantages and disadvantages as a passive bus topology.

Active Star

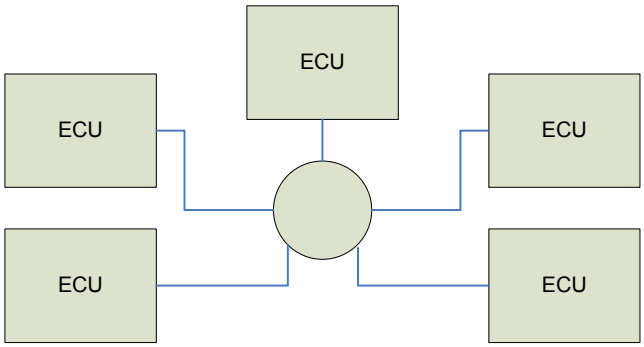

Figure 2.4: Active Star topology

The Active Star topology is superficially similar to the Passive Star, but its electrical properties are notably different (it is, in fact, more similar to a Tree). Instead of a cable splice, the individual node links are connected to a central switching fabric that broadcasts a message received from a given branch to all other branches. The central star coupler regenerates passing signals, and can optionally filter incoming messages such that they are broadcast only to nodes that declare an interest in the content of the message.

The central star coupler introduces additional complexity and cost into the network design, but it also improves signal quality (reducing corruption and retransmission) and allows for the possibility of message filtration, potentially increasing the maximum available bandwidth (if two messages have different source and destination nodes and the star coupler has sufficient capacity, they can occupy the network simultaneously as they need never share resources).

Redundant Active Star

The simplest form of a redundant active star network is a two–channel network in which each node has two communications interfaces, each interface connected to a different active star coupler (for

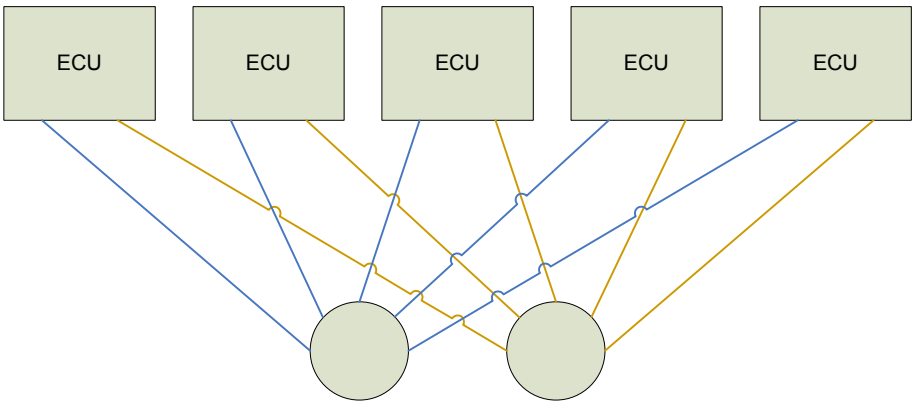

Figure 2.5: Redundant Active Star topology

a total of two star couplers, one per channel). Such a network is depicted in figure 2.5, and is often used in production FlexRay networks.

There are several possible benefits to network redundancy. A network is said to replicate another network if it contains the same messages at every point in time (being effectively a synchronised copy of the first network). If the two networks replicate each other, then physical damage or failure of one network will not compromise the ability of the cluster to transport messages between the ECUs that comprise it. It is possible to further mitigate the effects of damage on the network by routing the redundant cables through different physical areas, so that damage to one does not affect the other without having also terminally damaged one or more nodes in the process. The principle of redundancy in network design is further discussed later in this thesis, in section 3.2.3.

If the two networks do not replicate each other, the potential bandwidth of the cluster is doubled, as messages are no longer transmitted on both networks, meaning that it is possible to transmit two different messages at the same time, one on each network.

Multiplying the number of star couplers and associated link cables increases the cost of implementation in a more or less linear manner, but the benefits of doing so tend to be reduced once the number of redundant networks passes low unit numbers.

Cascaded Active Star

A Cascaded Active Star is a non–redundant configuration containing a single channel, but multiple active star couplers. Each node in the network is connected to one star coupler, and the star couplers themselves are connected together, resulting in a spanning tree. The configuration in

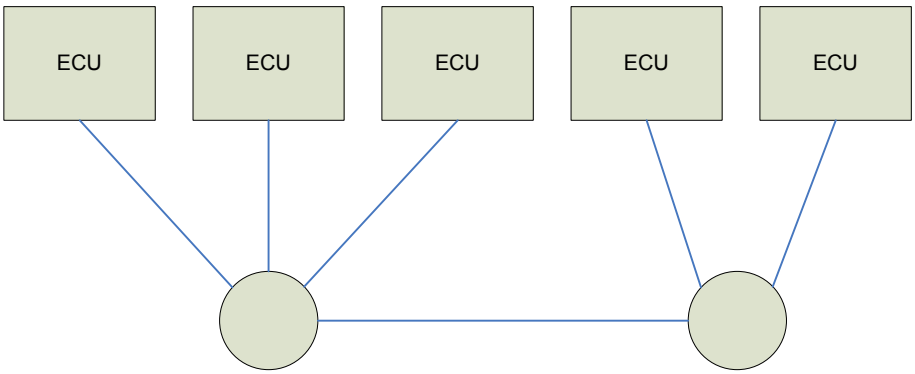

Figure 2.6: Cascaded Active Star topology

figure 2.6 has three nodes connected to one star coupler and two nodes connected to the other (in implementations, the link connecting the star couplers will often be set to operate at a higher bandwidth than the links on the periphery, reducing congestion at the couplers).

The Cascaded Active Star topology is more expensive in both cost and complexity than a single active star, but provides the potential benefit that it is possible to partition the network physically but not logically, using a single (or a small number) of "backbone" connections to link islands of nodes together, each with its own local star coupler. It should be noted, however, that each Active Star Coupler on a link between two nodes will add an additional overhead in terms of transmission time, as each Active Star must at least process and regenerate the signal, and may optionally take additional time to filter it and rebroadcast only on certain channels.

2.1.3 Triggering

In addition to the topology of the network, it is important to consider how it is triggered when studying its properties. Where topological design and cable length make it possible to estimate network bandwidth and latency arising from hardware concerns, triggering makes it possible to consider unavoidable protocol delays as well.

The triggering procedure for a network indicates how messages held in node transmission buffers are dispatched on to the bus. There are two common triggering procedures used in embedded networks: event triggering and time triggering. Both are explained briefly below.

Event–Triggered Networks

In an event triggered network, each node will attempt to transmit messages as soon as the application running on the node requests transmission. Exactly when the message is transmitted will depend on the collision resolution procedure of the protocol in use and network load, but it will be sent as soon as reasonably possible.

Event triggered designs have the advantage that messages always tend towards the minimum possible latency for the network conditions in which they operate. However, since there is no control over delivery times, messages can easily arrive out of order and, in heavily loaded networks, with significant delays (due to large numbers of other messages winning collision resolution against them before transmission is successful).

Time–Triggered Networks

In a time triggered network, there exists a global schedule that allocates periodic time slots to messages. A copy of this schedule is normally present on each node, and each node has a view of the global timebase, whether synchronised by a timekeeping master node or by group agreement. A message may then only be transmitted in its allocated time slot.

Time triggered networks are highly deterministic: it is possible to predict which message will be present on the bus at a given time with a high degree of accuracy. However, because messages must wait for their next scheduled slot before being dispatched on to the bus, a degree of additional latency is unavoidable (the exact amount of delay applied to each message being dependent on the current state of the schedule and the time remaining until an appropriate slot is available). Messages should never be lost in a properly scheduled time triggered network, as each message is assigned to a slot of appropriate length.

2.1.4 Segmentation and Bridging

The availability of spare network capacity can be a serious problem in non–trivial networks. A typical modern vehicle network includes a large number of systems and controllers, including (but not limited to) the following:

- Power–train controllers

- HVAC (Heating, Ventilation and Air–Conditioning)
- Navigation subsystems
- ABS (Anti–lock Braking System)
- Traction–Control subsystems
- Engine–management computers
- Security systems such as central locking and immobilisers
- HUMS (Health and Usage Monitoring Systems)
- Entertainment systems (radio, video screens in some high–end models, etc.)

When the communications requirements of all these systems and controllers are considered together, it is commonplace for the required network bandwidth and availability to be in excess of that which the backbone network is capable of supplying.

It is normally possible to reduce the communications requirements of the fieldbus network by splitting it into several functional segments, such that all the devices relevant to a particular function are connected exclusively to a single segment. This has the effect of confining the bandwidth load associated with a particular task or subsystem to a single network segment, reducing global network bandwidth usage and frame latency. In addition, this segmentation can have an effect not unlike modular abstraction in computer programming, forcing clear interfaces between components of the overall system of systems and making it simpler to add new subsystems to the network. One popular segmentation scheme divides the network into three segments: Automotive (containing systems such as power–train, ABS and Traction Control), Multimedia (radio, communications) and Utilities (navigation, diagnostics, HUMS, HVAC).

Although segmenting the network in this way results in a significant reduction in required network bandwidth and availability, it has its drawbacks in that it restricts data flow between seemingly unrelated systems (the obvious example in the segmentation scheme above being that the diagnostics subsystem is unable to receive diagnostic data from the power–train). The compromise solution is to perform functional segmentation as directed above, then connect the segments together by adding intelligent network bridges: devices with multiple network interfaces that transfer data between segments according to a set of mutable rules. In this way, vital data (such as diagnostics

and error–reporting) can be moved between segments without overloading any segment with data irrelevant to its function.

2.1.5 Example Fieldbus: CAN

Introduction

CAN, the Controller Area Network, is a more complex fieldbus originally designed for use in automotive applications, although it has now spread into the fields of industrial automation and medical equipment. A CAN network does not have a master-slave architecture: all nodes are equal, and message collisions are avoided by bitwise arbitration instead of a master node regulating the bus. Since a CAN network can support a large number of devices, and allows a longer maximum cable length (50 metres before signal degradation forces a slower data rate [9]), CAN is often used for vehicle chassis networks and diagnostics – the OBD-II vehicle diagnostics protocol uses CAN for the communications link, as does SAE J1939.

Topology

CAN specifies a terminated linear multi–drop passive bus, using differential transmission on a single twisted–pair cable to minimise EMI. The bus lines should be terminated by a 120Ω resistor. A CAN communications controller may operate on one or more channels simultaneously, each channel being a complete and separate bus. The following diagram (Figure 2.7)is drawn from the ASSC Guide to Digital Interface Standards ([10]), and shows the standard topology and terminal transmitter/receiver configurations as specified in ISO11898 (microcontrollers and devices attached to the bus are called ECUs, Electronic Control Units, in CAN terminology, and indeed often in the field of fieldbuses in general).

Addressing

Like many fieldbuses (but unlike most computer networks), CAN is message–addressed rather than node–addressed. That is to say, instead of each node having a unique address and a message being addressed to a particular node, each message has an address (or, more correctly, an identifier) and is broadcast to all nodes. The result is that the message identifier defines the type and thus meaning of the message, and any nodes needing the information contained within it can read it from the bus.

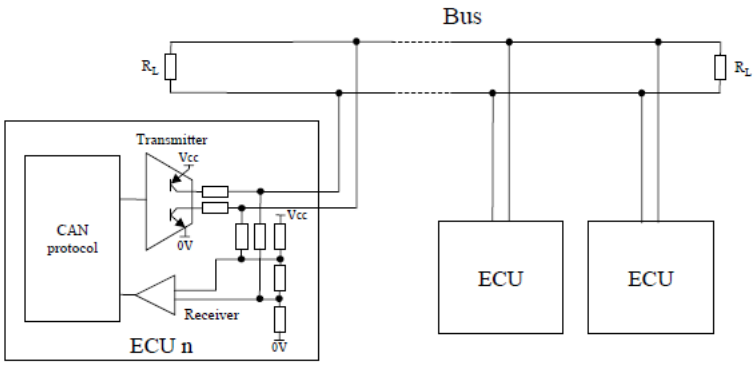

Figure 2.7: ISO11898 CAN Topology (from page 15 of [10])

Arbitration

As with all multi–master communications networks, arbitration is necessary when two controllers attempt to transmit different messages in the same time quantum. However, unlike in most networks, it is possible to resolve message collisions without corrupting data on the bus. This is achieved by the use of non–destructive bitwise arbitration, in which a logical 0 (defined as "dominant") overrules a logical 1 (defined as "recessive"). In CAN, message identifiers increase in priority as they approach zero (i.e. a message identifier that is numerically lower than another has a higher transmission priority). Since the message identifier is the first field transmitted whenever a message is placed on the bus (see Figure 2.8), the message with the higher priority (and thus the lower–valued identifier) will result in a 0 being placed on the bus before the message with the lower priority. Since all transmitting controllers are required to monitor the content of the bus after each bit transmitted, the controller transmitting the lower priority message will see the 1 it is transmitting overwritten by the 0 from another controller, and cease transmission until the bus is free again. Messages that fail arbitration are queued and retransmitted automatically. By the time the message identifier has been fully transmitted, only one message will remain on the bus, meaning that no corruption will occur (assuming the network was designed well, and each message identifier is transmitted by a maximum of one controller). Since only messages that fail arbitration cease transmission on collision (as opposed to all colliding messages in systems using CSMA–CD such as Ethernet), high–priority messages will tend to reach their destinations with minimal latency. There is, however, the risk that low–priority messages will be permanently held off the bus by higher priority messages (since the identifier of the message is the only attribute upon which arbitration is performed: messages held in the queue do not change priority with time,

resulting in what is essentially a form of scheduling starvation).

Variants

A CAN network can use one of two different message formats: the Basic frame format (CAN 2.0A) or the Extended frame format (CAN 2.0B). The Extended format is longer (64-128 bits, as opposed to 44-108 bits in the Basic format), and most of the additional length is in the message identifier; the Basic message identifier is 11 bits in length, while the Extended message identifier is divided into an 11-bit section and an 18-bit section. The formats can be differentiated by inspecting the fourteenth bit of the frame: in a Basic frame, this bit will be dominant (0), and in an Extended frame it will be recessive (1).

The increase in length of a CAN–B identifier (when compared to a CAN–A identifier) has both positive and negative effects on the network. A CAN–A identifier can contain a maximum of 2048 different values, meaning that the network can carry at most 2048 unique commands (with variable data). A CAN–B identifier can hold approximately 536 million different values. If the network has a large number of complex devices attached, or needs to be expandable for future use, 2048 message identifiers may prove insufficient, and a CAN–B based network may be more suitable. However, the use of longer messages without an increase in bus speed will mean that fewer messages can be carried in a given time period, and that bus latency will be increased by a small amount. Additionally, a longer message increases the risk that irregular corruption will damage the message, forcing retransmission.

Frame Formats

The CAN frame consists of a header, a payload and a trailer. The header begins with the start–of–frame field, then includes the message identifier (MID), a remote–transmission–request (RTR, allowing a node to request data from another node by transmitting a frame with this flag asserted) and a data–length code (DLC, indicating the number of bytes in the payload). The CAN–A and CAN–B frames differ in terms of their precise header layout (as shown in figure 2.8), but the elements listed above are common to both frames. The payload consists of between 0 and 8 bytes of application data, and the trailer consists of a a 15–bit CRC and an ACK bit, finally followed by the end–of–frame field.

There are two frame types that follow the formats described above in the CAN standard (the others,

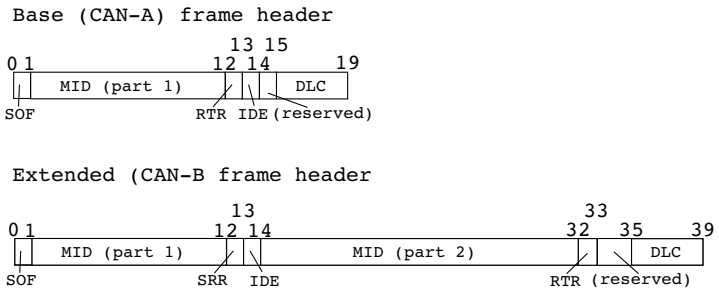

Figure 2.8: Basic (CAN–A) and Extended (CAN–B) frame formats

the Error and Overload frames, are substantially differently structured and used only to indicate fault conditions). The Data Frame carries data from the transmitting controller to the receivers, as one might expect: it has a dominant RTR and a payload of between 0 and 8 bytes. The Remote Frame, by contrast, is "transmitted" by the receiving controller. The receiving controller transmits a header which refers to particular data and has a recessive RTR, and at the beginning of the payload section, the remote controller is expected to take over and transmit the payload and trailer of the specified frame. In this way, both push and pull operations can be performed within the priority structure of the CAN protocol: if a node requires an item of data in order to perform calculations or operate machinery, it can request said data with a particular priority, rather than having to wait for periodic retransmission or sending a request and waiting for a response. Most importantly, if the remote transmission request fails priority arbitration due to a collision with a higher priority frame, it is possible for a particularly urgent information request to be retransmitted with a "higher" priority lower MID).

Since the CAN physical layer is conventionally based on a twisted–pair connection using differential transmission, it can be assumed to be moderately resistant to electromagnetic interference. Additionally, the frame CRC provides some protection against message corruption - the 15–bit frame CRC will allow message corruption to be detected and, in some cases, corrected.

2.1.6 Application Layer Concerns in CAN

Like many fieldbuses, the CAN standard specifies a transport network but does not include any kind of typical application–layer protocol: flow control, device addressing and packet fragmentation and reassembly are not a part of the standard. As a result, a variety of application–layer protocols have come into existence that use CAN for the transport layer and below, including CANOpen,

DeviceNet, J1939 and MilCAN. Since MilCAN is the implementing technology for a substantial proportion of the integration testbed, it will be covered in more detail later in this thesis.

2.2 Safety–Critical Systems

Now that concepts relevant to fieldbus systems have been addressed, an introduction to the properties and common implementations of safety–critical distributed control systems is presented. Once said introduction is complete, the discussion will move on to the allied field of X–by–Wire systems, a subset of safety–critical systems concerned primarily with vehicle control.

2.2.1 Safety–Critical Concepts

A safety–critical system, in its simplest form, is a system for which outright failure cannot be tolerated. Classic examples include certain classes of medical equipment, nuclear power plant management, and aircraft and ground vehicle control systems. In all of these cases, the system must always remain in a safe state, regardless of out–of–range inputs, contradictory information and component damage: the medical equipment must keep the patient alive, the nuclear power plant must remain safe (and, ideally, producing power) and the vehicles must remain under control or, in the worst possible case, in one piece.

In order to meet these requirements, a great many things must be in place:

Redundant Sensing. Typically, each sensor feeding data into the system is replicated at least once, allowing sensor failure to be detected. In practice, an odd number of sensors will normally be used, allowing majority voting to be performed to detect a single–sensor failure. Additionally, input values must be "sanity–checked" to ensure they fall within plausible ranges: an out–of–range value (such as, for example, a thermal sensor reading of -280°C) typically indicates a sensor malfunction, and should be ignored.

Redundant Function. System functions should all be replicated on at least two nodes, so that a single node failure does not impair functionality. This replication can be made more reliable by ensuring the nodes are significantly different (see section below, under "Functional, not Physical, Replication").

Redundant Communication. Any communications networks linking nodes in the system for the purposes of safety critical operation (as opposed to debugging, high bandwidth media transport etc.) should be safety–critical in themselves, including replicated communications buses, sanity–checking and guaranteed message delivery deadlines. Additionally, most safety–critical communications buses include a bus–guardian function of one form or another, specifically to prevent a faulty node from corrupting data on the bus. Finally, where possible, the network physical layer should be of a reliable type: differential twisted–pair or shielded co–axial cable or fibre–optic link are much less vulnerable to EMI than simple unshielded parallel or serial cable, and should be chosen over them to improve the reliability of communications.

Redundant Actuation. As the sensors feeding data in to the system must be replicated, so must the actuators controlled by the system. Typical implementations will involve fail–silent motors (which are designed to fail in a free–rotating rather than a locked or seized mode) on either a common shaft or a 1:1 gearbox, so that while all motors in the group are connected to the load, any one can fail without significantly affecting the performance of the system. Unlike in the sensor case, above, some allowances may have to be made for a reduction in torque, but in a properly designed system there will be sufficient excess capacity that this will not be a problem.

Reliable Power. It is a well–documented feature of electronic devices that they do not function well without a power source. In order to ensure the safety–criticality of the system as a whole, the power supply feeding all critical nodes must be stable and resilient. One way to ensure this is to design redundancy into the power supply system, as has been done in the other systems mentioned above.

It should be obvious from the above that redundancy of components, subsystems and functionality is the simplest way to improve the safety–criticality of a system.

Functional, not Physical, Replication

Subsystem and component replication is one of the most important stages involved in making a given system suitable to be used in a safety–critical role. However, simple physical replication only solves part of the problem.

If a given device fails due to a component malfunction, an exact physical copy of the failed device will be able to take over and continue the operation. However, if the failure is a result of a fault in the design of the device, a physical copy is likely to fail in the same way, thus negating the benefits of replicated design. So–called Common–Mode Failures can be avoided by replicating the device functionally, rather than physically. A functional replica is a device that replicates the designed function of its counterpart, but may do so in a different way, using a different algorithm or different hardware. The process of using functional, rather than purely physical, replication in such a system is commonly referred to as "design diversity".

The classic example of a functionally–replicated safety–critical computer system is the Digital Fly–by–Wire (DFBW) system used in commercial airliners. The Boeing 777 flight control computer (designed by GEC Avionics in the early 1990s) is a particularly good example: the initial design specified three quad–redundant computers, with each quad implemented in different hardware and programmed with different software, written in different programming languages ([11]). Any one of these twelve computers should, in an emergency, be capable of controlling the aircraft, and a control output cannot leave the computer cluster (and have an effect) unless it is agreed by the majority of the remaining functioning computers. The redundancy inherent in a single quad of identical computers should protect against component failure, and the presence of three very different quads, each doing the same task but in different ways, should protect against Common–Mode Failures. In practice, the 777 flight–control system is programmed throughout in Ada (a language designed with program reliability and maintenance as explicit design goals, see the introduction to [12]), but retains the overall design of three heterogeneous quads of computers, each quad running different software written to fulfill the same design specification.

Fault Containment

Redundancy alone does not guarantee reliability: in fact, a highly redundant system is more prone to failure, as there are more elements to fail. In order to improve reliability, some mechanism must be provided to manage system components, as a failed component must be removed from the system as quickly as possible to prevent it from interfering with the remaining functioning components. To address this problem, the system is typically divided into Fault–Containment Regions (FCRs). An FCR is a conceptual unit into or out of which faults cannot propagate: just as the elements within a region cannot be caused to fail by external faults or out of spec conditions,

equally the contents of an FCR cannot cause a failure in hardware outside its perimeter. The following diagram illustrates a replicated system making use of fault–containment strategies.

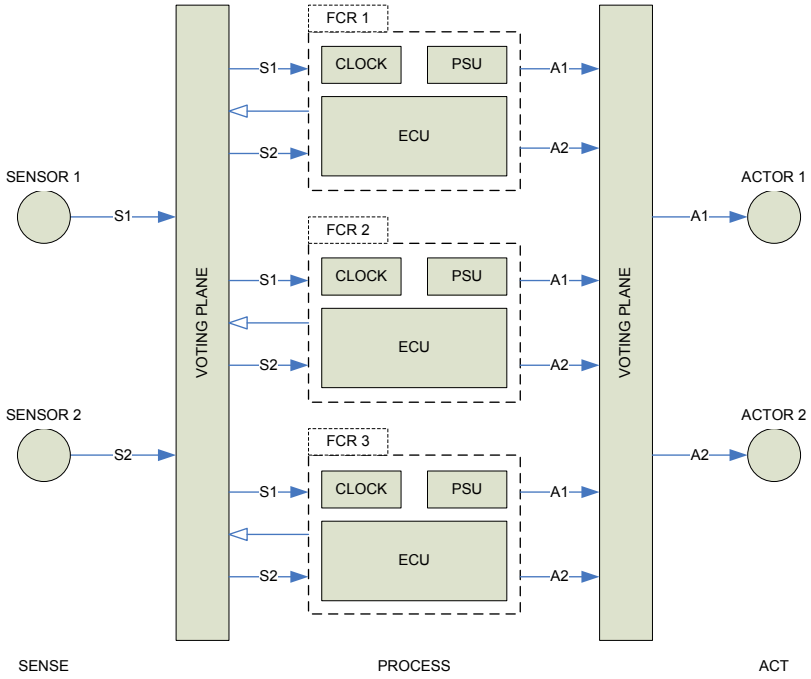

Figure 2.9: An example replicated system with fault–containment

Figure 2.9 shows three replicated ECUs, each in its own fault–containment region. The system reads a duplicated sensor, and commands a duplicated actor of some description (likely to be a pair of fail–silent motors driving a valve, or some similar mechanism). The two Voting Planes in the diagram may not be implemented in hardware (the alternative being some form of middleware on each ECU), but are required to allow management of signals passing from left to right in each case: the first plane allowing ECUs to "vote out" misbehaving sensors, blocking incorrect data from being used by the ECUs, while the second allows an ECU to prevent output from an erroneous sibling from reaching the actors.

It should be observed that although they are not present in figure 2.9, the power and clock resources feeding all components must also be partitioned for a functional element to be considered a true FCR. This should be self–evident: an FCR with proper fault–isolation on all input and output channels can still be caused to fail by a malformed clock signal, or by a loss of power supply. This requirement of fault–containment tends to result in a given FCR having its own internal clock, as well as some kind of power isolation system such as a simple UPS or on–board battery.

Preventing data corruption

Within a properly functionally replicated system, in which all sensors, processors and actuators have redundant counterparts and in which fault–containment boundaries are enforced, the engineer can be fairly confident in the reliability and safety–integrity of their design. However, although the processing hardware of the system is reliable, it is still possible for messages travelling on the communications networks linking individual processors to be corrupted.

Message corruption can be caused by transient electro–magnetic interference (EMI), cable damage or faults in the transmitting or receiving bus drivers (among other causes). An appropriate physical layer will minimise this risk: fibre-optic cable is not vulnerable to EMI, for example, and a differential twisted–pair or shielded cable can minimise its effect. Nonetheless, software approaches should also be considered, as they provide a way to recover from message corruption if and when it does occur.

It is possible to recover from message corruption by performing a Cyclic Redundancy Check (CRC) on the message or its payload. A CRC is a simple polynomial hash that is entirely dependent on its input data: thus, data with an incorrect CRC has been damaged in transit. By appending the CRC of a frame to the frame payload, it is possible to check the frame at the receiver by taking its CRC again and comparing the calculated CRC with the transmitted one. Due to the structure of the CRC algorithm, it is possible to recover damaged data under certain conditions, as the resulting CRC checksum changes in a predictable way as the input data changes. How many bits of damage can be recovered depends on the initialisation vector (IV) used when calculating the CRC and its resulting length – a longer IV will result in a longer checksum (and more overhead in communication), but will allow a greater amount of damage to be recovered before it becomes necessary to discard the frame and request retransmission.

In order to further mitigate the data corruption problem, safety–critical communications networks typically replicate message data both in space and in time. Spatial replication is the practice of sending more than one message by different routes (usually different physical buses). This approach addresses the problem of cable or transceiver malfunction, as such faults do not usually affect more than one bus or transceiver at a given time without having also caused catastrophic damage to the systems controlled by the safety–critical network, and the receiving processor can check the message CRCs to ascertain which is the undamaged frame. However, transient EMI will typically affect the entire network at the same time, without causing any physical damage, but for

a very short time period. Consequently, spatial replication does not improve the situation, as all spatial copies will often be similarly damaged.

In addition to spatial replication, then, something else is required. That something is temporal replication, replication in time as well as in space. Temporal replication results in an increase in message latency of more than 100% in the most reliable configuration: essentially every message is sent multiple times, on the fairly reasonable assumption that irregular transient EMI events are unlikely to damage both messages. Again, the message CRC can be used to find the undamaged message, or to repair any further minor damage.

Fail–Safe versus Fail–Operational

While a competent engineering team will attempt to prevent any form of fault from entering their delivered product, the fact remains that some failures will occur, due to user error, unexpected environmental conditions or component failure. In a safety–critical system, some degree of fault–tolerance is necessary, and there are several ways in which failures can be handled (descriptions drawn from [13]):

Fail Stop The simplest possible method of fault tolerance is simply to detect the failure and stop the system. While this is rarely desirable in an embedded control system, it prevents the system from continuing to operate in a malfunctioning state, and potentially behaving dangerously.

Fail Silent A fail–silent system is one that has the choice of producing a correct reponse or no result at all. While at first this may seem identical to the fail–stop system described above, a fail silent system has the important advantage that it handles transient failures gracefully. A transient failure that would stop a fail–stop system will simply cause a gap in function in a fail–silent system.

Fail Operational A fail–operational system continues to deliver service after a failure has occured, but will do so in a degraded mode, with known safety risks.

Fail Active In the ideal case, the system will recover from the failure without any degradation in performance or reliability.

In an ideal world, all safety–critical systems should be fail–active. In practice, this is not always possible due to the prohibitive costs of multiply replicating all system components, enforcing small, granular fault containment regions and running multiple redundant communications buses. It is often necessary to accept that a system will be able to recover the first major fault it suffers in a fail–active or fail–operational state (a so–called "limp–home" mode), then fall into a less satisfactory state on subsequent failures.

2.3 X–By–Wire

The term "X–by–wire" refers to a vehicle control system in which a traditionally mechanical or hydraulic linkage has been replaced by a digital electronic or network–electronic connection. Typically, the "X" is replaced by the name of the system so altered, leading to terms such as "steer–by–wire", "brake–by-wire", and "fly–by–wire" (the first two of these referring to vetronic systems, the last to avionics). In a digital–electronic X–by–wire system, a central controlling computer cluster is linked to the sensors and actuators that comprise the system's input and output by simple cable, with all signal conversion and processing taking place in the central computer cluster. In a network–electronic system, by contrast, most if not all sensors and actuators will be colocated with a controlling microprocessor, which then exchanges data with other microprocessors across a digital communications network. Control functionality is still traditionally provided by a central computer in networked systems, owing to the much greater simplicity of validating and testing a single computer system as compared to a system distributed across multiple physically remote processors. However, since input and output validation can be done by the local processors, the workload of the central computer cluster is reduced, allowing more advanced or computationally expensive software to be implemented in the freed space. Additionally, it is possible to implement some less–critical systems entirely in the distributed processors, bypassing the central computer altogether: Health and Usage Monitoring Systems are commonly implemented in this way.

In addition to the aforementioned software and communication benefits, network–electronic by–wire control has several other advantages. The majority of these can be traced back to the fact that network–electronic by–wire systems multiplex digital signals over a small number of bus cables, where the purely electronic system requires at least one cable per signal. Since the individual cables are fewer in number, they can be routed more efficiently through the vehicle frame. This

simplifies maintenance, as an individual cable can be more easily accessed when it is not in a large bundle of similar cables. Also, the increased ease of routing means that cables can be positioned to increase the vehicle's survivability, by ensuring that redundant buses are routed separately through the frame so that damage that severs one is unlikely to also sever the other without inflicting sufficient damage to independently cause vehicle loss. Finally, reducing the number of cables in the wiring looms throughout the vehicle will significantly reduce the contribution of the electronic system to the overall mass of the vehicle.

2.3.1 Fly–by–Wire

X–by–wire systems first came into use in military aviation, as a natural evolution of the power–assisted actuators that are necessary to allow a human pilot to control a large aircraft. Until recently, most fly–by–wire avionics systems have relied on a single central flight–computer cluster, linked to the relevant sensors and actuators directly. Now, with the advent of safety–critical digital data buses such as FlexRay and TTP, it is becoming possible to colocate processors with the sensors and actuators they control, and to distribute control functions across the avionics network, producing a true digital networked fly–by–wire system.

The first test of a digital fly–by–wire system with no mechanical backup was by NASA between 1971 and 1973 (as reported in Jaynarayan and Harper, [11]). A surplus F–8 fighter was converted into a flying research testbed, the mechanical links from cockpit to control surfaces completely removed and replaced with the guidance and navigation computer from the (then recently developed) Apollo command module. The resulting machine logged 58 flight hours before being decommissioned, proving that the concept of fly–by–wire control was implementable.

Following the success of the F–8 testbed, other implementations of the fly–by–wire concept began to emerge. One implementation of particular note is the YF–16 (later to enter production as the F–16), a prototype aircraft proposed by General Dynamics as an entry into the Lightweight Fighter development program. The program called for the design, development and testing of prototype aircraft to demonstrate new fighter technologies and their potential utility to the USAF. The YF–16 is designed to be unstable in the pitch axis, in order to maximise the usable lift generated by the airframe (Droste and Walker, [14]). This instability markedly improves the manoeuvrability of the aircraft, at the cost of making it near impossible for a human to fly unassisted (due to the speed of

reaction required to maintain level flight). The YF–16 makes use of a quad–redundant fly–by–wire flight control system to transmit commands from the pilot to the airframe actuators, and it is this system that provides the millisecond by millisecond corrections needed to keep the aircraft under control, freeing the pilot to concentrate on the mission.

In addition to making it possible for a human to fly the aircraft at all, the YF–16 flight–control system performs a set of assistive functions that further aid the pilot by protecting the airframe from certain undesirable situations. The Yaw Rate Limiter (YRL) is perhaps the best documented (on page 106 of [14]). Initial aerodynamic testing showed that the airframe was vulnerable to entering a flat spin under certain conditions, and further testing showed that the airframe was incapable of recovering from said spin if the spin rate was greater than approximately 0.35 revolutions per second. The Yaw Rate Limiter is a component of the flight control system that acts to maintain the yaw rate of the airframe below the critical value. When the Yaw Rate Limiter detects conditions that indicate a spin, it changes the mapping between pilot input and control surface commands, replacing pilot input with counterspin commands proportional to the rate of yaw; essentially, the YRL takes temporary command of the aircraft to recover the spin, then hands control back to the pilot. It is worth noting that this is a case of an electronic X–by–wire system containing features designed to explicitly assist the pilot, rather than simply translating their commands. The function of this system can thus be compared to Traction Control Systems and Anti–lock Braking Systems in motor vehicles (which will be studied briefly later, in section 2.3.2).

As might be expected, the field of fly–by–wire flight control systems has continued to develop since the Lightweight Fighter program concluded in 1975. The F117 ground–attack aircraft is of particular interest, as it uses a similar quad–redundant fly–by–wire flight control system to the F–16, but is unstable in all three axes (pitch, yaw and roll, as stated on page 48 of [15]). This instability is primarily due to the aerodynamic properties of the peculiar faceted fuselage design that is used in the F–117 to deflect incoming RADAR. As was the case in the YF–16, the F–117 requires continuous correction from the flight control system to maintain a stable attitude.

Civil aviation followed a similar development path to military aviation, and at roughly the same time. In the early 1970s, the Lockheed L1011 and the Douglas DC-10 were equipped with an auto–landing system that was capable of landing the aircraft in any and all visibility conditions (again, Jaynarayan and Harper, [11]). Both of these systems used a dual–dual redundant control system (meaning that each control axis is handled by a pair of identical channels, each composed

of a pair of fail–silent computers).

In modern airliners, such as the Airbus A–320 and the Boeing 777, multiply redundant digital fly–by–wire control systems with no mechanical backup are the norm rather than the exception. The Boeing 777 flight control computer (designed by GEC Avionics, UK in 1994) consists of a triple–triple redundant computer using design diversity in both software and hardware (see section 2.2.1 for a discussion of design diversity).

2.3.2 Drive–by–Wire

As fly–by–wire is to aircraft, so drive–by–wire is to road vehicles. Conventionally, the field encompasses steer–by–wire, brake–by–wire and throttle–by–wire, although additions such as shift–by–wire (control of the geartrain) and similar are becoming more commonplace.

Steer–by–wire systems replace the mechanical linkage between steering wheel and steering gear with an electronic connection. Although conceptually simple, steer–by–wire is likely the most complex problem to solve in the field of X–by–wire systems, since it is strongly reliant on a closed feedback loop between road, vehicle and driver. Small, rapid corrections at the steering wheel need to be transferred as quickly as possible to the wheels. Equally, small vibrations resulting from variation in the road surface and uncommanded deflections caused by ruts and small objects such as stones in the roadway cause the steering gear to assume an attitude other than that commanded by the driver. This situation, even if it occurs only momentarily before the system corrects it, must be fed back to the driver via the steering wheel in order for them to maintain control of the vehicle.

The electronic fuel injection system, introduced in 1979 ([16]), is the first drive–by–wire system to be introduced to the mass market, replacing the mechanically operated carburettor for throttle control. In a carburettor–based throttle system, the accelerator pedal pulls a cable that causes a series of butterfly valves to open, increasing the air and fuel flow entering the engine. The increase in fuel and air flow allows the engine to fire more frequently, increasing its rotational speed. In a vehicle with electronic fuel injection, the set point of the injector jets feeding each cylinder can be advanced or retarded individually to vary firing speed. The firing sequence is calculated dynamically by the engine control unit from a variety of inputs, one of which is a sensor that reports the position of the accelerator pedal.

By–wire braking systems tend to be the physically largest of the X–by–wire systems, containing

the most nodes and distributed across the widest area of the car. A typical brake–by–wire system will include a node dedicated to reading the pedal position (and offering haptic feedback to the driver) and a group of braking controller nodes, directly connected to the wheel brakes (and monitoring the speed of the relevant wheels). The prototype braking system mentioned in [16] uses one braking controller node per wheel and introduces a sixth node to act as an overall brake balance controller. This configuration has no obvious redundancy for brake controller node failure, but it would be entirely possible for an implementation of the design to include redundant sensing and acting connections cross-connecting the controller nodes at either end of each axle.

Until recently, EU roadworthiness legislation prevented steer–by–wire systems from being used on public roads, as a mechanical connection between steering wheel and steering gear was required by law. ECE Regulation 79, introduced in April 2005, permits an electronic–only link with no mechanical backup. ECE R79 also permits electronic driver aids (such as automatic lanekeeping or dynamic stability adjustment), on the condition that the driver of the vehicle can override these aids at any time.

Conclusion

In this chapter, I have studied the fields in which the rest of this thesis is set, and attempted to set out definitions and concepts that will be of use later in its reading. I have covered the key concepts involved in fieldbus systems, and presented a brief overview of the field of safety–critical systems, including examples of varying degrees of complexity and criticality.

Chapter 3

Standards, Guidelines and Methodologies

Introduction

In this chapter, the main technical considerations that affect the development of this project will be detailed. First, the basic theory behind network bridging will be investigated, and the properties of communications networks that must be considered when designing such a bridge. Then, the VSI Standards & Guidelines will be considered, and their impact on this work assessed. Finally, a brief investigation of software development methodologies (both past and present) will be presented, considering the benefits and drawbacks of each.

3.1 Technical Considerations

3.1.1 Packet Size and Payload

In most modern networks, data traversing the network is divided into packets. A packet consists of a section of the data being carried (the payload), and various pieces of metadata: payload length, sequence number, and a variable amount of addressing and routing information. This metadata

is usually collected into a header and/or trailer, giving a commonly used abstract packet format of the form “header, payload, trailer”. The original data can be reconstructed by concatenating together the received packet payloads in the order indicated by the sequence number (or other similar metadata).

A useful example can be found in the TCP/IP protocol stack that underpins the modern Internet. The TCP header contains some addressing information (source and destination ports) and a sequence number. The IP header contains the packet length, source and destination addresses. TCP packets are encapsulated in IP packets for transfer, and are routed over the network from source to destination according to the addresses in the IP header. As they arrive, the TCP packets are decapsulated, their payloads are concatenated together in order of sequence number and delivered to the destination port. Most packetisation systems will also include some form of error detection and recovery: in TCP/IP, this is achieved by checksumming at both the TCP packet and IP packet levels, supported by a frame-check sequence at the physical layer (a CRC32, in the case of Ethernet).

In TCP/IP and indeed in most networking systems the lowest–level packet, closest to the physical layer, is normally referred to as a frame. Embedded networks are normally designed and implemented at the frame level: it is unusual (but not unheard of) for higher levels of abstraction to be used, as they can incur significant overhead in encapsulation and unpacking, as well as payload fragmentation and reconstruction. In embedded networking, these additional delays are generally undesirable and should be avoided where possible.

When designing a network bridge, packet size and payload configuration must be considered. If the networks to be bridged have similar packet formats and transfer similar data in their current configuration, then bridging may simply be a case of copying data (in the payload and header) from one packet to another with minimal modification. If, however, the packet format used in one network is substantially different from the format used in the other, either in terms of size or format, bridging data between the networks is substantially more complicated because a direct copy is unlikely to be effective. If a size disparity exists, payloads must be fragmented when passing from the larger–payload network to the smaller, and reassembled when passing in the opposite direction. If the packet format is substantially different, some form of translation will be required – this problem, however, will be addressed later, in section 3.1.5.

3.1.2 Timings

The various types of networks have a wide range of transmission speeds and bandwidths, which can cause problems in bridging systems. If data is to be transferred from one network to another that operates at a significantly different speed, buffering or translation may be required even if the payload and packet formats of both networks are similar. For example, if three packets are received from one network in the time it takes to transmit one packet on the other, it is not possible to bridge the networks directly – some form of translation, compression or truncation is required.

When designing a network bridge, timing issues must be taken into account. If the networks being bridged use similar bitrates, no accounting is necessary. If they use substantially different bitrates, it will be necessary to mitigate the effects this produces by transforming the data as it travels between the two networks.

3.1.3 Latencies

Latency is the length of time that elapses between an action being performed and its effects being felt. In networking, it is commonly taken to mean the length of time between a packet being transmitted by the sender and the same packet being received by the receiver. In this respect, it can be used on event–triggered networks to estimate network load, on unloaded networks to estimate the distance between connected nodes, and so on. These options are only effectively available in event–triggered networks: a time–triggered network responds to signals when instructed to do so by its dispatch schedule, meaning that the latency of a message is often extended by a significant amount. Worse still, depending on the point in the dispatch cycle at which the message is received, the alteration that the dispatcher makes to the message latency varies noticeably.

High network latency has noticeable negative effects on most systems, but the problem is most visible in control systems that utilise an embedded network to transfer data between their nodes. A high latency may appear the same as data loss or, in the worst case, result in positive feedback and oscillation as the control algorithm acts on sensor data that is uselessly out of date.

Any bridging system should aim to minimise latency where possible.

3.1.4 End–to–End Delays

The end–to–end delay present in a network is the total wall–clock time taken for a message to travel from one network node to another and back again (and consequently is normally given for a connection between a pair of nodes rather than for an entire network). As end–to–end delay tends to be intrinsically related to network latency, it should similarly be minimised wherever possible.

3.1.5 Translation / Encapsulation / Association of messages and functionalities

As was stated in section 3.1.1, it is often not possible to directly transfer data from one network to another: some form of processing is required. There are essentially three options in this situation: translation, encapsulation and association. Two of these require the bridge to have advance knowledge of messages in use on both networks (translation and association). The three options are explained briefly below.

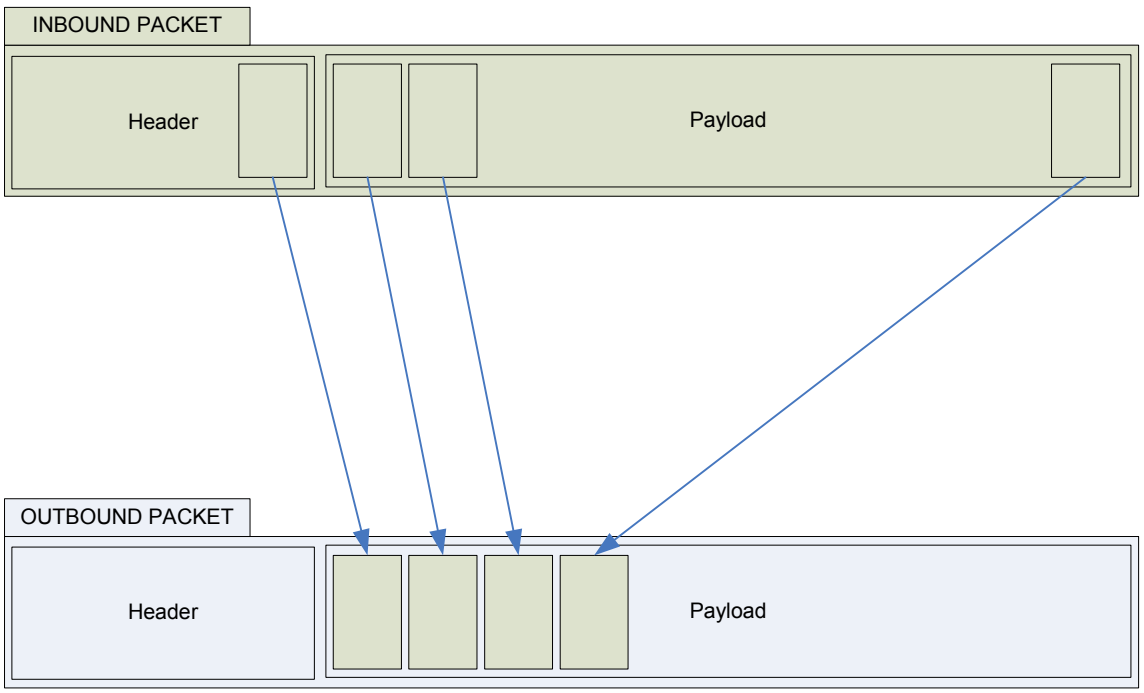

Figure 3.1: Translation of packets

Figure 3.1 shows packet transfer using Translation. The outbound packet is represented as a template and a translation function: data is then taken from the inbound packet, transformed according to the translation function and inserted into the template to produce the outbound packet. The translation function may simply copy data from a given location in the inbound packet to a given

location in the outbound packet. Alternatively, it may perform more complex operations depending on the meaning and content of the packet in question.

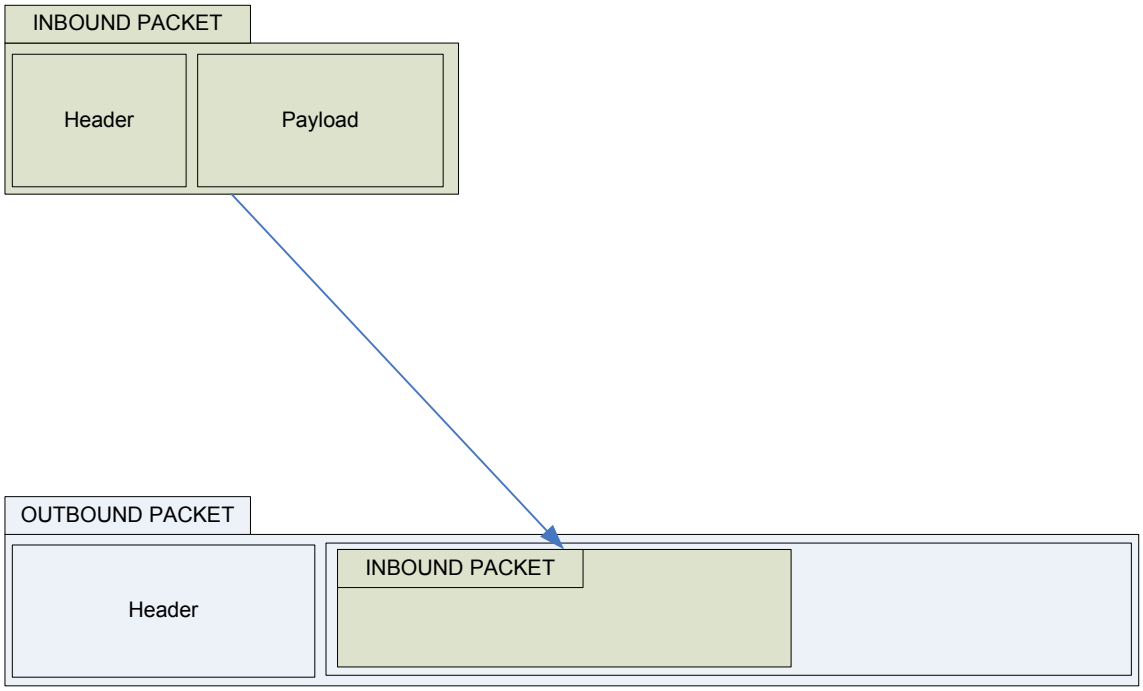

Figure 3.2: Encapsulation of packets

Figure 3.2 shows packet transfer using Encapsulation. The inbound packet is stored in the payload of the outbound packet(s) in its entirety. In the case in the diagram, the inbound packet is small enough that it will fit in the payload of a single outbound packet. If this is not possible, the inbound packet must be divided (fragmented) into data blocks small enough to fit into a single outbound payload, and reassembled into its original form at the other end of the connection. Encapsulation is best used when packets of one format are being tunnelled between two similar networks via a third which uses a different format, as the encapsulation/unpacking process is mostly overhead in cases where the data will not later be transmitted over a network using the original packet format.

Figure 3.3 shows packet transfer using Association. An association table is held in memory, containing a list of relationships between packets. When an inbound packet is received, a table lookup will be performed, and the associated packet sent on the outbound connection. Association is similar in concept to Translation, except that no data is carried over from the inbound to the outbound packet. Association is mainly useful for rapid translation of state messages, which contain little or no payload data (as table lookups are typically much faster than packet translation). Since Association is not particularly efficient in the case of data messages (packets containing a non–trivial

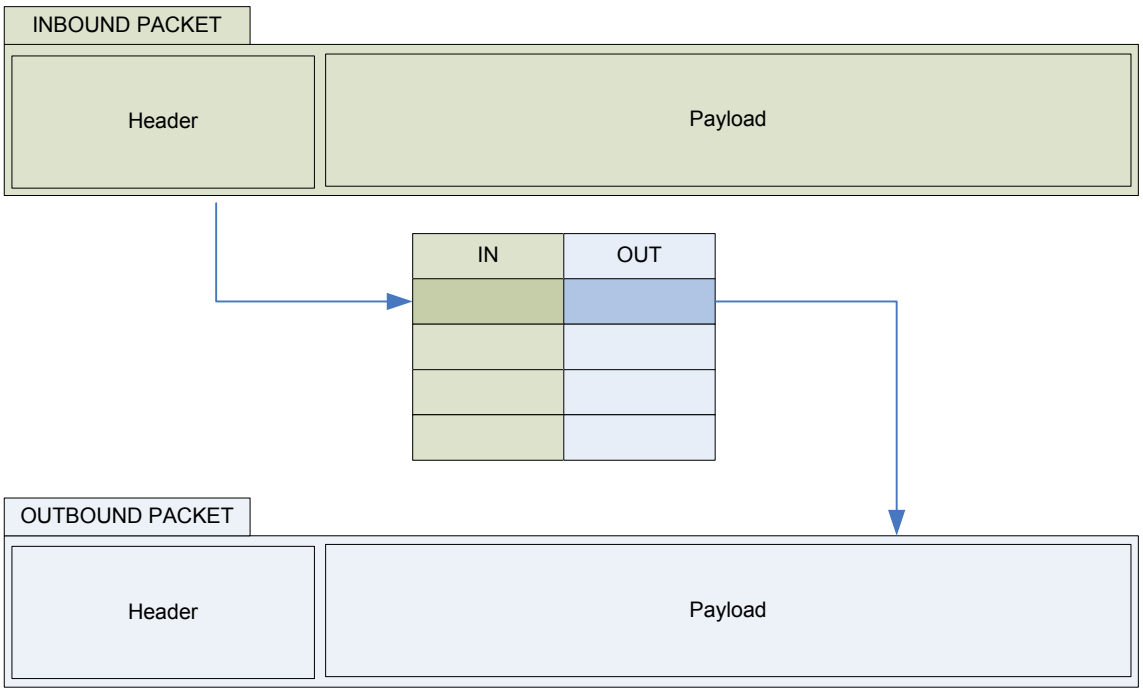

Figure 3.3: Association of packets

amount of payload data), hybrid Translation / Association bridges are quite common. It is worth noting that Translation and Association can be equivalent in certain specific cases - if no data needs to be translated from input to output packet (as might be the case in a simple state broadcast or heartbeat), the Translation function will do nothing and the packet template will be sent unaltered, like an associated message.

When designing a network bridge, it is important to consider how message(s) on the source network will map to message(s) on the destination network. Depending on packet size, bitrate and encoding, as well as other protocol and application specific concerns, it will often be necessary to transform the message data in some way as it moves between networks. The necessary transformations should be considered early and often during the design process, as the way in which data is transformed as it moves between networks is likely to influence the design of the application.

3.1.6 Addressing: Network–Node / –Bridge identification

A given message on a network may be considered either “broadcast” or “point–to–point”. A point–to–point message travels from sender to receiver, quite possibly avoiding all other nodes en route via a switching fabric or star coupler. A broadcast message travels through the entire network

when sent, being received at all nodes and ignored by the nodes that are not configured to receive it.

Ideally, a network bridge should act like a gateway: messages that are sent on one side and which must be received on the other should be transparently moved from one side to the other. This requires that the nodes of the networks be identifiable, and that the bridge can identify to which network a given node belongs. It also requires that the bridge can intercept the message in the first place, and that the sender(s) and receiver(s) of a given message can be identified. However, it is unusual for all of these requirements to be fulfilled in an embedded network. In MilCAN and TTCAN, while nodes can be identified and all messages are broadcast, it is not possible to identify the network segment on which a given node is present, or which nodes act on receipt of a given message (as CAN in general assumes that all messages are broadcast). In FlexRay and TTP, nodes can be identified and the sender and receiver of a message can be discovered, but most messages are point–to–point and must be copied to the bridge if they are to be processed.

As a result of the above problems, it is normally necessary to provide the bridge with a routing and transformation table, either at compile–time or at run–time. Since the bridge is an embedded system with minimal user interaction, little additional flexibility is gained by not compiling the table into the executable, unless frequent modification of the tables is anticipated. The routing and transformation table is an extension of the translation / association tables mentioned in the previous section, which additionally takes into account sender and receiver identity and network segment (where relevant).

3.1.7 Synchronous / Asynchronous Operation

All time–triggered (and some event–triggered) networks have the concept of cluster time, which is related to the node timebase. FlexRay and TTP maintain a global timebase by consensus agreement between nodes, while MilCAN and TTCAN use a Master Node which is responsible for maintaining the global clock and broadcasting periodic “ticks” to which the other nodes synchronise. Time–triggered networks, by their nature, require a global timebase to which their transmission schedule can be synchronised. Event–triggered networks, by contrast, are not normally synchronised to a global timebase, since the arrival time of a message does not affect the interpretation of its content.

Bridging synchronous (time–triggered) networks normally requires that the bridge waits until the appropriate point in the next time cycle before transmitting a bridged message, since keeping the inbound and outbound interfaces synchronised tends to be quite complicated. This, in turn, tends to lead to a minimum bridging delay of one outbound–network time–cycle, which must be taken into account when designing a bridged network system. Since asynchronous (event–triggered) networks need not wait for the appropriate time–slot before transmission, the bridge imposes only a minimal latency on messages bridged between them, the time required to physically move the message from one interface to the other (which in the case of cut–through bridges can be near zero [17]). Bridging from a synchronous to an asynchronous network can have mixed results: messages entering the synchronous network are delayed until the next safe injection point, while messages entering the asynchronous network can be injected immediately (subject to network load).

3.1.8 Absolute Time

In order to benchmark a bridge, it is necessary to gather accurate timing and latency information for messages being transported through it. This is best done by maintaining a unified view of real–world time across each network segment – absolute time, as opposed to elapsed time since an event or a count of ticks used to synchronise the system. If the sending and receiving timestamps for a frame (recorded against this absolute view of time) are compared, frame latency measurements can be made to a high degree of accuracy. Maintaining a stable view of real–world time can be difficult when attempting to benchmark a distributed embedded system, and is best done by an external system that is dedicated to debugging and analysis use. The debugging system can then maintain an absolute clock, and appropriate traps in the send and receive code on each node of the distributed system can be set to inform the debugging system as messages travel across the network. Since the internal time source of the debugging system is assumed to be accurate and it is dedicated to the task, accurate timing records for each test sequence should result.

It should be noted that this unified view of real–world time is to be maintained for the purposes of debugging, and for that only. Some embedded networks have a concept of absolute time as a component of their protocol: both FlexRay and TTP maintain an absolute timestamp as a part of their time-driven multiple access system. While an individual network segment may have a set view of absolute time, another segment may have a different view of absolute time (due to being started a little earlier or later, or to having a different internal clock speed). Bridging messages

that interfere with the segment's view of absolute time from one segment to another risks causing serious problems, up to and including TDMA violation, and must be avoided.

3.1.9 Technology–Independent Bridging

Where possible, a network bridge should not depend on any underlying technology, such as a particular networking configuration or operating system, being in use. As in most domains of engineering, a generic bridging system is normally of more value than a technology–dependent one, because it contains no implicit assumptions about the networks it bridges. Consequently, it can be adapted to interface with new networking technologies with a minimum of effort.

In practice, the goals of efficient operation (to which all useful products aspire) and technology independence tend to interfere with each other. In order to be as efficient as possible, the bridge is likely to require low–level device access. In order to maximise technology independence, the design of the bridge should restrict low–level access to a clearly defined set of operations common to all possible platforms. An implementation must, therefore, balance these two conflicting goals. The normal solution to the conflict is to define an abstraction layer as close to the hardware as possible, then write hardware drivers beneath the layer that are technology specific, while keeping the majority of the application (above the layer) technology independent.

3.1.10 Commands that generate Information

In previous sections, it has been assumed that all messages on the network are independent. In some cases, this is true (event broadcasts, system management messages and imperative commands). In other situations, a response is required – an acknowledgement for important messages, or the value read from a requested sensor, for example. A network bridge must bridge the response if it bridges the request, and should not bridge the response if it does not bridge the request.

3.1.11 Message Filtration

The simplest imaginable network bridge accepts all messages on each connected network and emits them on all others. This is a poor solution for several reasons. First, any network management frames on a given network (time synchronisation, node reconfiguration and TDMA modifi-

cation, to name a few examples) should not be bridged outside their source network, since they are not relevant to devices outside that network and may interfere with similar frames on destination networks. Furthermore, many messages would be bridged that need not be, because their sender(s) and receiver(s) are on the same network segment. This results in an unacceptable level of bus traffic on all the attached networks, with potentially unfortunate consequences including a significant increase in bus latency, an increased incidence of message loss or corruption, and message aliasing due to the same message identifier being used with different payloads on both networks.

The normal solution to this problem is message filtration, which requires the bridge to be at least minimally selective about the messages it forwards. Messages which have a source on one network and a destination on another (where the two networks in question are connected by the bridge) should be transferred, while messages that have both source and destination on the same network should not. This can be implemented either as a translation and association system, as detailed in section 3.1.5 above, or using a learning approach as indicated in [9]. In a broadcast network, the problem tends to be more complex, as a given message may contain only a partial list of its destination addresses, or even none at all. While a translation and association implementation requires total knowledge of the network and its communication flows, it handles broadcast messages better since their destinations can be written into the lookup table. A learning bridge requires a message to produce return data (such as an acknowledgement, or some other obviously related message) in order to determine which nodes receive it.

3.2 VSI Standards and Guidelines

The VSI Standards & Guidelines (R.M. Connor, July 2009, see [1]) set out a recommended framework for future development in the field of vehicle platform electronics in the UK Ministry of Defence. The "Standards" in the document are a set of open standards for networking, data compression, hardware and software that are recommended for use in new vehicles, as well as in upgrades to existing vehicles. The "Guidelines" are a set of procedures that are to be followed when upgrading vehicle platform electronics and networking.

Current military vehicles were not designed with the likelihood of in–service modification in mind. In modern theatres, however, the ability to adapt vehicles quickly and efficiently to handle changes in environmental or engagement conditions is essential to maintain an effective battlefield pres-

ence. The VSI Standards & Guidelines, then, are written with two particular goals in mind. First, they lay out suggested technologies and techniques that should be applied to the design of future vehicles to ensure that in–field updates and modifications can be carried out as quickly and simply as possible. Second, the specified technologies and techniques can also be applied to upgrades to current vehicles where practical, ensuring both that the upgraded subsystem can be used in future vehicles with a minimum of modification and that multiple upgrades to the same vehicle have a maximal chance of interoperating effectively. The overall effect of applying the Guidelines should be that vehicles become more modular and more intelligent, leading to simpler training and maintenance (due to the commonality of components and interfaces between multiple vehicles and vehicle types) and allowing subsystems to be transferred between vehicles without significant redesign.

3.2.1 Reduced Life–Cycle Costs

Where possible, a reduction in the manufacturing and maintenance costs associated with a given vehicle type is clearly desirable. The introduction of new technologies and subsystems to a design during its lifetime does not make it easy to achieve this goal. However, the use of standard interfaces and protocols means that component systems can be developed and tested independently of the vehicle, and their performance evaluated in abstracted testbeds. Although testing on the target platform will be necessary eventually, integration costs can be substantially reduced by the use of a testbed for initial development.

In addition to the above benefits, new and more intelligent technologies raise the possibility of integrated self–testing functionality being built in during development. These functions are typically divided into Built–In Test (BIT) and Health and Usage Monitoring Systems (HUMS). A built–in test is designed to be invoked by the user or by a supervising system periodically, to ensure correct operation of the unit under test and to determine its current operating parameters. A HUMS, by contrast, is designed to continually collect data about the operation of the unit and report back to the service depot periodically. Health and Usage Monitoring Systems allow the state and operational parameters of a fleet of vehicles to be assessed over time, and trends or marginal behaviour to be examined. Based on this data, it is possible to streamline maintenance schedules, perform preventative maintenance and arrange for spare parts to be delivered when they will be needed, rather than maintaining a large stock of potentially un–necessary parts.

The ability to pass messages between heterogeneous networks makes the implemention of self–test and HUMS much simpler, as the relevant data can be carried over the existing networks rather than requiring a test–systems network to be created.

3.2.2 Modularity

The VSI Standards & Guidelines indicate that modularity of design is desirable, both on the equipment and functional levels. Equipment modularity indicates that the equipment can be recombined to satisfy new requirements, greatly simplifying after–market modification and component reuse. Functional modularity indicates that functions can be combined within modules, either to reduce module count or to create new devices that fulfill specific requirements. The use of standard interfaces and protocols, both between devices and (where appropriate) within devices will aid the design of modular systems. This is because doing so provides a common platform on which the developer can build, meaning both that effort in building communications libraries need not be repeated, and also that when it becomes necessary to integrate a set of modules into a system, the work is already half done.

The ability to pass messages between heterogeneous networks makes the design of modular systems easier, since it removes the requirement that two interacting modules be connected to the same network.

3.2.3 Distributed Architecture

Once a minimum level of modularity has been introduced into the system, it becomes possible to distribute subsystems across physical space, rather than requiring that all components be closely colocated. The distribution of system function across multiple nodes has beneficial effects. A distributed system can achieve higher survivability, as its component nodes are typically not colocated, meaning that structural damage to the vehicle will tend to damage fewer of them. Fewer damaged nodes raises the likelihood that the system will continue to function, or at least be able to fall back to a safe state, rather than being disabled completely. In addition, the fact that data passes between nodes over a communications network means that other systems, either developed in parallel with the system in question or added on later, can read the same data from the network and act on it. This raises the possibility of multiple modular systems making use of the same input data,

thus reducing the component count and complexity of each system. It should be noted that the use of a distributed architecture does bring with it certain potential disadvantages, not least of which is a vulnerability to electro–magnetic interference. These disadvantages should be addressed by other components of the system, such as physical–layer redundancy, shielding and error recovery protocols. In turn, however, a fieldbus network containing multiple processing nodes may be able to recover from such interference by use of error–correction coding or temporal redundancy, in a way that an earlier non–fieldbus–based design may not be able to, as all of its messages are simple raw data values and states.

The ability to pass messages between heterogeneous networks allows data to be passed from one network segment to another, meaning that a subsystem can now be distributed across multiple networks instead of merely across multiple nodes.

3.2.4 Rapid Modification

Modularity and Distributed Architecture naturally give rise to an ease of modification in the finished system. This can be achieved either through reprogramming, redesign or replacement of existing nodes, or by adding new nodes that comply with the modular interface specification laid out for the system. Such modifications can potentially be made in the field, by the integration of an upgrade pack produced as a result of a UOR (Urgent Operational Requirement), without a need for any substantial re–engineering.

The VSI Standards & Guidelines indicate that rapid modification is a highly desirable trait in new vetronics systems and subsystems.

3.2.5 NEC

The NEC (Network Enabled Capability) programme is a relatively recent (November 2004) initiative within the UK MoD. Whereas current vetronics networks distribute data through the vehicle, an NEC–equipped vehicle can distribute data through the entire squadron and to local allied vehicles and infantry. The goal of the NEC programme is that data interchange between vehicles, squadrons and infantry should be as quick and simple as data interchange between computers in a network. This would not only simplify data distribution through the military command hierarchy,

Domain	Standard
Low Speed Serial Buses	Def Stan 00–18 (MIL-STD 1553) ISO 11898 (CAN) MilCAN TTP/C FlexRay
Point To Point Serial Interfaces	RS232 RS423 RS422
Programming Languages	ISO8652 (Ada) ISO9899 (C) MISRA-C ISO144882 (C++)
Middleware	Minimum CORBA RT-CORBA OMG-DDS

Table 3.1: A subset of the Recommended Standards list, from page 42 of [1]

but also enable peer–to–peer interchange, allowing units in the same immediate area to dynamically coordinate and cooperate without the delay inherent in said command hierarchy.

Since the NEC network is unlikely to use one of the network protocols specified in the VSI Standards & Guidelines (for a variety of reasons, including security, resistance to jamming and compatibility with existing systems), the ability to pass messages between the vetronics network and the NEC network is essential to future development.

3.2.6 Recommended Standards (summary)

The VSI Standards & Guidelines set out a list of recommended standards for use in certain situations. The purpose of this list is to guide the development process towards a subset of the available technologies, in the hope that different systems developed using the guidelines will have a greater chance of being based on the same technologies, and thus be much simpler to interconnect. A subset of the Recommended Standards list is given in Table 3.1.

The list of recommended standards informs platform choices in the Technology Mapping section of this thesis (MilCAN in section 5.3 and FlexRay in section 5.4).

3.2.7 Middleware

Generally, data that passes between components of a given system or platform will be constrained within the boundaries of that system or platform. In order for systems and platforms to interoperate (to meet the goals listed above, notably including NEC), it is necessary for some data to move beyond the boundaries of its originating system. From a more general viewpoint, it is desirable that *all* platform data be made accessible across system boundaries, allowing data aggregation and comparison across multiple deployed platforms. There are several ways to fulfil this goal, ranging from out–of–band debuggers and mandated interrogation interfaces through to intelligent middleware, either as a part of the network or as a part of each node.

The VSI Standards & Guidelines recommend that a middleware approach be taken in the design of all new vehicles and in cases of major modifications to existing vehicles. Several existing standards are recommended for new vehicles, including Realtime–CORBA, Minimum CORBA (a restricted subset of the CORBA specification for use in embedded systems) and the OMG Data–Distribution Service. Any middleware implementation can be generalised to one of two forms: network–based middleware and distributed node–based middleware.

A node–based middleware system requires modification to the software structure of the nodes: either the application should link a library capable of performing middleware functions, or else the operating system should be modified to allow remote queries to retrieve data. The Object Management Group's DDS (Data Distribution Service) is an example of a middleware system that exists as a part of the nodes of the system.

A network–based middleware system can be introduced without altering the nodes of the system, so long as the required data is placed on to the network by one or more nodes. This is done by adding a device to the network that monitors communications, gathers data from passing messages and makes this data available to external queries. The VSI Bridge (an intelligent network–agnostic message router developed in–house at Sussex University [2]) is an example of a middleware system that exists as a part of the network, rather than of the nodes. Note that the network–based system requires either that the source network be a broadcast network or that messages be copied to it, as it cannot bridge messages it does not read.

3.3 Development Methodologies

When developing a product, there are a variety of different developmental methodologies one can follow, to keep the work on schedule and ensure that all the necessary tasks are carried out in the right order. These methodologies are examined below, first with a brief historical perspective, then with an examination of current techniques. Which methodology is selected for a given application depends on a variety of conditions, not least of which the domain in which the product will serve. If the product is targetted at the domain of the desktop software, it may be acceptable to ship once most bugs have been removed from the product, and to fix the last few in the Maintenance phase. In the domain of embedded software, where once a product is deployed it will be much more difficult to maintain, a more stringent development methodology is typically called for. In a safety–critical system, even more cautious development methods are required, as shipping such products with faults risks compromising their safety–criticality and implies an immediate risk to human life. Cost is another condition that may drive selection: simpler methods are inherently cheaper to implement, but this must be balanced against their inherently higher risk.

3.3.1 Code and Fix

Whether the "code and fix" approach is in fact a software development methodology at all is a matter for debate. It is likely to be the oldest method in use today, originating as it does in the very early days of computer programming, but given its simplicity and lack of rigour, it may in fact be better considered an example of development **without** methodology.

Boehm (1988 [18]) describes the Code And Fix method very simply, as a two–stage process:

1. Write some code.
2. Fix the problems in the code.

Requirements, Design and Testing, stages which modern software engineering generally considers essential to the development of successful non–trivial software products, are all ignored by the Code And Fix method in favour of writing code as soon as possible. Parallels can easily be drawn to some modern development practices (eXtreme Programming and its focus on rapid application development is particularly similar), with the exception that in the Code and Fix method, the user's

needs are not typically considered. As a result of this code–before–all–else approach, products developed using the Code and Fix method are typically poorly designed, unsuited to the user's requirements and difficult to modify.

For the above reasons, the Code–and–Fix methodology should be considered deprecated, and it is rare to find a modern software product that has been developed by using it.

3.3.2 The Waterfall Model

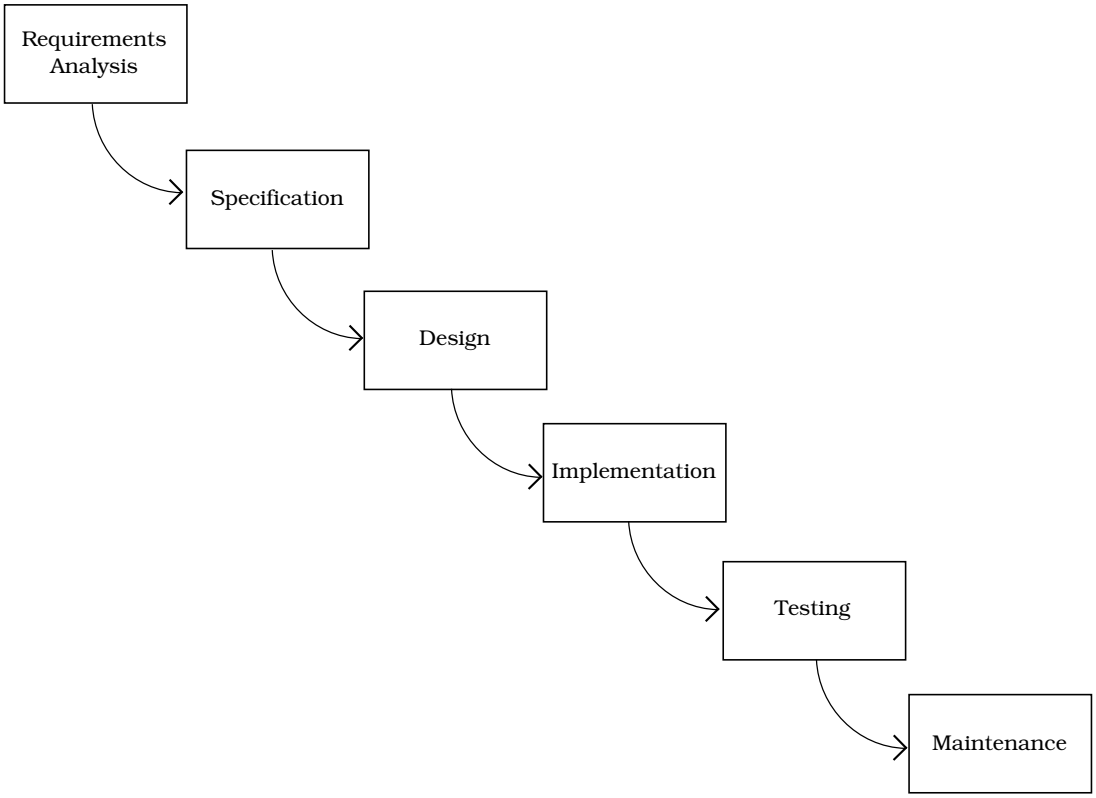

Figure 3.4: The Waterfall Model

The most intuitively obvious methodology, and the oldest if the Code and Fix method is discarded, is the Waterfall Model [19]. The Waterfall Model, shown in Figure 3.4, describes the development process as a series of subprocesses that cascade on to the next subprocess upon completion. The subprocesses are:

- **Requirements** In the Requirements Specification stage, the problem domain is explored and a set of requirements are codified, to which an acceptable product will conform. This specification guides development and testing, by acting as a description of what the product

must do, as opposed to how it does it.

- **Design** Where the Requirements stage consists of specifying what the product must do, in the Design stage the methods by which the product fulfils those requirements are defined. In the context of a software product, the design will include a list of algorithms and data structures, as well as data–flow diagrams and (where appropriate) file formats. In the context of a hardware product, the design will include specification of which materials, components and mechanisms will be used to construct the product, as well as how and where they will be fastened.

- **Implementation** In the Implementation stage, the product is produced according to the design. Parts are cut and assembled, software is written and compiled. If the design is well-written and comprehensive, the implementation stage should proceed without any significant setbacks or alteration to the design.

- **Verification (or Testing)** During the Testing stage, the (now manufactured) product is checked against the design and requirements specifications, to ensure that it is correct. It is first checked against the design, to ensure that it has been produced correctly, then against the requirements specification to ensure that it does what the customer wants. If the product passes testing, it can be released.

- **Maintenance** The Maintenance stage is often overlooked in software design methodologies, but is at least as important as any other stage. The maintenance stage consists of supporting the product while it is in use by the end–user, providing updates and bugfixes and eventually disposal (if necessary).

The process example given here is a generic, abstracted form that encompasses the common variations of the Waterfall Model: some variations expand the "Design" stage into System, Architecture and Component Design stages, others insert a prototyping stage into the middle of the Design stage to test assumptions made about the environment and performance of the product.

The intention is that each phase should be completed before the next one begins: the design is written based on a complete set of requirements, the product is produced from a feature–complete design, testing is done on the finished product and the product is released (and maintained) only after testing is complete. However, the waterfall model has its drawbacks.

The most significant revisions to the Waterfall Model can be found in (Royce, 1970 [19]), a paper which was written explicitly as a criticism of the Waterfall Model as described here. Royce argues that since the testing phase is at the end of an explicitly feed–forward process, it quickly becomes very expensive in terms of time and money to correct the product if it does not fulfil the specification correctly (as substantial parts of the design and codebase will need to be rewritten). In order to address this problem, Royce suggests that a set of retrograde paths be added to the model, allowing the development process to proceed back to an earlier stage in case of test failure, instead of returning all the way to the beginning. Additionally, Royce recommends a two–stage development process in which an initial prototype is designed, implemented and tested, then used to adjust requirements to take account of the real world. This method, it is suggested, should result in a much more realistic set of environmental requirements, as well as making conflicting requirements and poor design apparent before the final product is produced (at a significantly higher level of quality than the prototype).

3.3.3 The Spiral Model

The Spiral Model [18] is an evolution of the waterfall model designed to address these problems. It is primarily designed for complex, expensive projects, as it introduces significant overhead into the development process. In the spiral model, shown in figure 3.5, the waterfall model is turned in upon itself so that the testing–and–maintenance phases (now combined) feed into the requirements phase.

The **Planning** phase consists of making preparations for the next phase of development. It can be viewed as the Design Phase of the spiral model, such as one exists, because it is here that decisions about overall architecture and module layout tend to be made.

The **Determine Objectives** stage involves examining the design and selecting objectives whose achievement will indicate successful completion of the iteration.

Having determined the completion objectives for a given iteration, the penultimate stage of the cycle is to **Identify Risks** to the project. The definition of Risk tends to vary between projects, but will typically include cost, complexity and availability of equipment or personnel.

The **Development** phase of the Spiral model encompasses almost the entirety of the Waterfall Model, in that its final iteration is essentially a pass through the Waterfall (from Design to Imple-

mentation and Testing). In the Development phase, the design team is responsible for expanding knowledge from previous phases into a product, whether that product is a requirements specification, a prototype or a finished product.

While the spiral model addresses most of the criticisms levelled against the waterfall model, it in turn has some shortcomings, foremost among which is the fact that necessary changes cannot be made as the prototype is developed, only after it has been finished and tested and the requirements for the next cycle are being written. In some cases, this momentum of process is advantageous: since it is not possible to alter the product within an iteration, a failed test indicates a flaw in the iteration, and it will be possible to trace back from the test to the flaw. In other cases, it can be disadvantageous: if a mistake is spotted during the iteration, it cannot be corrected (since to do so would invalidate the test results) and must be allowed to exist uncompensated throughout the iteration so that the test results correspond to the design input for the iteration.

The Spiral Model is more responsive to necessary design changes than the Waterfall Model, in the sense that it is possible to make such changes at all while still adhering to the methodology. However, the requirement to complete the cycle before changes are made means that there is still significant latency in the model, particularly in large projects (to which the Spiral Model is often applied).

3.3.4 The V–Model

The V–Model, shown in the next figure[20], is an alternative evolution of the waterfall model that takes the concept of testing and test–driven development into account.

As the figures show, the V–Model is shaped as it sounds, with the design stages on the left–hand side (leading down to the implementation process) and the verification stages on the right–hand side leading back up toward eventual release, and with additional arcs connecting across the V at each level.

The model is drawn in this way because the testing stages that follow implementation are each directly related to one of the design stages that preceded it, as indicated by the undirected arcs that link stages between the arms. For example, Specification is linked to System Testing because any testing that investigates the properties of the complete system should test against the specification to ensure that the original requirements (from which the specification was derived) have been

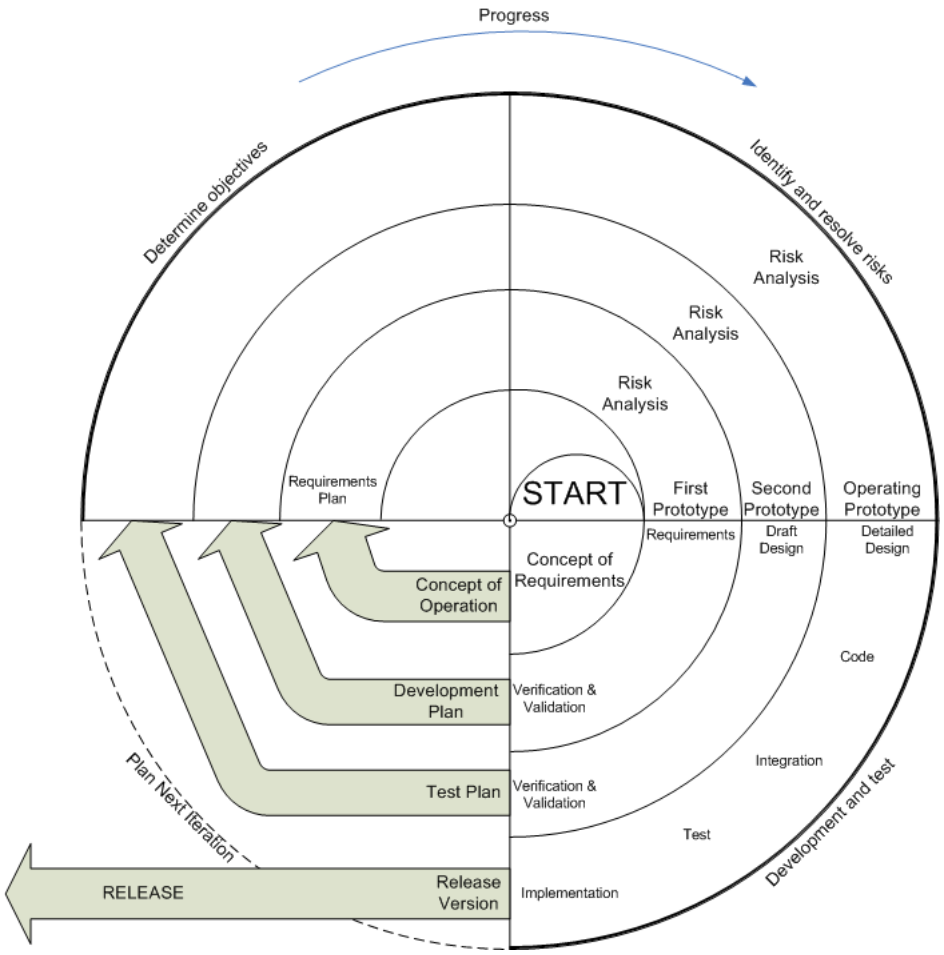

Figure 3.5: The Spiral Model, based on the diagram in [18]

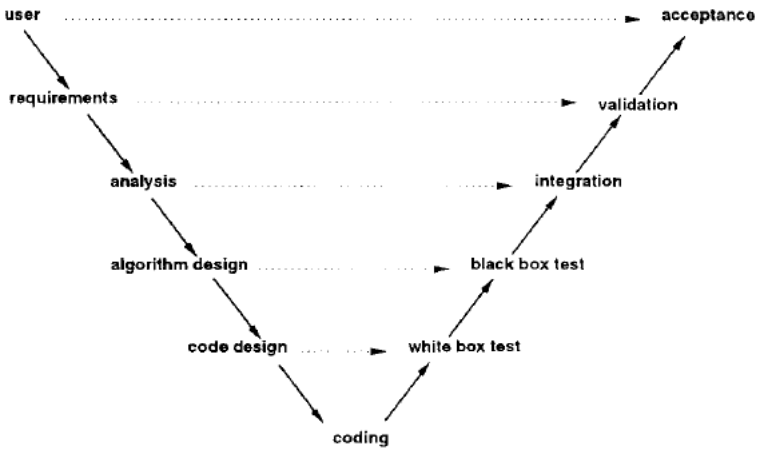

Figure 3.6: The V–Model from Hesselink, 1995 [20]

fulfilled correctly. Each design stage results in a set of design specifications for the product, to which it will be implemented. However, just as it is possible to transform the design specification into a product (by implementation), it is also possible to transform the design specification into a

testing specification, which details the tests a product must pass in order to conform to the design specification. The process of producing this testing specification and testing the product against it is known as *Verification*, normally defined as "*The process of determining whether or not the products of a given phase of the software development cycle fulfil the requirements established during the preceding phase*"[21]. Verification should be done at the end of each stage on the left (Design) arm of the V–Model, to ensure that the design process remains on course.

Once the product has been implemented, it must be tested to ensure that it behaves as the designers intended. This is most simply done by using the testing specification developed from the corresponding design stage (following the arcs linking the arms of the model), and is known as *Validation* (defined as "*The process of evaluating software at the end of the software development process to ensure compliance with software requirements.*"[21]). Validation should be done at a defined point within each stage on the right (Testing) arm of the V–Model.

If a given test is passed, the development process should move on to the next stage. If a verification test is failed, the failed stage should be restarted, as it does not comply with the requirements set by the previous stage, and will cause the development process to result in an incorrect or incomplete product if allowed to continue. If a validation test is failed, by way of contrast, more work will typically be involved as it indicates a more serious problem. In this case, the development process should follow the arc back across from the Testing arm to the corresponding stage on the Design arm, and repeat the loop back to the failed stage, allowing the results of the failed test to influence modifications to the design and implementation stages as appropriate to prevent the same failure from recurring.

The various labels shown in figures 3.6 are defined below:

- **Requirements Analysis** The customer's requirements are explored, with the intention of producing a list of specific, testable requirements that must be fulfilled for a solution to be considered acceptable. The requirements analysis tends to be quite complex, as it is not always possible to aggregate cases – often, similar requirements must be considered separately because aggregating them together would compromise precision in one or more cases.

- **System Design** The system design details the black–box properties of the product: how it will interact with its environment and what its external interfaces are. It is unusual for any

internal properties of the system to be considered at this stage, although it is sometimes necessary to specify features such as processor family or architecture at this stage.

- **Architecture Design** The architecture design details the white–box properties of the product: how features will be implemented, which components will be used and how the internal structure will be laid out. This includes internal interface specifications, but typically does not lay out the precise configuration of parts that will implement any component. Component, in this context, refers to either the smallest elements to which the design can be broken down, or a subassembly that will be manufactured as one piece and then inserted into the architecture.

- **Component Design** The component design details exactly how a given component will be implemented. It defines which parts and/or libraries will be used, how they will be connected, the algorithms to be implemented on any processors and all other relevant details. When the component design is complete, it should be possible to make minimal modifications to it to produce a physical layout (in the case of hardware) or a functional skeleton (in the case of software) for the component, ready to be implemented.

- **Implementation** During the implementation phase, the individual components are assembled and/or coded according to the unit design for that component.

- **Component Testing** Freshly implemented components are tested against their component design. Success here is defined as correctly fulfilling the component design: failure implies that the implemented component does not behave according to its design. Note that it is possible for a component to pass component testing without being suited for use in the final system if the component design is incomplete or incorrect. A testing failure during unit testing can indicate a flaw in the component design, or in the architecture design, or simply that the component was implemented poorly and must be modified.

- **Integration Testing** At this point, the components have passed unit testing, and are (in theory) complete and ready for use. They are now assembled together, and their interactions tested against the architecture design. The system may not have been fully assembled at this stage: it is necessary to test every possible interaction, but it may not be desirable to bring the entire system together at this point (for example, if the subsystem is being manufactured by a subcontractor, it may not be desirable or even possible to bring together all the subcontracted

subsystems to integration–test one of them). If a group of subsystems are to be tested without the entire product being assembled, a testbench of some form will be required to emulate the systems that are not present. A testing failure during integration testing implies that the architecture design was incorrect, which may be a self–contained problem or may imply in turn that the system or component design is flawed.

- **System Testing** The components and their direct interactions have now been tested, and they can finally be brought together in a complete system. This stage will typically be performed by the system integrator, gathering together any and all subcontracted components to do so. The completed system is tested against the system design. Success indicates that the assembled components implement the system design. Failure indicates that one or more of the system design requirements has not been addressed correctly.

- **Acceptance Testing** All testing stages prior to acceptance testing have been verification against the design (checking that the output of a given stage does what the design says it should do). The acceptance testing stage is the only stage at which the product is validated (checking that the product fulfils the specified customer requirements). If the acceptance tests are passed, the product meets the customer's specified requirements and can be released. If the acceptance tests are failed, then the system design did not correctly capture the customer's requirements. In this case, the system design must be modified and the modification propagated through the entire V–Model so that the modified product can eventually be revalidated. This process is typically very costly and time–consuming, and should be avoided by the production of a correct, complete system design whenever possible.

The ability to cross back to an earlier stage during prototyping means that the developers need not wait an entire iteration to test the results of a small modification to the design, but can see the results immediately by modifying the prototype. This speed of feedback, together with the rigour implied by a tightly–coupled verification and validation plan, makes the V–Model one of the most popular development methodologies currently used in the area of embedded systems.

The V–Model was eventually chosen as a basis for this project because every stage includes an iterative process and emphasises testing: this gives the methodology flexibility and the capacity for modification without the intricacies and momentum of process displayed by the spiral model.

Conclusion

In this chapter, the main technical considerations affecting the design and implementation of a network bridge linking heterogeneous networks have been listed, as well as considerations introduced by the United Kingdom's MoD, the funding agency for this work. Where possible, it has been indicated how this work will address these considerations, indications that will be discussed further in the next chapter. Also, the history of software development methodologies has been briefly considered, taking into account what might be considered "families" of methodologies, and the essentials of the process of verification and validation have been outlined.

Chapter 4

V&V Testbed - Abstract Design

4.1 Introduction

In this chapter, the high–level design for the systems that compose the integration testbed will be laid out, as well as the triggering and monitoring framework that will be used to collect results from the validation tests performed on it. An overall design (including the high–level application) will be described first, considering the high–level requirements that led to this design, then the triggering and network–bridging systems will be examined in more detail.

4.2 Initial Considerations

Before it is possible to lay out a system design, certain things must be considered. It is important to ensure that the design is independent of underlying technologies, to ensure that it can easily be ported to other platforms and networks if and when the need arises. Equally, it is wise to consider the use of modelling tools as a developmental aid: used correctly, they can highlight process failures and aid early prototyping without the need to develop complete subsystems for testing purposes. Finally, it is useful to consider using one of the developmental methodologies

discussed in section 3.3 to guide the design process, as without such guidance the project is likely to default to a Code–And–Fix approach, and end in failure.

These considerations will be studied in greater detail below.

4.2.1 Technology Independence

As was noted in the Theoretical Background (section 3.1.9), independence from any particular technology should be considered a goal. As a result, it is important that the design should not be reliant on the presence of a particular hardware platform, interface or networking system, so far as is possible.

As previously noted, true technology independence is difficult to achieve in practice. Where technology independence (a lack of reliance on any particular platform or interface) is not practically possible, many of its benefits can still be realised through the use of middleware instead. This middleware component is placed between the testbed application and its implementing technology to some extent, abstracting detail from the technology and allowing the application to work in general, rather than specific terms. The introduction of a middleware component produces a necessary dependency on said component, but if the middleware technology is well chosen, this will not significantly restrict the range of platforms, interfaces and networking systems the design can support.

The networks under test require some form of bridge device to interconnect them: for maximum flexibility and configurability a device based on the VSI Bridge was used (see Charchalakis, 2005 [2]). The VSI Bridge is a middleware application using a ruggedised x86 PC as a hardware platform, and can communicate with other VSI Bridges over a 100MBps Ethernet network. The VSI Bridge then loads customised interface modules for each vetronic networking standard present on the network under test (using additional hardware to mediate the interface where necessary). These interface modules are lightweight and communicate with the Bridge through a standardised API, thus providing an abstraction layer over protocols and simplifying the task of bridging additional protocols in the future.

4.2.2 Methodologies

A hybrid model was eventually created, based on the spiral model and the V–Model, and used as a guideline through the project. The hybrid was created since it is better suited to our needs: if for example there is a problem in the “Integration Testing” section, the system integrator will go back and test the system against the “Architectural Design” – hence verifying it. Depending on the seriousness of the issue, and how much a change will affect the rest of the system, he/she could choose to go back to “System Design” or proceed and recheck the “Component Design”.

4.2.3 Testbed Abstract Design (SysML)

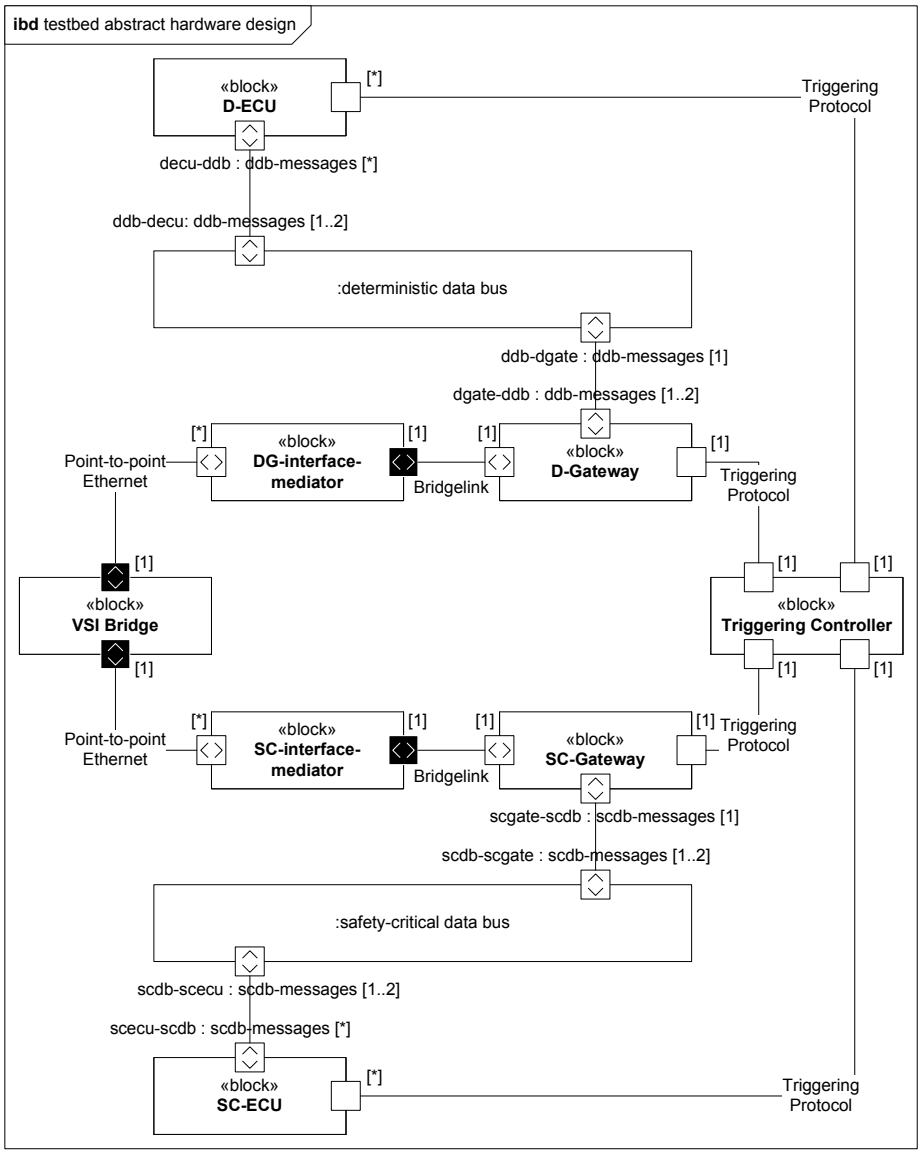

Figure 4.1: Internal Block Diagram showing the structure of the testbed

Figure 4.1 contains a SysML depiction of the overall structure of the testbed, at a reasonable level of abstraction. The diagram can be divided into four sections: centre, right, top and bottom. The central section is the section concerned with bridging message data between networks (see section 4.5). It is composed of a VSI Bridge and a gateway on each destination network, with an interface mediator to connect the two if necessary. The endpoint multiplicities in this section of the diagram indicate that each gateway is connected to a single bridge, but that a bridge may have multiple gateways (and, through them, networks) of either type attached to it. The section forming the right–hand side of the diagram is the section concerned with triggering (see section 4.4). A single Triggering Controller is linked to every node in the network, as can again be seen from the multiplicities. The top and bottom sections of the diagram are the different network types connected to the bridging and triggering systems. The top network is a generic deterministic network segment: any number of such segments can be connected to the bridge and triggering controller, up until one runs out of available ports. The bottom network is a generic safety–critical network segment, and subject to the same restrictions as the deterministic segment above (see 4.3 for a more detailed discussion of segment configuration). Each network is composed of one gateway ECU (which is responsible for forwarding message data between the network of which it is a part and the VSI Bridge) and one or more other ECUs (which will be running a minimal application for testing purposes). These sections are discussed in more detail below.

4.3 Application Networks

4.3.1 Hardware Design

Since this thesis is primarily concerned with producing a testbed for the integration of deterministic and safety–critical networks, at least one of each network is required. Thus, the simplest configuration of testbed consists of a deterministic network and a safety–critical network, interconnected by a single bridge, as was shown in the SysML diagram above.

As can be seen in figure 4.1, a single node on each network is set to act as a network gateway. This gateway is responsible for forwarding messages on the network to the VSI Bridge, and for injecting messages from the Bridge into the network. Since the VSI Bridge expects all messages to be transported as Ethernet frames, in many cases some form of interface mediation device is

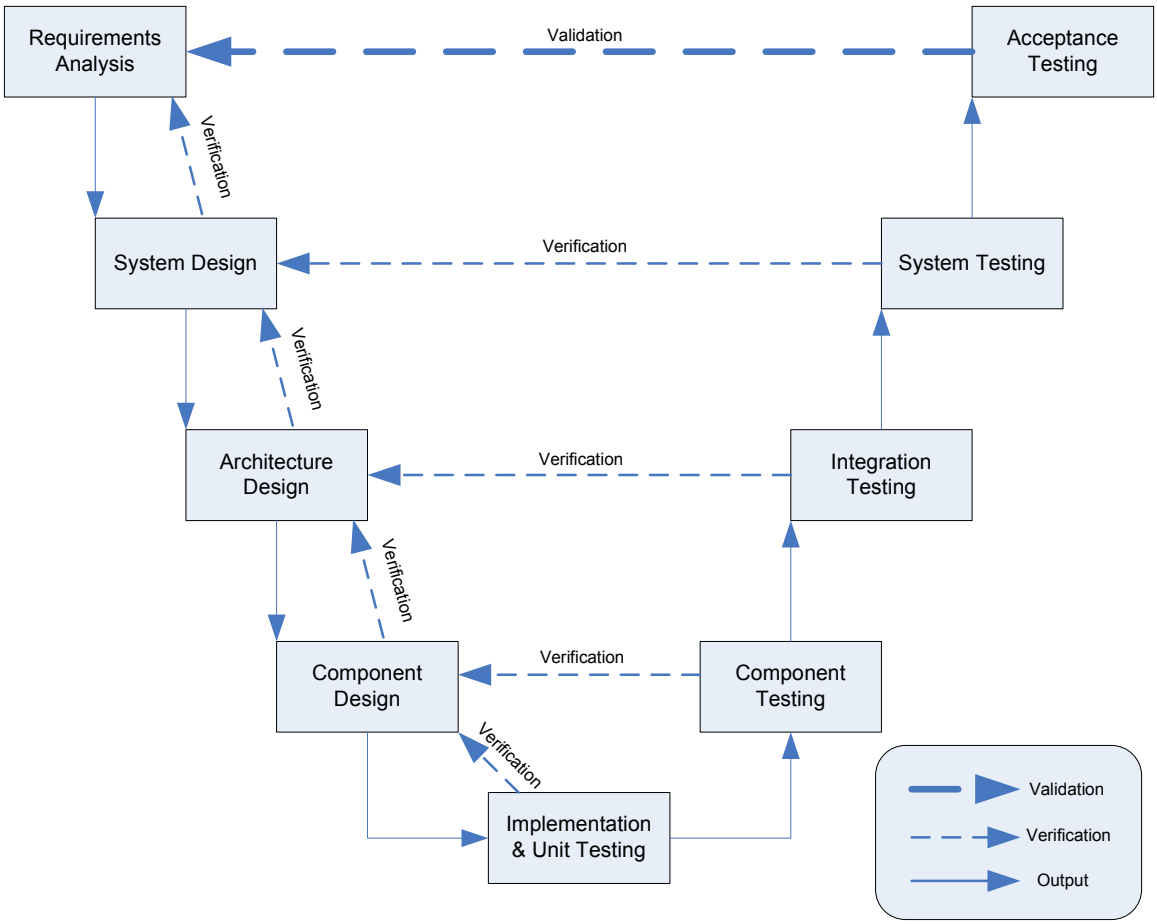

Figure 4.2: The Hybrid V–Model

required to encapsulate the frame format used on the deterministic or safety–critical network in an Ethernet frame. The interface mediator is in fact optional: if the ECU acting as a network gateway is capable of sending and receiving Ethernet frames natively, then the interface mediator can be removed from the design and replaced with a direct link. It is assumed that the interface mediator connects to the gateway by some form of intermediate protocol, the details of which will be considered in the Software Design section (section 4.3.2).

Where at all possible, no customised hardware should be constructed for this testbed. Standardised development kits should be used as the ECUs in each network, and the VSI Bridge should be run on a typical in–service platform (a ruggedised automotive x86 machine with an integrated Ethernet switch). The use of standardised hardware is intended to make experimental results gained from this testbed repeatable, as well as reducing assembly time.

4.3.2 Software Design

In an embedded network, the software design is of far more interest than the hardware design (not least in cases such as this, where all hardware is constrained to be as standardised as possible). Having considered the hardware design, it is now necessary to lay out an overall software design.

The deterministic network should be scheduled and programmed to a minimal operating state (a network schedule sufficient to gain and hold protocol synchronisation, per–node startup code, but no application code other than that required by the triggering system), but otherwise lightly loaded. Since the triggering system will act in place of a high–level application, by requesting frame transmissions from ECUs on the network to simulate the interchange of data that would normally take place, it will be possible to vary the network load to test the system under a variety of network conditions.

The safety–critical network should be scheduled and programmed to a minimal operating state in the same manner and for the same reasons as the deterministic network, above. Again, the triggering system can be used to substitute for a high–level application, although in the case of a time–triggered network, as a safety–critical system is likely to be, network loading should in no way affect the performance (dispatch time and latency) of messages. As such, manipulating the network load on the safety–critical side of the bridge is likely to have minimal impact.

As noted in section 3.1.6, it is possible for messages on the network to be transmitted in one of two modes, either “broadcast” (to all nodes on the network) or “point–to–point” (from the transmitting node to a single receiving node, or at most a small subset of the network). If either the deterministic or safety–critical networks operates in a solely point–to–point mode, it must be ensured that there is at least one message exchange connecting the network gateway to each other node, so that application data that must be bridged to the other network can be copied across that connection.

4.4 Triggering

4.4.1 Triggering Hub

In time, it will be necessary to test the performance of the bridged networks. There are many ways in which this could be done, ranging from instrumenting each ECU in the system and logging

messages sent and received through to implementing an entirely separate system to monitor the behaviour of the network under test (NUT). In this case, the decision was made to implement a separate system that is connected to all the ECUs of the network under test. This system (nicknamed the "Spider" due to its many connections and small central processor) is used to monitor the behaviour of the NUT, as well as to trigger frame transmission events (or one of several other simple functions) in the ECUs that comprise said network. This is done by inserting a simple interface shim into each of the ECUs and connecting it to the central command processor. Since the interface shim is simple and lightweight, it is hoped that its presence will have a minimal impact on the performance of the ECU.

The finalised Spider design consists of a central management node (based on an x86 PC) which extends a monitoring network of direct point–to–point connections to every ECU in the network under test. This monitoring network allows if to trigger events anywhere in the network and measure the results of these events without perturbing the NUT with management data (and thus potentially affecting the results). The design goals for the Spider are quite detailed, covering required latency, communications interfaces and underlying hardware, as well as requiring a degree of platform and network independence. The requirements, and their implications, are explored below.

The first set of requirements that must be considered are related to standardisation. Communication between the Spider and ECUs of the network under test should avoid introducing unnecessary complexity, and should make use of standard interfaces in preference to creating new ones. The Spider should be architecture independent where possible, relying on the presence of an x86 hardware platform with the appropriate interfaces, but as little else as possible, and making no assumptions about the capabilities and features of the endpoint ECUs to which it is connected. The controlling software should be written in standard C, with no more extensions or external libraries than are strictly necessary to support the designed functions. Finally, where possible, the Spider should be designed in such a way that it can be integrated into a larger system, so that existing testbed systems can be retrofitted if desired.

The second group of requirements are related to time. The Spider must maintain an absolute view of time, since as noted in the theoretical background (section 3.1.8), doing so across a heterogeneous distributed system is at best complex and unreliable. All operations within the Spider must have as small an overhead as possible, leading to a low communication and execution latency (most bridging operations execute in times on the order of tens of milliseconds, so the Spider's

operational latency should be in the unit millisecond range). These three requirements imply a composite requirement: since the Spider software is run on an x86 PC, it should use a real–time operating system (RTOS) of some description to ensure deterministic timing.

The final set of requirements is related to life–cycle management and ease of use. The Spider should be reconfigurable, so that message identifier lists, permitted and preferred responses, addressing and routing tables (etc.) can be altered without rebuilding the entire system. Further, it should maintain a unified configuration management system so that only one repository (configuration file, database, or another as–yet unforseen storage system) need be updated to change the system behaviour. Additonally, the system should be modular, with clearly defined interfaces between components, making it simple to upgrade if a new or better module becomes available.

4.4.2 ECU Triggering

In order for the central triggering hub to be able to manipulate the remote ECUs, each ECU must include a small amount of code to receive and act on messages sent by the hub. A simple message processor will exist on each ECU, sitting between the triggering port and the rest of the ECU application code. This message processor will be referred to in the rest of this chapter as an "interface shim", as it is a small piece of code that is inserted between the hub and the ECU application to adapt them to each other. The interface shim accepts commands from the central triggering node and manipulates the running system by sending frames, logging frame reception and altering the contents of the ECU's memory. It is important that the triggering interface shim be as lightweight as possible, since any reduction in available memory or processing time caused by the triggering shim may affect the behaviour of the application on that node in a way that is wholly due to the drain of resources (and as such would not occur in a real–life situation). While such changes in behaviour cannot be avoided in all circumstances, they should be minimised where possible by minimising the system load presented by the shim.

In order to operate as a useful component of the triggering system, the triggering interface shim must expose certain functionality. This functionality can easily be collected into categories (data manipulation, control, routing and event detection), as explained below.

Message Data Manipulation

To maintain simplicity of implementation, it has been decided to write the application in such a way that each ECU exchanges at least one message with each other ECU in the segment per TDMA cycle. Since the messages are transferred continuously, regardless of commands from the triggering system, some method other than message reception is needed to indicate that new data has been received. Consequently, the decision was made to keep a sequence number as the first byte of the payload of messages on the deterministic and safety–critical networks, and to include functions to manipulate this sequence number. The challenge of detecting reception of new messages is then reduced to monitoring the sequence number in each incoming message, and signalling reception if it has changed since the last message.

The triggering shim should, at a minimum, be capable of clearing the sequence number (setting it to 00000000), setting the sequence number (setting it to 11111111), toggling the sequence number by inverting each bit, and incrementing and decrementing it by one. The sequence number being referred to by these requirements is kept in node memory, and automatically copied into the first byte of each outbound message.

These message payload manipulations will cause a receiving ECU to generate a triggering event, as discussed later.

ECU Control

In addition to altering the properties of the messages exchanged between ECUs, it will frequently be necessary to alter properties of the ECUs themselves.

The triggering shim should include support for manipulating the indicator LEDs that can be found on almost all development kits on the market. This serves the purpose of simple error-reporting, as well as making it possible to check that logical addresses within the triggering system correctly match physical addresses (commanding a given node to switch on its indicator LED greatly simplifies the task of locating it in the network).

The triggering shim should contain a command that simply causes the ECU to reply immediately with a copy of that command. This command makes it possible to estimate the delay involved with sending data through the triggering system, by making such a loopback request and timing the wait for a response.

The triggering shim should contain a command that entirely switches off the triggering system on the ECU to which it is sent. This command simplifies tests involving point–to–point messages, as it will allow ECUs irrelevant to current measurements to be soft–disconnected from the triggering system for the duration of the test, rejoining the system at the next NUT restart.

Finally, the triggering shim should provide two acknowledgement messages, one positive and one negative. The positive acknowledgement should be sent back to the triggering controller when a command executes successfully, or when a command has no obvious response (such as an indicator LED request: such requests should be acknowledged positively to simplify the design of the triggering controller). The negative acknowledgement should only be sent to the triggering controller if a command fails to execute correctly.

Message Routing

The triggering shim should include some method of routing messages between networks and between ECUs. At the very least, it should be possible to send a message on one of the attached networks that includes an instruction to retransmit it, either back on the same network or through the VSI Bridge to the other network.

Event Detection

The last important category of requirements for the triggering shim is that of event detection. Thus far, every requirement has been concerned with messages entered into the network, with no provision for receiving messages from the network. Without event detection, it will not be possible to do any sensible measurement or analysis, as there will be no way to determine when a message arrives at a given ECU.

The triggering shim should include messages that are emitted upon the received message differing from its previous value. If the message is now all zeroes or all ones, or toggled from its previous value, the system should report this. Equally, if the message has been incremented or decremented by one, the system should report this. Finally, if the message has changed by more than one and not toggled, it should be considered corrupt and an error flag raised. The event detection system is discussed further in section 5.5.3.

4.5 Bridging

4.5.1 Overview

The connection between the VSI Bridge and the Gateway ECU on each network will be managed by a bespoke protocol that uses a simple frame structure to encapsulate and transfer the data, hereafter referred to as "Bridgelink". The Bridgelink protocol allows the creation of a packet–based communication link between the VSI Bridge interface and the Application Layer of a single ECU. This packet–based Application Programmer's Interface (API) should be common to all Bridgelink implementations, resulting in a modular interface beneath which the platform–specific communications process can be handled. As a result, the technology used as a physical layer can be changed at any time without significant modifications to the NUT firmware being necessary.

The reference implementation will use a high–speed parallel I/O interface capable of handling single–byte MTUs as a physical layer. It will form a point–to–point link from the application layer of the Gateway ECU to the interface mediator device (described further in 4.3.1). Once the Bridgelink frame arrives at the interface mediator, it will be then be encapsulated in an Ethernet frame and relayed onward to the VSI Bridge over a point–to–point Ethernet (raw IEEE802) link.

The reference implementation consists of a Physical Layer, a Data Link Layer and a Network Layer (OSI model levels 1, 2 and 3 respectively). The overall operation of the reference implementation is described in figure 4.3, which shows the sequence of operations for data transfer in both directions.

4.5.2 Bridgelink Network Layer

The Network Layer consists of a packet structure and an API to be used for data transfer by the node application. The Network–Layer packet (NL–P) supports variable–length payloads and includes a trailer checksum to protect the integrity of the packet, as shown in the packet structure diagram, figure 4.4.

- Header
  - Type (1 byte) Packet type (see type listing)
  - Seq# (1 byte) Packet sequence number

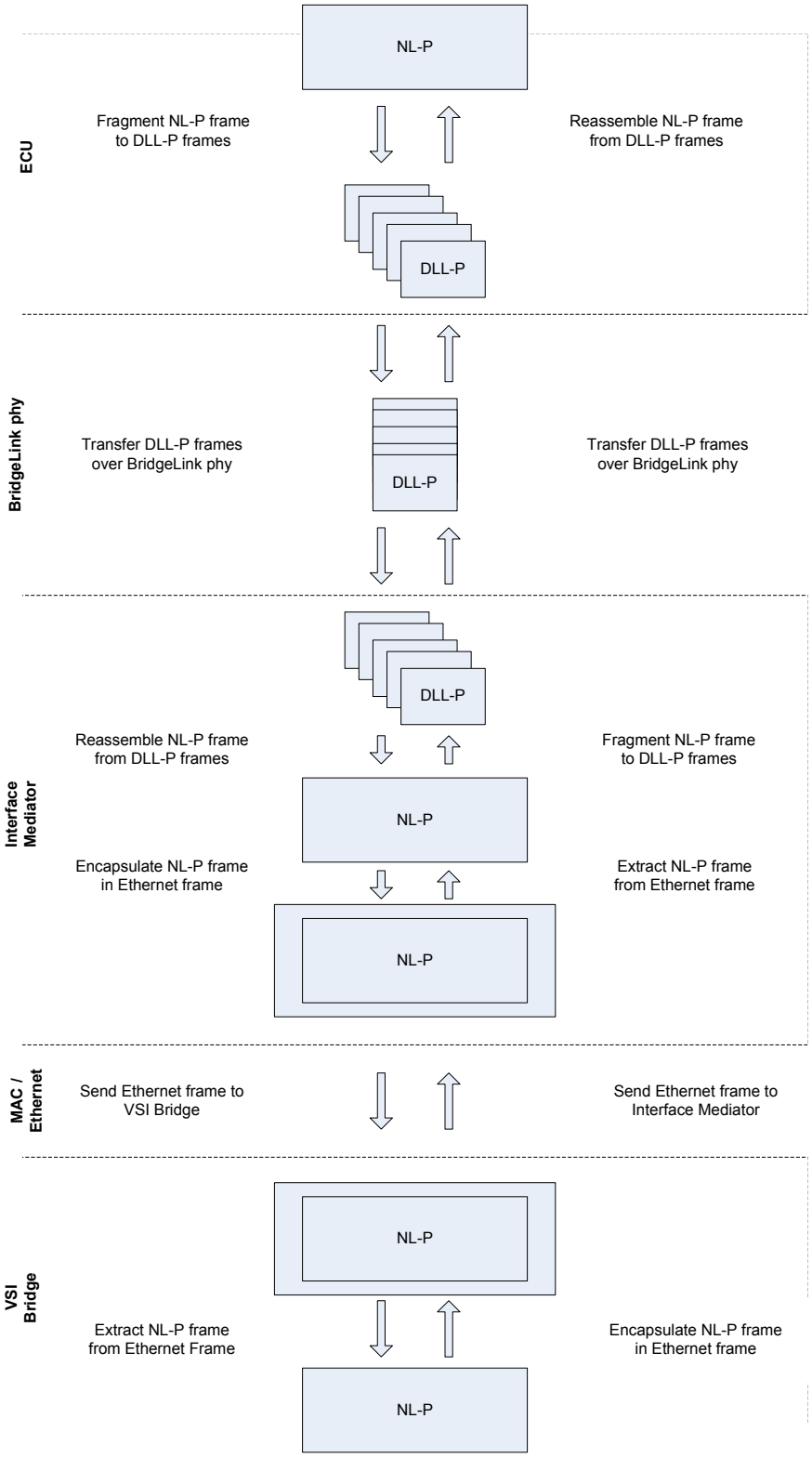

Figure 4.3: BridgeLink sequence of operations

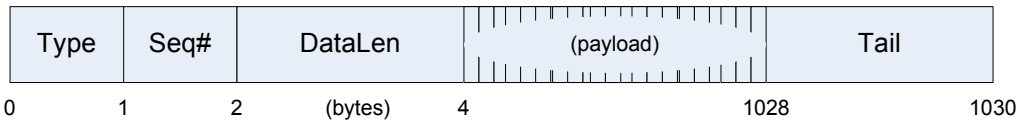

Figure 4.4: The NL–P frame structure

  - DataLen (2 bytes) Count of bytes in the payload

- Data (0-1024 bytes of application data)
- Tail
  - Checksum (2 bytes)

Whether it is sent directly by the gateway or via an interface mediator, at some point in its life any network–layer frame will be encapsulated in an Ethernet frame, for transfer to or from a VSI Bridge (over a point–to–point Ethernet link). The most common maximum payload size for an Ethernet frame is 1500 bytes (as stated in the IEEE 802.3 standard [22], so the network layer packet (NL–P) is deliberately specified to have a total length less than this: 1024 bytes of payload plus a six–byte header. As a result, it will be possible to encapsulate an NL–P in a standard Ethernet frame without fragmentation.

The NL–P header has a "Type" field, which must assume one of the following values.

Valid NL–P Header.Type values:

- **TYPE_INV:** Invalid
- **TYPE_DATA:** Application data
- **TYPE_CMD_COMMIT:** Acknowledge the reception of frames prior to this sequence number
- **TYPE_CMD_COMMIT_REQ:** Request acknowledgement of frame's reception
- **TYPE_CMD_RESEND:** Request retransmission of the most recent frame
- **TYPE_CMD_ID:** Request remote node's identification

A frame of type TYPE_INV should not be transmitted, but is a good default value for use in frames stored in a frame–pool, awaiting use. A frame of type TYPE_DATA contains data to be transferred between ECUs in its payload (this is the most common type of frame). If an ECU receives a frame of type TYPE_CMD_COMMIT_REQ, it should reply with a frame of type TYPE_CMD_COMMIT: a received "commit" frame confirms the reception of all frames transmitted prior to the request. A frame of type TYPE_CMD_RESEND indicates that the last frame was not correctly received, and should be retransmitted. Finally, a frame of type TYPE_CMD_ID is a request that the ECU should respond with its identity in the payload of the next frame it transmits (the details of said identity are currently implementation–dependent). The payload data of frames of command types is not application–relevant, and should be discarded during final frame decoding.

4.5.3 Bridgelink Data–Link Layer

The Data Link Layer is positioned between the Network–Layer API and the physical layer, and provides a packet structure (the "DLL–P") and the necessary algorithms to fragment and reassemble NL–Ps into DLL–Ps. Additionally, the Data Link Layer includes features to handle error–detection and resynchronisation. This additional fragmentation and reassembly stage in the communications process was added to provide an error–recovery system for the potentially error–prone parallel I/O physical layer. This is done by inserting additional checksumming steps after every 30 bytes of payload data, so that if a checksum fails, only 32 bytes need to be retransmitted rather than the entire length of the original NL–P. The Data–Link Layer inserts an additional 2 bytes into every 30 bytes of payload transmitted: a small overhead to bring greater reliability.

DLL Frame layout

The Data Link layer is based on a fixed–size frame (a DLL–P), which is checksummed to ensure reliable data transfer over the physical layer. The DLL–P has a fixed length of 32 bytes, and is structured as shown in figure 4.5.

- Header (1 byte)
  - Data Length field(bits 0–4): Count of bytes of application data in the payload.
  - Frame Type (bits 5–7): The type of the frame, selected from one of the six values listed

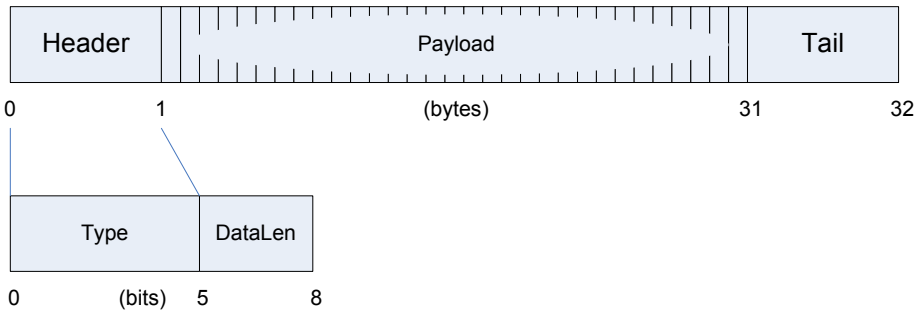

Figure 4.5: DLL–P frame layout

below..

- Payload (30 bytes, zero–padded if insufficient application data is supplied)
- Tail (1 byte)
  - Checksum (bits 0–7)

The payload section of a DLL frame is fixed at 30 bytes in length, to simplify the necessary frame buffering and error correction code. However, unless the NL–P being fragmented is an exact multiple of 30 bytes in length, it will be necessary to store less than 30 bytes in a DLL frame at some point. The Data Length field in the DLL frame header is designed to address this problem, allowing a data length less than thirty to be set, the appropriate number of bytes to be placed in the payload section, and the rest of the payload section to be padded with zeroes.

DLL–P frame transfers are potentially vulnerable to stream desynchronisation and corruption, due to the use of a parallel I/O physical layer without error checking or EMI screening. Since the DLL-P has a fixed length and is checksummed at both ends of the connection, it is possible to detect and correct these conditions in most circumstances. If correction is not possible, it may be necessary to resynchronise the connected ECUs, depending on the failure mode.

The resynchronisation mechanism is designed into the reception algorithm of the Data Link Layer, and can be triggered by the receiving ECU if necessary (using a DLL–P frame of type TYPE_CMD containing a resynch command). Ideally, the resynchronisation procedure should take place only during system startup, as it consumes two DLL–P frames and erases transmission and reception buffers during its execution, potentially presenting a significant disruption to ongoing operations in a real–time system. The resynchronisation sequence is a pair of 32–byte blocks of data (they are

not true DLL–P frames, as they have no header or trailer, and are composed entirely of payload). These data blocks contain 64 bytes of sequentially incrementing values (from 0x00 to 0x3f). When the receiving side detects this stream sequence, it should flush its buffers and restart all operations from last known checkpoints.

DLL–P packets can be one of the following types:

1. TYPE_INV: Invalid (for inactive frames)
2. TYPE_CMD: Command (command operand is stored in payload)
3. TYPE_NLP_F: First frame of a stream
4. TYPE_NLP_I: Intermediate frame within a stream
5. TYPE_NLP_L: Last frame of a stream
6. TYPE_NLP_S: Single frame, not part of a stream

The DLL–P (and indeed the entire Data–Link Layer) is designed to transport NL–P packets by fragmenting them into a sequence of DLL–P frames. The NL–P packet is divided into blocks of 30 bytes, each of which is then loaded into the payload of a DLL–P frame and enqueued for transmission. The Type assigned to each packet is dependent on the data that has been stored in it. If the NL–P packet is less than 30 bytes in overall length (including header and trailer), it can be stored in a single DLL–P, which will be labelled as TYPE_NLP_S. Larger NL–P packets will be fragmented into a stream of 30–byte blocks: the first frame of such a stream will be TYPE_NLP_F, the intermediate ones TYPE_NLP_I and the final frame TYPE_NLP_L (NLP_L frames will frequently be less than 30 bytes long, and zero–padded). If multiple long NL–Ps are transmitted together, the groups of DLL–Ps into which they are fragmented will be separated by TYPE_NLP_L / TYPE_NLP_F boundaries, allowing the Network Layer to detect and recover from lost or discarded DLL–P frames, as well as making packet reassembly possible.

Once frames have been placed in the transmission buffer, control passes to the Physical Layer, where the transmit operation proceeds according to the flow diagram in figure 4.6. DLL–P frames in the buffer are treated as raw 32–byte blocks, and they are processed and transmitted by the physical layer in an implementation–dependent manner.

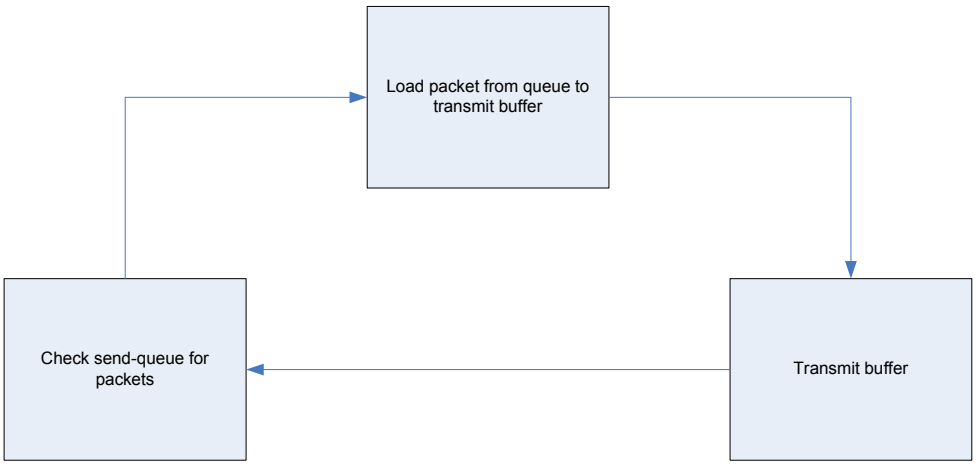

Figure 4.6: The algorithm responsible for DLL–P frame transmission

The algorithm responsible for handling DLL–P frame reception is rather more complicated, as it includes the aforementioned resynchronisation system, as well as checksumming incoming frames. It can be seen in figure 4.7.

The reception algorithm receives individual bytes from the underlying physical layer, and attempts to reassemble them into DLL–Ps. The algorithm maintains three items of data in memory: a "sync_count" byte, a 32–byte frame buffer, and a "framebuf_count" field that lists the current amount of valid data in the framebuffer. As each byte arrives, it is inspected. If it has a value one greater than the current value of sync_count, then sync_count is incremented. If it does not, sync_count is reset to zero: either way, the byte is appended to the frame buffer and the frame buffer counter is incremented. Then, if the new value of sync_count is equal to 0x3f, the frame buffer will be discarded, as this can only happen when the remote ECU requests resynchronisation, and the frame buffer is consequently full of the second resynchronisation frame, not useful data. All internal counters are zeroed, and the connection is reset to its initial configuration.

If the inbound byte does not signal a connection resynchronisation (is not equal to 0x3f), the frame buffer counter is checked to see if it is full: if it is, then a complete DLL–P has probably been received. The frame checksum is validated and, if it is valid, the frame is passed up to the network layer for further processing. If the checksum is invalid, the algorithm will request that the remote ECU retransmit the last frame. Either way, the frame buffer is then cleared, ready to accept the next inbound frame.

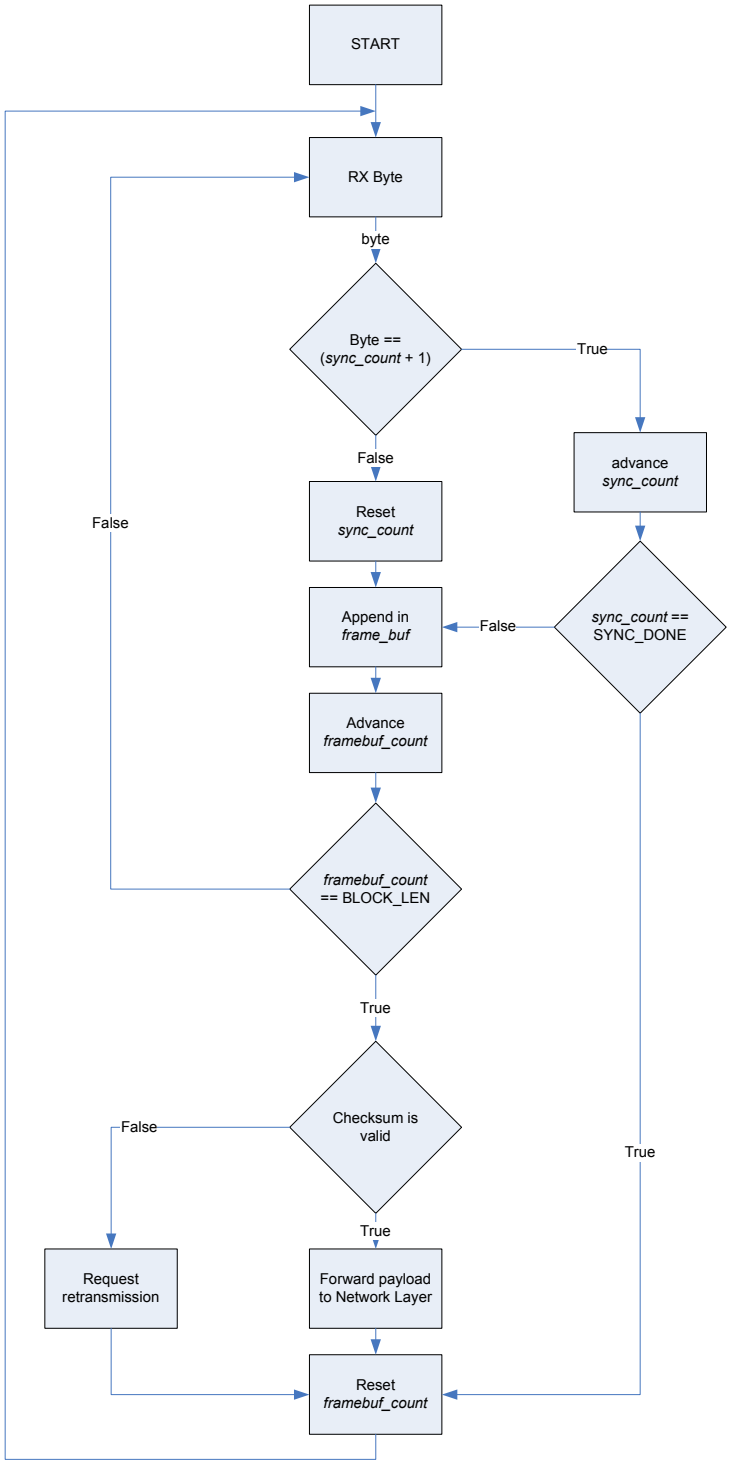

Figure 4.7: The algorithm responsible for DLL–P frame reception

If the checksum for a given DLL–P fails to validate, it is assumed that the frame was corrupted in transit. In this case, the Data–Link layer should discard the frame and request retransmission of that specific DLL–P. Since DLL–Ps are transmitted sequentially and one at a time, requesting retransmission is a simple matter of sending a negative acknowledgement rather than a positive one. If retransmission fails more than a set number of times, the data–link layer will attempt resynchronisation instead (by transmitting the resynchronisation sequence). Fault conditions that result in resynchronisation should be extremely rare, as the process is time–consuming and consumes time that would otherwise be used to transfer potentially time–critical data. As such, it should not be a regular or expected thing, outside of the necessary initial synchronisation step during system startup.

Now that the incoming data stream has been reassembled into DLL–Ps, it is necessary to process the frames according to their type. TYPE_NLP_F, TYPE_NLP_I and TYPE_NLP_L frames are the first, intermediate and last frames in a fragmented NLP frame, and should be extracted and assembled. TYPE_NLP_S frames contain single NLP frames that are small enough to be encapsulated in a single DLL–P, and should be extracted. In either case, the resulting NLP should be passed up to the network layer if it assembles correctly. Finally, TYPE_CMD frames are forwarded to the data–link layer command processor. The flow diagram in figure 4.8 shows the frame reassembly process.

4.5.4 Bridgelink Physical Layer

The Physical Layer of the protocol defines the hardware interface and the signalling system necessary to prevent data loss in transfer. The functional design and connections are shown in figure 4.9.

The physical layer connects two devices, device A and device B. In the testbed implementation, one of these devices will be a gateway ECU, and the other a VSI Bridge, but the protocol is sufficiently versatile that it can be used to interconnect any two devices over a short distance. Each device uses a pair of DATA ports and a pair of CONTROL ports (one in each direction), meaning that the protocol can support full–duplex operation. The DATA port combines 8 signal lines to supply a bytewise transfer path for data, while the CONTROL port uses four signal lines for bi–directional flow control (two in each direction). Fully bi–directional flow control allows the two devices to

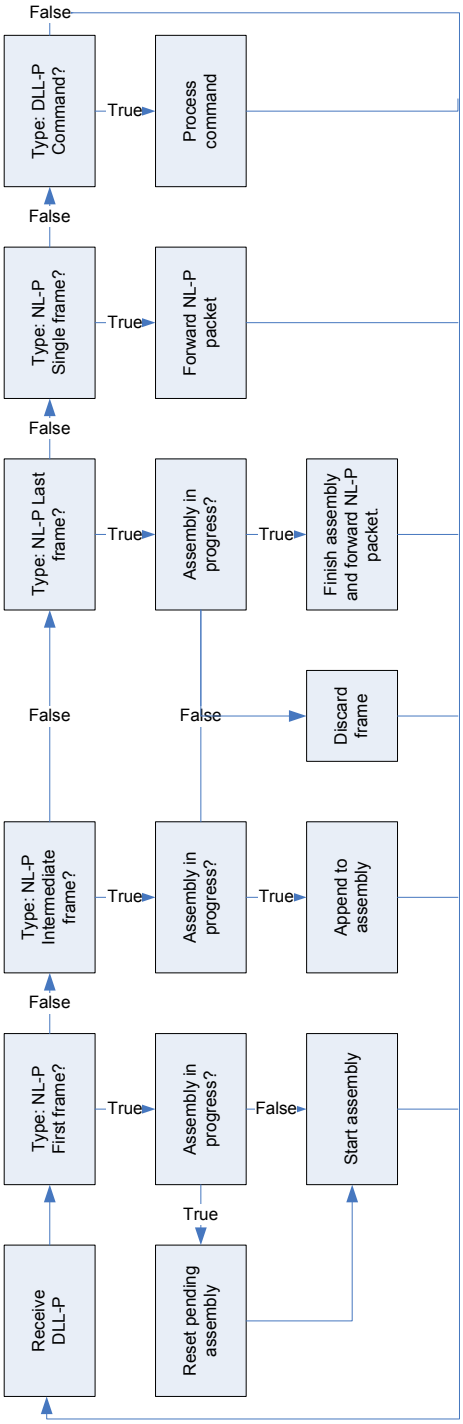

Figure 4.8: The algorithm responsible for NL–P frame assembly

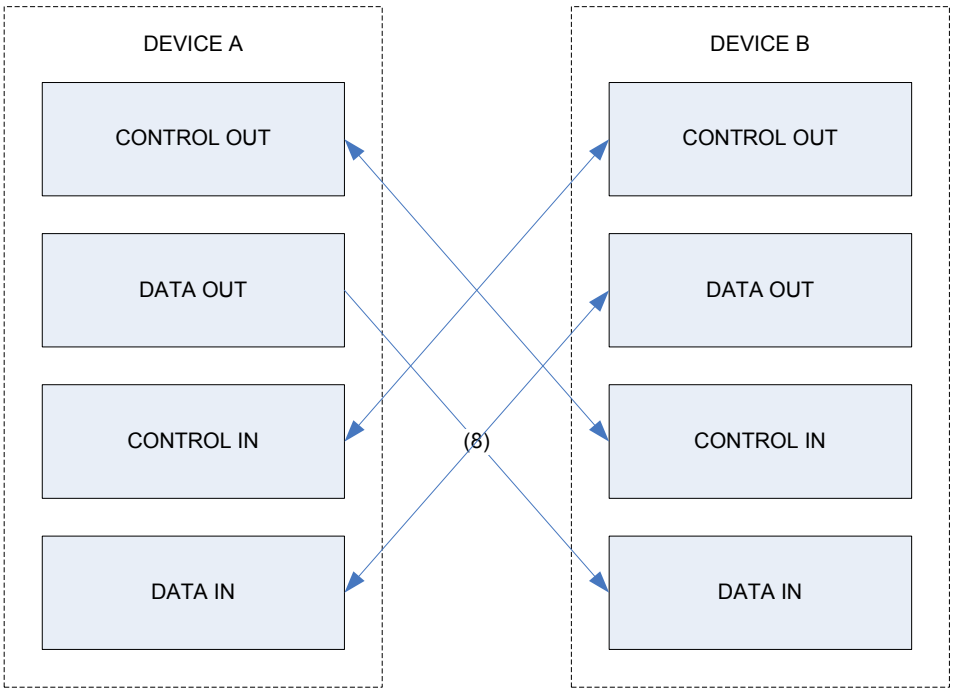

Figure 4.9: The physical–layer design for the BridgeLink protocol

operate asynchronously, meaning that the BridgeLink connection presents a minimal load on the resources of the host ECU. This is because it is possible to interrupt the protocol operation to execute another task - the flow control system contains all the state that is in transit, and allows transfer to resume later without loss of data. For a given transaction, the device putting data into the connection will be referred to as the Writer, while the device reading data out of the connection will be referred to as the Reader.

The two control lines relevant to a given direction of data transfer are referred to as ACT–R and ACT–W.

- **ACT–W:** Asserted by the Writer to indicate that valid data is present on the DATA bus.
- **ACT–R:** Asserted by the Reader to indicate that the DATA bus has been sampled.

A timing diagram demonstrating the operations necessary to transfer a single byte can be found in figure 4.10.

The numbered steps on the diagram are explained below:

1. The Writer sets valid data on the DATA bus.

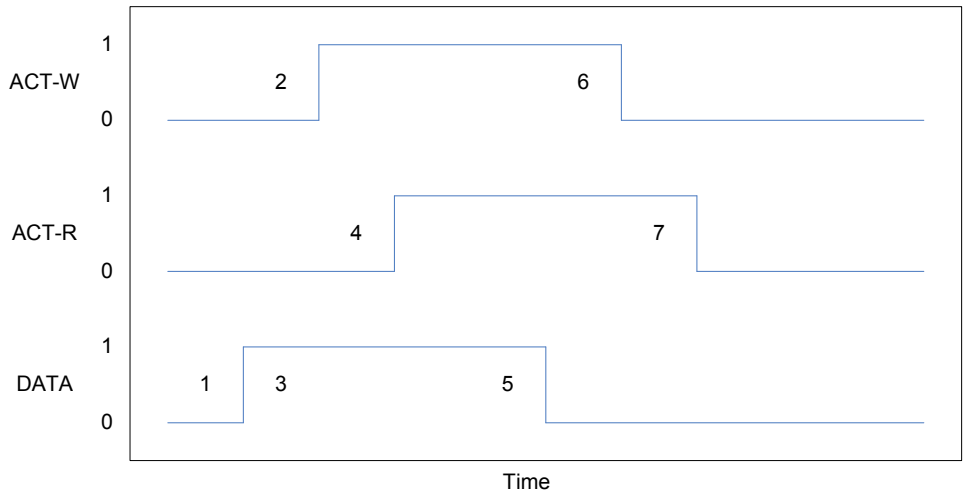

Figure 4.10: A unidirectional byte transfer using the BridgeLink protocol

2. ACT–W is asserted by the Writer to signal that data is ready.
3. Data is read from the DATA bus by the Reader.
4. The Reader asserts ACT–R to signal that it has read the data.
5. The Writer resets the DATA bus to its default value (or to the next byte if a multibyte transfer is in progress).
6. The Writer clears ACT–W to signal that the transmission is complete.
7. The Reader clears ACT–R to acknowledge the transmission.

4.6 Conclusion

In this chapter, a high–level design for the bridging testbed has been presented, along with a detailed design for the bridging and triggering subsystems. It is now necessary to map appropriate technologies on to the abstract design given here to produce a working testbed.

Chapter 5

V&V Testbed - Technology Mapping

5.1 Introduction

In the following few pages, the high–level design from the previous chapter will be revisited, and each abstract element of the design will be mapped to one or more technologies that will be used to implement the testbed. It should be noted that the choices made in the technology–mapping process are by no means absolute and immutable: the following presents the technology mapping used in this testbed, but it is entirely possible to produce a different mapping using different technologies to implement the same design (so long as design constraints are respected).

This Technology Mapping process is the first substage within the Implementation stage of the developmental methodologies. Once each stage has been mapped to an appropriate technology, implementation can proceed.

5.2 Technology Mapping – Overview

The technology–mapping process is essentially concerned with studying the requirements imposed by the system and system–of–system designs, and attempting to select a technology or technolo-

gies suited to implementing that component of the design. The selection process is guided by two sets of considerations, which may overlap or act against each other, depending on the system being designed and its environment. Design constraints will typically constrain implementation by requiring certain functional behaviours and properties (such as minimum latency, number and type of communications buses, overall system architecture and message types). By contrast, Implementation constraints will typically be concerned with more physical properties such as signal levels, operating voltages, component size and per–node cost.

It is entirely possible to design a system that meets all of the customer's requirements, but cannot be implemented (or would be prohibitively expensive to manufacture and maintain), just as it is possible to produce an implementable design that does not perform the desired function, as laid out by the customer. The technology mapping stage should address this as a part of the overall design process, and return unimplementable designs to the design stage for reconsideration.

5.3 Deterministic Network – MilCAN

Comparatively early in the development process, the decision was made to use MilCAN as the implementing technology for the deterministic network in the testbed. This decision was made for several reasons: first, because MilCAN is a widely adopted standard, used in multiple UK military vehicles. Second, MilCAN is a familiar platform with which the author has a reasonable amount of experience, which simplifies the programming and debugging of the ECUs composing the deterministic network. Third, since the laboratory in which this thesis was researched and written is primarily concerned with the MilCAN protocol (indeed, it is a member of the MilCAN Working Group, the steering committee that specifies the protocol and certifies devices as compatible), development hardware capable of operating as part of a MilCAN network was in plentiful supply. Finally, since the VSI Bridge is to be used for network diagnostics and bridging functions within the testbed, it seems wise to include one of the protocols for which it was designed. Doing so will remove an important variable from the testing procedure: that of whether or not the deterministic network is fully compatible with the VSI Bridge (the safety–critical network, however, remains a potential concern).

5.3.1 Protocol

MilCAN–A is an open standard interface implemented in devices mainly used in military vehicles[23]. It defines a deterministic communications protocol as a middleware layer on top of the popular Controller Area Network standard, allowing synchronised bus operation and predictable maximum message delivery latency. This known maximum latency of delivery means that MilCAN–compliant devices can safely be used in real–time or deterministic systems, since high priority messages are delivered within their latency contract even in cases of high bus load.

The underlying CAN network operates at a speed of 250, 500 or 1000 kbps, dependent on application and bus length, and supports 29–bit message identifiers, allowing MilCAN devices to be interoperated with SAE J1939 networks. The protocols are bus–compatible, (indeed, the frame formats are identical, with the exception of the 25th bit of the message identifier) in that they can be utilised on the same bus by different groups of devices without interfering, but communication between MilCAN–A and J1939 must be facilitated through an application–layer bridge.

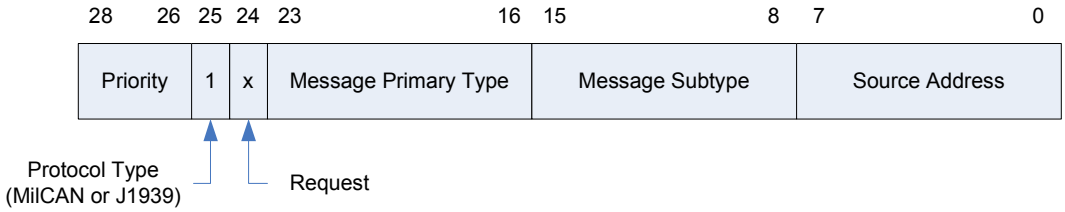

Figure 5.1: MilCAN Message Identifier Layout (p24 of [24])

The protocol achieves low–latency performance using a periodically repeating transmission schedule, synchronised by a master node (defined as the responsive node with the lowest source address). The schedule is divided into 1024 Primary Time Units (PTU), and messages scheduled to be transmitted in that PTU are queued for transmission at the sending node. MilCAN messages have a priority level ranging between zero and seven, each determining a maximum delivery latency from "in this PTU" to "whenever the bus is free". Message priority is arbitrated in a two–stage process: first, the highest priority message in the transmission queue at each node is placed on the bus; then CAN arbitration is applied to the message identifier to determine bus priority. Since message transmission takes place in order of priority, high priority messages will meet their delivery contract, even if the bus is heavily loaded. Note, of course, that this relies on the message schedule being designed sensibly: delivery failure can occur if too many messages are set to excessively high priorities.

Using the MilCAN layer on top of CAN adds a degree of determinism to the network, as well as the potential for a higher percentage utilisation, since frame transmission occurs at scheduled intervals rather than whenever a frame is ready to be sent. This means that transmission can be predicted, to a degree, and if the schedule is carefully designed, each PTU can be fully loaded with frames and every PTU in the cycle can be occupied.

5.3.2 Network Structure

Being based on a CAN network, a MilCAN network can support any topology that a CAN network would. For the purposes of this testbed, the MilCAN network will be configured as a group of three passive buses (see section 2.1.2 for a brief introduction to the bus topology), bridged over Ethernet and MilCAN backbones. The Utility and Automotive segments are bridged by VSI Bridges using XC167 interface mediators, while the Multimedia segment is connected to the MilCAN Backbone by a cut–through bridge of a type similar to that described in Kwok and Mukherjee's 1994 paper [17]. The final testbed will include a pair of MilCAN backbones and a single Ethernet backbone, connected as per the testbed diagram (Figure 5.2). These backbones will not all be in use at once: rather, they are all present to maximise the flexibility of the testbed, and will be enabled or disabled individually in software for particular tests.

During the normal operation of the testbed, the MilCAN network will use the blue channel (MilCAN 1) exclusively. Since each node in this network has two CAN interfaces, this means that all message data will be routed through just one of the ports, and the other port will be disabled in software. The red channel (MilCAN 2) will be present and connected, but unused. Messages travelling between segments, then, can be passed over MilCAN or over Ethernet through the VSI Bridge. It should be noted that the XC167 interface mediators on the first and second subnets do not operate as cut–through bridges in the same way that the third does: rather, messages arrive from the subnet, pass into the VSI Bridge via the interface mediator and are routed back out on to the MilCAN 1 backbone via the interface mediator's second CAN port. Consequently, despite appearances in the network diagram, all messages passing between MilCAN subnets via MilCAN 1 are routed through at least one VSI Bridge.

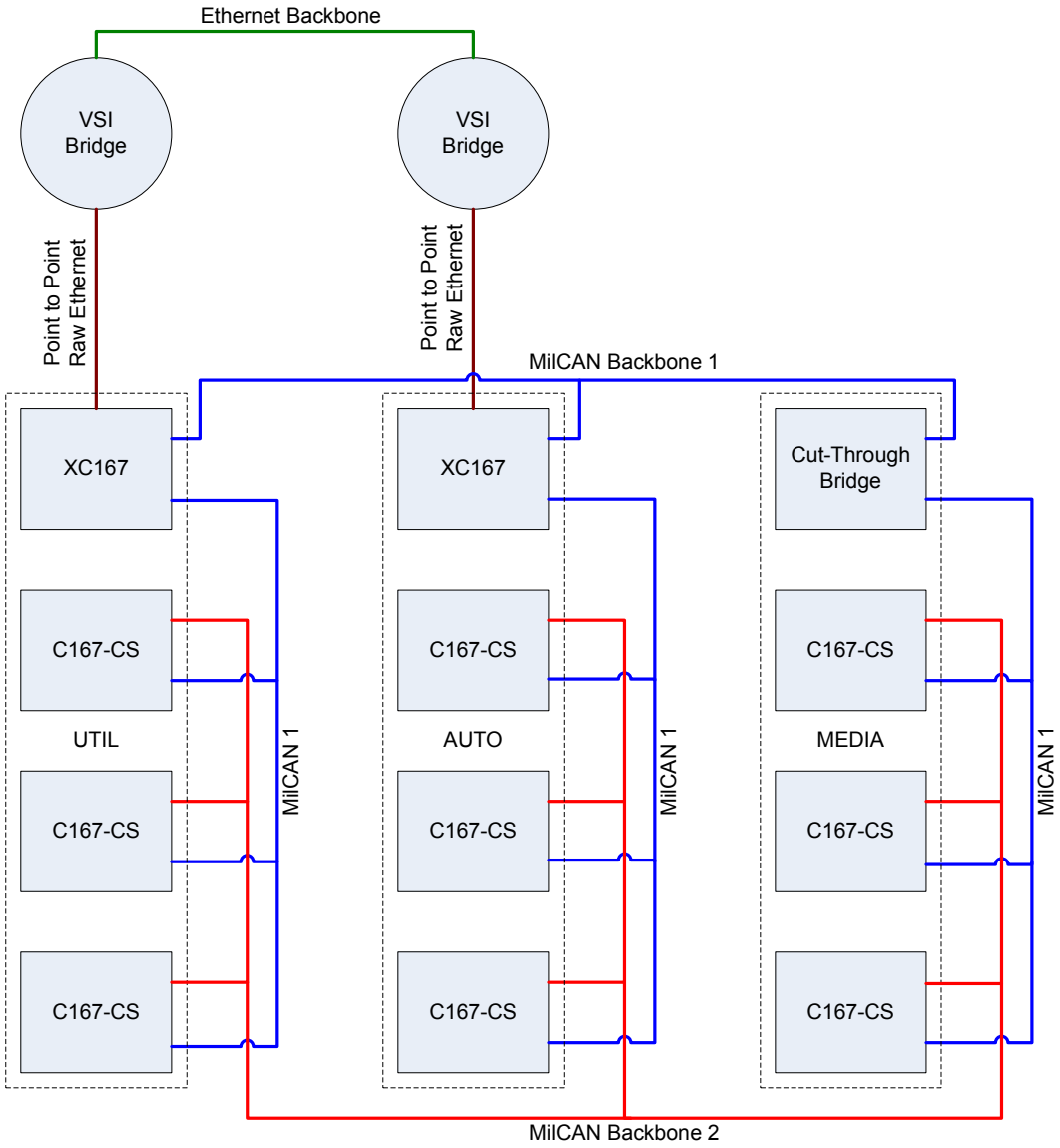

Figure 5.2: Simplified diagram of the MilCAN segment

5.3.3 Development Kits

The MilCAN network is based on a set of Phytec C167–CS development kits with dual CAN interfaces. The C167–CS kits are based around a single C167 microcontroller clocked at 20MHz, and have dual hardware CAN modems and dual hardware USARTs. Since these development kits are being used to provide network load rather than to implement a fully–featured embedded application, their other peripherals are not of particular interest.

It should be noted that the interface mediators on the MilCAN network will be development kits based on a slightly different microcontroller, the XC167, which has an integrated Ethernet controller and is capable of clocking at up to 40MHz. The choice to base only the interface mediators

(which are otherwise normal members of the MilCAN network) on a slightly different device was driven by the need for an Ethernet connection to the VSI Bridge. However, the XC167 was not used to implement the remaining nodes in the MilCAN network, since XC167 development kits have a substantially higher per–unit cost.

The C167–CS development kits have no on–board operating system: aside from a small startup script (written in C166 assembly language) that provides device configuration and minimal memory management, the programmer is required to work with the hardware directly. In addition to this startup script, all development kits in the MilCAN network are running a standard MilCAN protocol stack. This protocol stack uses the timers and interrupt mechanism of the C167 to provide a MilCAN API to the application firmware: once per MilCAN PTU, a time–driven interrupt service routine checks the device's transmission schedule and, if a message of higher priority than the one currently awaiting transmission is scheduled to be transmitted in this PTU, the lower priority message is pushed back into the message queue and the higher priority message is loaded into the transmission buffer. Separately, whenever the CAN modem raises a "transmission complete" flag, the next highest priority message in the queue is loaded into the transmission buffer (if and only if it is scheduled to be transmitted in a PTU prior to this one). In addition to the above, the protocol stack is responsible for achieving and maintaining synchronisation between the ECU and the network (including bus–master election, where necessary). This MilCAN stack is a product of the Vetronics Research Group (in which this thesis was researched), but was written and maintained by my colleague Dr Periklis Charchalakis (as a part of his thesis in 2005 [2]).

5.3.4 High–Level Application

For the purposes of this testbench platform, a minimal functioning network is required. Under the MilCAN protocol, a minimal functioning network would be one in which the only messages on the bus are Sync Frames emitted by the Sync Master. The MilCAN network in this testbed, however, was reused from an earlier project to save on setup time, and contained some messages related to that application. These messages will provide a realistic operating environment in that the MilCAN network will be performing a function, but should not affect the testbed. Since MilCAN is a deterministic network, the presence of other messages on the network will only affect the performance of these systems at all if they are both scheduled in the same slot and of a higher priority than the triggered frames. Knowing this, it is trivial to work around their presence.

The three subnets of the MilCAN network are each assigned a function in the overall application. The first subnet is a "Utility" network, carrying sensor information and monitoring the vehicle's power supply. The second subnet is an "Automotive" network, carrying information about individual wheel traction and controlling the engine, steering, drivetrain and lights. The third and final subnet is a "Multimedia" network, containing a camera and its attached crew–station screen. Each of the three segments carries a low but constant message load, simulating the behaviour of a vetronic network with such peripherals attached to it (the camera video stream would likely overwhelm the network, and as such is routed over Ethernet instead: only control data for the camera passes over MilCAN). The triggering commands will be injected into any of the three networks, at a high priority and in a slot not currently used by any other message.

5.4 Safety–Critical Network – FlexRay

Mapping a technology on to the safety–critical subnetwork of the testbed was not such a simple decision. The safety–critical subnetwork has several requirements which are met by comparatively few available technologies: reliable delivery, redundant signalling and strong time–triggering, among others. Additionally, the chosen technology should be available on the open market: mapping a technology that meets all of these requirements to the safety–critical subnetwork is of little use if development kits supporting that technology are not available.

Two technologies were eventually chosen to implement the safety–critical subnetwork, TTP and FlexRay. Each of these technologies was used to construct a small safety–critical network, which was then bridged to the deterministic network by means of a VSI Bridge and an XC167 interface mediator, using a Bridgelink connection to link the XC167 to the selected gateway device on the safety–critical network. The behaviour and performance of the TTP subnetwork is studied in detail in my colleague John Melentis' thesis (Melentis 2010 [25]). The behaviour and performance of the FlexRay subnetwork will form the focus of investigation for the remainder of this thesis. A brief introduction to the FlexRay protocol is presented below, as was for the MilCAN protocol above.

FlexRay is a flexible, safety–critical protocol designed to provide high availability, high bandwidth, and fault tolerance in next–generation X–By–Wire and power–train networks. The protocol itself is an emerging standard in the automotive sector, specified [26] by a consortium of controlling interests spanning the fields of embedded systems, networking and vehicle technology, including

companies such as Freescale, Bosch, BMW, General Motors, Volkswagen and Ford.

5.4.1 Protocol

The FlexRay protocol requirements include a need for both determinism and flexibility, two goals that at first glance appear entirely contradictory. Nevertheless, FlexRay strikes a reasonable balance by defining a repeating communications cycle, then dividing it into a static (deterministic) segment and a dynamic (flexible) segment.

The static segment is a strict TDMA in which nodes may only transmit messages in slots pre–allocated to a specific node/message combination. Since communications controllers cannot be reconfigured without taking the node off–bus, the TDMA is generally considered to be fixed short of reprogramming.

The dynamic segment is a much more flexible TDMA, in which messages of variable length may be transmitted if the scheduled node chooses to (and the message is provisionally scheduled in the current minislot). In the dynamic segment, different messages may be transmitted on each bus at the same time, assuming the system has multiple channels.

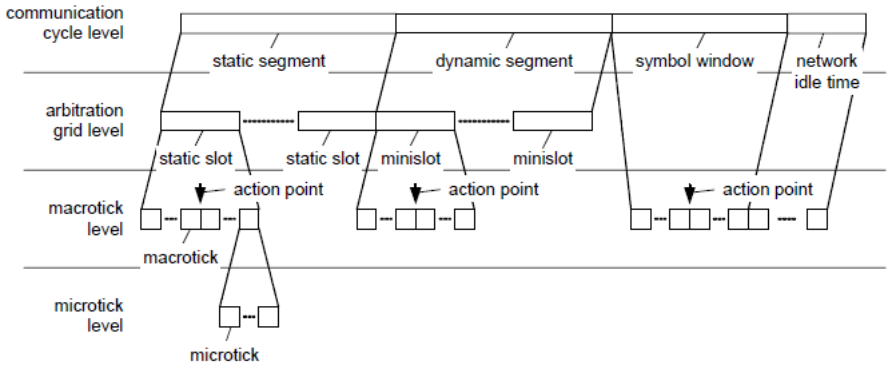

Figure 5.3: FlexRay Communication Cycle structure (page 100 of the Protocol Specification, [26])

Frames in both the static and dynamic segments have the same format, although dynamic frames conventionally include a message identifier as the first two bytes of the payload (used only in the application layer, not by the communications controller). The frame contains a series of flags to identify the type of frame, a frame identifier (that serves together with the cycle count to uniquely identify the frame in the TDMA), a payload of 0–254 bytes and a packet CRC to guard against corruption.

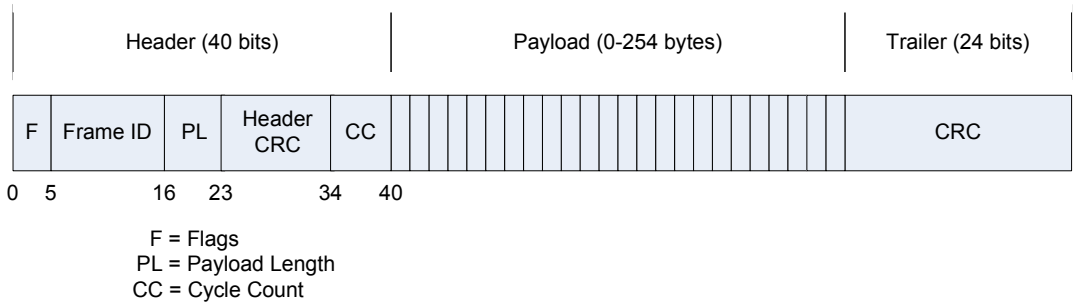

Figure 5.4: FlexRay Frame structure

5.4.2 Network Structure

Network Topology

The FlexRay cluster consists of a number of ECUs (“nodes”) and an optional number of active–star couplers, connected by two–wire differential buses. The optimal (and recommended) cluster configuration is a group of nodes connected via point–to–point links to an active–star coupler, which may in turn be connected onward to other active–stars. However, for development and for less–critical applications, the protocol also supports passive–star and passive–bus configurations, as well as hybrid topologies comprising several stars and/or buses in situations where the range or membership count of the cluster needs to be increased. In all cases, however, the topology should be a spanning tree, that is, there are no cycles in the cluster and every node can be reached from every other by at most one route.

The FlexRay protocol includes support for dual–bus configurations, to increase the redundancy of the physical layer and the cluster’s resilience to damage. FlexRay messages may be sent on one or both channels: in the case of safety–critical messages, it is strongly advised that the message be sent on both channels in case one is damaged. Additionally, if the dual–channel configuration is in use, it is recommended that the physical routing and connection graph for each channel is substantially different, to minimise the risk of physical damage partitioning the cluster.

Bus Guardian

The Bus Guardian [27] is an optional component of the FlexRay Communications Controller that enforces the programmed TDMA, preventing a damaged or misconfigured node from accessing the bus except during its scheduled transmission slots and thus preventing a node suffering from

the “Babbling Idiot” failure mode from corrupting the bus.

Structurally, the bus guardian is very similar to a communications controller, containing many of the same elements. At power on, it gains clock synchronisation independently from the local communications controllers, and loads an independent copy of the TDMA schedule. Once the ECU has synchronised with the cluster, the bus guardian monitors the TX_EN (“Transmit Enable”) lines that link the communications controller to its corresponding physical layer bus driver. If the TX_EN lines are asserted at a point in the TDMA schedule that does not contain a transmission slot for the appropriate controller, the bus guardian raises an immediate error interrupt and disables the communications controller (i.e. causes the communications controller to Fail Silent.).

Active Star

The active star configuration is an extension of the passive star network, with the exception that in place of a cable splice, the central connection is mediated through a piece of processing electronics. The passive star configuration is essentially a common bus, whereas an active star is comprised of a set of point–to–point connections linking each ECU to the star coupler. The star coupler contains separate bus drivers for each link, meaning that the branches are electrically decoupled from each other. However, no routing is performed: messages entering the star on one branch are transferred to all other branches (see page 29 and chapter 9 of [28] for further details).

Just as a FlexRay ECU may have a Bus Guardian monitoring each communications controller, so an Active Star may include a Bus Guardian on each link to monitor incoming traffic. If both the node and the star have bus guardians, both the node–local and the central bus guardians will monitor traffic on a given point–to–point link, increasing the likelihood that TDMA violations will be prevented. If the active–star coupler and the ECU are connected to different power supplies (not an unlikely case, since the devices are often physically separate), it is possible to place the star coupler and each ECU in separate Fault Containment Regions. Despite the presence of the Bus Guardian in a given ECU, it is not typically possible to place the ECU into its own FCR without an active star with appropriate bus guardians being present, as most if not all current ECU implementations power and clock the node–local bus guardian from the same sources as the ECU itself.

The simplest possible Active Star implementation can be constructed by coupling together appropriate bus drivers, such as the NXP TJA1080 ([29]) in a ring topology. Appropriate bus drivers

have an internal bus designed to carry frames between drivers, and will introduce a small but non–zero delay in frame transmissions passing through them (which should be accounted for in TDMA design).

Since active–star couplers capable of supporting the FlexRay protocol are still extremely difficult to acquire in quantities suited to testbed usage, the choice was made relatively early on in development to implement the safety–critical segment of the testbed using the passive linear bus network topology. As noted above, the FlexRay standard supports the use of a passive bus to interconnect ECUs despite recommending an active–star configuration. As a result, this configuration can be considered representative of a functional FlexRay network, despite the fact that a lack of active star couplers compromises fault–containment and may increase the risk of certain failure modes disabling the network.

5.4.3 Development Kits

Several companies produce FlexRay–capable ECUs, and consequently there is some variability in their hardware configuration. All FlexRay ECUs will consist of a CPU with memory and I/O, as well as a FlexRay communications controller, which mediates communication with the cluster.

The Elektrobit Node<ARM> development kits that are used in this testbed contain a 166MHz ARM9TDMI processor with 64MB of RAM and communications controllers for CAN, LIN and RS232 as well as a small selection of basic digital and analog I/O pins. The FlexRay communications interface is managed by a Motorola IP–module, the control interface for which is mapped into the ECU's memory map. The FlexRay controller (and its integrated physical layer / bus guardian) handles most of the interaction between the node and the cluster, including cluster–time synchronisation, message transmission and reception, and ensuring that messages are only transmitted in the correct TDMA slot (TDMA enforcement). This last is the responsibility of the integrated bus guardian, which will cause the controller to fail silent if it attempts to transmit outside of its allocated slots (as mentioned in 5.4.2).

Unlike the C167–CS development kits, the Elektrobit Node<ARM> contains a minimal operating system to provide features such as pre-emptive task scheduling, message queueing and timing (both wall–clock time and cluster time) and time translation. In earlier models of the Node<ARM>, this operating system was a minimal Linux installation with real–time extensions, and the application

software was built as a kernel module and loaded at run–time (either from local flash memory or from RAM). Later models of the Node<ARM> used a custom–written minimal time–driven dispatcher and memory manager known as "AESTPI" (the Application Execution System and Target Platform Infrastructure), supplied and maintained by Elektrobit: in this implementation, the application software is compiled alongside the AESTPI codebase to produce a firmware image that can again be loaded from flash or from RAM. In both cases, the popular open–source bootloader "u–boot"is responsible for bringing the development kit up, initialising peripherals to a basic level and then transferring control to the application firmware image. This firmware image can be loaded from local flash memory or over the network via TFTP (in which case it must be buffered in RAM). The development kits used to implement the testbed were based on the minimal Linux installation for the first set of tests, but were upgraded to AESTPI for the final tests. This upgrade altered the performance of the cluster for the better, as stated in section 5.6.

Regardless of which operating system is running on a given Node<ARM>, its function is the same. The communications controller is responsible for cluster time synchronisation, TDMA maintenance and message transmission and reception. The operating system is responsible for handling message buffering and providing a simple API to the other peripherals on the device, and the application software is responsible for everything else.

The task of writing firmware for distributed systems is historically very complex, much more so than for an equivalent system based on a single device. The designer must address concerns such as maintaining a global timebase (and with it global state) and passing messages reliably and without corruption. Additionally, the system must maintain reliability and function or fail gracefully as nodes are added to and removed from the network. Since the act of adding or removing a node can fundamentally alter the properties and behaviour of the distributed system, this task can become quite difficult. Fortunately, the Elektrobit FlexRay development environment simplifies many of these tasks by allowing the developer to specify the cluster design and message schedule through a graphical interface, and then automatically generating a code skeleton from that design that implements the required communications scheduling and buffering functionality. In addition to specifying the time–triggered structure of the FlexRay cluster, the design interface goes one step further and allows the designer to specify the time–triggered structure of each ECU within the cluster as a series of tasks, and times within the cluster cycle at which they execute. Thus, the process of designing a FlexRay cluster (and by extension the application that will run on it) is as

follows:

- **Define cluster logical structure.** First, it is necessary to define the logical structure of the application: the tasks that interoperate to produce the application behaviour and the signals that are exchanged by the tasks. At this stage, the physical cluster that will implement this structure is not considered. This stage includes setting some cluster-global properties such as the cluster cycle period - the amount of time it takes for the cluster to cycle once, see section 5.4.1. Once this logical structure (known as a "data–flow graph") is complete, it can be mapped to a physical cluster for implementation. The data–flow graph can be divided into two sub–sections:

  - **Define tasks to implement application.** Each task implements a component of the system behaviour, classic examples being tasks such as "read sensor" and "operate motor". The task specification includes a cluster–independent period and an offset, the former of which specifies the length of time between invocations of a task, and the latter specifies the time period between the start of the cluster cycle and the first invocation. Thus, if the cluster cycle time is 2000μs, the task period is 500μs and the offset 300μs, the task will be scheduled four times in a single cluster cycle (at 300μs, 800μs and 1300μs and 1800μs).

  - **Define messages and interconnect tasks.** Messages are defined much like Tasks, as blocks that have a defined period and offset. The message specification also includes the type and length of the data being transferred, whether it should be a static or dynamic segment transmission, and the redundancy of the message (whether it should be scheduled on one or both channels).

- **Define cluster physical structure.** The structure of the physical cluster is now specified, beginning with the "Cluster" root element (which specifies the properties of the FlexRay backbone buses), and moving on to specify each hardware node, its processor architecture, communications controllers and operating system.

- **Map logical structure to physical structure.** In this stage, the designer's task is to map the data flow graph to the physical cluster. Each task must be implemented on at least one node, but a node may implement as many tasks as can be scheduled to it within the cluster cycle.

- **Generate code.** Finally, the design software can be made to generate a code skeleton for each node in the cluster. This skeleton will include a function template for each Task and an API for sending, receiving and buffering each Message the node sends or receives. The application designer is then expected to write code to implement the application behaviour within each task, then to compile the expanded skeleton together with the object files implementing the operating system, task dispatcher and communications controller/peripheral device drivers. The result is a firmware image that can be directly uploaded and executed on the hardware cluster, assuming that the cluster was defined correctly.

5.4.4 High–Level Application

As noted in the Design chapter (section 4.3.2), there is no over–arching high–level application in either the safety–critical or deterministic networks: rather, there is a minimal framework of tasks and messages which the triggering system can then perturb to collect latency and throughput data.

Although the original design specified that the triggering system should be able to request frame transmission (see section 4.4.2), this will not be possible in the FlexRay cluster. In order to maintain safety–criticality and reliable performance, the operating system of the FlexRay cluster does not permit the message schedule to be altered at run–time. This is for two major reasons. First, locking the schedule during run–time prevents poorly written code from accidentally affecting it. Second, it is necessary to take the communications controller off–bus to update the schedule, and doing so for long enough to complete the update risks the controller losing synchronisation with the bus (and therefore, if the ECU being updated performs any safety–critical task, breaching safety–criticality). Since it is not possible to request arbitrary frame transmission, the FlexRay network will instead by fully inter–connected by messages (each ECU will send at least one message to every other ECU in the network), and arbitrary alterations to message payloads will take the place of arbitrary frame transmission. Each message will contain a sixteen–bit unsigned integer, the value of which can be altered in certain ways by the triggering system (see section 5.5.3, below).

In order to implement this minimal connected framework, the FlexRay cluster is lightly scheduled with a single application task, a single system task and a single communication task assigned to each node. The cluster hardware consists of three Node<ARM> connected in a passive linear bus configuration, with a single bus monitoring device (known as a “BusDoctor”). In addition to

the FlexRay backbone, the Nodes are connected to an Ethernet programming network, which also links the BusDoctor to its monitoring software, allowing events and messages on the network to be observed as they happen.

The communication task consists of code generated by the design tool to read messages from the communications controller's reception buffer into the application inbound message queue, and to move messages from the application outbound message queue in to the communications controller's transmission buffer. If a message is scheduled on the network and is present in the comms controller transmission buffer, it will be transmitted on the network in its specified slot. Likewise, if a message is received from the network and is scheduled to be received by the node, it will be stored in the comms controller reception buffer. As a result, the communications task can be seen as a cluster–synchronisation task, serving as the application programmer's interface (API) to the messages travelling on the network.

The system task functions differently in the two different node operating–systems. In the Linux–based version of the ECU operating system, the system task is used to maintain synchronisation between the node and the cluster, as well as handling memory management and a variety of housekeeping functions. The Linux–based system does not allow the developer to insert application code into the system task, which is an inconvenience that will be addressed in more detail later (in section 6.1.1). In the AESTPI version of the operating system, the system task is a hybrid of generated code and application code, and serves as the primary API to the node hardware. Maintenance tasks such as buffer clearance, synchronising the node timebase with the cluster timebase and other periodic operating–system operations should be included in the System Task implementation, and exist as API calls, but will not be included automatically. In both cases, the system task is invoked whenever no other task is running on the node (but must still be scheduled at a specific point to ensure that it runs at least once on heavily loaded nodes).

Finally, the application task consists entirely of application code, and it is here that the majority of the high–level application behaviour should take place.

All signals in the cluster application contained unsigned 16–bit integers as a payload. Signals connecting nodes 1 and 2 contained an incrementing count (each transmitter incrementing their transmitted value independently once per cluster cycle, and signalling an error if a significantly different number was received from the other). This count is intended to indicate that the cluster as a whole is running, and thus that any lack of response from the Gateway node is a node problem

rather than a network problem if the BusDoctor indicates that the count is still incrementing. Messages coming in from the gateway BridgeLink connection are duplicated and sent on all signals transmitted by the Gateway node. Messages received by the gateway node are deduplicated and sent over the BridgeLink connection.

5.5 Triggering

In this section, the event triggering system that will be used to gather data about the properties of the testbed is discussed. The system hardware is composed of two major elements: a central management node and a small common interface that is present on every ECU in the testbed, each ECU being connected to the management node by a simple standardised messaging interface.

5.5.1 Management Node

The first triggering system component to be addressed is the management node, which node will be based on an x86 PC, for several reasons. Hardware based on the x86 processor is widely available in the commercial sector, as well as being available in hardened and ruggedised versions should the need arise for deployment outside the lab. Additionally, since the triggering system requires a point–to–point link between the management node and every ECU in the testbed, and there are approximately 20 ECUs, a platform with a sizeable expansion bus is desirable. As has been previously noted, the development kits implementing both the safety–critical and deterministic segments have RS–232 interfaces. The particular x86 PC in use on this testbed has a sizeable PCI backplane, and multiport RS–232 expansion cards are readily available in the marketplace. For these reasons, RS–232 was chosen as the implementing technology for the connection between the triggering management node and each triggered ECU. The connection itself is made using conventional serial cables.

The management node uses a lightweight Linux operating system, chosen because it offers direct hardware access and can be configured to be highly deterministic. The operating system consists of a kernel and basic services, and as few extraneous applications and libraries as possible, as every piece of software running on the machine will consume processor cycles and memory that can otherwise be used by the triggering software. Since Linux is configurable down to the kernel

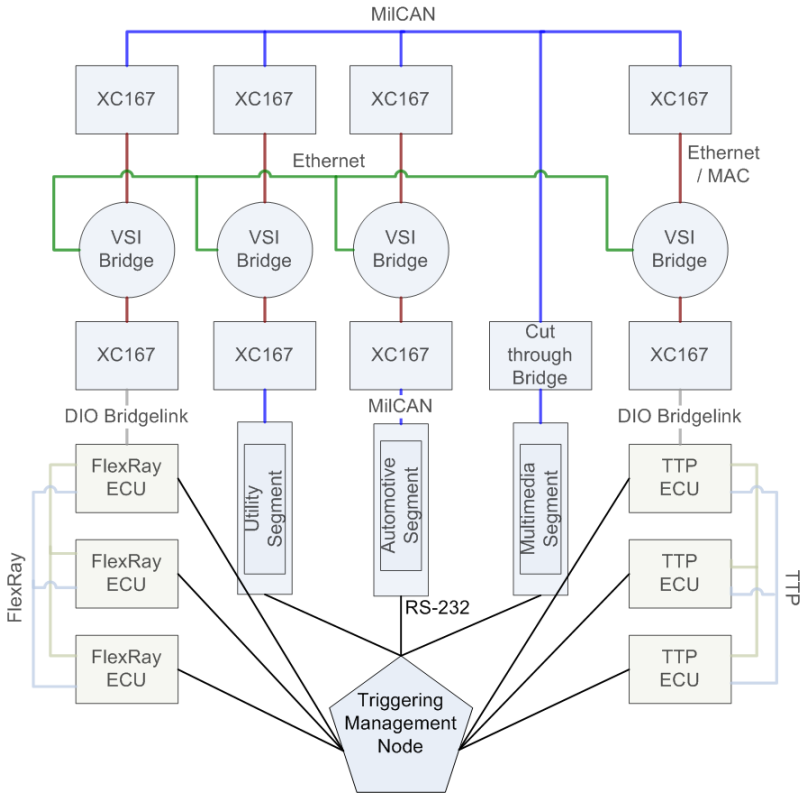

Figure 5.5: VVV Testbed with triggering

level, making it possible to strip out un-necessary elements at all levels of the node software, it is possible to produce a very efficient software image that is responsive and fast. In turn, this allows the triggering control software to respond to requests at the maximum possible speed, with minimal latency.

While it is true that real–time patches are available for the Linux kernel (the RTAI project), the triggering management node does not currently make use of them. Instead, the system was configured for minimal latency of operation. Since the x86 hardware implementing the management node is a modern desktop PC and clocks at approximately 3GHz, and the triggering management application is anticipated to be a very light load for such a machine, it should be perfectly adequate without the added complexity of an RTAI kernel.

A diagram of the hardware configuration of the triggering system can be found in Figure 5.5.

As shown in this figure, the central triggering management node is connected to every node in the deterministic and safety–critical segments. By triggering actions and receiving events it can accurately calculate response times and log message flows as they propagate from node to node

and from segment to segment. Through the generic triggering protocol, testing procedures can be scripted and controlled from the management node without it being necessary to modify the individual ECUs for each experiment. It should be noted that the triggering system will record "raw" timings rather than attempting to subtract the latency introduced by its presence (e.g. propagation delay from the triggering hub to the ECU). Consequently, all measurements presented in the later testing portion of this thesis should be considered upper bounds on performance rather than fully accurate timings.

Finally, the triggering management node will run an application that is capable of sending and receiving messages over the array of RS–232 ports, and uses the system Real–Time Clock to timestamp operations. This application will have a simple script interpreter as a user interface, allowing tests to be scripted as sequences of operations to be performed and events to be monitored, and will provide the behaviour for the management node.

5.5.2 ECU Interface

The other major component of the triggering system is the component that will run on the ECUs. While this component will necessarily be a little different for each ECU architecture, all versions of it will share defining characteristics and a functional interface, as follows. The ECU component will be as simple as possible, being designed to be polled quickly and frequently by the management node. It will use a simple, standardised interface that allows for short message lengths while still retaining a degree of flexibility (in this case, RS–232). All communications between the ECU and the management node will be based on single–character (8 bit) messages once the setup stage is complete, since a single character is the shortest message that can be transmitted over a standard RS232 connection while still offering a range of 256 different messages for use across the testbed.

It will not be possible to modify the ECU component of the system at run–time, since the additional complexity required to make this possible would risk slowing down the ECU–component execution more than is acceptable in a timing–based system. Instead, any modification to event or message assignments within the ECU component must be made at compile–time, and the resulting firmware image uploaded to the relevant ECU. Fortunately, it should not be necessary to modify the ECU component during the course of these tests, as the predefined message set should encompass all the atomic operations to be performed. Modifications to the test schedule, transmitted and received

messages and node properties that can be manipulated by the triggering system can all be made simply by altering the script executed by the management node so that it requests ECUs to perform different sequences of these atomic operations.

5.5.3 Operating Procedure

The triggering system operates in two discrete modes during a normal testbed run. In Setup mode, the management node sends a "ping" message to every attached ECU in sequence, to which the ECU should respond immediately with a "pong" message. It is imperative that this message exchange takes place while the testbed is running, as the latency of this operation will be used to establish the approximate end–to–end latency of the link between the management node and the ECU: if the ECUs are not performing their normal duties during the exchange, the result will not be representative of actual cluster conditions. Once message transfer latency has been measured for each link in the triggering system, the system can transition into operating mode.

In operating mode, the management node sends a sequence of messages (in the form of single characters, one per message) to trigger a response in the attached ECU. This response may be a message–transmission event (or rather, an alteration to a periodically transmitted payload, as noted in section 4.4.2), a request to clear buffers, or one of a small set of local state modifications (listed in section 4.4.2). Once the event has been processed, the ECU will respond with an acknowledgement if appropriate: positive if the event completed successfully, negative if it did not. Receiving a message loopback request and receiving an unrecognised triggering event are the only two situations in which a positive acknowledgement should not be generated. In the case of a loopback request (a "ping") the system should respond with an identical loopback request (a "pong"). In the case of an unrecognised triggering event, a negative acknowledgement should be generated to indicate a problem.

If an "on–reception" event is triggered on the ECU (caused by its receiving a payload that matches one of a set of conditions hard–coded in the application), it will send a message to the management node indicating receipt. It is thus possible to track a message through the network, by setting up reception triggers in the nodes on the expected route, then triggering a message transmission and timing the firing of each reception trigger.

There are five possible payload status changes that can trigger a message being sent to the man-

agement node from the receiving node if they arrive in a local network message, each one corresponding to an operation in the message transmission command set:

Payload Zeroed The uint16 in the payload holds 16 zero bits, where before it held something else.

Payload Oned The uint16 in the payload holds 16 one bits, where before it held something else.

Payload Toggled The uint16 in the payload holds the bitwise inverse of the value most recently received.

Payload Incremented The uint16 in the payload holds a value that is the result of adding 0x1 to the previous value.

Payload Decremented The uint16 in the payload holds a value that is the result of subtracting 0x1 from the previous value.

It should be noted that there is a sixth possible response, that of "Error Detected". This response is generated either if the values received from both channels do not match (as all messages in the FlexRay cluster are bus–redundant, a mismatch indicates corruption), or if an unexpected value is received. An unexpected value is classified as a value that alters the payload without conforming to one of the five cases above.

5.6 Proof of Concept

The connection between the safety–critical and deterministic segments of the testbed was iterated three times during development until a final, acceptable solution was found. The initial prototype used CAN as a linkage, bridging the FlexRay ECU directly to a C167–CS without involving the VSI Bridge. The second iteration used a byte–wide digital I/O interface to connect the FlexRay ECU to an XC167 and on from there to a VSI Bridge, which was linked to the Ethernet backbone. The third and final iteration moved the FlexRay–Ethernet translation into the FlexRay ECU, cutting the XC167 out of the link between the FlexRay ECU and the VSI Bridge entirely. Each of the three iterations will now be examined in more detail: the CAN proof–of–concept in the rest of this chapter, and the two more complete iterations in the following two chapters.

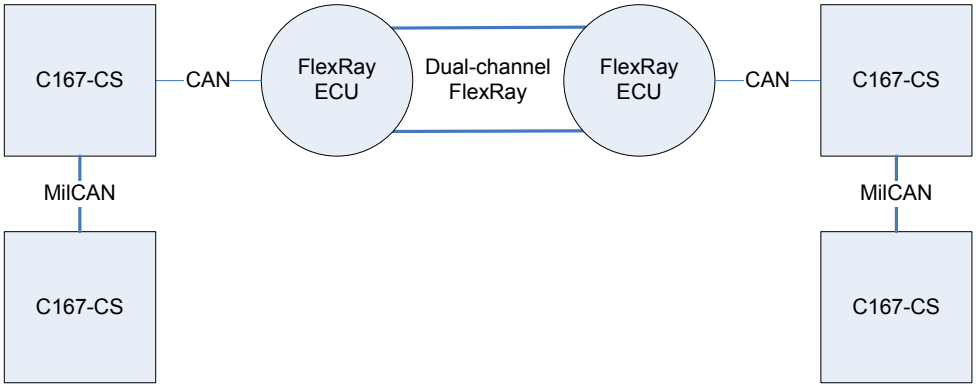

Figure 5.6: CAN proof of concept demonstrator

5.6.1 CAN proof–of–concept

The first implementation of message bridging between MilCAN and FlexRay was a simple proof of concept rather than a fully featured testbed (described more completely in [30]). The intent was to explore the capabilities of the platforms and to investigate whether or not it would be possible to transfer messages between MilCAN and FlexRay at an acceptably fast rate. The test configuration was comparatively simple: a pair of MilCAN subnets, each containing two nodes, was established and a FlexRay network of two nodes was placed between them. The configuration can be seen in figure 5.6.

As can be seen from the figure, the proof–of–concept demonstrator was somewhat complex in configuration, despite its simple intent. A total of three protocols were used over five segments: MilCAN in the source and sink networks, FlexRay in the transport network and CAN to interconnect the two protocols. This slightly peculiar network configuration was necessitated by the fact that at the time the demonstrator was constructed, the research group possessed a total of two, Linux–based, FlexRay ECUs, but a reasonable number of MilCAN ECUs. Since two is the minimum number of nodes in a functioning network in the case of both MilCAN and FlexRay (a single–node "network" will fail to initialise), the FlexRay network was placed in the position of Transport network, carrying bridged messages between left and right MilCAN networks. This configuration has the additional, coincidental advantage that it allows the higher–bandwidth network to transport messages rather than generating them, removing the risk of transport network congestion (since the FlexRay network has a bandwidth capacity an order of magnitude greater than that of the MilCAN network, this is not an insignificant concern).

The operating sequence of the demonstrator is as follows:

- Each MilCAN network maintains two message exchanges, one “local” and one “remote”, each exchange being composed of one message transfer in each direction. The Local message (MID 0x3FA & 0x3FB on the left–hand network, 0x3FE & 0x3FF on the right) is incremented and retransmitted each time it is received, while the Remote message is simply retransmitted.
- When the MiLCAN gateway node receives the Local message, it copies the message in its entirety to the connected FlexRay node over CAN.
- The FlexRay network transports the content of the message, along with an incremented sequence number to indicate that new data is present.
- When the FlexRay gateway node receives a message containing new data, it copies the message to the connected MilCAN node over CAN.
- The MilCAN gateway node places the contents of the received message into the Remote message on the destination network and transmits the message.

The visible result of this operating sequence is that both message exchanges are up to date and synchronised between the two MilCAN networks, ignoring the slight transmission delay caused by the message data traversing the transport network. This delay should be a maximum of one MilCAN Sync Cycle under normal conditions: if the frame traverses the transport network in less than one MilCAN PTU, it may be transmitted in the same cycle, otherwise it will be held and transmitted in the next one. Note that frames entering a MilCAN network via the gateway overwrite the contents of their destination frame on arrival, so a frame that is held for transmission will not be corrupted by the arrival of an out–of–date message from the opposite node in the network.

Bus diagnostic equipment (in this case, the Vector CANoe software) confirmed that messages were transferred successfully from one MilCAN network to the other. However, the demonstrator suffered an unexpected periodic failure: after approximately 300 message exchanges, the FlexRay network began to corrupt the message payload. This was eventually traced back to a known flaw in the CAN controllers used on the FlexRay development kits: although they are rated at up to 1Mbps, the silicon version in use is not capable of sustained correct operation at that speed. Since basing a testbed on equipment known to be flawed is a poor decision, the use of the development

kit CAN controllers was removed from the list of possible implementation choices, despite the success of the testbed in proving that message transfer was possible.

Indicative results from the testing of this hardware configuration can be found below.

For the purposes of the CAN–based prototype, all that was needed was a provably working system with a delivery latency within acceptable bounds. The Vector CANoe network analysis package was used to monitor frame transfers on both networks. Since CANoe timestamps all reception events with a microsecond granularity, the transfer time can be found quite accurately by subtracting the transmission timestamp (on the source network) from the reception timestamp (on the sink network). Given the unloaded state of the network, all message latencies measured during the course of this test should be considered "best–case" performance.

Five independent trials were performed, yielding an overall mean average end–to–end latency of 3.54ms, and a standard deviation of 0.0928. Given that the FlexRay cluster has a cycle time of 2ms, and that the bridging process requires one FlexRay cluster cycle to complete, this is an acceptable proof–of–concept for FlexRay – MilCAN bridging. Unfortunately, as noted, the CAN controllers of the FlexRay nodes were not capable of sustained operation at the necessary speed: while the devices are capable of supporting a bridging system, a different communications technology will required in a production implementation. Knowing this, it was possible to continue with the research and to design a fully–featured bridge, as will be seen in subsequent sections.

5.7 Conclusion

In this chapter, the safety–critical and deterministic networks referred to in the Design chapter have been mapped to specific technologies and development kits, and the implementation of the triggering system has been described in some detail. The implementation of both the triggering master node (an x86 PC running a minimal Linux with a simple script interpreter and a large number of RS–232 ports) and the ECU–specific triggering interface has been discussed. The construction of a CAN–based proof of concept testbed has also been mentioned.

It is now possible to present a description of the implementation and testing of the full bridging testbed, first with a cross–platform Bridgelink connection based on fast parallel I/O, then with a faster and more efficient Ethernet–based Bridgelink connection.

Chapter 6

V&V Testbed - Testing

Introduction

In this chapter, the implementation and testing of two successive iterations of the V&V Testbed is discussed. In the initial configuration, the FlexRay segment of the testbed is assembled from ECUs using the Linux–based operating system, and communicates with the VSI Bridge using the Bridgelink protocol over byte–wide parallel I/O with the aid of an XC167 interface mediator, as described in chapter 4. As will be shown, the parallel interface is reliable over a large number of trials and easily adaptable for use with different technology mappings, but suffers from low overall performance. It should be noted that the Triggering System used to stimulate the testbed and collect results in this testing sequence was not a full implementation, and only integrated completely with the VSI Bridges and MilCAN network, necessitating the measurement pattern seen below in which all measurements start and end either on the deterministic subnetwork or in the VSI Bridge.

In the final configuration, the layout of the testbed is altered. The FlexRay ECUs are switched over to a lighter operating system ("AESTPI", produced by the manufacturers of the development kits as a low–latency API over the hardware). The Bridgelink connection and its associated interface mediator are removed from the link between the FlexRay gateway and the VSI Bridge: since the

FlexRay development kits used to compose the FlexRay segment possess an Ethernet port, the decision was made to implement the entire bridging system over Ethernet. As will be seen in the Bridgelink test series, the link between the XC167 and the FlexRay gateway appears to be the largest source of latency in the testbed by a fair margin. The Ethernet–based implementation (referred to as "Ethlink" from this point onward) should remove the primary source of latency from the connection while maintaining the original Bridgelink API: the only indication that a testbed is running in one or other configuration should be the difference in bridging speed.

6.1 Initial Configuration (Bridgelink)

6.1.1 Testbed Configuration

Since the proof of concept has demonstrated that the target platforms are capable of performing basic translation at an acceptable speed, development can now proceed to implementing the Bridgelink design as laid out in section 4.5. The hardware configuration was as specified in Figure 6.1.

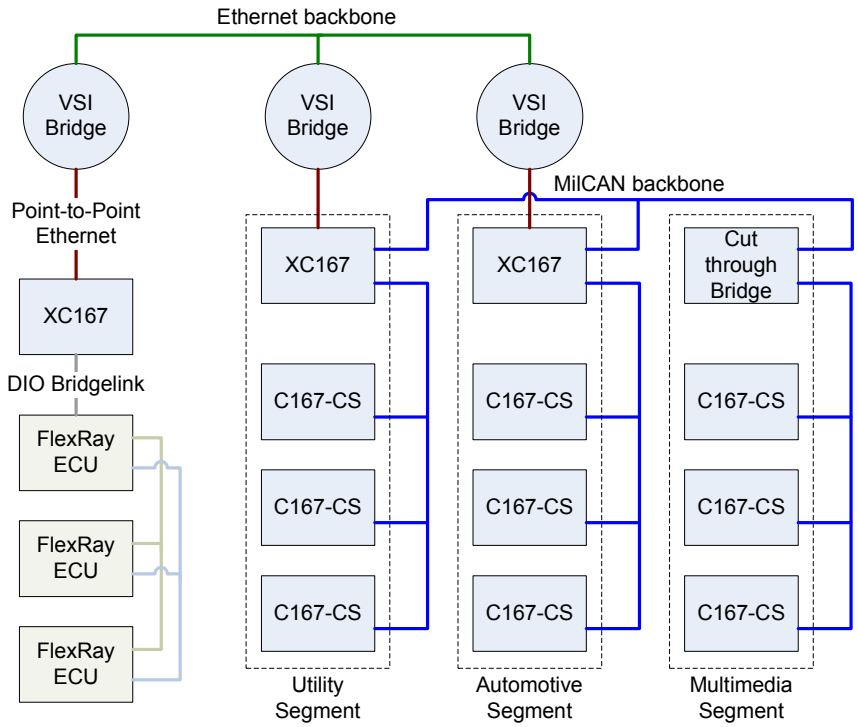

Figure 6.1: First iteration of the Bridgelink testbed

The MilCAN network is configured exactly as specified in section 5.3 (figure 5.2), as three subnetworks interconnected through a MilCAN backbone (via a pair of VSI Bridges and a Cut–Through bridge). The redundant MilCAN network referred to previously as MilCAN Backbone 2 has been elided from this diagram, as the triggering application on the MilCAN cluster routes all messages solely through MilCAN 1.

The FlexRay cluster is configured for dual–redundant linear–bus operation and a cluster cycle time of 2000μs, and contains a total of three nodes plus one BusDoctor (also elided from the diagram, as it has no relevance to the testbed configuration). One FlexRay ECU is designated the network gateway, and is connected to an XC167 interface mediator by 24 channels of one–bit digital I/O. These channels are bound together as a Bridgelink physical layer, with eight data lines and four control lines in each direction, and communication over them proceeds using the Bridgelink protocol exactly as specified in section 4.5. The cluster application is configured according to the framework laid out in the High–Level Application section (section 5.4.4), with three tasks per ECU and two channel–replicated messages transmitted per ECU, one to each of the other two. The gateway node reads the content of messages transmitted to it and exports said content over the Bridgelink connection whenever its value changes. While the gateway firmware supports the transfer of messages both out from and in to the safety–critical network, it is not wise to transfer messages in to the network without detailed checking, as a corrupt or out–of–specification message risks compromising the criticality of the network. Nonetheless, the testbed exercises both input and output paths to the safety–critical network in the following test series, in the interest of completeness.

The XC167 interface mediator is connected to a VSI Bridge as might be expected, using a point–to–point Ethernet connection. This connection uses simple Ethernet MAC frames in accordance with IEEE802 to encapsulate NL–P frames from the Bridgelink connection and forward them to the Bridge, there to be routed to the Bridges in the MilCAN cluster (via the Ethernet Backbone that connects the Bridges to each other). Since Ethernet frames have a minimum size of 64 bytes (including header and payload), the frame is zero–padded to minimum size if necessary.

It has been noted that the Linux–based version of the development kits (from which this version of the testbed was created) does not support the insertion of application code into the System Task, and that as a result, the great majority of the application support code (including the bridging and triggering systems) was present in the application tasks instead. This caused several problems,

foremost of which was that Bridgelink transactions could only proceed when the application task was executing. Since the TDMA schedule is non–preemptive, if the Bridgelink transaction took longer to complete than the allocated worst–case execution time of the task, the node scheduler would crash and cause the node to halt. This problem was eventually solved by limiting the number of bytewise transfers the Bridgelink layer was permitted to make in each application task invocation to 8, meaning that a Bridgelink DLL frame would be transferred in four consecutive executions of the application task. While this will introduce additional latency into Bridgelink operations (the FlexRay cluster cycles on a 2ms timebase, so 8ms per frame), the fact that the triggering system is not capable of requesting transmission this frequently means that the additional latency will not adversely affect the testbed's operation. An alternative solution would be to reschedule the application task with a worst–case execution time long enough to encompass the transmission of a single Bridgelink frame, but this would likely require the entire cluster to be rescheduled and the cluster cycle time extended, again resulting in additional latency.

Additionally, it should be noted that the Triggering System as discussed in section 5.5 was not in full operation in this testbed configuration, as it was still under heavy development at the time these results were taken. Consequently, the tests in series 1 (Bridgelink latencies) were timed by the VSI Bridge host, although tests in series 2 (Bridge latencies) were timed by the triggering system (since the MilCAN ECU triggering component was in a usable state).

6.1.2 Testing Protocol

The following tests were performed:

1. Bridgelink latencies
    (a) VSI Bridge to XC167 interface mediator
    (b) VSI Bridge to FlexRay Gateway
2. Bridge latencies
    (a) MilCAN network direct loopback
    (b) MilCAN network Bridge loopback

The first test series (Bridgelink latencies) is intended to find the latencies present in the Bridgelink system itself (hardware and software components) by measuring the time taken for a packet sent over Bridgelink to loop back to the sender. The first test measures the latency of the point to point raw Ethernet link between the interface mediator and the VSI Bridge. The second test measures the time taken for a packet to travel from the VSI Bridge to the interface mediator, onward to the Gateway node (over Bridgelink) and back to the interface mediator and Bridge. The result from the first test can then be subtracted from the result from the second to find an approximation of the latency of a DIO–Bridgelink connection.

The second test series (Bridge latencies) involve the transmission of a message on a given network which is then picked up by the gateway node, forwarded on to the bridging system, then looped back to the transmitting network. The latency of this operation is measured for later analysis. The first version of this test performs the loopback operation in the interface mediator, using a simple routine that reflects any inbound message back through the port by which it entered. The second version of this test performs the loopback in the VSI Bridge itself, making use of the Bridge's internal routing and filtration system to send the message back on its source interface.

Once both test series are complete, the results will undergo analysis to piece together an approximate figure for the end–to–end latency of the bridging system. Since the tests in test series 1 and 2 cover the entire bridging system, it will be possible to assemble a timing approximation for message transfer from the MilCAN gateway, through the bridging system to the FlexRay gateway and back to the MilCAN gateway.

6.1.3 Results

Bridgelink latencies

This subsection shows the results from test series 1. These tests study latencies in the connection linking the VSI Bridge to the network gateways on each vetronics network. In all the tests in this series, the nodes in the testbed were members of a running cluster of the appropriate protocol, and handling their protocol responsibilities in addition to the test procedure. It should be noted that the VSI Bridge software is **not** running on the VSI Bridge host during these tests: instead, a simple script was used to transmit and receive NL–P frames without the overhead of the VSI Bridge, since the objective was to gather results for the communications protocols, not the Bridge software (see

also section 6.1.3, below).

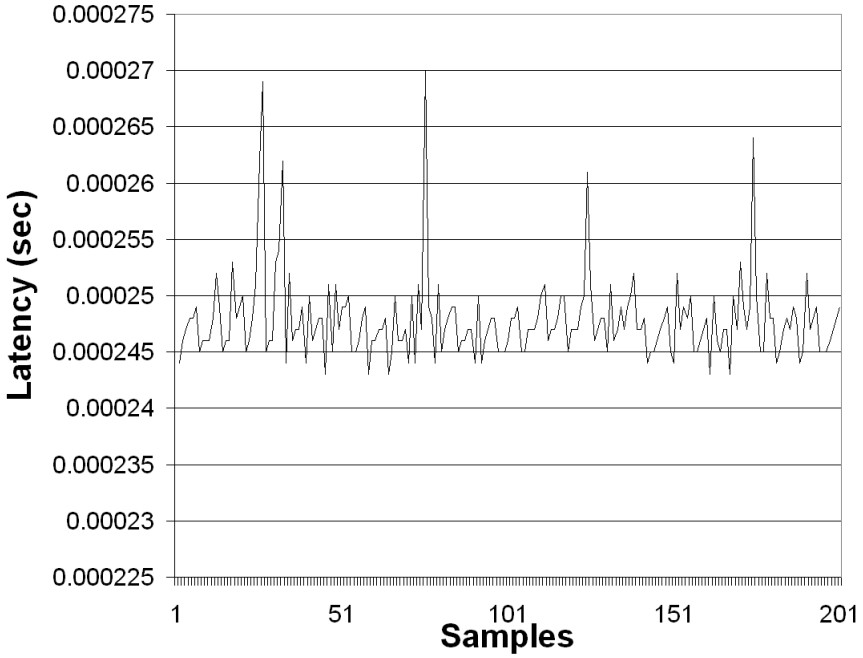

Figure 6.2: Test 1.a. Loop latency from VSI Bridge host to XC167

Figure 6.2 shows the results of test protocol 1.a. The COTS x86 PC hosting the VSI Bridge software emits a Bridgelink NL–P frame, which is wrapped in an Ethernet frame and sent to an XC167 interface mediator (which XC167 of those available is immaterial, as they are all identical hardware, running identical software). The interface mediator reflects the message back to the VSI Bridge, where its reception is recorded. The y–axis of this graph shows the round–trip times for these NL–P frames over 200 trials. This test studies the latency of the link between the VSI Bridge and its associated interface mediators. The mean average frame latency was 0.000248 seconds (0.24ms), with a standard deviation of 3.7 $\times 10^{-6}$.

Figure 6.3 shows the results of test protocol 1.b. In this test, the VSI Bridge host emits an NL–P frame as before, transmitting it to the interface mediator over Ethernet. Instead of reflecting the frame back, however, the interface mediator retransmits the message over the Bridgelink/DIO link to the FlexRay gateway. The FlexRay gateway then reflects the frame back to the bridge by the same route. This test studies the latency of the connection between the FlexRay gateway and the VSI Bridge. Four individual sources of latency contribute to the latency of this connection:

- The Ethernet link between the VSI Bridge host and the interface mediator.

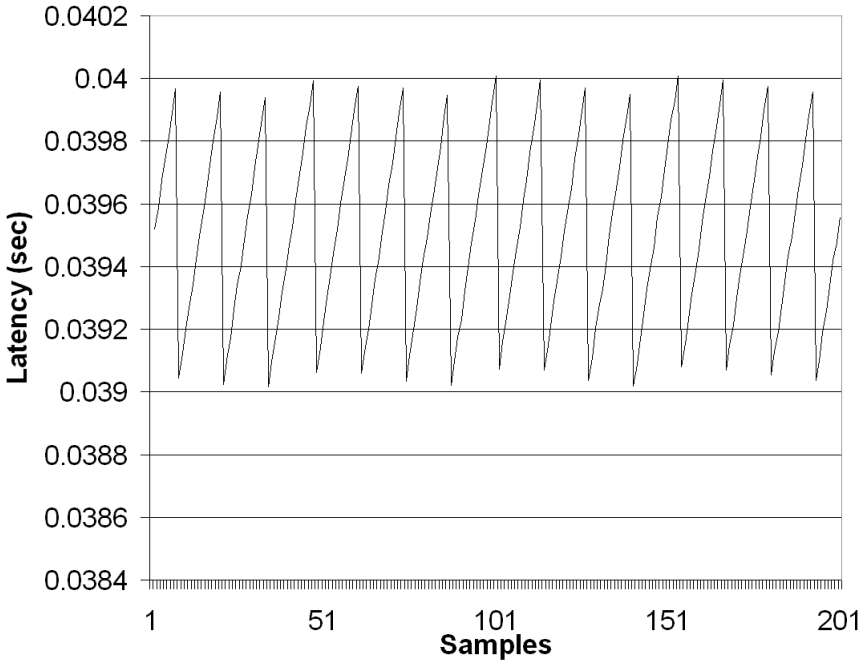

Figure 6.3: Test 1.b. Loop latency from VSI Bridge host to FlexRay gateway

- The interface mediator frame–retransmission firmware.
- The Bridgelink/DIO link between the interface mediator and the FlexRay gateway.
- The software of the FlexRay gateway.

The mean average frame latency in this test was 0.0395s (39.5ms), with a standard deviation of 0.000289. This is a substantial increase in latency from the previous test, and since the first two latency sources were also present in the previous test, it can be assumed that the additional delays originate in the Bridgelink/DIO connection and in the FlexRay gateway software. When performed on the TTP subnetwork, this test indicated a loop latency approximately 4ms, further indicating that the additional delay is likely to be in the FlexRay gateway software and the FlexRay–segment implementation of the Bridgelink protocol.

Bridge latencies

This second test series is designed to study the contributions made by the VSI Bridge to overall system latency.

In Figure 6.4 (test 2.a.), the MilCAN gateway sends a Bridgelink frame over Bridgelink/DIO to the

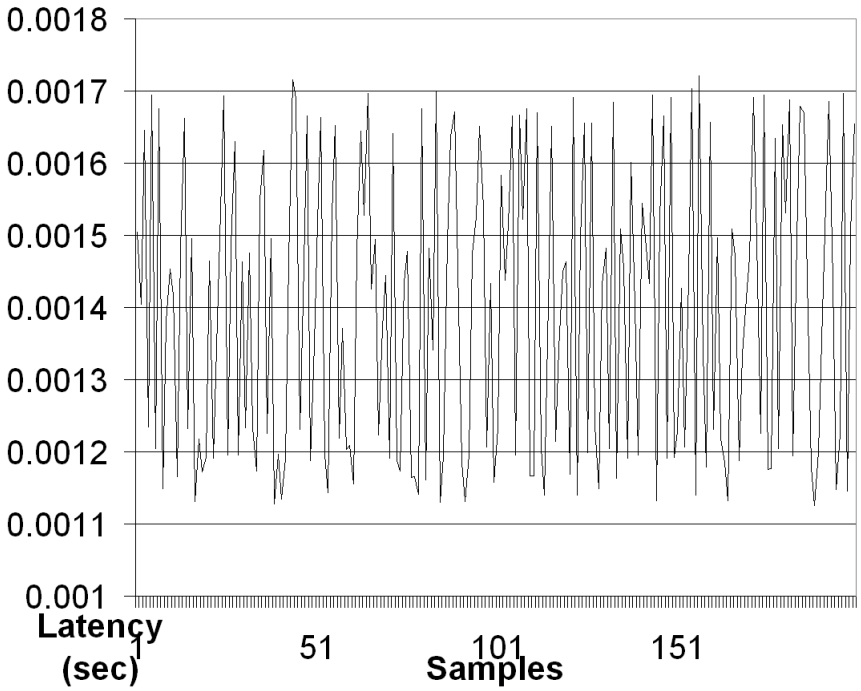

Figure 6.4: Test 2.a. MilCAN loopback, via VSI Bridge host only

XC167 interface mediator, from where it is forwarded on to the x86 VSI Bridge host. In place of the VSI Bridge, the host is running a small piece of in–house software that receives the frame and immediately reflects it back out through the port through which it arrived. This test is very similar to the one that produced Figure 6.3, in which the VSI Bridge host and the FlexRay gateway are the end stations. As before there are four sources of latency: the Bridgelink/DIO link, the interface mediator firmware, the Ethernet link and the reflecting software. As can be seen from the graph, the mean average frame latency when the VSI Bridge software is not running is around 0.0014s (1.4ms).

Figure 6.5 depicts test 2.b., which is almost identical to the previous test. The only difference is that in test 2.a, a small port reflector reflected inbound frames on their arrival ports. In test 2.b, the VSI Bridge software is running and configured to loop inbound frames through its internal routing system. As a result, these two tests (when considered together) should indicate the latency introduced by the VSI Bridge software itself. When the two sets of results are compared, it can be seen that the VSI Bridge introduces only a small amount of additional latency to the frame loop operation, approximately 200μs. While this amount can be expected to vary slightly depending on system load and the total amount and type of traffic being processed by the bridge, it is unlikely

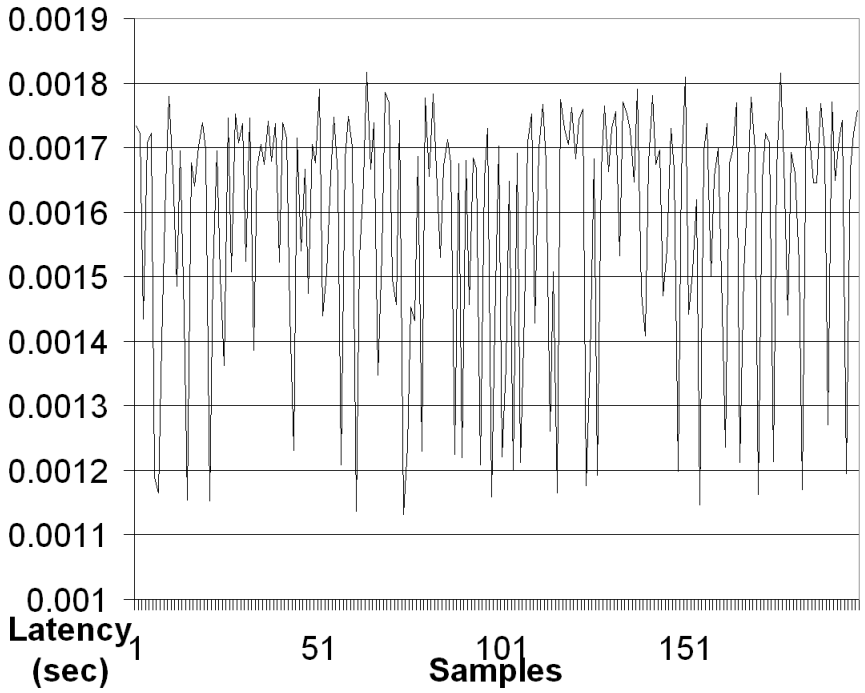

Figure 6.5: Test 2.b. MilCAN loopback, with VSI Bridge software routing

to significantly affect results, given the fact that the Bridgelink/DIO link has been indicated as the source of the great majority of network latency in the connection.

End–to–End latencies

It is possible to calculate the approximate round–trip latency of the FlexRay/MilCAN internetwork using the results above. The latency from the FlexRay gateway node to the VSI Bridge averages 39.5ms, and from the MilCAN gateway to the VSI Bridge averages 1.5ms. As before, the XC167–VSI Bridge link has an average latency of 0.25ms. Adding the segment latencies together, one can observe a round–trip latency of 41ms, without any FlexRay and MilCAN network latencies. This analysis is represented visually in figure 6.6.

6.1.4 Analysis

From these results, it is apparent that the majority of the delay in the FlexRay–MilCAN bridge demonstrated here is in the Bridgelink connection between the FlexRay gateway node and the XC167 interface mediator. However, since the Bridgelink connection on the FlexRay side of the network takes 39.5ms to cycle (a little under one twenty–fifth of a second), and the Bridgelink

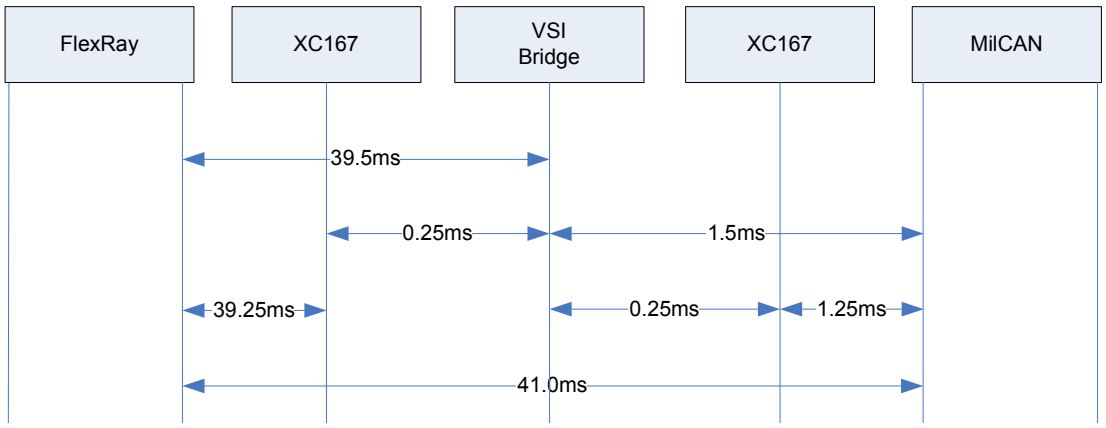

Figure 6.6: FlexRay to MilCAN end to end round–trip average latencies breakdown

connection on the MilCAN side of the network takes only 1.5ms to cycle, it would seem apparent that the Bridgelink connection itself is not at fault. Rather, it is likely that the operating system of the FlexRay gateway ECU is causing a significant delay due to the fact that the ECU Bridgelink component can only be executed in the application task, a small fragment of the overall cluster cycle. Furthermore, in order to prevent the application task from over–running its allocated time, it was necessary to limit the Bridgelink component so that it would only transmit 8 bytes at any one time (see section 6.1.1), causing an additional 8ms delay in each direction (since the cluster must cycle 4 times to transfer a 32–byte DLL frame. This limitation alone accounts for almost half of the latency in the FlexRay Bridgelink implementation.

This implementation is serviceable, reliable and portable and may be perfectly acceptable if the bridge is being used only for monitoring of the safety–critical segment, even with a constant delay of 41ms (obviously, if a resolution greater than 1ms is required, variations in bridge and triggering latency will cause the result to be degraded). If the bridge is being used to allow the safety–critical segment to control devices on the deterministic segment, it seems less likely to be a workable solution. It should be noted, however, that 40ms is at or below the typical human visual perceptive limit (persistence of vision occurring at between twenty and twenty–five events per second). Consequently, if the message being bridged originates from the crew–station interface, a 40ms message latency may be perfectly acceptable.

6.2 Final Configuration (Ethlink)

6.2.1 Testbed Configuration

In the previous testbed configuration, the safety–critical segment gateway was connected to the VSI Bridge using an interface mediator, which translated the Bridgelink PIO messages into point–to–point raw Ethernet frames. This interface mediator has now been removed, and the connection between the FlexRay Gateway and the VSI Bridge is run directly over Ethernet, making use of the Ethernet controller present on the FlexRay Gateway. A diagram of the new testbed configuration can be seen in figure 6.7. In addition to the removal of the interface mediator, the operating system on the FlexRay nodes has been upgraded to Elektrobit's latest release, as described below. The configuration of the testbed is otherwise the same as in the previous tests.

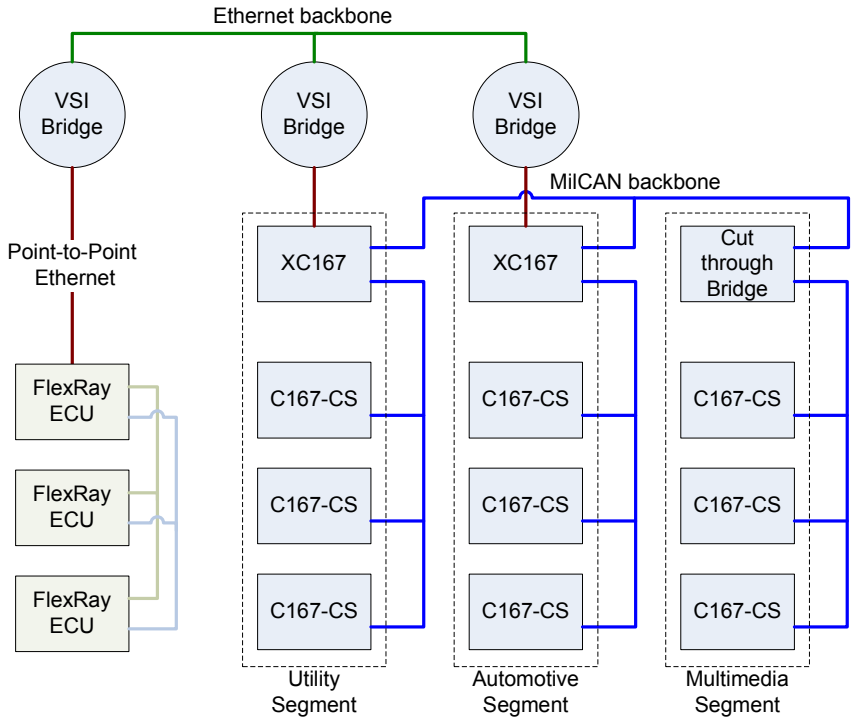

Figure 6.7: Final iteration of the Bridgelink testbed

The changes to the connection between the FlexRay Gateway and the VSI Bridge require that both the physical layer and the data–link layer be re-written. The Bridgelink PIO design fragmented the network–layer packet into data–link layer packets on the FlexRay Gateway, before passing them over the Bridgelink PIO connection to the interface mediator where they were reassembled into a network–layer packet, encapsulated in an Ethernet frame and sent to the VSI Bridge. It is

wasted effort to fragment and reassemble the network–layer packet without transmitting it in its fragmented state, so the Bridgelink Ethernet system simply encapsulates the network–layer packet directly into an Ethernet frame and transfers it to the VSI Bridge. The original network–layer API is preserved, for portability, meaning that the application running on the FlexRay segment need not be modified in any way, merely linked with the new version of the bridging library to enable Ethernet–based communication.

Operating System

In addition to altering the links between segments, this second testbed configuration also alters the operating system in use on the Flexray nodes. In place of the real–time Linux used in the initial Bridgelink/DIO tests, the FlexRay nodes now run a custom–written minimal operating system named AESTPI, distributed by Elektrobit.

The Application Execution System and Target Platform Infrastructure is a minimal operating system written specifically to target the hardware configuration of Elektrobit FlexRay nodes. AES is a simple interrupt–based time–driven task despatcher that allows "tasks" (application elements) to be scheduled to run at particular times in the FlexRay segment absolute time cycle. TPI, by contrast, is a driver layer over the peripherals of the node, allowing the application to configure the FlexRay controllers, send and receive messages and operate the serial port and I/O without having to work at the bare–metal level. It provides basic buffering, type translation and a user–friendly API. The two components form a skeleton within which a series of user–written tasks can interoperate to implement the required application, as was the case for the previous Linux–based operating system. The Linux–based system, however, was significantly more complex and restrictive in the operations that tasks could perform (it was not possible to schedule tasks to run in the Idle Time, the time in which the node is otherwise not doing anything, for example). Additionally, the Linux–based system required more upkeep time per application cycle than the AESTPI system does, meaning that there is now more processor time available to the user application. The AESTPI operating system is not certified for deployment in safety–critical applications, but serves as a workable infrastructure for performance testing: in a real–world use of the testbed, the safety–critical segment would need to make use of whatever operating system is specified for the vehicle being modelled.

The substantially simpler nature of the new operating system, combined with the fact that it is now

possible to execute application code in the idle time (not possible in the Linux–based version), means that it was possible to reduce the FlexRay cycle time by almost half (to 1100μs) and still transmit an entire Bridgelink frame in each direction in each cycle. A shorter application cycle is desirable where possible, as it means the system will be more responsive (there will be less elapsed time between consecutive executions of a given application task). It should, of course, be noted that the Ethernet driver does not necessarily transmit at that rate due to delays in the memory allocation and data transfer processes within the node: messages are queued and dequeued in the application task, and the Ethlink system (as the Ethernet–based implementation has been named) transfers data continuously between message queues in the gateway ECU and the VSI Bridge in the idle time, but a direct correspondence between FlexRay cycles and message transfers cannot be assumed.

The application task schedule used in this final version of the FlexRay segment application is shown in figure 6.8, with their connecting signals. Each node has three scheduled tasks: the Application, System and Communications Tasks. In the Communications Task, inbound message data is read from the FlexRay controller's internal buffers and placed in the application buffers, and outbound messages are flushed from the application buffer to the FlexRay controller. The Application Task contains the application logic, operating on messages received and data from peripheral sensors to control peripheral actuators and transmit messages. Finally, the System Task is used by the node operating system to synchronise its internal timekeeping with the FlexRay network and to do any necessary housekeeping, as well as being used by the Ethlink system to transfer data to and from the VSI Bridge. The System Task retriggers continuously whenever no other task is active on the node (being interrupted whenever necessary so that other tasks can run), but is guaranteed to execute at least once at the specified point. It should be noted that signal transmission and reception points are not marked on this diagram, but that they were scheduled around the communications task so that signal reception occurred shortly before the task and signal transmission shortly after. Consequently, the data that is passed to the application from the communication task is as close a reflection of bus state as is possible (and indeed vice versa).

Alteration to Bridgelink design

The Ethlink protocol consists of Physical Layer, Data Link Layer and Network Layer specifications, as did Bridgelink before it. The Physical Layer defines the wire protocol used to carry data

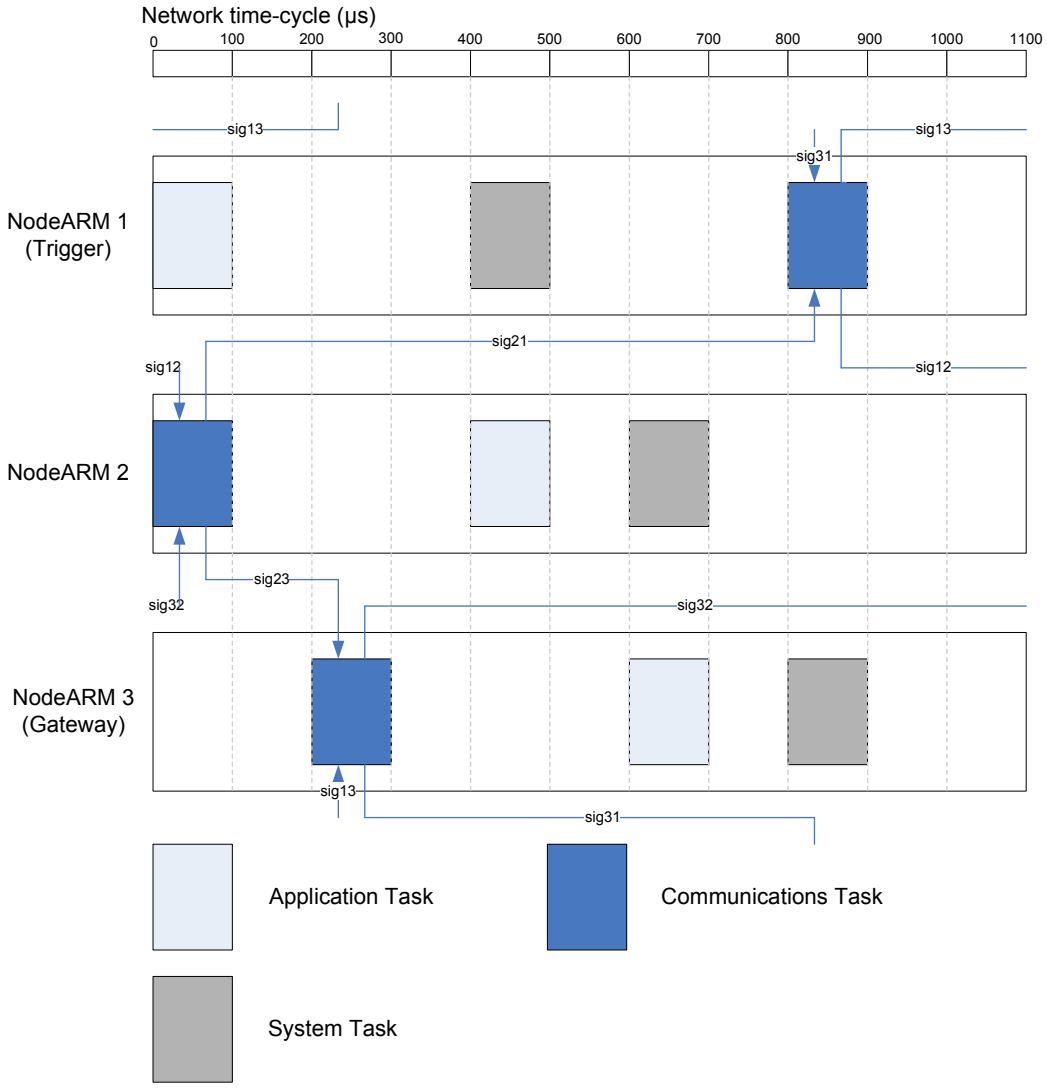

Figure 6.8: Application skeleton for the FlexRay cluster

between the gateway node and the VSI Bridge, in this case MAC/Ethernet. The Data Link Layer specifies a software protocol intended to detect lost data and retransmit it. Since the Ethernet link is a point–to–point link between the two nodes, frame loss should not occur: nevertheless, provision is made for it, as no link is perfect. Finally, the Network Layer specifies how data is encoded in the frame payload, and is identical to the previous Bridgelink implementation.

Ethlink Physical Layer

The Ethlink physical layer is a standard type–II Ethernet frame, carried over a point–to–point CAT–5 cable. The Ethernet frame consists of a six–octet destination address, a six–octet source address, an Ethertype (0x88b6 in all Ethlink frames, a reserved value in the IEEE802.3 standard for local experimental use), the payload (padded with zeroes to a minimum length of 46 octets as required by the standard) and a four–octet CRC. The CRC is calculated and checked by the integrated circuitry implementing the MAC layer of Ethernet: Ethlink is not concerned with it, but should never receive frames that do not have a valid CRC.

Ethlink Data Link Layer

In the Bridgelink implementation, the data–link layer was primarily concerned with fragmentation and error recovery. Fragmentation is not necessary in Ethlink, as the network–layer packet is deliberately sized to fit in a single Ethernet frame, but error recovery is still a concern. To this end, the data–link layer frame consists simply of a four–octet header inserted at the beginning of the payload. This header contains a sixteen–bit payload length (equal to the application data length plus four octets, to allow space for said header), an eight–bit sequence number and an eight–bit frame type. The sixteen–bit payload length limits frame payload lengths in this version to 65535 octets, but since the maximum payload size of an Ethernet frame is 1500 octets and the payload length is never greater than seven octets in the current implementation, this will not be a concern. The sequence number increments once for each frame sent, and wraps without incident on overflow, since if is used mainly for packet loss detection and 256 frames can never be on the bus simultaneously. Finally, the frame type should be either FT_DATA (0x01) or FT_DACK (0x02), the former indicating that a frame contains valid data and the latter being an acknowledgement of the sequence number indicated in the DACK packet.

Ethlink Network Layer

On the network layer, the data is simply an array of octets, the meaning of which is entirely irrelevant to the Ethlink system. In the current testing application, the first octet indicates the semantic content of the message (the signal being bridged) and the following two octets contain the content of that message.

Sequence of operations

The sequence of operations as specified in the protocol design is as follows. Node A sends an FT_DATA frame, containing application data. On successful sending, the buffer holding that frame is moved from the "transmitting" queue to the "waiting for ack" queue. When Node B receives the FT_DATA frame, it immediately generates an FT_DACK frame with the same sequence number as the FT_DATA frame and a truncated payload (containing only payload length, sequence number and frame type, which will of course be padded to the minimum 802.3 frame length by the physical layer) and sends it back. If Node A does not receive an FT_DACK in a timely fashion, it will retransmit the frame in question. If Node A does receive an FT_DACK before timeout, the frame will be removed from the "waiting for ack" queue and recycled.

Ethernet driver

The Node<ARM> utilises an SMC91111 Ethernet controller, a moderately well understood device with a good amount of documentation. Since the u–boot bootloader successfully uses the Ethernet controller to retrieve a boot image over a TCP/IP network, it can be assumed that it contains all the essential components necessary to produce a simple Ethernet driver, despite being based on a very different platform (a multi–threaded minimal Linux rather than the bare–metal single–threaded approach of AESTPI). The initialisation and shutdown functions of the Ethernet based Bridging system are almost identical to those found for the SMC91111 in the u–boot source tree, as this codebase provably works and a clean–room reimplementation was not considered necessary. The only significant alteration found to be necessary was a change to the base address of the Ethernet controller, since while the device is mapped into memory, u–boot appears to maintain a small offset between the address it uses and the physical address of the device. The fact that it is based on code from the u–boot project means that if it is released to the public at any point, the Ethernet–based bridging system would have to be released under the GNU General Public License

(since u–boot uses said license, and the license requires that derivative projects inherit the license).

The transmission and reception elements of the bridging system are more problematic, as u–boot's version is based on multi–threaded code and requires a listener thread for both functions. Multi–threaded operation is not advisable for safety–critical systems, as such systems are characterised by their rapid response (among other things) and a multi–threaded approach presents the possibility that the "wrong" thread may be running when an urgent message comes in, causing an unacceptable delay while the thread proceeds to a pre-emptible point and the message handler is context–switched in. It should be noted that a single–threaded solution is not inherently immune to this problem, but can at least be written in a deterministic fashion so that interrupts to the message handler are processed at predictable intervals. In place of a threaded message handler, the bridging system uses a pair of finite state automata to handle message interchange: the SMC91111 is of considerable help here, because it has an internal 8kB packet buffer, allowing messages to be transferred to it quickly and stored. The SMC91111 will then act on those messages as appropriate, transmitting outbound messages as the bus becomes free and releasing the Node<ARM> from having to control the message interchange in any way. The Ethernet transmission function is then implemented much like the transmission function in u–boot, with the exception that wherever the u–boot function waits for the SMC91111 to return a ready code (or indeed performs any kind of timed wait), the bridging system function transitions the FSM to a wait state and returns. When the FSM is next polled, it attempts a gated "next" transition: it checks the SMC91111 return–code register and, if the Ethernet controller is not ready, returns immediately. If the Ethernet controller has completed the requested operation, the FSM transitions to the next state in the transmission process. Consequently, the transmission function can be polled rapidly within the System Task, and will advance through the transmission process in the same way as the u–boot driver, with the exception that it breaks up the transmission process into small sections and returns to an interruptible state while waiting. Since the System Task retriggers continuously when no other task is scheduled, the transmission function can safely be called from within the System Task and will run with a reasonably minimal delay in transmission while remaining highly interruptible for the purposes of FlexRay application processing. The FlexRay application task places messages into a message queue for transmission, and when the transmission function is called in the System Task, it takes a message from that queue and loads it into the SMC91111 packet buffer, then flags it for transmission. The actual transmission process is somewhat involved (hence the finite state machines) and largely irrelevant to this thesis: buffer memory must be requested and allocated, the

SMC91111 must be given an address from which to DMA the packet contents, and a series of flags must be set.

The Ethernet–based bridging system was a success, and proved to be significantly faster than the Bridgelink system (see section 6.2.1), a fact that should be no surprise given that the underlying network is capable of higher speeds than the parallel I/O system and that the message interchange is simpler, with no interface mediator and no fragmentation/reassembly step.

6.2.2 Testing Protocol

As with the previous testbed configuration, it is necessary to collect results to indicate the performance of the system. Since the MilCAN segment of the testbed has remained the same, it was not thought necessary to repeat the MilCAN tests, but the FlexRay segment has undergone significant alteration as described in section 6.2.1 of this chapter. The communications cycle has been reduced to almost half its length, and the schedule compacted accordingly (figure 6.8 shows the current communications schedule).

In addition, the triggering system is now fully integrated into the testbed: the FlexRay gateway and one other node (designated the FlexRay Trigger) are attached to the triggering system. The ECU triggering component runs continuously in the System Task alongside the bridging component, and can be used to alter the payload of a subset of the messages on the FlexRay network (those linking the Gateway and Trigger), effectively simulating data changing in the vetronics network under test. As before, the Gateway bridges the payload of a certain message (sig13, as it is labelled in figure 6.8) to the MilCAN network, and inserts data received from the MilCAN network into the payload of the complementary message (sig31).

The series of modifications detailed above constitute substantial changes to sections of the FlexRay segment that are function– and timing–critical. Consequently, it was necessary to gather results from the FlexRay segment again, to study how these changes had affected the performance of the testbed. The schedule of tests was as follows:

1. VSI Bridge to FlexRay Gateway
2. VSI Bridge to FlexRay Trigger

3. Triggering System internal latency

4. Triggering System to FlexRay Trigger

5. Triggering System to FlexRay Gateway

6. MilCAN to FlexRay Trigger

The first test involves the VSI Bridge sending a message over Ethlink to the FlexRay Gateway, where it is reflected back. In the second test, the FlexRay Gateway sends the message from the Bridge over the FlexRay network. A node on the network picks up the message and reflects it back across the FlexRay network, at which point the Bridge relays the message from the FlexRay network to the Bridge. In the third test, the Triggering system transmits a "ping" to the designated test node on the network, which immediately returns a "pong" to allow the latency of the serial line to be established (note that while any node on the FlexRay network could be used, a single node was designated the test node for ease of administration). In the fourth test, the Triggering system requests that the test node transmit a FlexRay message: the Gateway node will reflect the transmitted message back to the test node and the overall loop latency will be logged. The final test exercises the entire network of networks: the MilCAN network generates a frame which travels through the MilCAN gateway, VSI Bridge and FlexRay Gateway before being received and looped back by the FlexRay test node.

6.2.3 Results

Test 1: VSI Bridge to FlexRay Gateway

Results for test 1 follow (figure 6.9). This test studies the direct replacement of the Bridgelink and interface mediator setup from the previous testbed with an Ethernet connection, using the internal Ethernet controller of the FlexRay Gateway. For the purposes of the test, the VSI Bridge sends an Ethlink frame to the FlexRay gateway, which immediately sends it back. This test is the same as test 1.b. from the Bridgelink test series, but must be retaken because of the changes in both hardware and software. Significantly, since the bridging component is now serviced in the System Task (during idle time) rather than just in the Application Task, the link latency should be markedly improved. Also, since the gateway node spends most of its time idle now that the load

of maintaining the Linux operating system has been removed, the response time of the gateway is unlikely to be affected by the cluster cycle time unless a task is triggered during the operation, in which case it will be deferred until that task completes. This test was timed against the VSI Bridge's internal timebase.

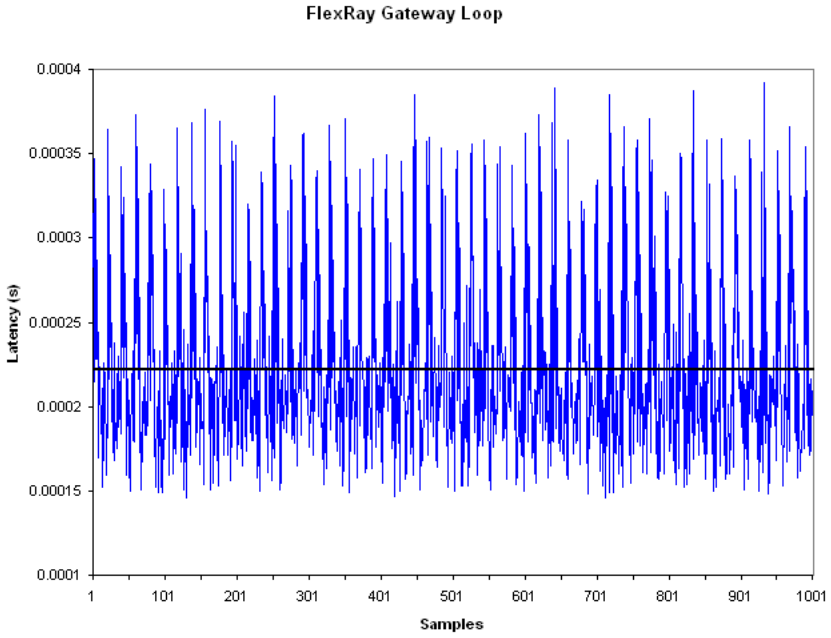

Figure 6.9: VSI Bridge to FlexRay GW loopback

In the graph (figure 6.9), one can see a maximum latency of 0.392ms, an average latency of 0.222ms (indicated by a bold black line), and a minimum of 0.146ms. The previous (Bridgelink) implementation had an average latency in the region of 0.0395 seconds, which can be directly compared with the new average latency of 0.222ms (or 0.000222 seconds). Using the percentage change formula, we see a percentage decrease in latency of 99% (see Equation 6.1, below).

$$\begin{aligned} \%change &= \left(\frac{new\_value - old\_value}{old\_value}\right) \\ &= \left(\frac{0.0395 - 0.000222}{0.0395}\right) \\ &= 0.994 \end{aligned}$$

Equation 7.1. Percentage decrease in latency

From this average latency, it is possible to estimate the average bandwidth usage of the Ethlink system. Assuming that the communication is perfectly symmetrical and that node–processing delays are zero, the one–way latency of an Ethlink frame can be approximated as 0.111ms. Taking the reciprocal of this value yields the number of packets per second, 9009.009... Since we know the frames are all 64 octets in length, the minimum permitted frame length under IEEE802, we can multiply the frame count per second by 512 (64 * 8) to estimate the maximum half–duplex bandwidth of the Ethlink connection at 4.61Mbps.

A loop latency of 0.2ms is a significant improvement on previous tests, as it corresponds to approximately one five–thousandth of a second as opposed to one twenty–fifth. A connection with this latency is much more likely to be usable in a vetronic network (outside the previously stated human interface case).

Test 2: VSI Bridge to FlexRay Trigger

The test results for test 2 follow (figure 6.10). This test requires the VSI Bridge to send an Ethlink frame to the FlexRay Gateway, as in the previous test. The Gateway then forwards the frame to the Trigger node on the FlexRay network, which turns it around and sends it back. The Gateway then sends the frame back to the VSI Bridge. This test was timed by the VSI Bridge.

Although Ethlink receives and buffers the frame immediately, due to the time–driven nature of the cluster the frame payload can only be studied and moved around the FlexRay network at certain clearly defined times. Since there is ony one communication task in the schedule of each node, a received frame will take an entire TDMA cycle (1100μs) to be reflected. This is because the node receives and buffers the frame from the network during the communications task, processes it and queues it for retransmission during the application task, but will not actually retransmit it until the communications task is next executed. In the best possible case, with the current FlexRay schedule, frame data should pass from the Ethlink buffers through the FlexRay network and back to the Ethlink buffers in approximately 3300μs (as can be calculated from the FlexRay Schedule in section 6.8, assuming that the Ethlink frame arrives immediately before the Gateway Node application task executes). In contrast, the worst possible case assumes the Ethlink frame arriving immediately after the Gateway Node application task, and thus waiting in the buffers for almost another entire communications cycle before being transmitted on the FlexRay bus (giving a worst–case loop time of approximately 4400μs).

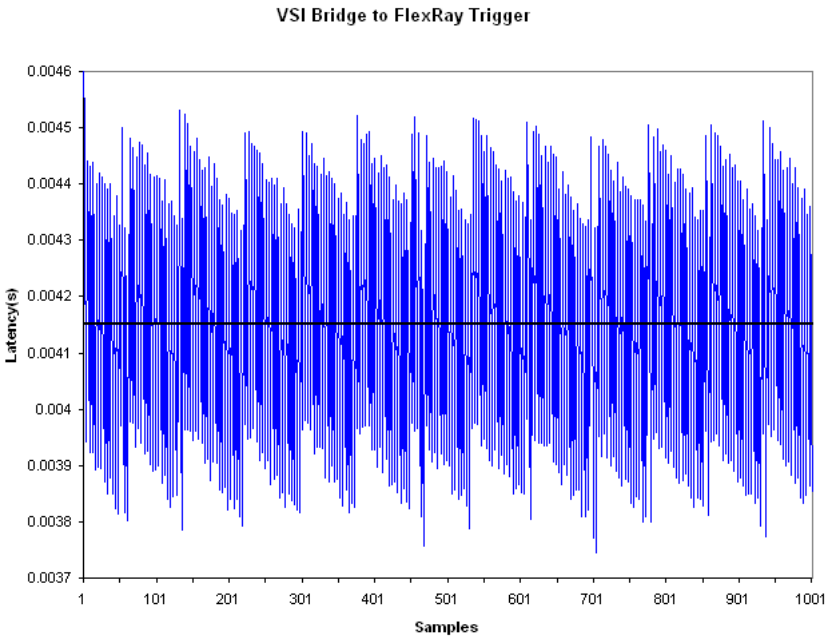

Figure 6.10: VSI Bridge to FlexRay Trigger via Ethlink and FlexRay

The Ethlink round–trip time must be added to these values to get an overall estimate of the time taken for a frame to travel the loop described in the test. Since the average RTT calculated in test 1 was 222μs, we can assume a best–case overall RTT estimate of 3522μs and a worst–case overall RTT estimate of 4622μs. The graph for this test matches these predictions well: the worst–case RTT reported in the test was 4635μs, and the best–case was 3746μs. There is a significant amount of variation in the results (a decreasing trend cycling approximately once every 150 samples), which is likely to be due to the retransmission interval of the frame–sending script on the VSI Bridge not being equal to the cycle length of the FlexRay cluster. As a result, the cycle point at which the frame arrives in the FlexRay cluster will precess with time, probably making one trip around the schedule approximately once every 150 cycles.

These results further confirm indications from the last test that the Ethlink system introduces a negligible latency into connections on which it is used. This is obviously a desirable property when real–time systems are being bridged.

Test 3: Triggering System internal latency

This test establishes the approximate latency of the Triggering System itself. In place of a serial cable linking the triggering hub to a network ECU, a crossover serial cable returns the transmitted

data directly back to the hub. The latency of this message exchange is timed against the hub's internal RTC (Real–Time Clock), to ensure the accuracy of the measurement. Since the serial ports of the FlexRay nodes are capable of a maximum operating speed of 19'200bps, the triggering hub serial ports will be limited to that speed for the test.

The results for this test are shown in figure 6.11. In a test of 1000 samples, the maximum loopback latency was 0.003115 seconds, the average was 0.003070 seconds, and the minimum latency was 0.003052 seconds.

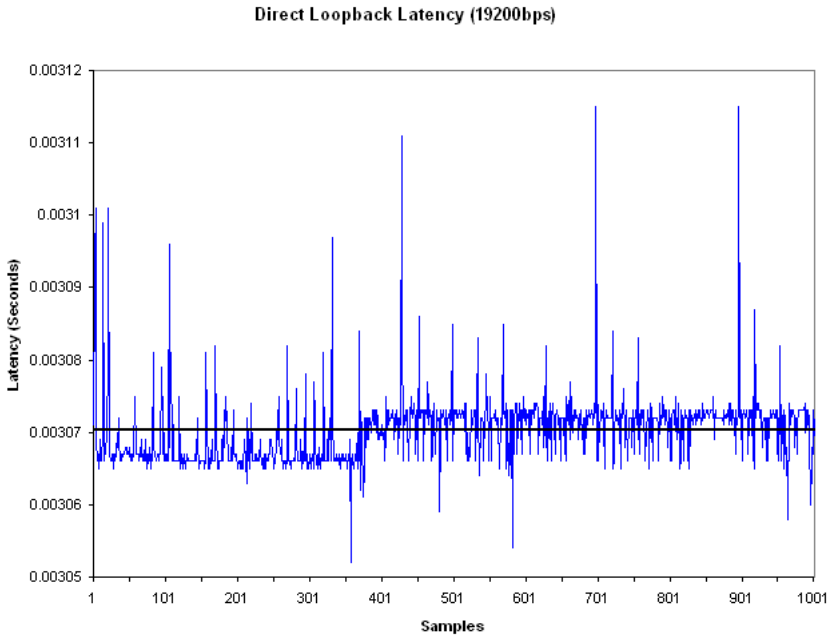

Figure 6.11: Stimulus system direct loopback latency

In theory, a serial frame of ten bits will take 10/19200 = 0.0005208 seconds, 520μs, to traverse the cable. However, the fact that the triggering hub is based on a multi–tasking x86 PC will naturally introduce some additional delays, likely causing the higher latencies seen in this test.

Test 4: Triggering System to FlexRay Trigger

In this test, the loopback latency between the Triggering hub and the FlexRay Trigger node was tested, using the "ping-pong" functionality built into the triggering system. The RTC of the triggering hub is read on message transmission and reception, resulting in an accurate latency measurement (since the trigger node responds immediately upon receipt).

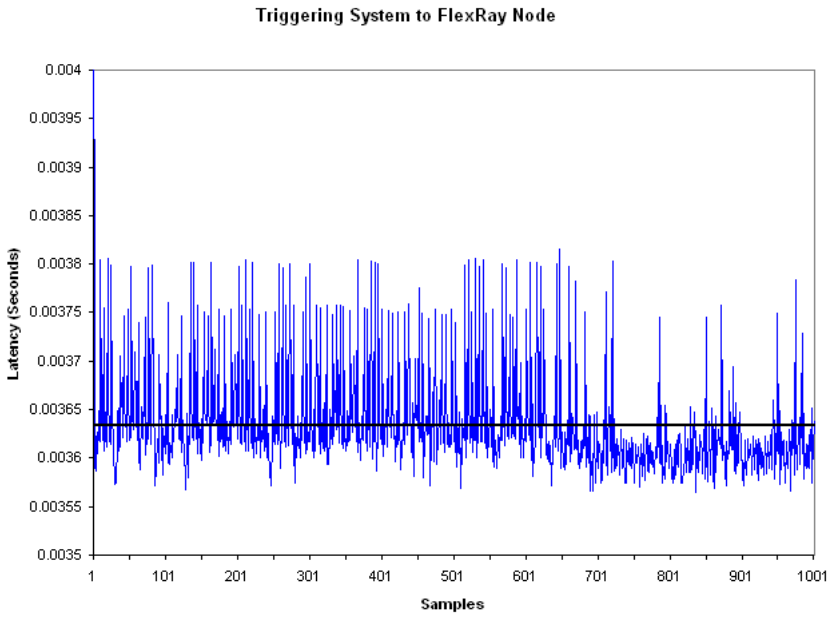

Figure 6.12: Triggering System latency - FlexRay Trigger

The results (in Figure 6.12) show a maximum latency of 0.004269 seconds, an average of 0.003633 seconds and a minimum latency of 0.003565 seconds. The variation in response time is likely partially due to the response time of the triggering hub and partly due to the response time of the FlexRay node: the triggering system is present in both the Idle Task (which can be skipped or truncated if another time–triggered task is due to be dispatched) and in the time–triggered Application Task, which cannot be skipped or truncated. Since the triggering system is serviced at most once in each task, and each task will be executed at least once in each cluster cycle, a response is guaranteed but its servicing point is somewhat variable. In all triggering–related tests and calculations, this average latency should be considered as part of the round–trip time, and factored out where appropriate.

The results from this test include no compensation for the inherent latency of the triggering hub and cables, as was measured in section 6.2.3. If the naive subtraction is performed based on the results from this test and the previous test, the triggering system appears to have an internal latency of around 0.000563s, or 0.5ms. It is the overall average that is of interest, however: any measurement that is taken by the triggering system will include an overhead of approximately 3.6ms.

Test 5: Triggering System to FlexRay Gateway

In test 4, a stimulus request issued by the triggering hub is immediately returned on reception by the Trigger Node. In this test, the Trigger Node emits a frame on the FlexRay network when it receives the stimulus request. That frame will be received by the FlexRay Gateway, and a response returned immediately. When the Trigger Node receives that response, it then emits a signal to the triggering hub: it is this overall loop (from hub to trigger node to gateway to trigger node to hub) that is under measurement. A graph of the results can be seen in figure 6.13.

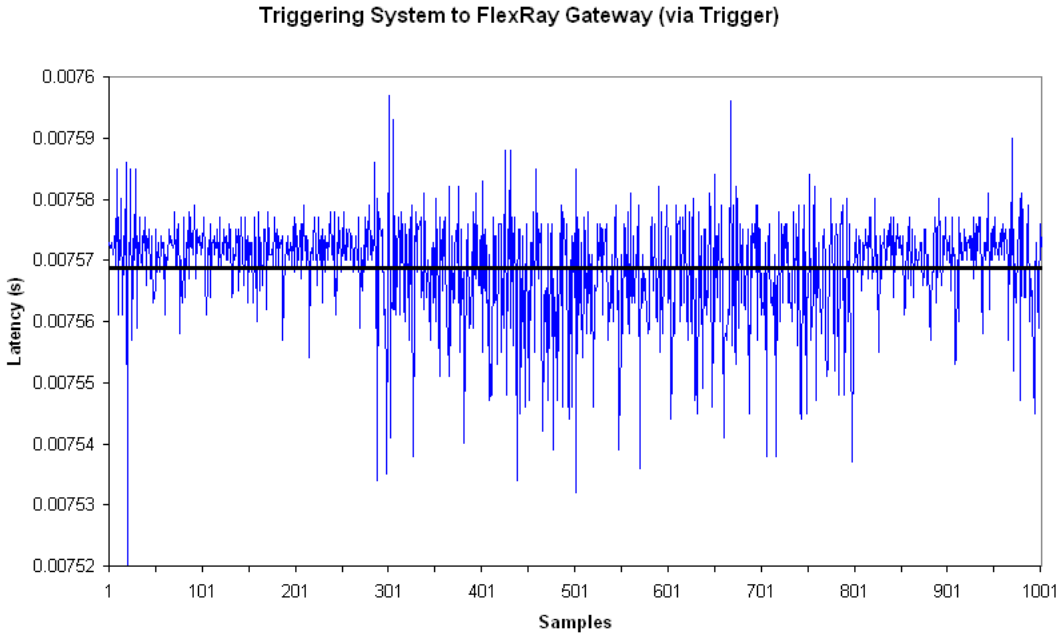

Figure 6.13: Triggering System latency - FlexRay Gateway

The results show a maximum latency of 0.007597 seconds, an average of 0.007568 seconds and a minimum latency of 0.00752 seconds. Given that the FlexRay segment has a previously demonstrated loop latency of approximately 0.00393 seconds (4.151 - 0.222, from tests 1 and 2) and the triggering system has a previously demonstrated loop latency of 0.00363 seconds, 0.007568 seconds is a plausible number (0.00363 + 0.00393 = 0.00756 seconds, and node processing delays can account for the remainder).

Test 6: MilCAN Trigger to FlexRay Trigger

This final test involves the entire MilCAN/FlexRay testbed. The triggering hub causes the MilCAN node to emit a message, which travels across the MilCAN network, through the VSI Bridge to the

FlexRay Gateway, across the FlexRay network to the FlexRay trigger, and then back along the same route. The measured latency is the elapsed time between the triggering system triggering the message and its return being signalled by the MilCAN node. A graph of the results can be seen in figure 6.14.

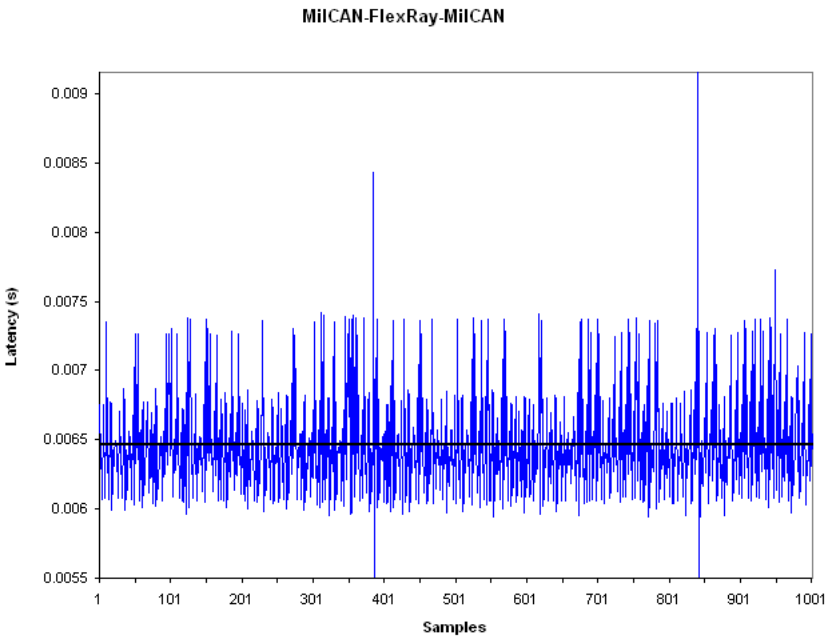

Figure 6.14: MilCAN - FlexRay loopback

The results show an average latency of communication of 6.465ms, with a maximum latency of 9.155ms (likely a momentary fault) and a minimum of 5.428ms. Since the previous version of the system (as documented above, in section 6.1.1) has an end to end loopback latency of approximately 44ms, the new system shows a substantial improvement: an 85% percentage decrease in latency.

Based on previous tests, these results seem plausible if a little low: the FlexRay segment average loopback latency of 3.9ms and the MilCAN average loopback latency of 1.5ms sum to give 5.4ms, a little less than the minimum time taken for a message to travel through the testbed in practice. The substantial improvement in loopback latency means that this implementation of the bridging system is a much better candidate for use in bridging real–time, deterministic and safety–critical networks. The improvement in performance is a result of the new operating system and the upgraded communications link between the FlexRay cluster and the VSI Bridge, as Ethlink has proven to be notably faster than Bridgelink in these tests.

Consolidated Results

In figure 6.15, the results from the previous tests have been gathered with the MilCAN results from previous tests and consolidated into a single diagram for ease of comparison with the approximate latencies shown in section 6.1.3. Each double–headed arrow indicates the average round–trip time for messages travelling between the different segments. It should be noted that these times do not always add up correctly, as they are drawn from experiments that necessarily altered the configuration of the system slightly between runs (mainly, different routes taken through the gateways and bridge follow different code paths, which will take varying amounts of time to execute). Nonetheless, the consolidated results indicate substantially lower message latencies in the second version of the testbed, which will result in an improvement in determinism in all attached segments, when compared against the first version..

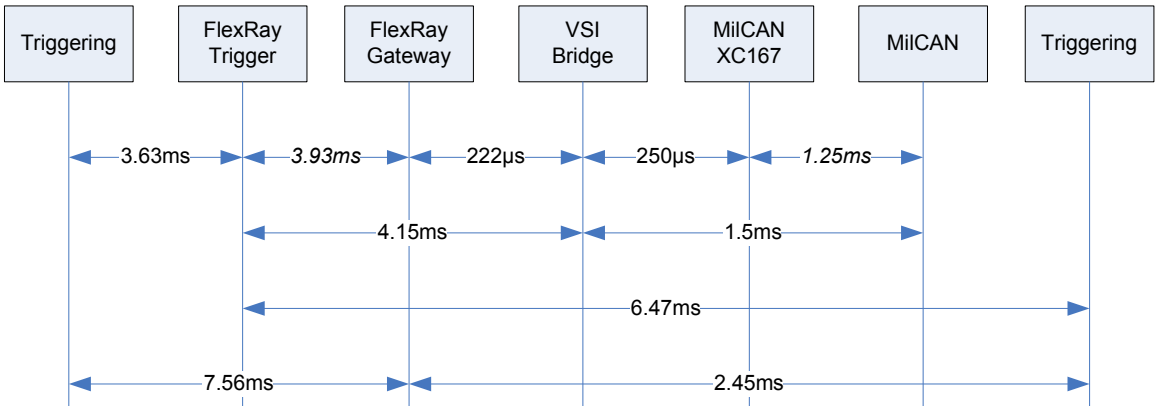

Figure 6.15: Consolidated timings for the FlexRay/MilCAN testbed

6.2.4 Analysis

The Ethlink connection is almost an order of magnitude faster than the Bridgelink connection and retains the same reliability and ease of implementation, assuming that Ethernet drivers appropriate to the platform are available. Some of this improvement is due to the upgraded operating system in use on the FlexRay ECUs, and some of it due to the substantially faster and less processor–intensive Ethlink connection. While the Ethlink–based testbed remains slower than the initial CAN–based concept, it is likely that the majority of the additional latency is due to the comparatively slow RS–232 connection used by the triggering system. Furthermore, a small drop in performance to move

from the hard–wired, inflexible CAN concept to the highly flexible and run–time reconfigurable VSI Bridge based system is entirely acceptable.

6.3 The Mobile Testbed

In addition to the workbench testbed, the V&V testing system was also partially implemented in a mobile demonstrator, nicknamed "The Buggy". The Buggy contains a deterministic network, implemented as a pair of MilCAN segments (Crew Station and Utilities), and a safety–critical network in the form of a TTP cluster. The deterministic network is responsible for managing vehicle lights and indicators (and a small amount of status monitoring and telemetry), while the safety–critical network forms the core of a simple drive–by–wire system. The two networks are bridged by a VSI Bridge in exactly the same way as the segments in the V&V testbed, and the testbed was in fact used to develop the control system for the Buggy in exactly the way one might expect. This work could easily be continued to include further devices or additional subnetworks, although a new chassis may be required, as the current vehicle has little available space for further additions.

Although both FlexRay and TTP are present on the V&V testbed, the drive–by–wire system on the Buggy was implemented using TTP. There were several reasons for this decision. First, TTP is a relatively well–established technology, with known behaviour and limitations. In comparison, FlexRay is a new protocol which is not yet in widespread use. Additionally, ruggedised TTP nodes were readily available for use on a demonstrator that involves large amounts of vibration and dirt that would likely damage desktop development kits. At the time the Buggy was implemented and integrated, no ruggedised FlexRay nodes were available on the open market. With these two considerations in mind, the choice to implement the drive–by–wire system in TTP is not a difficult one.

In addition to the networks described above, a simple "glass cockpit" was implemented by the author of this thesis and integrated into the network. A pair of LCD touchscreens was mounted above the crew–station, one in front of the driver and one in front of the passenger. Each touchscreen was connected to a dedicated vehicle–PC (an x86–based computer that is vibration protected and designed for use in vehicles), which hosted a VSI Bridge with a Qt–based push–button interface capable of controlling features of the Buggy. In each case, the VSI Bridge was connected to the

vehicle MilCAN backbone. The driver interface hosted a speedometer and controls to manipulate the headlights and indicators, as well as an engine start / stop button. Since the size of the force–feedback steering wheel assembly that was installed as part of the drive–by–wire network necessitated the removal of the dashboard and steering column and associated controls, this interface held considerable utility. When pressed, buttons on the interface caused specific messages to be emitted on to the MilCAN backbone, which were then picked up by devices in the Utilities segment, causing the relevant action to be taken. The passenger interface hosted an "engineer's screen" that gave a scrolling dump of messages on the MilCAN network and basic statistics, allowing the status of the vehicle and its attached peripherals to be assessed. This status information could serve as a simple input to a Through–Life Capability Management system, allowing the ongoing health and maintenance of the Buggy to be assessed over time, and appropriate maintenance scheduled.

Conclusion

As presented above, the Bridgelink–based testbed is serviceable, but not well suited to all applications due to its comparatively high average transfer latency of approximately 40ms. The Ethlink–based testbed configuration is capable of much lower latency in transfer, taking between 6 and 9ms to loop a frame. Both timings are below the threshold of human visual perception, and as such are perfectly acceptable for a human interface system or an engineer's network monitoring view. However, in the more exacting kinds of embedded network (actuator and process control, vehicle steering etc.), the Bridgelink result is unlikely to be acceptable. The Ethlink result may be, not least considering that when the FlexRay protocol loop and all the node overheads are removed (as in test 6.2.3) the connection is capable of a minimum operating latency of 0.222ms.

The TTP subnetwork was similarly upgraded to a faster interface, the fastest available on the TTP development kits being a synchronous serial connection (SSC, resulting in Ethlink's cousin SSClink). In the Bridgelink test series, the link between the TTP segment and the VSI Bridge was substantially faster than the link from the FlexRay segment (5.5ms rather than 39.5ms). In the second test configuration, however, the two networks are comparable, as the FlexRay Ethlink achieves an average latency of 6.47ms while the TTP SSClink achieves an average latency of 9.12ms.

Finally, the testbed configuration was implemented on a mobile demonstrator, where it performed well. The safety–critical network segment carried X–by–wire data for the steering, braking and throttle control systems, while the deterministic segment carried sensor and management data for the other onboard systems. Messages from the safety–critical segment are bridged to the deterministic segment for display on the status screens, and to control the light bars at either end of the vehicle. The mobile testbed implements the design of the V&V Testbed on a real–world platform, demonstrating that its use is a viable development technique.

Chapter 7

Conclusions & Further Work

7.1 Concluding Remarks

In the introduction to this thesis, it was stated that a vetronics networking testbed would be specified and created, first in the abstract and then as a set of implementing technologies. A set of interfaces and procedures for integrating heterogeneous embedded networks were to be defined, and a triggering and monitoring system to gather performance data was to be produced. All of these goals have been achieved.

The Verification and Validation testbed makes use of standardised interfaces and procedures to interconnect a safety–critical segment and a deterministic segment through a middleware bridge. The deterministic segment of the V&V Testbed is implemented as a MilCAN segment, while the safety–critical segment is implemented twice to the same design: once as a FlexRay segment and once as a TTP segment, allowing both technologies to be tested. This thesis is concerned with the FlexRay and MilCAN segments only.

The development of the Verification and Validation Testbed is chronicled in chapters 4, 5 and 6 of this thesis. As might be suggested by the name, this testbed is intended to support the verification and validation process codified in the V–Model and its variants, by making it possible to create

a simulation of the target architecture, then verify components under development by subjecting them to simulated load. Assuming that the target vetronic network can be faithfully simulated on the testbed, it is also possible to validate components and systems.

A triggering and monitoring system is included, allowing performance data to be gathered from the testbed in a repeatable manner. A central, scriptable hub sends commands over RS–232 to a set of simple command interpreters installed on each node in the testbed, allowing frame sending to be triggered and frame reception logged.

Since the testbed can theoretically be programmed to replicate a wide variety of vetronics networks, new components and subsystems can be verified and validated through it without going to the considerable expense and difficulty of testing on the target platform itself. Final stage validation and acceptance testing is still recommended to be performed on the target platform, in case edge cases become apparent upon installation.

7.1.1 The V&V Testbed

The V&V Testbed is comprised of a group of heterogeneous network segments of varying classes, some being deterministic (MilCAN), some being safety–critical (FlexRay and TTP) and some being neither (Ethernet). These segments are interconnected through a set of highly configurable programmable bridges and gateways, and linked to a fully–integrated triggering, testing and monitoring system. This triggering system is capable of providing a wide variety of stimuli to the attached network nodes and reading back responses with millisecond accuracy. This means that a wide variety of network configurations and architectures can be simulated by the simple expedient of altering the routing tables in the VSI Bridges and loading different testing scripts into the triggering system hub.

Two versions of the connection between the FlexRay and MilCAN segments are described in this thesis: the first is simpler to implement and slightly more general, while the second is significantly more efficient but requires the presence of certain hardware. In the first testbed, the connection from the safety–critical segment to the VSI Bridge is handled over a byte–wide parallel I/O connection, with an overlaid packetisation system to handle data loss and error recovery. This first implementation requires no specialist hardware on the implementing node and is reliable, but is extremely slow in comparison to most modern communications interfaces, taking up to 40ms to

transfer data to and from the gateway node. The second implementation relies on the presence of an Ethernet controller on the gateway device (likely restricting implementability to mid–range and high–end devices), but is much more responsive (taking only 0.2ms to transfer data) without being significantly less reliable.

The triggering system implemented in the V&V testbed consists of a controlling central hub and a portable interface component on the controlled node. The components are interconnected by an RS–232 connection, as RS–232 is available on the great majority of development kits, and as such is a good choice of communications link for use in a testbed to which many different types of hardware may be attached. Unfortunately, RS–232 may not have been the best choice in hindsight, as its comparatively low bitrate (19'200kbps in the case of the FlexRay network) limits the resolution of measurements taken. Additionally, latency in the comm channel risks masking smaller but important latencies in the equipment under test. Since the new implementation of the testbed bridges messages over Ethernet, freeing the digital I/O, one possibility would be to investigate the use of digital I/O pins for triggering and monitoring.

Both implementations of the FlexRay to MilCAN configuration of the testbed were tested to establish their envelope of operation. As noted above, the first implementation displayed an average latency of 44ms for end–to–end messages and little variation, which would be serviceable for monitoring purposes but is likely to be of strictly limited utility if the deterministic network is expected to use data from the safety critical network for control purposes. The second implementation is notably better, averaging 6.5ms end–to–end. Furthermore, it is suspected that that 6.5ms is in fact at best an upper bound on performance, and that actual performance is markedly better, but that the triggering system is masking smaller latencies.

In light of the above, it is safe to state that integrating heterogeneous networks is a difficult problem, as one might perhaps expect. The V&V testbed is an approach to solving the problem, by providing a set of standard interfaces (the VSI Bridge, Bridgelink and Ethlink, and the Triggering System) and a recommended configuration for testing.

7.1.2 Challenges Met

The implementation of the FlexRay segment was not without its challenges: FlexRay is a comparatively new technology, few development kits are available and the integrated development

environments used to specify and program the cluster tend to be restrictive or feature–incomplete.

Both the node operating system and the development environment underlying the FlexRay segment went through major changes part of the way through the development of the testbed. The new version of the operating system (AESTPI, replacing the minimal Linux) imposed a significantly lessened processing and resource load on the development kit hardware, and was installed for this reason. Unfortunately, the new version of the development environment was significantly different, and several weeks were lost to retraining.

7.1.3 TTP & FlexRay

During the research that resulted in this thesis, it became apparent that the performance of TTP and FlexRay is often quite similar. Ignoring the original Bridgelink latency measurements, measurements involving both networks have always tended to end up resulting in similar values (the original latency measurement is ignored because the TTP development kits have always had an operating system of similar simplicity to AESTPI, rather than the complexity of Linux that may have adversely affected early FlexRay tests). It seems, from observing the workflow involved in configuring and programming the FlexRay and TTP clusters, that the TTP development system is more detailed and able to give more useful metrics on the application under development, but that the FlexRay development system is much more flexible in terms of programming and node configuration, other than the options supplied in the integrated development environment UI.

7.2 Further Work

7.2.1 Alternate Technology Mappings

The testbed configuration investigated during the course of this research forms just one possible technology mapping of the abstract design. It would be of interest to map a different set of technologies to the testbed design and compare the implementations, as doing so is likely to bring out features that are features of the testbed rather than features of one or other protocol. Even if a full reimplementation is not performed, the addition of more networks and networking technologies to the testbed is likely to highlight behaviours worthy of further investigation.

7.2.2 Upgraded Triggering System

Given the problems exhibited by the triggering system (as listed above), it would be of considerable interest to implement an alternative triggering system using a faster technology, then use it to repeat tests on the Ethlink implementation and investigate whether the same behaviour is exhibited. The obvious candidate for implementation is parallel I/O, since the bridging system no longer places any demand upon it, and if the appropriate pins are chosen (preferably with hardware interrupts) there is a potential for considerable speed. The majority of triggering technologies other than RS–232, however, indicate a need for some form of interface device between the triggering PC and the triggered devices, as the x86 architecture does not provide a useful number of appropriate digital interfaces.

7.2.3 Generic Vehicle Architectures

When drive–by–wire and similar systems are considered in relation to the testbed, it becomes easy to see further uses to which this research may be put. The United Kingdom Ministry of Defence has mandated that all future UK MoD vehicles will be based around the concept of a Generic Vehicle Architecture[4]: an abstract structure and set of interfaces and technologies that would be used to construct military vehicles. Such an architecture has the potential to greatly increase modularity and simplify field upgrades, as devices designed to this GVA could simply be attached to the vehicle's internal network and, after a minimal amount of alteration to that network, operate at full design capacity despite not being designed specifically for the particular class of vehicle and attachment point. This degree of flexibility will require that testing and development be done within an appropriate framework, and the V&V testbed will be easily adapted to represent the GVA or the subset of it being tested.

7.2.4 Possible Future Vehicle Systems

From a broader implementation perspective, the application of network integration concepts to the domain of military vehicles presents a number of interesting possibilities. Some possible vehicle upgrades are listed below.

Platooning The vetronic networks of multiple vehicles can be linked so that the vehicles act as a

group. The implementation currently being explored in the civilian world is that of motorway driving: since platooned vehicles are capable of negotiating road position, intervehicle separation and speed, trials are being held involving multiple vehicles travelling as a group at highway speeds with a 6" separation. This degree of control is unattainable for all but the very best human drivers, but the driving style is extremely fuel efficient as air resistance is reduced for all but the first vehicle. Additionally, drivers who are not leading the platoon are free to do something else, a potentially useful feature in both a civilian commute and a military patrol.

Assisted Maintenance The vetronic network need not only be active when the vehicle is travelling or on mission. By including a "maintenance mode" or "Built–in Test" in their programming, it will be possible for ECUs to assist maintenance crews in repairing the attached peripherals by performing simple built–in tests and reporting the results, logging conditions encountered in use and so forth.

ISTAR Integration Given that sensors attached to the vetronic network are always available, whether the vehicle crew is currently monitoring them or not, the opportunity exists to integrate the sensors of a group of vehicles to provide a mobile sensor station and improve situational awareness, without impacting the vehicle's primary mission at all. This approach simply requires that networked sensors be permitted to send data to a central information fusion unit, which is then likely to send the resulting intelligence to group and section commanders.

Limp–Home Reconfiguration Military vehicles, indeed vehicles in general, can be damaged in the line of duty. A conventional "dumb" vehicle can be modified by its crew to some extent in order that it and they may return to base for repairs. Under the right conditions, such modifications could be made wholly in software, partially or entirely automatically, to compensate for the loss of damaged or destroyed components.

Remote Control Finally, as the vehicle approaches the end of its lifespan, the ability to remote control it may prove useful, either as an unmanned logistics vehicle, or for mine clearance, or even as a target drone. In the case of a drive–by–wire vehicle, altering the vehicle for remote control is as simple as reprogramming the crew station to accept remote guidance input and to echo it to the drive–by–wire network in an appropriate format.

References & Bibliography

[1] R M Connor. Vetronics Standards & Guidelines. Technical report, Qinetiq, UK, June 2009.

[2] Periklis Charchalakis. *Integrated vetronic systems - Intelligent bridging of vehicle networks over high speed backbones*. PhD thesis, Engineering & Design, University of Sussex, UK, September 2005.

[3] George Valsamakis. *Vetronic Systems Integration: Network Management and (Re) Configurability Integration on MilCAN based systems*. PhD thesis, University of Sussex, 2006.

[4] United Kingdom Ministry of Defence. DEF-STAN 23-09: Generic Vehicle Architecture (GVA). Technical report, United Kingdom Ministry of Defence, 2010.

[5] Amos Albert. Comparison of event–triggered and time–triggered concepts with regard to distributed control systems. In *Embedded World*, pages 235–252, 2004.

[6] Gabriel Leen. Expanding automotive electronic systems. *IEEE Computer*, 35:88–93, 2002.

[7] David A. Glanzer and Charles A. Cianfrani. Interoperable fieldbus devices: a technical overview. In *ISA Transactions*, volume 35, pages 147–151. Elsevier Science Ltd., 1996.

[8] Max Felser. The fieldbus standard: History and structure. Technical report, MICROSWISS Network, HTA Luzern, Switzerland, October 2002.

[9] H. Ekiz, A. Kutlu, and E.T. Powner. Design and implementation of a can/can bridge. In *Parallel Architectures, Algorithms and Networks*, pages 507–513, June 1996.

[10] Avionic Systems Standardisation Committee. Guide to digital interface standards for military avionic applications. Technical Report ASSC/110/6/2-ISSUE 3, ASSC, 2006.

[11] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82(1):25–40, Jan 1994.

[12] Ada-Europe, editor. *Ada Reference Manual*. The MITRE Corporation / Ada-Europe, November 2006.

[13] Johan Hedberg. Methods for verification and validation of time–triggered embedded systems. Technical report, Nordtest, Nordic Innovation Centre, Stensberggata 25, 0170 OSLO, December 2005.

[14] Carl S. Droste and James E. Walker. The general dynamics case study on the f–16 fly–by–wire control system. Technical report, American Institute of Aeronautics and Astronautics, 1986.

[15] Ben R. Rich and Leo Janos. *Skunk Works*. Sphere, 1995.

[16] Rolf Isermann, R Schwarz, and S Stolzl. Fault-tolerant drive-by-wire systems. In *IEEE Control Systems Magazine*. IEEE, 2002.

[17] Conrad K. Kwok and Biswanath Mukherjee. Cut-through bridging for csma/cd local area networks. In *IEEE Transactions on Communications*, volume 38, pages 938–942. IEEE, July 1990.

[18] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[19] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. Reprinted from Proceedings of IEEE WESCON, August 1970, pages 1-9.

[20] H. H. Hesselink. A comparison of standards for software engineering based on do-178b for certification of avionics systems. *Microprocessors and Microsystems*, 19(10):559 – 563, 1995.

[21] Standards Coordinating Committeee of the Computer Society of the IEEE. IEEE standard glossary of software engineering terminology. Technical report, The Institute of Electrical and Electronic Engineers, Inc., Dec 1990.

[22] IEEE LAN/MAN Standards Committee. 802.3: Part 3: Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. Technical report, The Institute of Electrical and Electronic Engineers, Inc., 2008.

[23] P. Charchalakis, G. Valsamakis, B. Connor, and E. Stipidis. Milcan and ethernet. In *9th International CAN Conference*, October 2003.

[24] UK Ministry of Defence. MilCAN A Specification (Revision 2). Technical report, October 2007.

[25] Ioannis Melentis. *An investigation of compositing mixed-integrity vetronic distributed systems*. PhD thesis, Engineering & Design, University of Sussex, UK, 2010.

[26] FlexRay Consortium. *FlexRay Communications System - Protocol Specification - Version 2.1A*, 2.1a edition, December 2005.

[27] FlexRay Consortium. *FlexRay Communications System - Preliminary Node-Local Bus Guardian Specification - Version 2.0.9*, 2.0.9 edition, December 2005.

[28] FlexRay Consortium. *FlexRay Communications System - Electrical Physical Layer Specification - Version 2.1B*, 2.1b edition, November 2006.

[29] NXP Semiconductor. *TJA1080 FlexRay Transceiver datasheet*, 2 edition, July 2007.

[30] D Summers, P Charchalakis, E Stipidis, and F.H. Ali. Flexray milcan bridging. In *IEEE Vehicle Power and Propulsion Conference, 2006.*, September 2006.