

University of Dundee

DOCTOR OF PHILOSOPHY

Deterministic SpaceWire Networks

Gibson, David James

Award date:
2017

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Deterministic SpaceWire Networks

David James Gibson

Doctor of Philosophy

University of Dundee

October 2017

Table of Contents

List of Figures	xiii
List of Tables.....	xix
List of Acronyms	xxi
Glossary	xxv
Acknowledgements	xxvii
Declaration of the Candidate.....	xxviii
Declaration of the Supervisor.....	xxix
Abstract	xxx
Introduction	1
1.1 Research Questions	2
1.2 Outcomes	2
1.3 Thesis Structure.....	4
Background	5
2.1 Spacecraft Subsystems	5
2.1.1 Attitude and Orbit Control	5
2.1.2 Communications	6
2.1.3 On-Board Computers	7
2.1.4 Scientific Instruments.....	7
2.2 SpaceWire	8

2.2.1	Development History	9
2.2.2	Protocol Stack	9
2.2.2.1	Physical Level.....	10
2.2.2.2	Signal Level	10
2.2.2.3	Character Level.....	11
2.2.2.4	Exchange Level.....	12
2.2.2.5	Packet Level.....	15
2.2.2.6	Network Level	15
2.3	SpaceWire-D	17
2.3.1	Motivation	17
2.3.2	Operation.....	18
2.3.2.1	Time-Slots.....	18
2.3.2.2	Virtual Buses.....	19
2.3.2.3	Schedules	28
2.4	Missions	29
2.4.1	Magnetospheric Multiscale Mission	29
2.4.2	ASTRO-H	32
2.4.3	JUpiter ICy moons Explorer	34
2.5	Other Communication Networks	36
2.5.1	MIL-STD-1553	36
2.5.1.1	Node Types	36

2.5.1.2	Transfer Types	37
2.5.1.3	ECSS Standardisation	39
2.5.2	Controller Area Network.....	42
2.5.2.1	Message Identifiers	42
2.5.2.2	Arbitration.....	42
2.5.2.3	Transfer Types	44
2.5.3	Other Networks	46
2.5.4	Comparison	47
2.5.4.1	Time-Division Multiplexing	47
2.5.4.2	Exclusive or Non-Conflicting Access to Network	48
2.5.4.3	Data Rates and Protocol Overhead	49
2.5.4.4	Acknowledgements.....	51
2.5.4.5	Multiple Initiators	51
2.5.4.6	Fault Detection, Isolation and Recovery.....	52
2.5.4.7	Comparison Summary	53
2.6	On-Board Data Systems	55
2.7	Summary	57
	Research Questions	60
3.1	Research Questions	60
3.1.1	Designing a SpaceWire-D Software Layer	61
3.1.2	Designing a SpaceWire-D Demonstrator.....	61

3.1.3	Scheduling SpaceWire-D Networks	62
3.1.4	Summary	63
	SpaceWire-D Software Layer	64
4.1	Overview	65
4.2	RTEMS Board Support Package.....	66
4.2.1	RTEMS in Space.....	67
4.2.2	Porting Process.....	68
4.2.2.1	Existing LEON Support.....	68
4.2.2.2	Cross-Compiling Toolchain.....	69
4.2.2.3	Loading and Debugging.....	69
4.2.3	Creating the BSP	70
4.2.3.1	Configuration Files	70
4.2.3.2	Linker Command Script.....	71
4.2.3.3	Board Initialisation	73
4.2.3.4	Interrupt Vectoring	74
4.2.3.5	Example: Ticker.....	77
4.3	SpaceWire-D Layer.....	79
4.3.1	Architecture.....	80
4.3.1.1	Application.....	80
4.3.1.2	API.....	81
4.3.1.3	Script Interpreter	81

4.3.1.4	Script.....	81
4.3.1.5	Command Handler.....	81
4.3.1.6	RMAP Driver.....	82
4.3.1.7	Dispatcher	82
4.3.1.8	Error Handler	82
4.3.1.9	Notification Handler	82
4.3.1.10	Schedule.....	82
4.3.1.11	Control	82
4.3.1.12	Memory Map	82
4.3.1.13	Static Bus	83
4.3.1.14	Dynamic Bus.....	83
4.3.1.15	Asynchronous Bus	83
4.3.1.16	Packet Bus.....	83
4.3.1.17	Asynchronous Queue.....	84
4.3.1.18	Packet Queue	84
4.3.1.19	Packet Operation.....	84
4.3.1.20	Transaction.....	84
4.3.1.21	Transaction Group	85
4.3.2	Virtual Buses	85
4.3.3	Management.....	86
4.3.4	Executing Time-Slots.....	87

4.3.4.1	Initiator Processing Time.....	88
4.3.4.2	Optimisation.....	89
4.3.5	Local-Timer Synchronisation.....	92
4.3.6	Error Detection and Reporting.....	93
4.3.7	Notifications.....	94
4.3.8	Testing and Verification.....	94
4.3.8.1	Unit Testing	94
4.3.8.2	Protocol Verification.....	95
4.4	Summary	95
SpaceWire-D Demonstrator		97
5.1	Overview	97
5.2	Initiators	102
5.2.1	Automated Test Scripting	103
5.2.1.1	Scripting Language	103
5.2.1.2	Examples.....	105
5.3	Targets.....	119
5.3.1	Command Authorisation	119
5.3.2	Notifications.....	120
5.4	Routers	121
5.5	Network Manager.....	121
5.6	Host PC	122

5.6.1	Initiator Configuration	122
5.6.2	Target Configuration	124
5.6.3	Network Manager.....	125
5.6.4	Target Monitor	126
5.7	Summary	131
Verification of the SpaceWire-D Demonstrator		133
6.1	Running Example Script 1	134
6.2	Running Example Script 2	138
6.3	Running Example Script 3	140
6.4	Testing and Validation	152
6.4.1	Test 1 - Single Static Bus	152
6.4.2	Test 2 – Multiple Static Buses	153
6.4.3	Test 3 - Multiple Different Buses	153
6.4.4	Test 4 – Slow Link	153
6.4.5	Test 5 – Concurrent Slots	153
6.4.6	Test 6 – Common Link	154
6.4.7	Test 7 – Multi-Slots.....	154
6.4.8	Test 8 – Changing Schedules	154
6.4.9	Test 9 – Start and Stop Schedules	154
6.4.10	Test 10 – Reset Initiator	154
6.4.11	Test 11 – Error Injection	155

6.5	Summary	155
Scheduling SpaceWire-D Networks		156
7.1	Problem Specification	156
7.1.1	Bandwidth Requirements	157
7.1.1.1	Periodic Traffic	157
7.1.1.2	Aperiodic Traffic	157
7.1.1.3	Payload Data Traffic	158
7.1.2	Network Topology	159
7.1.3	Network Parameters	159
7.1.4	RMAP Execution Time.....	161
7.1.5	Solution Format.....	163
7.2	Solving the Problem.....	163
7.2.1	Selecting Paths	165
7.2.1.1	Breadth-First Search	165
7.2.1.2	Weighted Search	167
7.2.2	Scheduling Transactions	170
7.2.2.1	Conflict Graph	171
7.2.2.2	Periodic Traffic	171
7.2.2.3	Aperiodic Traffic	176
7.2.2.4	Payload Data Traffic	179
7.2.3	Complete Algorithm	187

7.3	Summary	189
Evaluating the Scheduling Strategy		190
8.1	Test Cases	190
8.1.1	Generating Network Topologies	190
8.1.2	Generating Bandwidth Requirements	193
8.1.3	Test Case Generation Algorithm.....	194
8.1.4	File Format	197
8.2	Experimental Setup	199
8.3	Results	200
8.3.1	Experiment 1: BFS Path Selection.....	200
8.3.1.1	Results.....	201
8.3.1.2	Analysis	202
8.3.1.3	Detailed Result Description	204
8.3.2	Experiment 2: Weighted Search Path Selection.....	208
8.3.2.1	Results.....	209
8.3.2.2	Analysis	210
8.4	Case Study: JUPiter ICy moons Explorer (JUICE).....	215
8.4.1	Network Topology	215
8.4.2	Time-Slot Duration	216
8.4.3	Bandwidth Requirements	217
8.4.3.1	Periodic	217

8.4.3.2	Aperiodic	218
8.4.3.3	Payload Data	219
8.4.4	Scheduling.....	220
8.4.4.1	Path Selection	220
8.4.4.2	Initial Results	221
8.4.4.3	Load Balancing Dynamic Penalties.....	222
8.5	Load Balancing Dynamic Penalties	225
8.6	Summary	226
Future Work		227
9.1	SpaceWire-D Efficiency Improvements	227
9.2	SpaceWire-D Hardware Controller.....	228
9.3	SpaceWire-D FDIR	229
9.4	SpaceWire-D On-Board Scheduler	229
9.5	SpaceFibre Scheduling.....	230
9.6	Summary	230
Conclusions.....		231
10.1	Research Summary.....	231
10.1.1	Designing a SpaceWire-D Software Layer	231
10.1.2	Designing a SpaceWire-D Demonstrator	232
10.1.3	Scheduling SpaceWire-D Networks	233
10.2	Contributions.....	236

10.3 Outcomes	237
Bibliography.....	238
Appendix 1	250
LEON2-FT Processor Board.....	250
RMAP Engines	251
SpaceWire DMA Channels	253
Embedded SpaceWire Router	254
Packet Demultiplexer	254
LEON2-FT in Space	255
Appendix 2	258
Multiple Initiators and Static Buses Results	258

List of Figures

Figure 2-1: SpaceWire Protocol Stack (ECSS 2008 A).....	10
Figure 2-2: Data-Strobe Encoding	11
Figure 2-3: Link Interface State Machine (ECSS 2008 A).....	13
Figure 2-4: Physical Addressing	16
Figure 2-5: Logical Addressing	17
Figure 2-6: Time-Slots	19
Figure 2-7: Overall Network Topology	20
Figure 2-8: Example Virtual Buses.....	20
Figure 2-9: Static Bus Operation	22
Figure 2-10: Dynamic Bus Operation	23
Figure 2-11: Asynchronous Bus Operation.....	25
Figure 2-12: Packet Bus Operation	27
Figure 2-13: MMS Network (Raphael, et al. 2014).....	30
Figure 2-14: ASTRO-H Network (Ozaki, et al. 2010).....	33
Figure 2-15: JUICE Network (Airbus Defence & Space 2015)	35
Figure 2-16: Example MIL-STD-1553B Bus Architecture	37
Figure 2-17: BC to RT Transfer (US Department of Defense 1978).....	38
Figure 2-18: RT to BC Transfer (US Department of Defense 1978).....	38
Figure 2-19: RT to RT Transfer (US Department of Defense 1978).....	39
Figure 2-20: On-Board Data Systems (European Space Agency 2014 A)	56
Figure 4-1: Initiator Layers	65
Figure 4-2: Loading an RTEMS Program to the LEON2-FT	70

Figure 4-3: RTEMS Program Memory Layout.....	72
Figure 4-4: SpaceWire Protocol Engine Interrupt Vectoring	76
Figure 4-5: Using STAR-Gate to Connect to the LEON2-FT Processor Board.....	78
Figure 4-6: Ticker Program UART Output.....	79
Figure 4-7: SpaceWire-D Layer Software Architecture	80
Figure 5-1: SpaceWire-D Demonstrator PXI Rack	98
Figure 5-2: SpaceWire-D Demonstrator Network Topology.....	99
Figure 5-3: SpaceWire-D Demonstrator Interactions	102
Figure 5-4: Example Script 1 – Simple Static Bus Schedule.....	105
Figure 5-5: Transaction Command	106
Figure 5-6: Transaction Group Command	107
Figure 5-7: Slot Command.....	107
Figure 5-8: Open Static Bus Command	108
Figure 5-9: Controlling Command Execution.....	109
Figure 5-10: Load Static Bus Command.....	111
Figure 5-11: Example Script 2 – More Complex Static Bus Schedule.....	112
Figure 5-12: Example Script 3 – Multiple Difference Types of Buses.....	116
Figure 5-13: Packet Bus Operation Command	116
Figure 5-14: Target Internal Authorisation Parameters	120
Figure 5-15: Initiator Configuration Program.....	123
Figure 5-16: Target Configuration Program	124
Figure 5-17: Network Manager Program.....	125
Figure 5-18: Target Monitor Program.....	127
Figure 5-19: Target Monitor Schedule View	128
Figure 5-20: Updated Target Monitor Schedule View	129

Figure 5-21: Target Monitor Target Statistics View	130
Figure 5-22: Target Monitor Command List View	131
Figure 6-1: Example Script 1 – Target Monitor Schedule View	134
Figure 6-2: Example Script 1 – Time-Slot 0.....	135
Figure 6-3: Example Script 1 – Static Bus 0, Transaction 0.....	135
Figure 6-4: Example Script 1 – Static Bus 0, Transaction 1	136
Figure 6-5: Example Script 1 – Static Bus 0, Transaction 2.....	136
Figure 6-6: Example Script 1 – Network Manager Statistics View.....	137
Figure 6-7: Example Script 2 – Target Monitor Schedule View	138
Figure 6-8: Example Script 2 – Target Monitor Command List View	139
Figure 6-9: Example Script 3 – Target Monitor Schedule View	141
Figure 6-10: Example Script 3 – Time-Slot 16.....	142
Figure 6-11: Example Script 3 – Asynchronous Bus 16, Transaction 0.....	142
Figure 6-12: Example Script 3 – Asynchronous Bus 16, Transaction 1	143
Figure 6-13: Example Script 3 – Asynchronous Bus 16, Transaction 2.....	143
Figure 6-14: Example Script 3 – Asynchronous Bus 16, Transaction 3.....	144
Figure 6-15: Example Script 3 – Asynchronous Bus 16, Transaction 4.....	144
Figure 6-16: Example Script 3 – Asynchronous Bus 16, Transaction 5.....	145
Figure 6-17: Example Script 3 – Time-Slot 32.....	145
Figure 6-18: Example Script 3 – Packet Bus 32, Packet Channel Status Read 0	146
Figure 6-19: Example Script 3 – Packet Bus 32, Packet Channel Status Read 1	146
Figure 6-20: Example Script 3 – Packet Bus 32, Packet Channel Status Read 2	147
Figure 6-21: Example Script 3 – Packet Bus 32, Packet Channel Status Read 3	147
Figure 6-22: Example Script 3 – Packet Bus 32, Packet Channel Status Read 4	148
Figure 6-23: Example Script 3 – Packet Bus 32, Packet Channel Status Read 5	149

Figure 6-24: Example Script 3 – Time-Slot 34.....	149
Figure 6-25: Example Script 3 – Packet Segment Transfer.....	150
Figure 6-26: Example Script 3 – Time-Slot 36.....	151
Figure 6-27: Example Script 3 – EOP Transaction.....	151
Figure 7-1: Problem Solving Overview	164
Figure 7-2: Network Topology with Potential Collisions.....	165
Figure 7-3: Shortest Path Selection with Collisions	166
Figure 7-4: Path Selection with No Collisions.....	166
Figure 7-5: Weighted Search Initial Costs	167
Figure 7-6: Weighted Search Costs after First Path Selected	169
Figure 7-7: Weighted Search Costs after All Paths Selected.....	170
Figure 7-8: Example Conflict Graph.....	171
Figure 7-9: Periodic Bandwidth Scheduling Algorithm	173
Figure 7-10: Periodic Bandwidth Scheduling Algorithm Block Diagram.....	174
Figure 7-11: Example Periodic Bandwidth Requirements Conflict Graph	175
Figure 7-12: Aperiodic Bandwidth Requirement Scheduling Algorithm	177
Figure 7-13: Aperiodic Bandwidth Scheduling Algorithm Block Diagram	178
Figure 7-14: Example Payload Data Conflict Graph	180
Figure 7-15: Example Payload Data Allocation	181
Figure 7-16: First-Fit Heuristic Block Diagram	183
Figure 7-17: First-Fit Heuristic Allocation	184
Figure 7-18: Best-Fit Heuristic Allocation	185
Figure 7-19: Best-Fit Heuristic Block Diagram.....	185
Figure 7-20: Least-Conflicting Heuristic Block Diagram	187
Figure 7-21: Complete SpaceWire-D Scheduling Algorithm.....	188

Figure 8-1: Generating Network Topology Stages	192
Figure 8-2: Network Topology Generation Algorithm Block Diagram	194
Figure 8-3: Requirement Generation Algorithm Block Diagram	195
Figure 8-4: Test Case Generation Software Flow	196
Figure 8-5: Test Case File Format	197
Figure 8-6: Example Test Case File	198
Figure 8-7: Example Test Case Architecture	199
Figure 8-8: Network Topology for Test Case (small_001).....	204
Figure 8-9: Number of Conflicts Ratios	210
Figure 8-10: Payload Slots Used Ratios	212
Figure 8-11: Experimenting with Dynamic Penalty Values	214
Figure 8-12: Load Balancing Dynamic Penalties	225
Figure 12-1: LEON2-FT Processor Board Architecture (Parkes, McClements and Mantelet, et al. 2013)	251
Figure 12-2: RMAP Initiator Descriptor Array	252
Figure 13-1: Example Script 2 – Time-Slot 0.....	258
Figure 13-2: Example Script 2 – Static Bus 0, Transaction 0.....	258
Figure 13-3: Example Script 2 – Static Bus 0, Transaction 1	259
Figure 13-4: Example Script 2 – Static Bus 0, Transaction 2.....	259
Figure 13-5: Example Script 2 – Time-Slot 16.....	260
Figure 13-6: Example Script 2 – Static Bus 16, Transaction 0.....	260
Figure 13-7: Example Script 2 – Static Bus 16, Transaction 1	261
Figure 13-8: Example Script 2 – Static Bus 16, Transaction 2.....	261
Figure 13-9: Example Script 2 – Time-Slot 32.....	262
Figure 13-10: Example Script 2 – Static Bus 32, Transaction 0.....	262

Figure 13-11: Example Script 2 – Static Bus 32, Transaction 1	263
Figure 13-12: Example Script 2 – Static Bus 32, Transaction 2	263
Figure 13-13: Example Script 2 – Multi-Slot Time-Code	264
Figure 13-14: Example Script 2 – Time-Slot 48	264
Figure 13-15: Example Script 2 – Static Bus 48, Transaction 0	264
Figure 13-16: Example Script 2 – Static Bus 48, Transaction 1	265
Figure 13-17: Example Script 2 – Static Bus 48, Transaction 2	265

List of Tables

Table 2-1: Character Sequences	12
Table 2-2: Routing Tables.....	17
Table 2-3: Initiator Schedules	28
Table 2-4: CAN Bus Identifier Arbitration.....	43
Table 2-5: Communication Network Comparison.....	54
Table 5-1: SpaceWire-D Demonstrator Address Scheme.....	101
Table 5-2: Example Command Execution Parameters	109
Table 7-1: Initiator Parameters	160
Table 7-2: Target Parameters	160
Table 7-3: System Parameters.....	161
Table 7-4: RMAP Command and Reply Packet Sizes.....	161
Table 7-5: Paths Selected with the BFS Algorithm	168
Table 7-6: Paths Selected with Dijkstra's Algorithm	168
Table 7-7: Example Periodic Bandwidth Requirements.....	175
Table 7-8: Example Periodic Bandwidth Requirements Static Buses	175
Table 7-9: Example Payload Data Bandwidth Requirements.....	180
Table 8-1: Test Case Classes.....	200
Table 8-2: Experiment 1: BFS Path Selection Results.....	201
Table 8-3: Paths for Test Case (small_000).....	205
Table 8-4: Schedule for Test Case (small_000).....	206
Table 8-5: Experiment 2: Weighted Search Path Selection Results	209
Table 8-6: JUICE Periodic Bandwidth Requirements	218

Table 8-7: JUICE Transformed Aperiodic Bandwidth Requirements	219
Table 8-8: JUICE Peak Throughput Payload Data Bandwidth Requirements	220
Table 8-9: Initial Result for JUICE Schedule	221
Table 8-10: MAJIS to SSMM Conflicting Paths	223
Table 8-11: JUICE Schedule with Load Balancing Dynamic Penalties	224
Table 12-1: Example RMAP Reply Demultiplexer Configuration	254
Table 12-2: Example RMAP Command Demultiplexer Configuration	255

List of Acronyms

ACS	Attitude Control Subsystem
AHB	Advanced High-Performance Bus
AOCP	Attitude and Orbital Control Processor
AOCS	Attitude and Orbital Control Subsystem
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BC	Bus Controller (MIL-STD-1553B)
BF	Best-Fit
BFS	Breadth-First Search
BM	Bus Monitor (MIL-STD-1553B)
BPP	Bin Packing Problem
BSP	Board Support Package
CAN	Controller Area Network
CCSDS	Consultative Committee for Space Data Systems
CIDP	Central Instrument Data Processor
CRC	Cyclic Redundancy Check
C & C	Command and Control
DCL	Debug Communications Link
DMA	Direct Memory Access
DR	Data Recorder

DS	Data-Strobe
DSU	Debug Support Unit
EEP	Error-End-of-Packet
ELF	Executable and Linkable File
EOP	End-of-Packet
ESA	European Space Agency
ESC	Escape Character
ESTEC	European Space Research and Technology Centre
FCT	Flow-Control Token
FF	First-Fit
FPGA	Field-Programmable Gate Array
GDB	GNU Debugger
GEO	Geostationary Orbit
GPS	Global Positioning System
GSFC	Goddard Space Flight Center
HXI	Hard X-Ray Imaging System
IEEE	Institute of Electrical and Electronics Engineers
INI	Initiator
IPT	Initiator Processing Time
IRQ	Interrupt Request
ISR	Interrupt Service Routine
IXV	Intermediate eXperimental Vehicle
JUICE	JUpiter ICy moons Explorer
LC	Least-Conflicting

LEO	Low Earth Orbit
LVDS	Low-Voltage Differential Signalling
MAJIS	Moons and Jupiter Imaging Spectrometer
MMS	Magnetospheric Multiscale Mission
NASA	National Aeronautics and Space Administration
NULL	Null Character
OBC	On-Board Computer
OBDAH	On-Board Data-Handling
OCS	Orbit Control Subsystem
OD	Object Dictionary
PDO	Process Data Object
PROBA	Project for On-Board Autonomy
QoS	Quality of Service
RMAP	Remote Memory Access Protocol
RT	Remote Terminal (MIL-STD-1553B)
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real-Time Operating System
RTR	Router
SAR	Synthetic Aperture Radar
SDO	Service Data Object
SGD	Soft Gamma-Ray Detector
SMU	Satellite Management Unit
SOIS	Spacecraft On-Board Interface Services
SSMM	Solid-State Mass-Memory
SXI	Soft X-Ray Imaging System

SXS	Soft X-Ray Spectrometer System
TAR	Target
TBR	Trap Base Register
TC	Time-Code
TCIM	Telemetry Command Interface Module
TID	Transaction ID
TTC	Telemetry, Tracking and Command
UART	Universal Asynchronous Receiver/Transmitter
WCET	Worst-Case Execution Time

Glossary

Blocked	When a SpaceWire packet is held in one or more buffers while it waits for a required SpaceWire interface to be freed.
Cargo	The section of a SpaceWire packet between the destination address and the EOP that contains the packet's data.
Epoch	A single iteration of 64 time-slots of a SpaceWire-D initiator's schedule.
L-char	A SpaceWire link character.
Logical Address	A byte with a value of 32-254 at the head of a SpaceWire packet that indicates the entry within the routing table that describes how the packet should be routed.
N-char	A SpaceWire normal character.
Physical Address	A byte with a value of 0-31 at the head of a SpaceWire packet that indicates the next physical port that the packet should be transmitted out of.
SpaceWire Link	A connection between two SpaceWire interfaces.
SpaceWire Node	A source or destination of SpaceWire traffic.

SpaceWire Network	A collection of SpaceWire nodes and routers connected by SpaceWire links.
SpaceWire Router	A device used to switch packets between two or more SpaceWire interfaces.
RMAP Initiator	A device that transmits RMAP commands and processes RMAP replies.
RMAP Target	A device that processes RMAP commands and transmits RMAP replies.
RMAP Transaction	The process of an initiator transmitting an RMAP command, a target receiving and executing the command, the target transmitting an RMAP reply, and the initiator receiving and handling the reply.

Acknowledgements

I would like to thank my principal supervisor, Prof. Steve Parkes, for allowing me the opportunity to work on this project and for his support during the completion of this research. I would also like to thank my second supervisor, Dr. Karen Petrie, for her support and feedback throughout this project.

My colleagues at STAR-Dundee and the Space Technology Centre were extremely helpful and supportive during the last four years and I would like to thank them for their advice, feedback and for providing their technical expertise when working with the many SpaceWire devices that were used on this project.

Finally, I'd like to thank my family and friends for their support and encouragement.

Declaration of the Candidate

I hereby declare that I am the author of this thesis; that, unless otherwise stated, all references cited have been consulted by me; that the work of which this thesis is a record has been done by me, and that it has not been previously accepted for a higher degree.

David James Gibson

October, 2017

Declaration of the Supervisor

I hereby declare that David James Gibson has satisfied all the terms and conditions of the regulations made under Ordinances 12 and 39, and has completed the required nine terms of research to qualify in submitting this thesis in application for the degree of Doctor of Philosophy.

Steve M Parkes

October, 2017

Abstract

SpaceWire-D is an extension to the SpaceWire protocol that adds deterministic capabilities over existing equipment. It does this by using time-division multiplexing, controlled by the sequential broadcasting of time-codes by a network manager. A virtual bus abstraction is then used to divide the network architecture into segments in which all traffic is controlled by a single Remote Memory Access Protocol (RMAP) transaction initiator. Virtual buses are then allocated a number of time-slots in which they are allowed to operate, forming the SpaceWire-D schedule.

This research starts by contributing an efficient embedded SpaceWire-D software layer, running on top of the RTEMS real-time operating system, for use in the initiators of a SpaceWire-D network. Next, the SpaceWire-D software layer was used in two LEON2-FT processor boards in combination with multiple other RMAP target boards, routers, a network manager, and a host PC running a suite of applications to create a SpaceWire-D Demonstrator. The SpaceWire-D software layer and SpaceWire-D Demonstrator were used to verify and demonstrate the SpaceWire-D protocol during the ESA SpaceWire-D project and resulted in multiple deliverables to ESA.

Finally, this research contributes a novel SpaceWire-D scheduling strategy using a combination of path selection and transaction allocation algorithms. This strategy allows for a SpaceWire-D network to be defined as a list of periodic, aperiodic and payload data bandwidth requirements and outputs a list of paths and an allocation of transactions to time-slots which satisfy the networking requirements of a mission.

Chapter 1

Introduction

SpaceWire is a communication network used on-board spacecraft to facilitate digital communication between scientific instruments, mass-memory storage devices, on-board computers, downlink telemetry and other subsystems (ECSS 2008 A). To allow data to flow between multiple devices, SpaceWire networks use packet-switching routers to direct traffic through the network. These routers use a mechanism called wormhole routing which means that as soon as a packet enters an input port on the router, it is immediately switched through the output port, assuming the port is not already in use. This routing mechanism simplifies the memory requirements within the routers as there is reduced packet buffering. However, it may cause a packet to be delayed if the packet's required output port is already in use. This problem is known as blocking and it can introduce variable packet propagation times as well as network congestion if multiple packets are strung out across the network, possibly causing further blocking. This means that an unscheduled SpaceWire network is not suitable for handling critical real-time traffic, which requires determinism. Therefore, additional quality-of-service (QoS) mechanisms need to be in place before a SpaceWire network can be used to handle both payload data-handling and command and control traffic.

To provide deterministic features on top of a network consisting of existing SpaceWire devices, a new protocol called SpaceWire-D (Parkes, Gibson and Ferrer 2015 B) can be used. SpaceWire-D slices a SpaceWire network in two dimensions. Firstly, the network topology is divided into segments called virtual buses, which are sections of the network where all traffic is controlled by a single initiator. Secondly, network time is divided into time-slots, controlled by the broadcasting of time-codes by a time-code master and in which Remote Memory Access Protocol (RMAP) (ECSS 2010 B) transactions are executed. Allocating time-slots to virtual buses can prevent blocking as long as virtual buses that are allocated the same time-slot do not share any links.

SpaceWire-D networks must be designed so that they satisfy the bandwidth requirements of a mission whilst adhering to the rules of the standard. There is a need for computational methods to generate the schedules to meet these goals.

1.1 Research Questions

This project aims to answer the following question, divided into three parts:

How can the SpaceWire-D protocol be used to fulfil the bandwidth requirements of a space mission?

1. How can an efficient SpaceWire-D software layer be designed on top of existing SpaceWire devices?
2. How can a system using the SpaceWire-D protocol be prototyped in order to demonstrate the standard?
3. How can a SpaceWire-D mission's bandwidth requirements be represented and satisfied computationally?

1.2 Outcomes

Work related to this project has appeared in the following publications:

- D. Gibson, S. Parkes, and K. Petrie. Modeling Deterministic Spacecraft Networks with Constraint Programming. *19th International Conference on Principles and Practices of Constraint Programming (Doctoral Program)*, Uppsala, Sweden, 2013, (Gibson, Parkes and Petrie 2013)
- D. Gibson, S. Parkes, C. McClements, S. Mills, D. Paterson. SpaceWire-D on the Castor Spaceflight Processor. *6th International SpaceWire Conference*, Athens, Greece, 2014, (Gibson, Parkes, et al. 2014)
- D. Paterson, D. Gibson, S. Parkes. An RTEMS Port for the AT6981 SpaceWire-Enabled Processor: Features and Performance. *6th International SpaceWire Conference*, Athens, Greece, 2014, (Paterson, Gibson and Parkes 2014)
- S. Parkes, D. Gibson, A. Ferrer. SpaceWire-D: Deterministic Data Delivery over SpaceWire. *DASIA International Space System Engineering Conference*, Warsaw, Poland, 2014, (Parkes, Gibson and Ferrer 2014)
- S. Parkes, D. Gibson, A. Ferrer. Experimental Results for SpaceWire-D. *DASIA International Space System Engineering Conference*, Barcelona, Spain, 2015, (Parkes, Gibson and Ferrer 2015 A)
- D. Gibson, S. Parkes, C. McClements, S. Mills. SpaceWire-D Prototype and Demonstration System. *7th International SpaceWire Conference*, Yokohama, Japan, 2016, (Gibson, Parkes, et al. 2016)

In addition, the SpaceWire-D Demonstrator described in Chapter 5 was used to complete the verification activity of the ESA SpaceWire-D project. It was then delivered to ESA and installed at ESTEC in Noordwijk, The Netherlands.

Furthermore, the author has presented work related to this thesis at two SpaceWire Working Group meetings at ESTEC and also gave an invited tutorial on SpaceWire-D at the 7th International SpaceWire Conference in Yokohama, Japan, 2016.

1.3 Thesis Structure

Chapter 2 gives an overview of spacecraft systems, the SpaceWire standard and how SpaceWire-D provides determinism over existing SpaceWire networks. A number of missions using SpaceWire and SpaceWire-D are also described. The chapter concludes with a comparison between SpaceWire-D and other existing deterministic networks used for command and control traffic in space.

Chapter 3 presents the research questions in more detail and describes how the following chapters answer them.

Chapter 4 describes the design of an efficient SpaceWire-D software layer.

Chapter 5 describes the design of the SpaceWire-D Demonstrator and Chapter 6 presents results gathered whilst performing a series of experiments using the Demonstrator.

Chapter 7 describes computational methods for scheduling SpaceWire-D networks and Chapter 8 presents results for randomised test cases and a case study of the JUICE mission.

Finally, Chapter 9 describes some directions for future work and Chapter 10 provides a conclusion to the thesis.

Chapter 2

Background

An on-board data-handling (OBDH) network allows the subsystems within a spacecraft to communicate with each other. Scientific instruments generate data which is sent through the network to be stored in one or more mass-memory devices before being sent to a ground receiving station on Earth via the Telemetry, Tracking and Command (TTC) system during a communication window. Telecommands sent from the spacecraft's operators on Earth are received via the TTC system and forwarded through the network to an on-board computer for execution. Housekeeping telemetry is either passed to or gathered by the on-board computer so that it can be transmitted to a ground station to allow the spacecraft operators to monitor its status.

2.1 Spacecraft Subsystems

A spacecraft is composed of multiple subsystems which each have their own responsibilities. For example, ensuring that the correct position and orientation are maintained, facilitating communication with the spacecraft operators on Earth, controlling other devices and generating and storing scientific data.

2.1.1 Attitude and Orbit Control

The attitude of a spacecraft is its rotational position around its centre of mass. A spacecraft must be pointed in the correct direction so that it can fulfil its mission. For example, a spacecraft that is capturing images of Earth from a low earth orbit (LEO)

must have its instruments directed at the required location on Earth and when a communication window opens between the spacecraft and one of its ground stations, the spacecraft's downlink antenna must be directed to the station.

The attitude control subsystem (ACS) is responsible for controlling the orientation of the spacecraft by making adjustments. Attitude is measured through the use of sensors such as star trackers which use known star patterns as references in order to calculate the rotational position of the spacecraft. The on-board computer can use these measurements to calculate any adjustments that need to be made and if so, they are passed as commands to actuators such as thrusters or reaction wheels for execution.

Similarly, the orbit control subsystem (OCS) allows the spacecraft to adjust its velocity in order to maintain its current orbit or possibly transition into another orbit if that is a requirement of the mission. Velocity changes are made using thrusters commanded by the on-board computer.

2.1.2 Communications

The communication between a spacecraft and its operators on Earth is provided by the TTC system. Housekeeping information and scientific data is sent from the spacecraft to one or more ground stations on Earth. Telecommands are generated by the spacecraft operators and sent from one or more ground stations to the spacecraft for execution.

Due to the position of the spacecraft and the curvature and rotation of Earth, communication may not be possible at all times. For example, if a spacecraft enters a position behind a body such as a planet, it will not be able to communicate with a ground station. As another example, if a spacecraft is in a low Earth orbit (LEO), its view to a ground station may be obscured by the curvature of the Earth. Additionally,

in a LEO, the spacecraft may not pass over the same locations during its orbit due to the rotation of the Earth. This further reduces the communication windows between the spacecraft and its ground station.

The communication windows available between a spacecraft and its ground stations affects the data storage requirements of a mission. If there are few communication windows, the spacecraft will require more mass-memory storage space in order to store data until it can be sent to a ground station.

2.1.3 On-Board Computers

An on-board computer (OBC) is responsible for executing embedded flight software that controls a part or all of the spacecraft. It may have one or more processors, timers, interrupt controllers, memory units and integrated peripherals such as network/bus controllers or ADC/DAC interfaces.

Typically, the OBC will execute its flight software on top of a real-time operating system such as VxWorks (Wind River 2016) or RTEMS (The RTEMS Project 2017) due to the deterministic features and relatively small memory footprint in comparison to general purpose operating systems.

2.1.4 Scientific Instruments

The main objective of a space mission is to deploy its scientific instruments so that data can be gathered and sent back to Earth. There is a wide range of different scientific instruments such as cameras, telescopes, telecommunications antennae, spectrometers, magnetometers and many others. For example, a satellite in a geostationary orbit (GEO) may include, in its payload, a large antenna to facilitate communication links between continents or a satellite in a LEO may include a number of cameras to perform earth observation.

In a science mission, each instrument generates scientific data packets which are typically transferred across a communications network and stored within a mass-memory device. When a communication window arrives, the spacecraft transmits its stored data back to Earth for interpretation and analysis.

For Earth observation missions, instruments may be capturing images of the Earth and therefore they may require very high data-rates compared to science missions which require relatively low data-rates. Depending on the communication windows, as described above, they may send their data back to Earth directly or store it in mass-memory first.

In addition to the science or Earth observation data, an instrument may generate housekeeping telemetry to inform the OBC of its status and receive telecommands to perform various actions.

For all types of space missions that require two or more devices to be interconnected, a technology such as SpaceWire may be used. SpaceWire packets are content-agnostic and can be used to transfer any type of data between devices. For future Earth observation missions with extremely high data-rate requirements, other networks may be required as described in Section 2.5.3.

2.2 SpaceWire

SpaceWire is a communication network used on-board spacecraft to facilitate communication between the mission payload's scientific instruments, mass-memory storage devices, on-board computers, downlink telemetry and other subsystems (ECSS 2008 A). SpaceWire enabled devices are connected by full-duplex data links, providing bi-directional data-flow at variable transmission rates of between 2 Mbit/s and 200 Mbit/s. The simplest SpaceWire network consists of two nodes with a point-

to-point link between them. If more complex network topologies are required, packet switching routers can be used to direct traffic between nodes.

2.2.1 Development History

In 1996, the IEEE-1355-1995 serial link standard (IEEE Computer Society 1996 A) was published and subsequently considered for use in OBDH networks. However, some problems were identified that would prevent it being used for space applications. Some of the potential improvements that were identified were the use of low voltage differential signalling (LVDS) (IEEE Computer Society 1996 B), (Telecommunications Industry Association 2012), improved cables and connectors and improved link initialisation and restart mechanisms (Guasch, Parkes and Christen 1999). Taking into consideration the IEEE-1355-1995 and LVDS standards as well as the identified optimisations, work began at the University of Dundee on the first draft of the SpaceWire standard (S. Parkes 1999). Revision A of the full standard was published by the ECSS in 2003 (ECSS 2003), revision B was not released and revision C was published in 2008 (ECSS 2008 A).

2.2.2 Protocol Stack

The SpaceWire standard defines several levels that form a protocol stack allowing communication between the physical transmission medium and the software application. As the protocol stack progresses upwards, beginning at the physical level, each level provides further abstraction from the hardware and ends at the network level which describes a system of SpaceWire devices. The protocol stack in a system with two applications is shown in Figure 2-1.

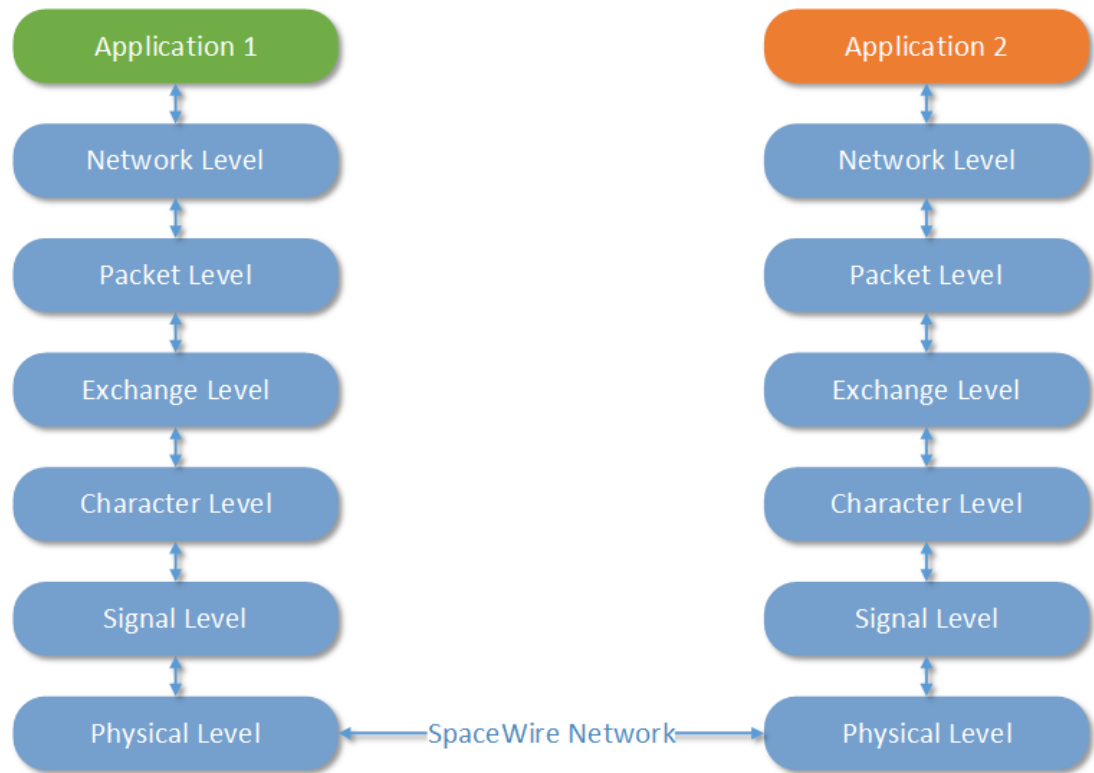


Figure 2-1: SpaceWire Protocol Stack (ECSS 2008 A)

As shown in Figure 2-1, in this example there are two applications running on SpaceWire enabled devices connected by a SpaceWire link or network at the physical level. Each level talks to the levels immediately above and below it. The following sections briefly describe each level.

2.2.2.1 Physical Level

The physical level describes the mechanical and electrical requirements of SpaceWire cables, connectors and printed circuit boards. Each cable consists of four shielded twisted pairs with an overall shield and can be up to 10 metres in length.

2.2.2.2 Signal Level

The signal level describes the encoding, voltage level and noise margin requirements of the signals driven along the cables. SpaceWire performs signalling using LVDS

(IEEE Computer Society 1996 B) which provides low power consumption and high immunity to noise.

SpaceWire signals are encoded using data-strobe (DS) which encodes the transmission clock along with the data signal as a pair of signals, data and strobe, as illustrated in Figure 2-2.

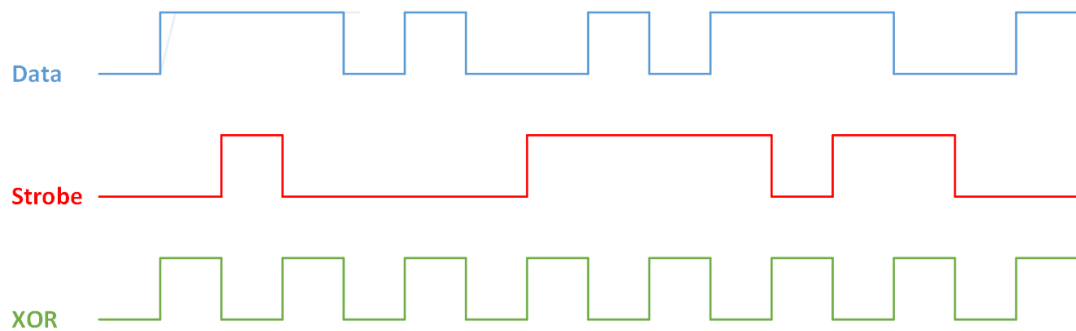


Figure 2-2: Data-Strobe Encoding

As shown in Figure 2-2, the data signal is unmodified and the strobe is a signal that changes voltage levels whenever the data signal is at the same value for two consecutive bits. The transmission clock can be recovered by XORing the data and strobe signals.

2.2.2.3 Character Level

At the character level, bits are grouped into different combinations that describe two classes of characters: data and control. A data character is a 10-bit sequence that combines an 8-bit block of data with a data-control flag set to zero to indicate a data character and an odd parity bit. A control character is a 4-bit sequence split into a 2-bit control code, a data-control flag set to one to indicate a control character and an odd parity bit. The different characters defined at this level are listed in Table 2-1.

Table 2-1: Character Sequences

	Bit													
	P	C	0	1	2	3	4	5	6	7	8	9	10	11
Data	P	0	D0	D1	D2	D3	D4	D5	D6	D7	-	-	-	-
FCT	P	1	0	0	-	-	-	-	-	-	-	-	-	-
EOP	P	1	0	1	-	-	-	-	-	-	-	-	-	-
EEP	P	1	1	0	-	-	-	-	-	-	-	-	-	-
ESC	P	1	1	1	-	-	-	-	-	-	-	-	-	-
NULL	P	1	1	1	0	1	0	0	-	-	-	-	-	-
TC	P	1	1	1	1	0	T0	T1	T2	T3	T4	T5	T6	T7

As shown in Table 2-1, the character level defines seven different characters: one data character, four normal control characters and two extended control characters. Each data character contains eight bits of data listed as D0-D7. The flow-control token (FCT) control character is used for flow control between two SpaceWire interfaces and the end-of-packet (EOP) and error-end-of-packet (EEP) characters are used to indicate the normal or erroneous termination of a packet. The two extended control characters, the null (NULL) and time-code (TC), are formed by concatenating the escape (ESC) control character with additional bits. The NULL character is sent continuously while a link is not being used, in order to keep the link active and avoid disconnections. The TC control character contains an 8-bit time-code where the first six least-significant bits, listed as T0-T5, indicate a time value and the two most significant bits, T6 and T7, contain control flags.

2.2.2.4 Exchange Level

The exchange level defines the mechanisms for controlling the state of a link during initialisation, nominal activity and recovery as well as handling flow-control and system time distribution. The control characters used for these mechanisms: the ESC,

NULL, FCT and TC characters are known as link-characters or L-chars and are not passed up the stack to the packet level.

A SpaceWire link can be in one of a number of states as illustrated by the link interface state machine from the SpaceWire standard (ECSS 2008 A), in Figure 2-3.

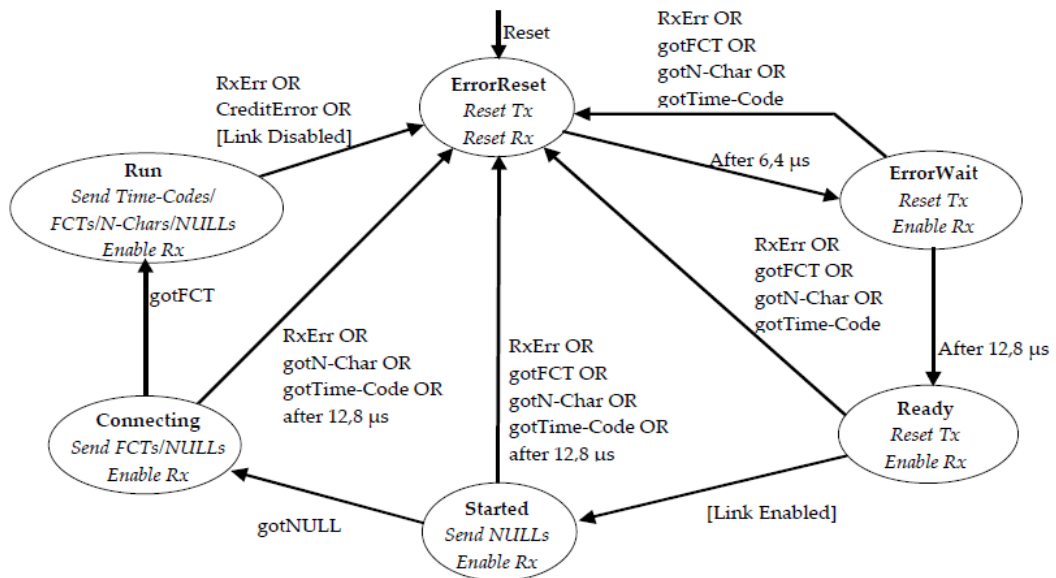


Figure 2-3: Link Interface State Machine (ECSS 2008 A)

As shown in Figure 2-3, when a link is reset, it enters the *ErrorReset* state and the transmitter and receiver are reset. After the reset signal is de-asserted, the state machine transitions to the *ErrorWait* state after a 6.4 µs delay. In the *ErrorWait* state, the transmitter is reset again and the receiver is enabled. After a 12.8 µs delay, the state transitions into the *Ready* state unless an error occurs or a character other than a NULL is received in which case the state returns to *ErrorReset*. The link waits in the *Ready* state until the link is enabled, either by the host or started automatically upon receipt of a NULL character if the AutoStart flag has been set. As in the *ErrorWait* state, if an error occurs or a character other than a NULL is received, the state transitions back to *ErrorReset*. Once the link is enabled, the link transitions into the

Started state, enabling the transmitter which begins sending NULL characters. Once a NULL has been received in the *Started* state or previously, in the *ErrorWait* or *Started* states, the link moves into the *Connecting* state. If an error occurs or a character other than a NULL is received or 12.8 μ s passes without receiving a NULL, the link returns to the *ErrorReset* state. In the *Connecting* state, the transmitter is enabled to allow transmission of FCTs as well as NULLs. When an FCT is received the link moves to the *Run* state. If an error occurs or any character other than a NULL or FCT is received or 12.8 μ s passes without receiving an FCT, the link transitions back to the *ErrorReset* state. Once in the *Run* state, the transmitter is fully enabled to send Time-Codes, FCTs, N-Chars and NULLs. If an error occurs, the state transitions back to *ErrorReset*.

Flow-control is handled by the exchange of FCTs sent across a link to indicate that the transmitter has enough space in the receive buffer to hold another eight N-chars. A credit count is maintained by the link interface which is updated each time an FCT is received or an N-char is transmitted. When there is no credit left, a link cannot transmit anymore N-chars until it receives another FCT. An outstanding counter is maintained by the link interface to indicate how many N-chars it is expecting to receive, determined by how many FCTs the link has transmitted and how many N-chars have been received. An FCT is transmitted whenever there is enough room to receive eight more N-chars.

System time distribution is implemented through the use of the TC extended control character. In a SpaceWire network, one device can be designated as the time-code master and is responsible for initiating the broadcasting of sequential time-codes. Each routing switch contains an internal time-code counter. When a link receives a time-code that is one more than the counter's current value, modulo 64, it increments the

counter. The time-code is then transmitted out of all other ports in the device, if they are configured to transmit time-codes. If the time-code received has the same value as the internal counter, it is ignored to prevent the circular distribution of time-codes. Otherwise, if the time-code received is not the next expected value, the counter is updated but the time-code is not transmitted out of the other links.

2.2.2.5 Packet Level

The packet level defines the composition of SpaceWire packets. A packet consists of a list of zero or more physical address bytes and a cargo section carrying the packet's data. Each packet is terminated by a marker to indicate the end of the packet which is an EOP in a packet terminating correctly and an EEP in a packet terminating erroneously.

2.2.2.6 Network Level

The network level defines the mechanisms used to allow the construction of more complex networks consisting of multiple nodes, links and routing switches.

A SpaceWire routing switch is a device with two or more links that allows a packet to be switched between an input and output link on the routing switch using a mechanism called wormhole routing. When a packet enters an input port on the routing switch, the first byte is read to determine the required output port for the packet. If the output port is free, the header byte is deleted from the packet and the remainder of the packet is immediately sent out of the output port, marking the output port as busy. If the output port is already busy, the packet must wait until the output port is free before being forwarded. Once the packet is terminated by either an EOP or EEP, the output port is freed, ready to be used by another packet if required. If the next device in the

path is a routing switch, the process repeats with the next header byte until the packet reaches the destination node.

There are two types of addressing defined in the packet level. The first is physical addressing which has already been described as a list of output ports which a packet must be forwarded out of to travel from the source of the packet to the destination node. At each step of the journey the leading physical address byte is stripped from the packet. An example of a packet being routed between a source, three routers and a destination node is shown in Figure 2-4. In this figure, the routers are named RTR0, RTR1 and RTR2.

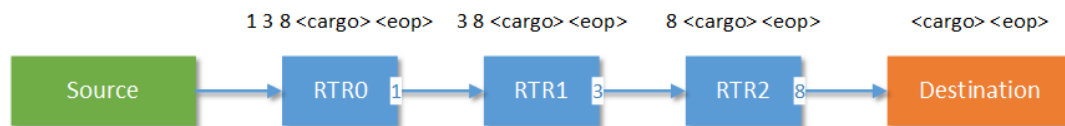


Figure 2-4: Physical Addressing

As shown in Figure 2-4, when the packet leaves the source and enters RTR0, the destination address list is 1, 3 and 8. The first address byte is deleted and the packet is forwarded out of output port 1. This process continues through the RTR1 and RTR2 until the packet finally reaches the destination node.

The second type of addressing is logical addressing where a single destination address, which is not deleted at each step, is used to route a packet through the network. When using logical addressing, each node has a unique identifier between 32 and 254 and each routing switch has a table that defines one or more ports to forward a packet addressed with a specific logical address to. This allows even a very large network to use a single byte for the destination address section of each packet rather than a potentially large list of physical addresses. If the example network from Figure 2-4 is used with a destination logical address of 32, the routing table is listed in Table 2-2.

Table 2-2: Routing Tables

Router	Logical Address	Ports
RTR0	32	1
RTR1	32	3
RTR2	32	8

As listed in Table 2-2, when a packet enters RTR0 with a logical address of 32, it will be forwarded out of port 1. In RTR1 and RTR2, the packet will be forwarded out of ports 3 and 8, respectively. The packet being sent from the source to the destination using logical addressing is shown in Figure 2-5.

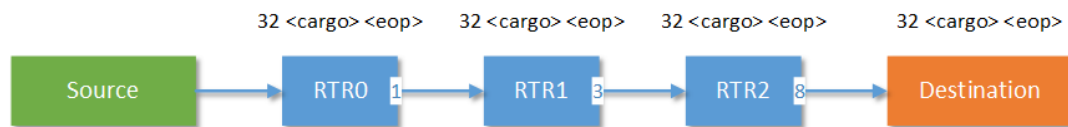


Figure 2-5: Logical Addressing

As shown in Figure 2-5, the packet now has only a single logical address byte at the start of the packet. When the packet enters a routing switch, it checks its routing table entry for logical address 32 and forwards it to the correct output port.

2.3 SpaceWire-D

SpaceWire-D is a protocol which extends SpaceWire networks with deterministic capabilities using a software layer on top of existing SpaceWire equipment (Parkes, Gibson and Ferrer 2015 B).

2.3.1 Motivation

As described in Section 1, unscheduled SpaceWire networks can suffer from blocking which can cause the network to be non-deterministic. If the network is being used for real-time traffic, such as command and control, this may cause critical deadlines to be

violated. The aim of SpaceWire-D is to solve this problem by providing deterministic features in order to ensure that blocking does not cause deadlines to be missed. It also aims to allow deterministic and non-deterministic traffic to share the same network. If these goals can be achieved, then complexity and cost may be reduced as the spacecraft now only requires a single network.

2.3.2 Operation

SpaceWire-D operates by controlling which parts of the network are allowed to send and receive traffic at specific times. Firstly, network time is divided into time-slots which are controlled by the distribution of consecutive SpaceWire time-codes. Secondly, the network is divided into segments called virtual buses where all traffic is controlled by a single initiator. Finally, each initiator has a schedule which describes which time-slots are allocated to the virtual buses. If a non-conflicting link rule is adhered to when creating the schedules, the possibility of blocking can be removed. This allows SpaceWire-D to satisfy the deterministic requirements of a communication network used for command and control applications.

2.3.2.1 Time-Slots

A SpaceWire-D time-slot is a period of time that begins when an initiator receives a time-code and ends when the initiator receives the next time-code. Time-codes contain a 6-bit time-value so there are 64 time-slots. For example, if an initiator receives time-code n , this signals the end of the previous time-slot, $(n - 1) \bmod 64$, and the start of time-slot n , as illustrated in Figure 2-6.

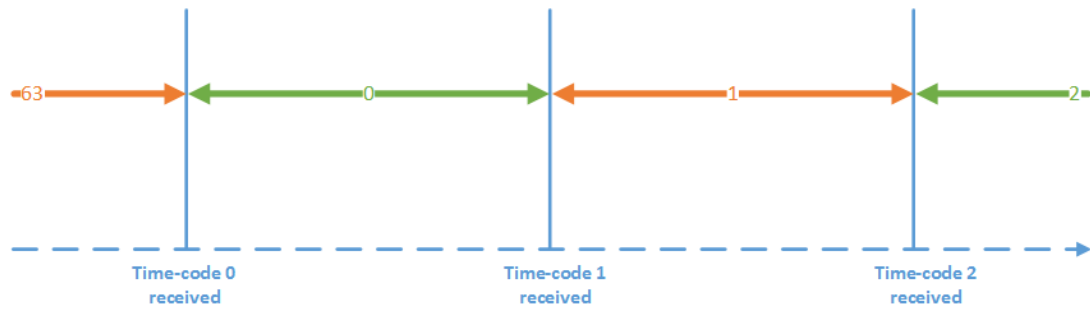


Figure 2-6: Time-Slots

In Figure 2-6, there is a timeline going left to right on the horizontal axis showing when time-codes are received by an initiator. At the start of the illustration, time-slot 63 is currently active. When time-code 0 is received by the initiator, this terminates time-slot 63 and signals the beginning of time-slot 0. The same process is repeated for the other time-codes.

A single time-code master is used to synchronise the initiators by sending out time-codes at fixed-length intervals, at a rate of 1-1024 Hz. This allows for between 1 and 16 schedule epochs per second, where an epoch is a single iteration of the schedule from time-slot 0 to time-slot 63. Each initiator listens for time-codes being received by, for example, installing an interrupt service routine (ISR) that is called whenever a time-code interrupt is raised. Alternatively, a time-code status flag could be polled if interrupts are discouraged. The initiator can then tell its SpaceWire-D layer that a new time-slot should be executed, which will, in turn, initiate any scheduled transactions.

2.3.2.2 *Virtual Buses*

Virtual buses are segments of the overall network that have a specific structure. They consist of a single RMAP initiator, one or more RMAP targets and the SpaceWire links that make up the paths between the initiator and the targets. For example, consider the network topology illustrated in Figure 2-7.

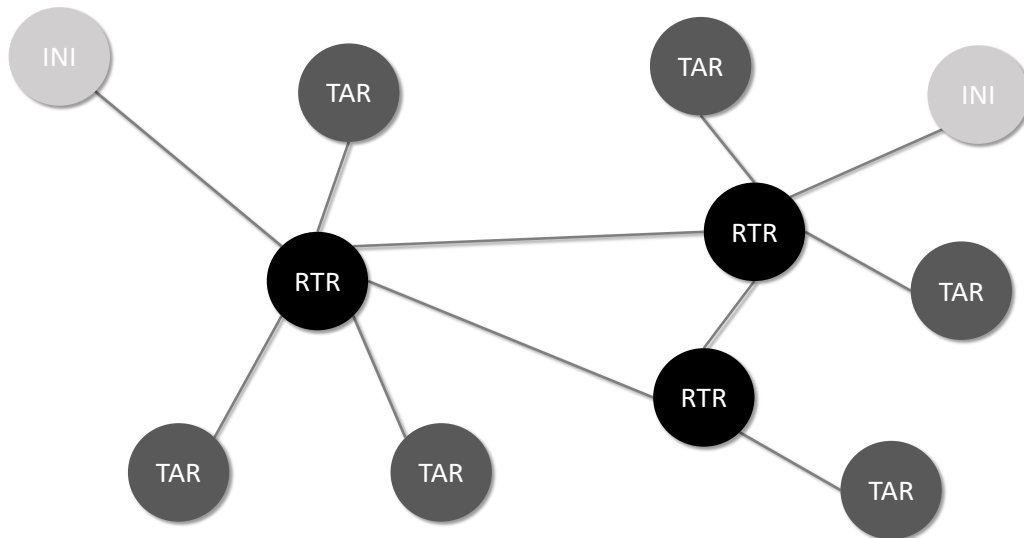


Figure 2-7: Overall Network Topology

In Figure 2-7, there is a network containing two initiators, six targets, three routers and some links to connect the different nodes and routers. Two possible virtual buses are shown in Figure 2-8.

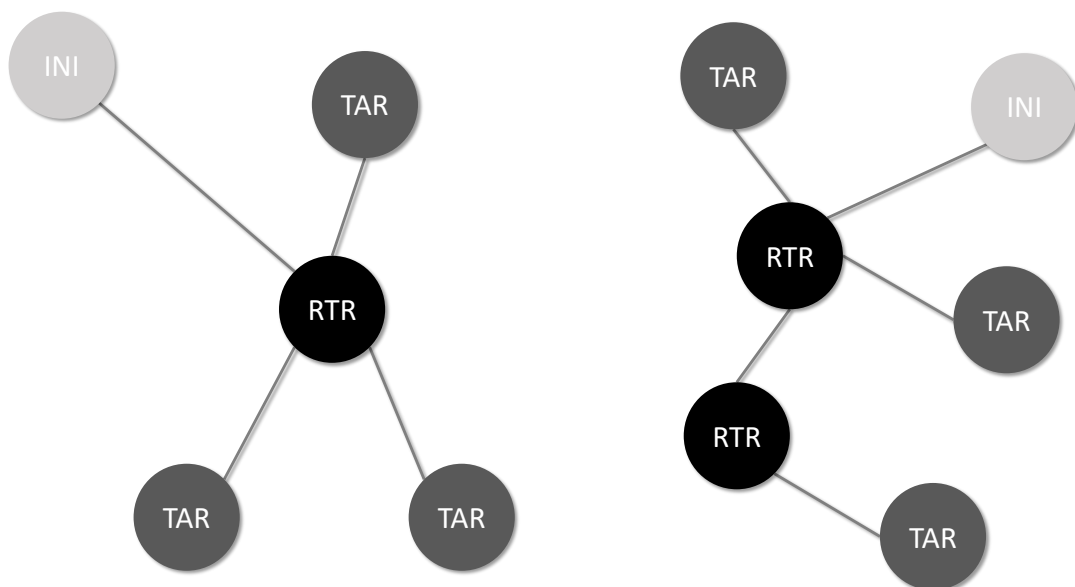


Figure 2-8: Example Virtual Buses

As shown in Figure 2-8, there are two virtual buses, each consisting of one initiator, three targets and the links between the nodes. In this example, the two virtual buses

have no shared links so they can be thought of as independent i.e. they can operate at the same time without packets on one virtual bus interfering with packets on the other.

A virtual bus is assigned to an initiator's schedule by allocating the virtual bus one or more time-slots in which it can operate. Virtual buses can change from one time-slot to the next and an initiator's schedule can contain a combination of any virtual bus types as long as their allocated time-slots don't overlap.

Initiators have four different functions related to virtual buses: an initiator opens a bus, defining its configuration and allocating it to the schedule; loads it, telling the bus what it should do in the next allocated time-slot; executes it during an allocated time-slot; and closes it when it's no longer required. There are four different types of virtual bus, each with their own implementations of the load and execute functions which provide features related to different classes of traffic which exist on an OBDH/C&C network.

Static Bus

The static bus is the simplest type of virtual bus. It is a segment of the network that is allowed to operate in a single time-slot in each schedule epoch. This means that transactions on a static bus are executed at the same time in each schedule, making the static bus the most deterministic type of virtual bus. The static bus is suited to repeating, periodic traffic such as housekeeping telemetry being gathered by an OBC.

When a static bus is opened by an initiator, the user passes a time-slot, a list of allowed targets and a slot size to the SpaceWire-D layer. The schedule is then checked to determine if the request to open a new static bus is valid. If the newly requested bus does not interfere with any existing virtual buses in the schedule, the slot is allocated.

A static bus is loaded with a group of one or more RMAP transactions. The transaction group can be repeated in every occurrence of the allocated time-slot or executed once, as a single-shot transaction group.

When the time-code master signals that the allocated time-slot should be executed, the SpaceWire-D layer looks at the static bus to determine if there is a transaction group to be executed and, if so, the initiator executes the transactions.

When a static bus is closed by the initiator, the user passes the identifier of the static bus and the SpaceWire-D layer frees the slot allocated to the bus.

The operation of the static bus is illustrated in Figure 2-9.

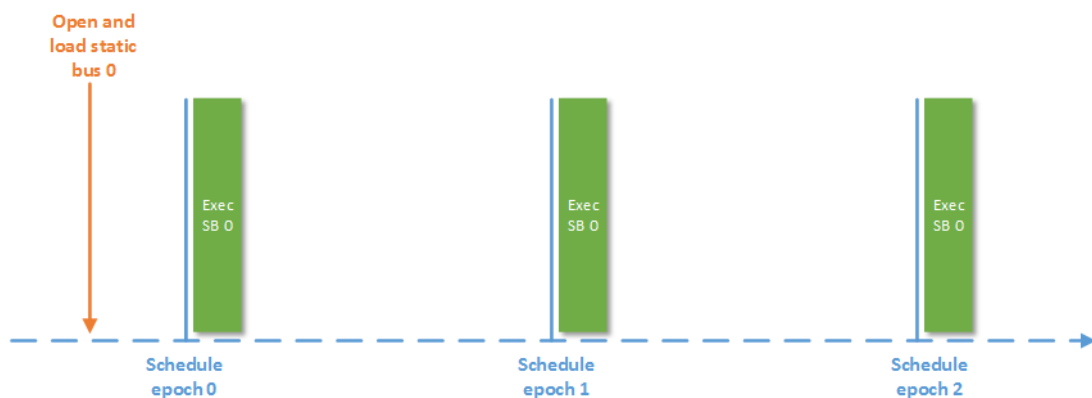


Figure 2-9: Static Bus Operation

In Figure 2-9, there is a timeline showing three schedule epochs. At some point before the first schedule epoch begins, the initiator opens a static bus in time-slot 0 and loads it with a transaction group. As time-codes are generated by the time-code master, the static bus is executed in each occurrence of time-slot 0 as shown by the green rectangles. This illustrates that a static bus executes a transaction group at the same time in each epoch of the schedule.

Dynamic Bus

The dynamic bus differs from the static bus in that it can be allocated multiple slots and transaction groups can be executed in any of those allocated. This means that it is less deterministic than the static bus because the time-slot in which a transaction group is executed may differ between schedule epochs.

When a dynamic bus is loaded with a transaction group, the group is executed once in the next allocated time-slot. The dynamic bus is suited to aperiodic traffic with deadlines, such as commands that must reach an instrument within a certain length of time after an event is detected by a sensor. If a dynamic bus has its slots allocated at intervals less than the required deadline, then the deadline will be satisfied no matter when the transaction group is loaded into the dynamic bus.

The operation of the dynamic bus is illustrated in Figure 2-10.



Figure 2-10: Dynamic Bus Operation

In Figure 2-10, an initiator has opened dynamic bus 0 and allocated it time-slots 0, 15 and 42. The filled rectangles show the slots in which transaction groups are executed and the non-filled rectangles show the slots where no transaction groups have been executed. The number of the time-slot is shown in parenthesis below the name of the dynamic bus. At four points in the figure, the initiator loads the dynamic bus with a

transaction group which is executed in the next available time-slot in the schedule. This illustrates that a dynamic bus can execute a transaction group at multiple different times in the schedule.

Asynchronous Bus

The asynchronous bus differs from the static and dynamic buses in that it is loaded with individual, prioritised transactions rather than transaction groups. The transactions are held in a queue and, just before an allocated time-slot occurs, a subset of the transactions are pulled from the head of the queue to form a transaction group to be executed in the next slot. When pulling the highest priority transaction from the queue, its execution time is added to a running total execution time. If the new total is less than the duration of a slot, the transaction is added to the transaction group. If a high priority transaction does not fit in the time-slot, it may be skipped in preference of lower priority transactions if they are small enough to fit in the remainder of the slot's capacity.

The asynchronous bus is non-deterministic because of the problem of priority starvation. This means that lower priority transactions cannot have their execution guaranteed because they may be overlooked in preference of higher priority transactions. It is suited to payload data traffic where, for example, an instrument is generating data packets that are to be written to a solid-state mass memory (SSMM) device for storage. As packets are generated, they can be added to the asynchronous bus transaction queue and sent in a best-effort manner with no deterministic guarantees.

The operation of the asynchronous bus is illustrated in Figure 2-11.

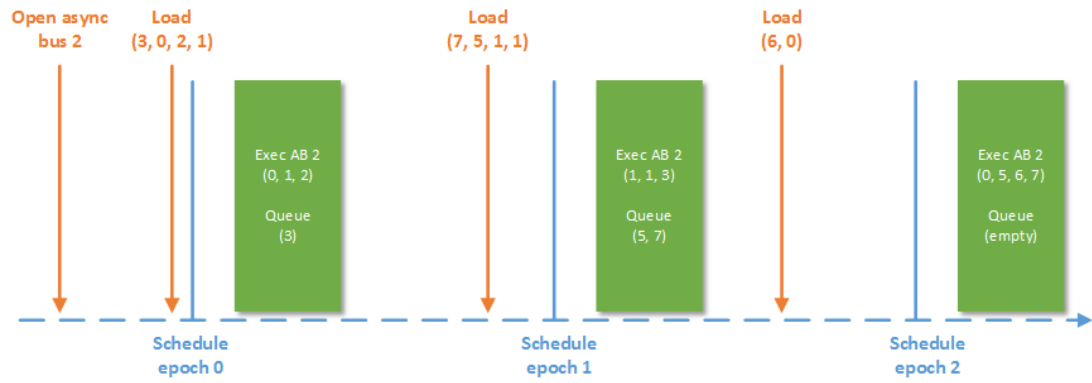


Figure 2-11: Asynchronous Bus Operation

In Figure 2-11, the initiator has opened an asynchronous bus and allocated it time-slot 2. Just before schedule epoch 0 begins, the initiator loads the asynchronous bus with four individual transactions with priorities 3, 0, 2 and 1. At this point the priority queue has reordered the transactions in the correct order, where the highest priority level is 0 and the lowest is 7. In time-slot 1 of schedule epoch 0, just before the allocated time-slot, a transaction group is prepared by pulling transactions from the head of the priority queue. In this example, three transactions are pulled, with priority levels 0, 1 and 2, to form the transaction group which is executed in the following time-slot. This leaves the queue with a single remaining transaction with priority level 3. This process is repeated in the next two schedule epochs.

Packet Bus

The targets in a packet bus take an active role in order to provide flow-control with the initiator. This is different to the static, dynamic and asynchronous buses, where the only action required by the targets is to return RMAP replies.

Rather than be loaded with transactions or groups of transactions, a packet bus is loaded with prioritised requests to transfer a packet between the initiator and the target. The packet bus does this using a flow-control abstraction called a packet channel. A

packet transfer operation is completed in three stages. Firstly, the initiator checks that the required packet channel in the target is ready to send or receive a packet. Secondly, the packet is transferred between the initiator and the target in one or more segments. Lastly, the operation is terminated by indicating to the target that the process is complete. This frees the packet channel up to be used for another transfer operation if required.

In the first stage, the initiator checks that the target's packet channel is ready to send or receive a packet. This is done to provide end-to-end flow-control between the initiator and the target and to ensure that a packet is not transferred between the two nodes until both sides are ready. The initiator signals that it is ready by pulling the transfer request from the packet bus queue. The target indicates its readiness by writing status information in a set of registers, or a data structure, in a known location for the packet channel. The initiator can then use an RMAP read command to read this status information and determine if the target has a buffer ready for the operation.

Once both sides are ready, they can move to stage two of the transfer operation. If the packet is too large to transfer in a single slot, then it must be segmented and transferred over multiple slots. In this stage, the segments are sent to the target via RMAP write transactions, or read from the target via RMAP read transactions.

Once all of the packet segments have been transferred, the initiator needs to tell the target that the process has been completed. This indicates to the target that the packet channel can now be used in another transfer operation. This stage could be done, for example, by using an RMAP write transaction to reset the status information in the packet channel's registers or data structure.

Packet transfer operations are prioritised in the same manner as transactions in the asynchronous bus. Just before a slot allocated to a packet bus occurs, a number of packet transfer operations are pulled from the head of the queue. Their transaction execution times, for the relevant stage, are checked and summed using the same method as the asynchronous bus. This means that a packet bus transaction group can contain a combination of stage one, two or three transactions depending on the current status of each operation. Again, because of the problem of priority starvation, this is a non-deterministic bus and would be suited to payload data traffic that requires end-to-end flow-control between the initiator and target.

The operation of the packet bus is illustrated in Figure 2-12.

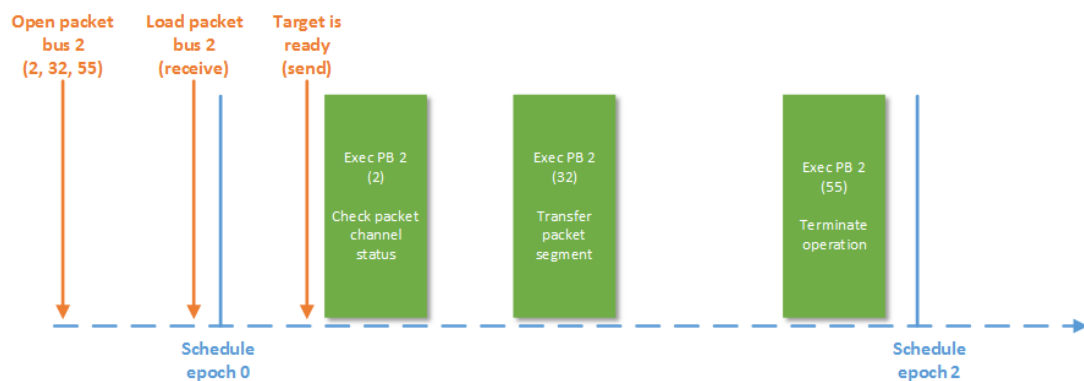


Figure 2-12: Packet Bus Operation

In Figure 2-12, the initiator has opened packet bus 2 and allocated it time-slots 2, 32 and 55. Before the first allocated time-slot occurs, the initiator loads the packet bus with a request to receive a packet from the packet channel in the target. The target then sets its packet channel status to indicate that it is ready to send a packet. When the first allocated time-slot begins, the initiator doesn't yet know that the target is ready to send a packet so it must read the packet channel status. Once the packet channel status has been read, the initiator knows that the target packet channel is ready as well as the

location and size of the packet. The initiator then uses this information to form an RMAP read transaction to transfer the packet from the target to the initiator which is executed in the next allocated time-slot. The operation has now been completed so, in the third allocated time-slot, the initiator uses an RMAP write transaction to indicate to the target that the packet channel is now free.

2.3.2.3 Schedules

Each initiator in a SpaceWire-D network contains its own schedule, in the form of time-slots allocated to virtual buses. The schedule controls which network resources and targets the initiator can interact with and when. An example of a set of initiator schedules is shown in Table 2-3.

Table 2-3: Initiator Schedules

Time-Slot	Initiator 1	Initiator 2	Initiator 3
0	Static bus 0	Empty	Dynamic bus 0
1	Async bus 1	Packet bus 1	Dynamic bus 0
2	Empty	Async bus 2	Empty
3	Empty	Empty	Static bus 3
4	Empty	Async bus 2	Packet bus 4
...
62	Async bus 1	Async bus 62	Packet bus 4
63	Static bus 63	Static bus 63	Static bus 63

In Table 2-3, the schedules for three initiators are listed. A schedule can contain any combination of virtual buses and different initiators can have different virtual buses in the same slot. The schedules should be designed so that they adhere to the no-blocking rule of SpaceWire-D. For example, in time-slot 1, each initiator has a virtual bus allocated. These virtual buses should be independent i.e. no two virtual buses share a common link. Otherwise, it would be possible for blocking to occur between packets

on different virtual buses, which could result in extended transaction execution times and missed deadlines.

2.4 Missions

SpaceWire has been widely used for payload communication networks by the world's major space agencies including ESA, NASA, JAXA and Roscosmos (European Space Agency 2015 B).

The following sections focus on three specific SpaceWire missions: the Magnetospheric Multiscale (MMS) Mission, ASTRO-H and the JUpiter ICy moons Explorer (JUICE). These missions were chosen as they use SpaceWire to handle payload communications as well as telemetry and telecommands and they are NASA, JAXA and ESA missions, respectively, which gives an insight into the international use of SpaceWire.

2.4.1 Magnetospheric Multiscale Mission

The aim of the MMS mission is to research the transfer of energy between the Sun and Earth's magnetic fields, known as magnetic reconnection. The mission intends to improve space weather forecasting which will, in turn, improve technologies that are affected by it such as communications networks, Global Positioning System (GPS) and power grids (NASA GSFC 2015).

The MMS mission consists of four satellites with identical instrument suites and was launched on the 12th of March 2015. It was carried by an Atlas V launch vehicle and inserted into an elliptical orbit around Earth. The four spacecraft fly in a pyramid formation allowing them to collect three dimensional data.

SpaceWire is used in the MMS spacecraft to handle telemetry, telecommands and payload data using a single network (Raphael, et al. 2014). The nodes include a signal

processing card, communication card, processor card, analogue card, instrument data multiplexer, engine valve drive and power subsystem monitor. Each device is redundant in case of failure of the nominal device. Redundancy is cold for all devices except the redundant communication board which is powered on so that it can receive and decode telecommands at any time. However, for simplicity, only the nominal devices are shown in Figure 2-13.

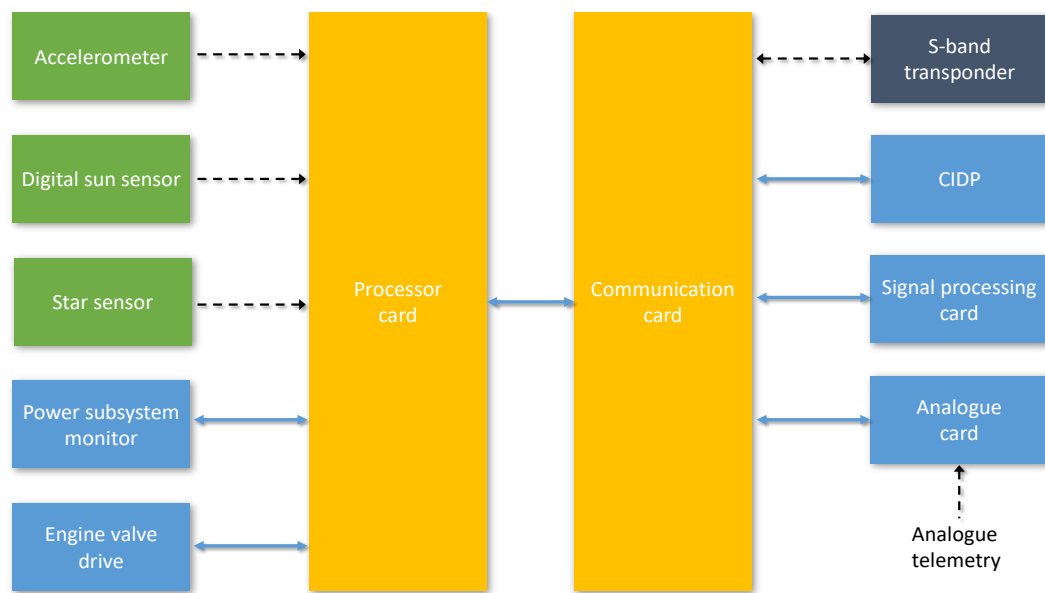


Figure 2-13: MMS Network (Raphael, et al. 2014)

In Figure 2-13, the unbroken blue arrows represent SpaceWire links and the dashed black arrows represent other digital or analogue connections. There are two SpaceWire routers, one inside the communication card and one inside the processor card. The processor card is connected by SpaceWire links to the communication card, power subsystem monitor and engine valve drive and connected to the sensors via Universal Asynchronous Receiver/Transceiver (UART) interfaces. The communication card and the S-band transponder are also connected via UART.

Each MMS satellite contains an identical suite of 11 instruments divided into three classes: Hot Plasma, Energetic Particles Detector and Fields. The Hot Plasma

instruments are used to investigate plasma detected during magnetic reconnection, the Energetic Particles Detector instruments record fast-moving particles and the Fields instruments measure electric and magnetic fields. The instruments are connected to the SpaceWire network through the Central Instrument Data Processor (CIDP) shown in Figure 2-13. The CIDP contains a SPARC processor, a 96 Gbyte mass-memory module and a power/analog card (Klar, et al. 2013).

The processor card contains a radiation-hardened ColdFire microprocessor operating at a clock rate of 40 MHz. Additional processing is provided by an Arbiter FPGA which implements features such as an interrupt controller, DMA and memory controllers and a flight software bootloader as well as interfaces to peripherals like the SpaceWire router and sensors.

The accelerometer and the star sensor send data periodically across the UART links between the sensors and the processor card at a rate of 4 Hz. Data is sent from the digital sun sensor in a similar manner but at a rate of three times a minute, once per spacecraft revolution. This data is used by the ACS flight software to calculate thruster commands which are transmitted over SpaceWire to the engine valve drive. The communication card receives housekeeping telemetry from the S-band transponder via UART which is sent to the processor card over SpaceWire. Housekeeping telemetry is gathered from the communication card by the processor card using SpaceWire at a rate of 1 Hz. Instrument telemetry and payload data is gathered by the CIDP and stored in mass-memory before being sent to the communication card over SpaceWire and downlinked to Earth via the S-band transponder.

To help with spacecraft rotation, system-time synchronisation and flow-control, GSFC have modified the SpaceWire protocol to implement distributed event signalling using time-codes.

2.4.2 ASTRO-H

ASTRO-H is an x-ray space observatory developed by JAXA, in collaboration with NASA. Its main objectives are to investigate the evolution of large astronomical structures such as galaxy clusters and supermassive black holes; to research extreme environments such as the effects of matter near black holes; to explore the role of astronomical events such as collisions on the energising of cosmic rays; and to study dark matter and energy and their effects on the evolution of galaxy clusters (Takahashi, et al. 2012).

The ASTRO-H mission is a single spacecraft and was launched in February 2016 on-board an H-IIA launcher. The scientific instruments suite consists of a hard x-ray imaging system (HXI) containing two identical pairs of hard x-ray telescopes and imagers; a soft x-ray spectrometer system (SXS) with a soft x-ray telescope and a calorimeter spectrometer; a soft x-ray imaging system (SXI) containing a soft x-ray telescope and an imager; and, finally, a soft gamma-ray detector (SGD).

SpaceWire is used on-board ASTRO-H to handle telemetry, telecommands and payload data using a single network. The network is connected physically but logically divided into two subnets. The first subnet contains the instruments, downlink telemetry, mass-memory and the main on-board computer. The second subnet contains the attitude and orbit control sensors and actuators (Yuasa, et al. 2011). The attitude and orbit control computer connects the two subnets, but no packets from either subnet

can reach the other. The network topology for ASTRO-H, simplified with no redundancy, is illustrated in Figure 2-14.

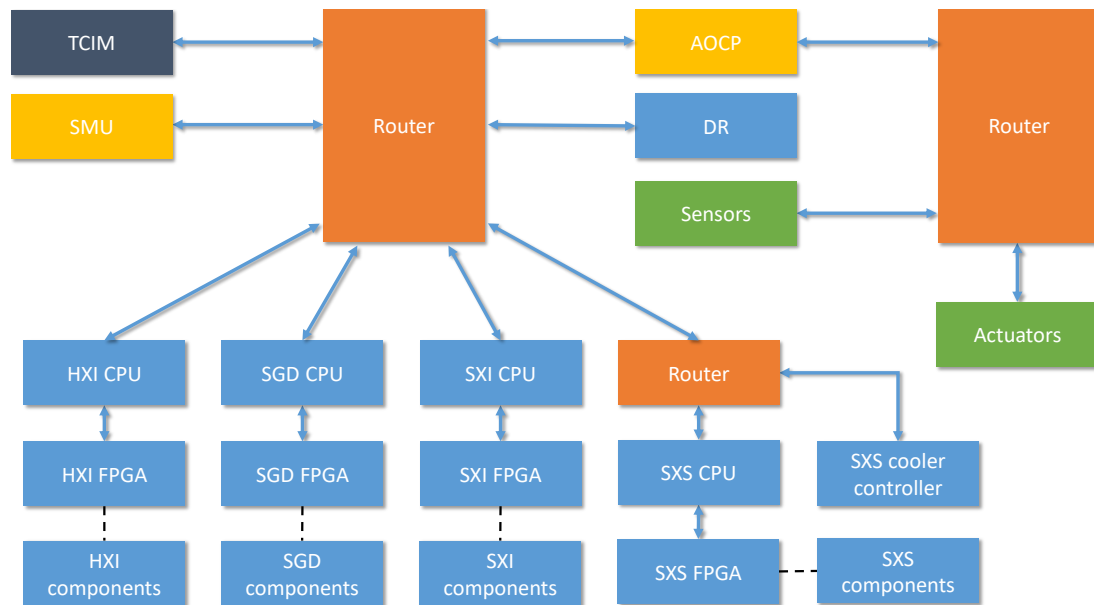


Figure 2-14: ASTRO-H Network (Ozaki, et al. 2010)

Similar to the previous network topology diagram for the MMS mission, the blue arrows in Figure 2-14 represent SpaceWire links and the dashed black lines represent non-SpaceWire links. The telemetry command interface module (TCIM) is used to receive telecommands and downlink housekeeping telemetry and payload data. The satellite management unit (SMU) is the main on-board computer and the initiator of all traffic on the data-handling subnet. The data recorder (DR) is the mass-memory device and is used to store housekeeping telemetry and payload data before being downlinked to a ground station. Each instrument has a SpaceWire enabled CPU board to connect it to the network which is also connected, via SpaceWire, to an FPGA board. The SXS system also has its own router to connect to both the SXS instrument and the cooling controller. The FPGA boards are then connected to a number of

instrument components via non-SpaceWire links. The attitude and orbit control processor (AOCP) is the barrier between the data-handling and control subnets and it is the initiator of all traffic to the control sensors and actuators.

ASTRO-H uses a deterministic network protocol based on an early version of SpaceWire-D to schedule its traffic. All communication on the network is done with RMAP transactions initiated by the SMU on the data-handling subnet and the AOCP on the control subnet. The SMU acts as the time-code master and distributes time-codes at a rate of 64 Hz, giving one schedule epoch per second. The schedule for the SMU consists of slots dedicated to telecommand distribution, housekeeping telemetry gathering, time data distribution, auxiliary data distribution and payload data gathering. Every fourth slot is dedicated to command distribution and housekeeping telemetry gathering. The remaining groups of three slots are dedicated to payload data gathering except for the first and last group. The former is used to distribute time and auxiliary data to other nodes and the latter is used to collect housekeeping telemetry from the routers (Yuasa, et al. 2011).

After ASTRO-H was launched in February 2016, an attitude adjustment was executed on 26th March 2016. The ACS then reported incorrect attitude values which caused a reaction wheel to erroneously attempt to correct the attitude. This caused ASTRO-H to start rotating and the increasing rotation resulted in the satellite separating into several pieces and ASTRO-H being lost (JAXA 2016).

2.4.3 Jupiter ICy moons Explorer

The JUICE mission is being developed by ESA under the Cosmic Vision Program. Its aim is to investigate Jupiter and three of its moons: Ganymede, Europa and Callisto, to determine if they have potential to support life (Grasset, et al. 2013).

JUICE is a single spacecraft due to be launched in 2022 on an Ariane 5 launcher in order to reach Jupiter by 2030. It will then travel to Callisto, execute multiple flybys around Jupiter and Europa before eventually transferring into orbit around Ganymede in 2032 (European Space Agency 2011). The instrument suite contains 10 different instruments: a camera, two spectrometers, a wave instrument, a laser altimeter, an ice penetrating radar, a magnetometer, a particle environment sensor package, a radio plasma wave instrument and a radio science package. One additional experiment makes use of the spacecraft's telecommunication device to pinpoint the spacecraft's position and velocity.

SpaceWire is used on-board JUICE to handle telemetry, instrument telecommands and payload data. The network architecture is illustrated in Figure 2-15.

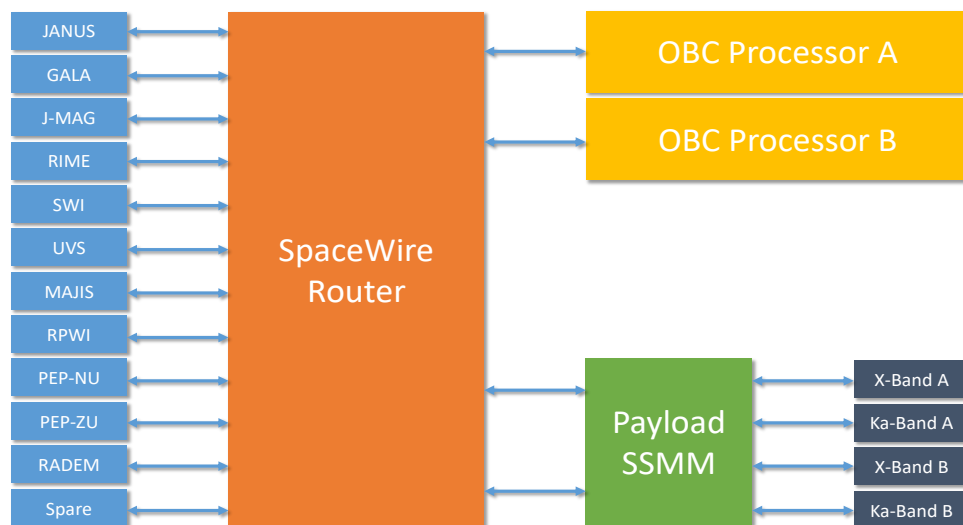


Figure 2-15: JUICE Network (Airbus Defence & Space 2015)

As shown in Figure 2-15, the network is based around a central router. The router is connected, via redundant SpaceWire links, to each of the instruments, the redundant OBCs and the mass-memory device. For simplicity, the redundant SpaceWire links

are not shown in Figure 2-15. In addition, there are point-to-point redundant SpaceWire links between the SSMM and the telecommunications devices.

Each instrument generates a number of payload data packets, ranging mostly from 10.6 to 310 per second with one more demanding instrument, Moons and Jupiter Imaging Spectrometer (MAJIS), generating a burst-rate of 1500 per second. The payload data packets are sent to the SSMM for storage before downlinking. The instruments also send two housekeeping telemetry packets per second to the OBC and receive up to five telecommands per second from the OBC.

2.5 Other Communication Networks

In addition to SpaceWire-D, there are a number of communication networks that have been used historically for command and control traffic in space applications. This section looks at two technologies that have been adopted for use in space by ESA and extended by ECSS standardisation, then compares them with SpaceWire-D.

2.5.1 MIL-STD-1553

The MIL-STD-1553 serial data bus is a US military standard that was first published in 1975 with a revision, referred to as MIL-STD-1553B, published in 1978 (US Department of Defense 1978). Originally designed for use in real-time systems in aircraft, it has been widely adopted for space applications and the ECSS published a standard in 2008 to extend the standard and improve compatibility between technologies utilising MIL-STD-1553B in ESA missions (ECSS 2008 B).

2.5.1.1 Node Types

A MIL-STD-1553B bus can consist of three types of nodes: a bus controller (BC) which controls the flow of data by initiating transfers or commanding other nodes to execute a transfer; up to 31 remote terminals (RT) which are the data sources and

destinations; and bus monitors (BM) which are passive and used to capture and analyse bus traffic (Bracknell 1988). Figure 2-16 illustrates an example MIL-STD-1553B bus architecture containing each type of node.

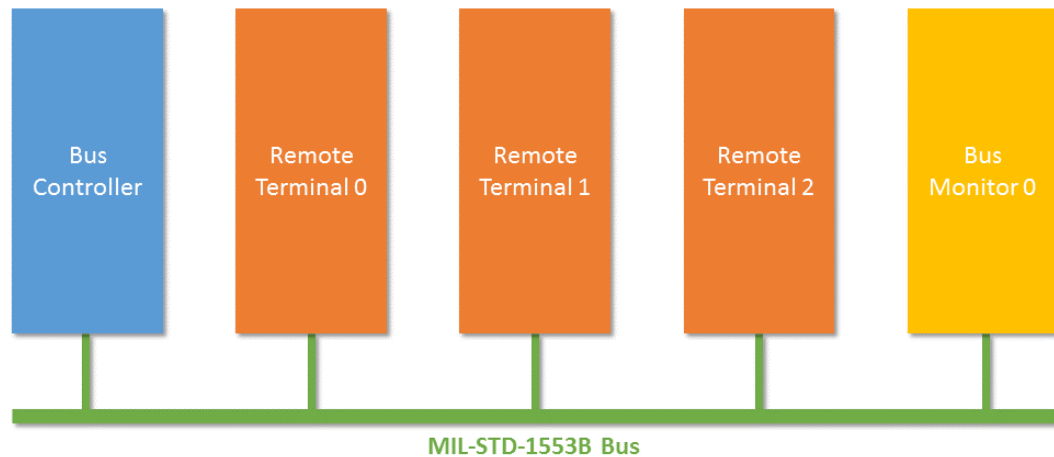


Figure 2-16: Example MIL-STD-1553B Bus Architecture

As shown in Figure 2-16, the example bus contains a single BC, three RT's indexed from 0 to 2, and a single BM. The BC may initiate transfers to any of the RT's or command any of them to execute transfers with another. The BM has a passive role and can capture any of the transfers being executed between the BC and the three RT's for analysis.

2.5.1.2 Transfer Types

The operation of a MIL-STD-1553B bus is controlled by the BC. The BC is responsible for transferring data from the BC to an RT, commanding an RT to transfer data back to the BC or commanding an RT to send data to another RT. These three types of transfers are implemented using three types of words: command words which are sent only by the BC and contain an RT address and information about the required transfer; data words which can be sent by both the BC and RTs and contain the actual

data to be transferred; and status words which are sent only by RTs and contain a response to a command word.

BC to RT

The BC to RT transfer type allows the BC to send between 1 and 32 16-bit data words to an RT, as illustrated in Figure 2-17.



Figure 2-17: BC to RT Transfer (US Department of Defense 1978)

As shown in Figure 2-17, the BC first transmits a receive command word which is addressed to an RT. Following this are the data words to be transferred. The RT then verifies the command and data words and, after a response time delay, transmits a status word to indicate if the transfer was successful or not.

RT to BC

The RT to BC transfer type allows the BC to command an RT to send between 1 and 32 16-bit data words to the BC, as illustrated in Figure 2-18.

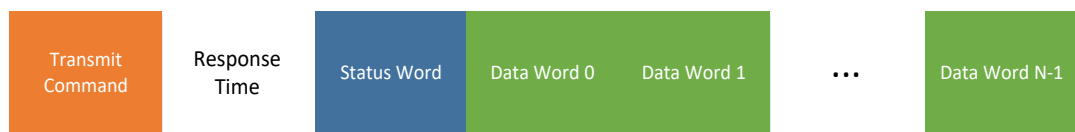


Figure 2-18: RT to BC Transfer (US Department of Defense 1978)

As shown in Figure 2-18, the BC first transmits a transmit command word which is addressed to an RT and contains information about the data to be transferred. The RT then verifies the command and, after a response time delay, transmits a status word followed by the requested data words.

RT to RT

The RT to RT transfer type allows the BC to command an RT to send between 1 and 32 16-bit data words to another RT, as illustrated in Figure 2-19.

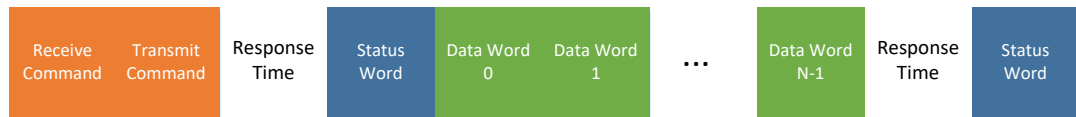


Figure 2-19: RT to RT Transfer (US Department of Defense 1978)

As shown in Figure 2-19, the BC first transmits a receive command word addressed to the receiving RT followed by a transmit command word addressed to the transmitting RT. The transmitting RT verifies the command then, after a response time delay, transmits a status word followed by the requested data words. After the data transfer is complete, the receiving RT verifies the data words and, after a response time delay, transmits a status word to indicate if the transfer was successful or not.

2.5.1.3 ECSS Standardisation

The ECSS MIL-STD-1553B standard defines a set of services to provide time distribution, time-division multiplexing, data transfers and node management for devices implementing MIL-STD-1553B. The standard's aim is to improve reusability across missions and provide guidance to engineers (ECSS 2008 B).

Time Distribution

Time distribution is provided by the BC acting as a time-master which synchronises time in a two-step process. The first step is to transmit time data to the RTs and the second step is to transmit a time synchronisation command to tell the RTs to update their on-board time based on the data from the first step.

Time-Division Multiplexing

Time-division multiplexing is implemented by dividing network time into communication frames which are controlled by the time synchronisation commands transmitted by the BC. These communication frames can be further divided into minor frames. A schedule can then be designed to transmit messages in specific minor frames in order to meet the bandwidth requirements of a mission. The ECSS standard describes two types of scheduled data transfers. The first is pre-allocated bandwidth with populated content, which could be periodic traffic or aperiodic traffic with deadlines. This type of data transfer is preconfigured in the schedule to ensure that periodic traffic can meet its rate requirements and that aperiodic traffic can satisfy its latency requirements. The second type of data transfer is pre-allocated bandwidth with unpopulated content, which could be used for rarely seen traffic or payload data. This type of data transfer has bandwidth reserved in the schedule but no data transfers are preconfigured. When a data transfer is required, it is inserted into the next free unpopulated content frame.

Data Transfers

The ECSS standard defines Set Data and Get Data services which implement BC to RT and RT to BC transfers, respectively. In addition, a Data Block Transfer service is used to transfer blocks of data that exceed the basic length limitation of a MIL-STD-1553B data transfer.

To transmit data from a BC to an RT using the Set Data service, the user application calls the SendData.Submit primitive and passes it the data to transfer. If the requested transfer is intended for a populated minor frame, it is executed within the relevant populated minor frame in the next communication frame. Otherwise, it is executed in

the next unpopulated minor frame, in the next communication frame, as long as there are no higher priority transfers waiting to be executed. The Get Data service operates in a similar manner with the user application using the ReceiveData.Submit primitive to set up a receive data operation and the ReadData.Submit primitive to read the received data.

The Data Block Transfer service allows for data transfers with a maximum data length of 4096 bytes, compared to the maximum of 64 bytes for basic data transfers. Data blocks are segmented into a number of individual transfers and transferred from the BC to an RT using Data Distribution transfers and from an RT to the BC using Data Acquisition transfers.

When the user application wants to send a block of data to an RT, it calls the SendData.Submit primitive and passes it information about the required data block and its destination in the RT. The BC then sends the data block and also sends a descriptor to a specific location in the RT. Meanwhile, the RT polls the descriptor location to see if any new data has been written. If so, it writes a confirmation to a specific location which is then read by the BC to check the status of the operation.

Receiving a block of data works in reverse. First the RT calls the SendData.Submit primitive and passes it information about the required data block which is stored in a specific location in the RT. The BC polls all RTs implementing the Block Transfer service to determine if there is a data block to transfer. If so, the BC reads the data block and sends a confirmation to a specific location in the RT to indicate the status of the operation.

Management

The management of remote terminals is mission-specific and can be done by reading status information via the standard Get Data and Set Data services. The basic MIL-STD-1553B standard also defines some flags within the status words transmitted by remote terminals after receiving a command or after being probed directly by the BC, which can be used to signal problems to the management software.

2.5.2 Controller Area Network

The Controller Area Network (CAN) bus standard was originally designed for use in the automotive industry. The first version was developed by Robert Bosch GmbH and published in 1986, with a second and the latest version released in 1991 (Robert Bosch GmbH 1991). In addition to the automotive industry, CAN bus has been adopted for use in embedded systems in other industries, including space, and the ECSS is in the process of publishing an extension standard (ECSS 2013).

2.5.2.1 Message Identifiers

Nodes are not assigned individual addresses in a CAN bus. Instead, nodes receive messages they are interested in by filtering them based on the unique message identifiers. If a node is interested in a message based on its identifier, it can receive it and notify the user application, otherwise, the message can be ignored.

2.5.2.2 Arbitration

In a CAN bus, each node can initiate data transfers with any other node. Access to the bus is controlled by a priority-based arbitration mechanism to ensure that two nodes can't attempt to send different messages at the same time.

An implementation of a CAN bus defines one binary value as the dominant value and the other as the recessive value. For example, in an implementation with a dominant

zero-bit, the bus acts as an AND gate where the value of the bus is zero if one or more nodes transmit a value of zero and one if all nodes transmit a value of one. If two or more nodes wish to transmit at the same time, they can each transmit the first bit of their message identifiers then immediately check the value of the bus. If the value does not match the one transmitted, the node has been dominated by another and it must wait until the next time the bus is free to try again. If the value matches the one transmitted, the node can continue with the next value of the identifier, and so on until only one node remains. The final node wins access to the bus after its entire message identifier has been transmitted without being dominated. Table 2-4 shows an example of arbitration between four 11-bit message identifiers on an AND gate CAN bus implementation where identifiers with lower values have higher priority.

Table 2-4: CAN Bus Identifier Arbitration

Message Identifier Value										
0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	1	1	1
0	1	1	0	0	1	1	1	0	0	0
1	0	1	1	1	1	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1	1
Bus Value										
0	0	0	0	0	0	0	0	0	1	1

In Table 2-4, there are four message identifiers being transmitted at the same time by four different nodes. The bottom row shows the value output on the bus after each value is transmitted. After the first value is transmitted, the third message identifier is dominated by the others so the message transmission is cancelled by its node. The second message is cancelled after the second value is transmitted and the first message is cancelled after the ninth value is transmitted. Therefore, the fourth message successfully gains access to the bus as its identifier was fully transmitted.

2.5.2.3 *Transfer Types*

CAN bus messages are transferred using different types of frames, each with either an 11-bit standard identifier or 29-bit extended identifier. Each message is separated by an inter-frame space which allows the receiving nodes to deal with a message before receiving the next.

Data Frame

Data frames are used to transmit up to eight bytes of data from a source node to one or more destination nodes. Each data frame has a 16-bit cyclic redundancy check (CRC) to allow the receiving nodes to perform error checking. In addition, each data frame has an acknowledgement slot to allow receiving nodes to signal to the transmitter that they have correctly received the message.

Remote Frame

Remote frames are transmitted by a node to request the transmission of a data frame, with the same message identifier as the remote frame, from another node. If a remote frame and data frame with the same message identifier are sent at the same time, the data frame wins the arbitration. This is because it has a dominant flag after the message identifier to indicate that it is a data frame and not a remote frame.

Error Frame

If a receiving node detects an error in a message, it can transmit an error frame during the inter-frame space after the erroneous message. The error frame is detected by the other nodes and causes them to transmit their own error frames. After all nodes have transmitted their error frames, the node that transmitted the original erroneous frame can attempt to retransmit the original frame.

Overload Frame

Overload frames are transmitted by a node during the inter-frame space to indicate that the node requires an additional delay before the next frame is transmitted.

ECSS Standardisation

The ECSS CAN bus standard (ECSS 2013) extends the original standard to describe the use of the CANopen protocol (CAN in Automation 2011). CANopen provides an object abstraction on top of CAN messages and defines methods for time distribution and redundancy management.

CANopen

The CANopen protocol is a higher layer protocol used on top of a standard CAN bus which defines how the message identifiers and frames are used. The version of CANopen recommended for use in the ECSS standard is a minimal implementation in order to reduce the hardware and software resources required.

CANopen functions are implemented as a number of different objects. Each device has an Object Dictionary (OD) which has a number of entries containing configuration, status or other data. Each OD entry can be accessed using Service Data Objects (SDO) which request to read or write to an entry in the OD by embedding an entry identifier in the data section of the frame. Process Data Objects (PDO) allow a node to query another node's OD entry or transmit an entry from its own OD without being prompted. PDOs remove the protocol overhead of SDOs by using a direct mapping between a message identifier and an OD entry. Synchronisation Objects are used to signal a time-dependent event to other nodes. Emergency Objects are used to report any detected errors across the bus. Finally, Network Management Objects allow a management node to control and monitor the other nodes.

Time Distribution

Time is distributed by a single time-master node that transmits a PDO containing the system time information which is received by any other interested nodes. In addition, if a node also has a local time it can be read by using another PDO. Finally, Synchronisation Objects can be used to inform other nodes that a time-dependent event has occurred or that an action should be taken.

Redundancy and Network Management

A single node can be assigned as the Redundancy and Network Management master, which is responsible for monitoring and controlling the other nodes. Monitoring the status of the nodes is done by each node transmitting a heartbeat at regular intervals. If a node misses a heartbeat, the Redundancy and Network Management master can perform a mission-specific redundancy management function to recover. In addition to redundancy management, the master can use Network Management objects to start, stop and reset the other nodes on the network.

2.5.3 Other Networks

In addition to the communication networks being standardised by ECSS, there are other networks that have already been used in space or are currently under investigation for use in space.

SpaceFibre is the successor to SpaceWire and provides very high data-rate links as well as built-in quality of service mechanisms (Parkes, McClements and McLaren, et al. 2015) and runs over electrical or fibre cables. There are three levels of quality of service included in SpaceFibre: prioritised traffic, bandwidth reservation and scheduled traffic. Time-Triggered Ethernet is an extension to Ethernet that provides deterministic mechanisms for real-time systems (Kopetz, Ademaj, et al. 2005).

Wireless networking technologies have been investigated for use in space (Vladimirova, et al. 2007) but have not yet been widely adapted for on-board communications.

2.5.4 Comparison

In this section, a comparison between MIL-STD-1553B, CAN and SpaceWire-D is given in relation to various features that are important for a deterministic network. These two communications networks were chosen for this comparison because they, in addition to SpaceWire-D, are in the process of standardisation for use in space by the ECSS. The versions of MIL-STD-1553B and CAN used for this comparison are described in (ECSS 2008 B) and (ECSS 2013), respectively.

2.5.4.1 Time-Division Multiplexing

MIL-STD-1553B provides TDM by dividing system time into major and minor communication frames. The bus controller sends time information to each node on the bus then distributes a synchronise command that tells each node to update their time. Repeating, periodic traffic or aperiodic traffic with deadlines can be scheduled within populated pre-allocated bandwidth. Sporadic or payload traffic can be prioritised and scheduled within unpopulated pre-allocated bandwidth.

CAN provides TDM by having a SYNC producer which transmits Synchronise Objects. This indicates to the SYNC consumers, i.e. the other nodes, that a time-dependent event like a time-slot should begin. Within the time-slots, traffic can be scheduled using the message identifier arbitration system to ensure that traffic is prioritised and deterministic.

SpaceWire-D provides TDM using time-slots controlled by the distribution of consecutive time-codes by a single time-code master. When a new time-slot begins,

the initiators execute any transaction groups for the virtual buses, if any, allocated to the time-slot.

2.5.4.2 Exclusive or Non-Conflicting Access to Network

MIL-STD-1553B provides exclusive and non-conflicting access to the network by ensuring that there is a single bus controller in operation at any one time. The bus controller controls all traffic on the bus so there are no conflicts with other nodes attempting to transmit at the same time.

CAN provides exclusive and non-conflicting access to the network through its message identifier arbitration system. The system must adhere to the rule that each message is assigned a unique identifier. Then, if two or more nodes attempt to transmit a message, the node with the highest priority message will win access to the bus. The other nodes will wait until the bus is free again to re-attempt to transmit.

SpaceWire-D provides exclusive and non-conflicting access to the network through its virtual bus abstraction and scheduling. The system must adhere to the rule that no two virtual buses are allocated the same time-slot if they share one or more SpaceWire links. Then, in each time-slot, the initiators have exclusive access to their segment of the network. Therefore, packets on one virtual bus cannot interfere with packets on any other.

Each of the networks allow for exclusive or non-conflicting access to the network, with MIL-STD-1553B providing the most robust option because it is reliant solely on the bus controller. CAN and SpaceWire-D also rely on the other nodes adhering to the unique message identifier rule or non-conflicting virtual bus scheduling rule, respectively.

2.5.4.3 *Data Rates and Protocol Overhead*

MIL-STD-1553B operates at 1 Mbit/s and has a minimum transmit and receive data size of 2 Bytes and a maximum of 64 Bytes per transfer. Transmit operations are done through BC-RT transfers and consist of a receive command, one or more data words each containing 2 bytes of data and one status word. Each word is 20 bits, giving a protocol overhead of 73.33% for a 2 Byte BC-RT transfer and 24.71% for a 64 Byte BC-RT transfer. Receive operations are done through RT-BC transfers and have the same protocol overhead as BC-RT transfers. The final transfer type, RT-RT, allows a BC to request one RT to transmit data to another and consists of a receive command sent to one RT and a transmit command to the other, a status word from the transmitting RT, one or more data words and a status word from the receiving RT. This gives a protocol overhead of 84% for a 2 byte RT-RT transfer and 28.89% for a 64 Byte RT-RT transfer.

CAN operates at 1 Mbit/s and has a minimum transmit and receive data size of 1 Byte and a maximum of 8 Bytes per message. Each CAN frame consists of a start-of-frame bit, an arbitration field, a control field, a data field, a CRC field, an acknowledgement field and an end-of-frame field, giving a total of 44 bits of protocol overhead for standard frames and 64 bits for extended frames. Transmit operations, for PDOs, are done through data frames, giving a protocol overhead of (84.62%, 88.89%), where the two parenthesised values are for normal and extended frames, for a 1 Byte message and (40.74%, 50%) for an 8 Byte message. Receive operations, for PDOs, are done through a combination of a remote frame sent by the destination node and a data frame sent by the source node, giving a protocol overhead of (91.67%, 94.12%) for a 1 Byte message and (57.89%, 66.67%) for an 8 Byte message. For SDOs, 4 bytes of the data field are taken up by the OD addressing information, therefore the protocol overhead

is (90.48%, 92.31%) for a 1 Byte transmit message, (70.37%, 75%) for a 4 Byte transmit message, (95%, 96%) for a 1 Byte receive message and (82.61%, 87.5%) for a 4 Byte receive message.

SpaceWire-D operates at variable transmission rates of between 2 and 200 Mbit/s per link. SpaceWire-D uses RMAP transactions which have a minimum transmit and receive data size of 1 Byte and a maximum of 16 Mbytes per transaction. Transmit operations are done through acknowledged RMAP write transactions which consist of a write command header, a write data section, a command EOP, a reply header and a reply EOP, giving a total of 258 bits of protocol overhead and an additional 2 bits for every byte in the data section. For an acknowledged RMAP write transaction using a logical address, this gives a protocol overhead of 97.07% for a 1 Byte write and 20% for a 16 Mbyte write. Receive operations are done through RMAP read transactions which consist of a read command header, a command EOP, a reply header, a read data section and a reply EOP, giving a total of 298 bits of protocol overhead and an additional 2 bits for every byte in the data section. For an RMAP read transaction using a logical address, this gives a protocol overhead of 97.4% for a 1 Byte read and 20% for a 16 Mbyte read. Although, technically, RMAP transactions can contain up to 16 Mbytes of data, this is implicitly limited by the length of the slots used by the virtual bus executing the transaction. In practice, it is likely that the maximum transaction size would be smaller. For a payload data transfer containing 4 Kbytes of data, the protocol overhead would be 20.51% for a transmit operation and 20.58% for a receive operation. To compare the protocol overhead with the largest MIL-STD-1553 transfer, the protocol overhead is 43.3% for a 64 Byte write transaction and 45.42% for a 64 Byte read transaction. In addition, if the RMAP transactions are using path addressing, this will result in higher protocol overhead as a list of path address bytes will be

required at the start of each packet and in the return path section of the command header.

2.5.4.4 Acknowledgements

MIL-STD-1553B provides acknowledgements through remote terminals transmitting status words after receiving commands. A status word contains the 5-bit address of the remote terminal and 11 single bit flags. The flags are used to report a successful command, an error in the received message or inform the bus controller of errors in the remote terminal itself.

CAN has an acknowledgement slot in each of its messages during which any receiving nodes may transmit a dominant bit if it detects an error in the message. The transmitting node can read the value on the bus after the acknowledgement slot to determine if the message was successfully received by all interested nodes or if any receiving nodes have reported an error.

SpaceWire-D provides acknowledgements through RMAP replies sent by a target that has received a command. Each RMAP reply contains an 8-bit status field that describes if the command was received and authorised successfully or if there was an error to be reported to the initiator.

2.5.4.5 Multiple Initiators

MIL-STD-1553B has a single bus controller responsible for controlling all traffic on the bus. However, the role of the bus controller can be switched between nodes using dynamic bus control. If the bus controller wants to give control of the bus to another node it issues a dynamic bus control request command to the node. If the node is able to accept the role of bus controller it issues a status word with the dynamic bus control

acceptance bit set. If the bit is set when the initial bus controller receives the status word, it ceases to act as the bus controller and the new node takes over the role.

Any CAN node can act as an initiator, assuming it wins the message identifier arbitration at the start of transmission and gains access to the bus. However, as with MIL-STD-1553B, only one node can act as the initiator at any one time due to CAN being a bus and not a point-to-point network.

SpaceWire-D networks can have multiple concurrent initiators but this is dependent on the network architecture and the schedule. The number of possible concurrent initiators in each time-slot is the number of independent virtual buses allocated to that time-slot.

2.5.4.6 Fault Detection, Isolation and Recovery

MIL-STD-1553B provides status words for each command, as described in Section 2.5.4.4, which are used to report errors to the bus controller. The handling of the errors is mission specific and left to the system designer. However, MIL-STD-1553B does provide various mode commands to aid in the handling of errors and recovery. These mode commands provide features such as dynamic bus control, as explained in Section 2.5.4.5; reading the last status word transmitted, or the last command received from a remote terminal; self-test initiation, to instruct a remote terminal to perform diagnostic tests; remote terminal transmission enable or disable; and remote terminal reset. In addition, there can be a redundant bus in case the primary bus or a node fails. If a node is babbling, commands can be sent to the babbling node on the redundant bus to disable or reset it.

CAN provides an acknowledgement slot in all of its messages, as described in Section 2.5.4.4, to report that one or more receiving nodes have detected an error. In addition

to acknowledgement errors, bit errors are detected if the transmitting node senses an incorrect bit after transmission, stuff errors are detected if there are 6 consecutive bits with the same value, CRC errors are detected if the 15-bit CRC calculated by the receiver differs from that of the transmitter and form errors are detected if a fixed-value bit has an unexpected value. The ECSS CAN standard describes two methods for redundancy using either a selective bus architecture where only one bus is active at a time or a parallel bus architecture where nodes can transmit on both buses simultaneously. A single node acts as the redundancy master and is responsible for transmitting master heartbeat messages on the active bus so that the other nodes know which bus is regarded as the primary. The redundancy master is also responsible for receiving heartbeat messages from the other nodes so that it knows if a node has failed.

SpaceWire-D targets provide a reply for each RMAP command, as described in Section 2.5.4.4, to report if any errors were detected in the command packet or in the authorisation or execution of the command. RMAP packets also contain 8-bit CRCs which cover the header values of all packets and data sections of write command packets and read reply packets. RMAP reply errors, as well as any others detected by the initiators such as incomplete transaction errors, are compiled into a list and reported to the network manager at the end of each schedule epoch. The detection and reporting of errors is defined in the SpaceWire-D standard but the handling of the errors is mission specific and left to the system designer. SpaceWire networks provide flexible redundancy capabilities through redundant links, nodes or routers, depending on the requirements of the mission.

2.5.4.7 Comparison Summary

The comparison between MIL-STD-1553B, CAN bus and SpaceWire-D is summarised in Table 2-5.

Table 2-5: Communication Network Comparison

Feature	MIL-STD-1553B	CAN	SpaceWire-D
Time-Division Multiplexing	+	+	+
Exclusive Network Access	++	+	+
Data Rates and Overhead	-	-	++
Acknowledgements	+	+	+
Multiple Initiators	+	+	++
FDIR	+	+	+

In Table 2-5, each row lists a feature and gives a corresponding score for MIL-STD-1553B, CAN bus and SpaceWire-D.

The networks score equally with regards to TDM because they each provide the capability to divide time into time-slots in which scheduled and prioritised traffic can be executed.

MIL-STD-1553B has an advantage when it comes to exclusive or non-conflicting access to the network because it depends only on the bus controller. CAN and SpaceWire-D rely on all of the nodes on the network adhering to rules that ensure that traffic doesn't conflict.

SpaceWire-D has an advantage in relation to the data rates and protocol overhead as it can operate at up to 200 Mbit/s compared to 1 Mbit/s for MIL-STD-1553B and CAN bus. SpaceWire-D also has a much larger maximum single transfer size at 16 Mbytes compared to 64 Bytes for MIL-STD-1553B and 8 Bytes for CAN bus, allowing for reduced protocol overhead.

The networks score equally with regards to acknowledgements because MIL-STD-1553B and SpaceWire-D provide acknowledgements with status and error information for every transfer. CAN bus provides an acknowledgement slot and error frames containing the error information if any remote terminal detects an error.

Although MIL-STD-1553B and CAN bus can have multiple initiators, only one can operate at any time due to the bus architecture of these networks. SpaceWire-D is at an advantage because it can have multiple concurrent initiators as long as no two initiators use a common SpaceWire link during the same time-slot.

MIL-STD-1553B provides FDIR through its status words and various mode commands to aid in the handling of errors and recovery. The manner in which these are used are mission specific and left to the system designer. Each MIL-STD-1553B data word has a parity bit but the commands, data words and status words are not protected by a CRC. CAN bus provides FDIR through its acknowledgement slots and error frames and each CAN bus message is protected by a 15-bit CRC. SpaceWire-D provides FDIR through RMAP replies and checks at various points during the execution of the schedule. RMAP commands are checked for correct parameter values before transmission and RMAP transaction groups are checked to make sure they will fit within their intended time-slots. If time-codes arrive early, late or go missing an error is recorded. At the end of each schedule epoch, the list of errors is reported to the network manager. Each of the networks provides their own methods for redundancy.

2.6 On-Board Data Systems

Current on-board data systems are typically divided into two segments: the platform and the payload (Hult and Parkes 2014). The platform is responsible for critical tasks

such as receiving and handling telecommands, generating telemetry frames, performing housekeeping and performing control applications. The payload is responsible for moving data between the spacecraft's instruments and its data storage, and possibly processing the data and generating the payload telemetry frames. The architecture of a typical on-board data system is illustrated in Figure 2-20.

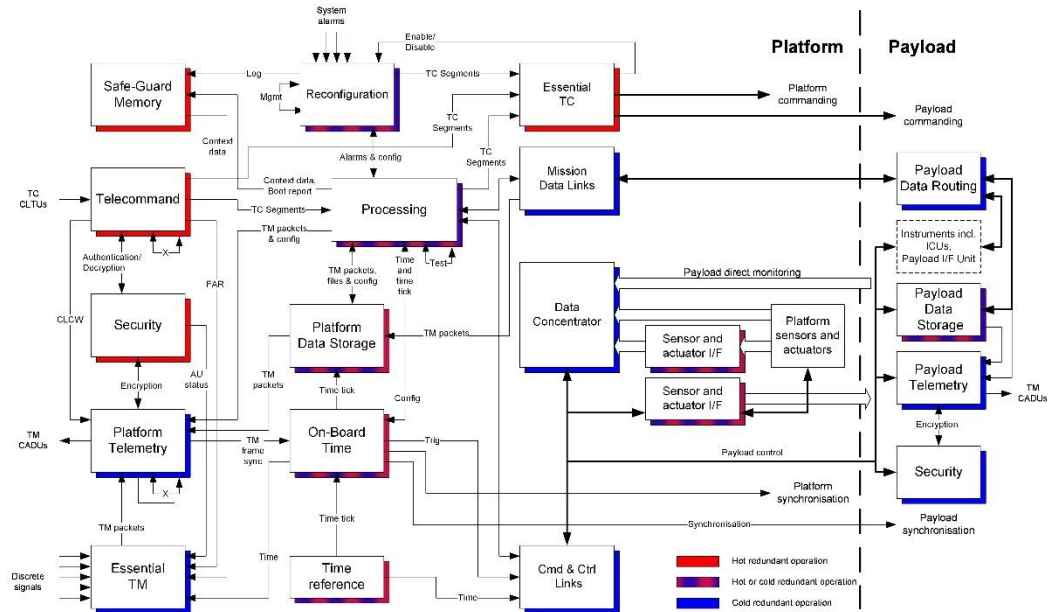


Figure 2-20: On-Board Data Systems (European Space Agency 2014 A)

As shown in Figure 2-20, the on-board data system is divided into the platform segment on the left and the payload on the right. The communication network used in the platform may require determinism, FDIR and QoS mechanisms, but relatively low data-rates. In contrast, the communication network used in the payload may require much higher data-rates but lower levels of FDIR and QoS.

Currently, MIL-STD-1553B is most often used for the platform communication network because it provides determinism. However, it supports relatively low data-rates so for missions with higher data-rate requirements it is not also suitable for use

in the payload. Instead, a high data-rate communication network like SpaceWire or SpaceFibre is used.

As explained in Section 2.3.1, SpaceWire is not deterministic due to the problem of blocking and therefore, it may not be suitable for use in the platform. The aim of SpaceWire-D is to solve this problem and allow a single communication network to support determinism in the platform and high data-rates in the payload.

The Consultative Committee for Space Data Systems (CCSDS) Spacecraft On-Board Interface Services (SOIS) document (CCSDS 2013) describes a set of standard on-board data services which are abstracted in layers above the specific communication networks that are used to implement them e.g. MIL-STD-1553B or SpaceWire. Some of these services, such as the Command and Data Acquisition Service, require deterministic and prioritised access to the underlying network which is not provided by a regular SpaceWire network but is provided by the virtual bus system in a SpaceWire-D network.

This thesis demonstrates that SpaceWire-D can be used to support deterministic platform traffic and high data-rate payload traffic on the same communication network. The benefits of achieving this include reducing complexity by removing the need for multiple communication networks and possibly reducing mass by reducing the number of cables and connectors required.

2.7 Summary

This chapter started with a description of the subsystems found on-board spacecraft. Attitude and orbit control subsystems measure and adjust the orientation and velocity of the spacecraft through the use of various sensors and actuators. Communications subsystems provide a link between the spacecraft and Earth in order to send data and

telemetry to its operators and receive telecommands. On-board computers manage and control the other subsystems on the spacecraft. Finally, scientific instruments, which are the payload of a spacecraft, gather data or provide a service in space.

Next, the SpaceWire and SpaceWire-D protocols were described. SpaceWire allows for communication between the different subsystems on-board a spacecraft and is primarily used to handle payload data. SpaceWire-D was designed to provide deterministic capabilities on top of existing SpaceWire equipment. It uses time-slots, a virtual bus abstraction and scheduling to regulate traffic so that there are no conflicts on the network which may cause non-deterministic delays.

Following the description of SpaceWire and SpaceWire-D, three missions that utilise SpaceWire were described. These three missions use SpaceWire for handling payload data as well as more critical traffic such as telecommands or telemetry. MMS is a NASA mission consisting of four satellites used to research the transfer of energy between the Sun and Earth's magnetic fields. ASTRO-H was a JAXA mission which had the intention of investigating large astronomical structures, to research extreme space environments, to measure the effect of astronomical events on the energising of cosmic rays and to study dark matter and energy's effect on the evolution of galaxy clusters. Finally, JUICE is an ESA mission that aims to investigate Jupiter and three of its icy moons in order to determine if they have the potential to support life.

In the final sections of the chapter, two existing communication networks were described and compared to SpaceWire-D. The first, MIL-STD-1553, is a US military standard originally published in 1975 with a revision, MIL-STD-1553B, published in 1978. It is a bus architecture that uses a single bus controller to send messages to remote terminals at a data-rate of up to 1 Mbit/s. The second, CAN, is a standard

originally developed for the automotive industry. Again, it is a bus architecture and uses a message identifier arbitration system to allow all nodes to act as transmitters and compete for prioritised access to the network. Both of these networks have been extended for use in space by ECSS standardisation.

Chapter 3

Research Questions

SpaceWire-D is a deterministic extension to the SpaceWire protocol that uses time-division multiplexing, a virtual bus abstraction and scheduling to allow payload and real-time traffic to share a network. The main research aim is to answer the following question:

How can the SpaceWire-D protocol be used to fulfil the bandwidth requirements of a space mission?

As SpaceWire-D is a new technology, it is necessary to explore how to design an efficient SpaceWire-D software layer and how to prototype a system that utilises the protocol. In addition, once a prototype system has been designed, an investigation in to how the protocol can be used to meet the demands of a space mission is required. Therefore, the main research question above was further divided into three sub-questions which are listed in the following sections along with a description of how they will be answered.

3.1 Research Questions

The research questions are as follows:

3.1.1 Designing a SpaceWire-D Software Layer

Question 1: How can an efficient SpaceWire-D software layer be designed on top of existing SpaceWire devices?

An initiator that is using the SpaceWire-D protocol must minimise the section of each time-slot consumed by the software overhead of the protocol in order to maximise the available time spent executing RMAP transactions. In addition, the design of the SpaceWire-D layer must take into account the deterministic requirements of a system utilising the protocol as well as the relatively low amount of memory and CPU power available in an on-board computer.

In Chapter 4, an efficient SpaceWire-D software layer is described, built on-top of the RTEMS real-time operating system and running on a LEON2-FT based processor board. RTEMS support for the processor board was required so the chapter begins by explaining how the operating system was ported to the board based on existing LEON support in the RTEMS source tree.

After RTEMS was successfully running on the processor board, a SpaceWire-D software layer was designed to allow on-board software to open, load and close virtual buses which are executed during each time-slot. The time-slot processing in the SpaceWire-D layer went through several designs until the initiator processing time was reduced to a suitable level.

3.1.2 Designing a SpaceWire-D Demonstrator

Question 2: How can a system using the SpaceWire-D protocol be prototyped in order to demonstrate the standard?

After the SpaceWire-D layer was designed for a single initiator board, a prototype SpaceWire-D system was designed in order to test the protocol in a representative spacecraft on-board data-handling network.

In Chapter 5, the design of a SpaceWire-D Demonstrator is described. The system consists of two LEON2-FT based processor boards acting as the initiators, three RMAP interface boards each containing four RMAP targets, a network manager board, two router boards and a host PC processor board. The host PC runs a suite of software used to configure, control and monitor the other devices on the network.

During this activity, a time-slot triggered scripting system was designed in order to automate tests using the SpaceWire-D Demonstrator. A description of three example scripts is given and the results from running the automated tests are presented and described in Chapter 6. This system and the Demonstrator were used to create and run all of the tests during the verification activity of the ESA SpaceWire-D project and the Demonstrator was delivered to ESA and installed at ESTEC.

3.1.3 Scheduling SpaceWire-D Networks

Question 3: How can a SpaceWire-D mission's bandwidth requirements be represented and satisfied computationally?

After SpaceWire-D was prototyped in a full system, scheduling mechanisms for space missions were explored. The scheduler takes a network specification consisting of a list of bandwidth requirements, a network topology and network parameters. The scheduler then transforms the specification into a selection of paths between initiators and targets and an allocation of transactions into time-slots to form the required virtual buses and network schedule.

Firstly, SpaceWire-D scheduling was formalised by describing the different types of bandwidth requirements, the network topology format, the network parameters used to calculate RMAP execution times and the format of a solution. Secondly, methods for selecting the paths between the initiators and targets were explored in order to attempt to reduce the number of possible collisions between initiator/target pairs. Thirdly, methods for allocating periodic and aperiodic bandwidth requirements were designed and the problem of allocating payload data bandwidth requirements was transformed into a variation of the classic bin-packing problem. The scheduling algorithm was then evaluated on a suite of test cases as well as a case study of the JUICE mission.

3.1.4 Summary

In this chapter, the research questions were listed and a description of how they were answered was given. The research started by looking at how a SpaceWire-D software layer could be designed using a LEON2-FT based processor board on top of the RTEMS real-time operating system. Next, two initiators using the SpaceWire-D software layer were combined with several target boards, a network manager, routers and a host PC running a suite of programs to create a prototype SpaceWire-D system that was used to demonstrate and verify the SpaceWire-D standard. Lastly, scheduling mechanisms were explored to allow SpaceWire-D to satisfy the bandwidth requirements of space missions. Missions are specified as a list of bandwidth requirements, a network topology and a list of network parameters. The scheduler then transforms the specification into a set of paths between initiators and targets and a SpaceWire-D network schedule.

Chapter 4

SpaceWire-D Software Layer

In order to evaluate, verify and demonstrate the SpaceWire-D protocol, a prototype system needed to be designed and created to integrate initiators, targets, routers and analysis devices into a SpaceWire network using real, standard-conforming equipment. The system consists of multiple devices taking on the various roles within a SpaceWire-D network, one such role being that of the initiator. Initiators are responsible for controlling all traffic within the SpaceWire-D network. They transmit RMAP commands based on a schedule, and they process RMAP replies returned from target devices.

If a SpaceWire-D network is to operate efficiently and responsively, the SpaceWire-D layer within each initiator must be designed to reduce software overhead. This means minimising the processing time required for an initiator to begin executing transactions when a time-code is received. This reduces the delay at the start of each time-slot and increases the percentage of each time-slot available to the initiators to execute transactions.

This chapter describes the design of a SpaceWire-D layer running on top of the Real-Time Executive for Multiprocessor Systems (RTEMS) (The RTEMS Project 2017) real-time operating system (RTOS). The SpaceWire-D layer executes within each of

the LEON2-FT processor boards, provided by STAR-Dundee, acting as initiators in the SpaceWire-D prototype system.

4.1 Overview

The initiator is a multi-layered system comprising of the LEON2-FT processor board hardware at the lowest layer, including the embedded peripherals that allow the initiator to communicate with other devices. Above that is the RTEMS and driver layer, connecting the hardware to the software through registers and interrupts. Next is the SpaceWire-D layer which implements the protocol and provides an application programming interface (API) to the final layer, the application. The multi-layered system is illustrated in Figure 4-1.

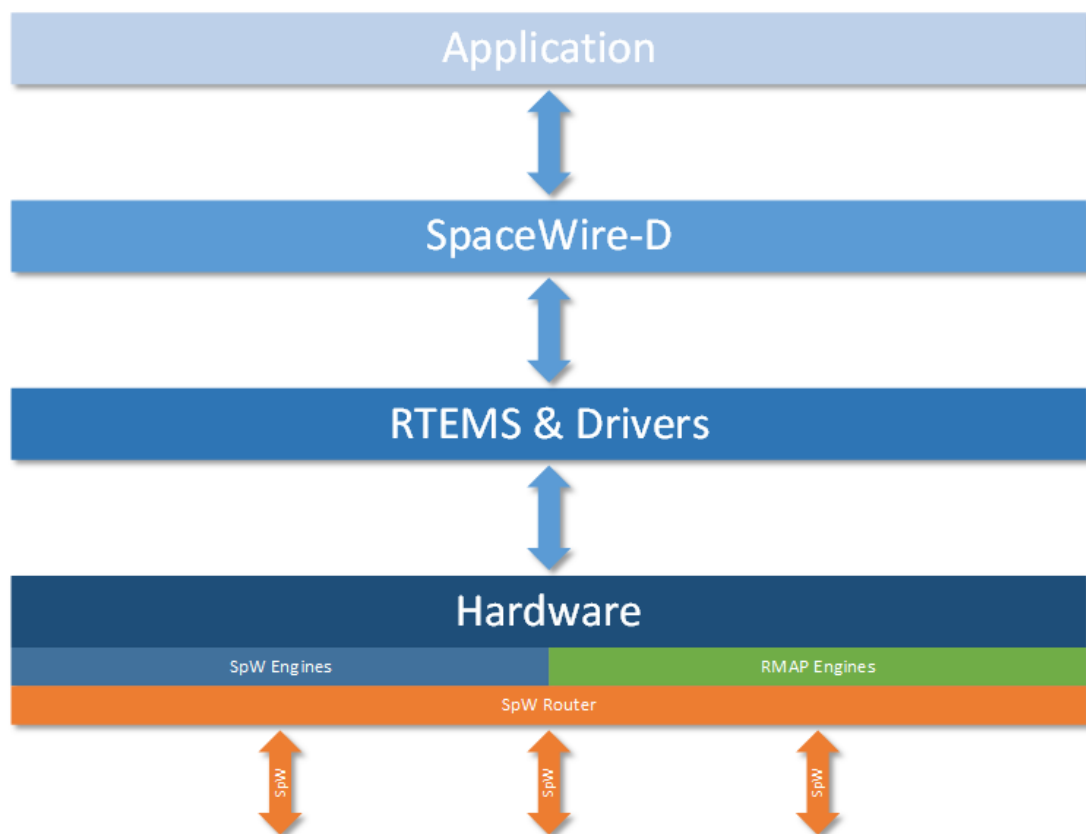


Figure 4-1: Initiator Layers

In Figure 4-1, each layer talks to the layers immediately above and below. All SpaceWire traffic travels through links connected to the SpaceWire router and one of the SpaceWire or RMAP engines. This causes a change in one or more status registers as well as an interrupt to be raised. The interrupt may or may not be signalled to the RTEMS board support package (BSP) depending on if the specific interrupt has been enabled. If an interrupt is signalled to the BSP, the operating system's interrupt handler is responsible for calling a peripheral driver's interrupt service routine (ISR) which handles it then clears the interrupt.

The SpaceWire-D layer issues system calls to the RTEMS layer to deal with operating system constructs such as semaphores and invokes driver functions to perform IO operations or configure hardware.

Between the SpaceWire-D layer and the application, an API is provided to allow the user to call the various services of the SpaceWire-D protocol such as opening, loading and closing different types of virtual buses and configuring management parameters.

A more in-depth description of the LEON2-FT processor board at the hardware layer is given in Appendix 1. The following sections provide a description of the RTEMS BSP then the SpaceWire-D layer.

4.2 RTEMS Board Support Package

RTEMS is a real-time operating system, meaning that it prioritises predictability, precise scheduling of tasks and the satisfaction of temporal constraints over raw power and generality (Stankovic and Rajkumar 2004). The design and development of RTEMS began as a United States Army Missile Command project in 1988. It has since grown into a free and open-source software project used in a variety of industries including military, space and automotive (The RTEMS Project 2015).

Compared to large, general-purpose operating systems like Windows or Linux, RTEMS has a relatively small set of features. It provides real-time task scheduling algorithms, standard data structures and a range of libraries for providing common operating system mechanisms such as synchronisation, inter-task communication and event-handling.

Real-time systems are usually built for a specific purpose and embedded in an environment to run for long periods of time. This might mean that a specialised hardware configuration was designed for the system's mission. If so, a layer of software is required to connect the different parts of the operating system to the hardware. This layer is known as a board support package, and RTEMS provides a set of guidelines to aid in the design of BSPs for different hardware configurations. An RTEMS software project is the combination of the RTEMS operating system, the BSP and the application code.

4.2.1 RTEMS in Space

NASA Goddard Space Flight Center (GSFC) developed a platform independent software framework called the Core Flight Executive to promote reusability and standardisation during the creation of flight software (McComas 2012). RTEMS is one of the three operating systems supported by the framework alongside VxWorks and Linux.

EADS Astrium developed two versions of a space-qualified version of RTEMS by performing source code verification, extensive testing and documentation before arriving at what they call RTEMS Product 1.0 and RTEMS Product 1.0.1 (Rossignol and Seronie-Vivien 2012). RTEMS Product Version 1.0 for the ERC32 platform was used in TerraSAR X, Pleiades, and SMOS Instrument. Version 1.0.1 for the ERC32

platform was used in Galileo IOV, Aeolus, Lisa PF, Bepi Colombo, Sentinel 2 and Earthcare. The payload controller on-board Alphasat and the ExoMars rover use Version 1.0.1 for the LEON2 processor. Finally, Spot 6, Spot 7, KRS, Sentinel 5P and CSO use Version 1.0.1 for the LEON3 processor.

The RTEMS Project maintains a list of space missions that use RTEMS. The list currently contains over 30 ESA and NASA missions as of October 2013 (The RTEMS Project 2013).

As described above, RTEMS or a qualified derivative has been used in a range of space missions. In addition, RTEMS has support for multiple LEON processor boards included in its source tree. This made it an ideal choice to be used as the initiator operating system in the SpaceWire-D prototype system.

4.2.2 Porting Process

To port RTEMS to a new hardware system, the intended version of the operating system should first be checked for existing support. RTEMS may already support the processor architecture, the specific system-on-chip or any embedded peripherals. If a BSP or drivers exist for the required hardware, it may be possible to reuse some or all of the software without creating it from scratch.

4.2.2.1 Existing LEON Support

The SPARC architecture and LEON processor family has extensive support within RTEMS. This is demonstrated by the use of a SPARC instruction simulator for GDB in the example tutorials and the existence of BSPs for LEON1, LEON2 and LEON3 boards.

However, the LEON2-FT processor board used in the SpaceWire-D prototype system is quite different from the board in the existing LEON2 BSP. The registers and

memory architecture are different as are most of the embedded peripherals. This meant that it was simpler to create a new BSP using the shared SPARC functionality, with the existing LEON2 BSP as a starting-point and reference.

4.2.2.2 Cross-Compiling Toolchain

A cross-compiler takes a program written in a high-level programming language like C, and translates it into an executable and linkable format (ELF) file for a specific architecture. The compiler framework used in the RTEMS project is the GNU Compiler Collection (GCC) (Stallman 2001). For the creation of the LEON2-FT processor board BSP and SpaceWire-D layer, the Windows port of GCC, MinGW (MinGW 2015), was used for cross-compilation.

4.2.2.3 Loading and Debugging

The LEON2-FT processor provides hardware debug support through its debug communications link (DCL) and debug support unit (DSU). The DCL uses the processor's dedicated debug UART and a simple protocol to allow an external device to read and write to the processor's on-chip memory or any other device connected to the Advanced High-Performance Bus (AHB). The DSU provides common debugging operations such as setting breakpoints and stepping through execution.

To utilise the debugging and loading capabilities of the LEON2-FT, the STAR-Dundee STAR-Gate (STAR-Dundee Ltd 2015) software is used. STAR-Gate is a GDB RSP program and a USB-to-UART bridge is used to connect the development and debugging PC to the LEON2-FT processor board's debugging UART.

The process to load a program from a development environment host to the target LEON2-FT processor board is illustrated in Figure 4-2.

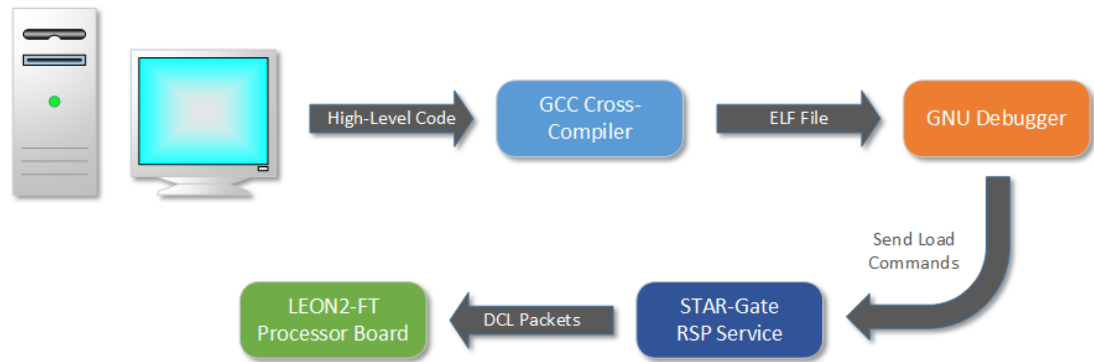


Figure 4-2: Loading an RTEMS Program to the LEON2-FT

In Figure 4-2, the loading process begins at the development environment host where a program has been written in C. Next, the high-level code is sent to the cross-compiler and linked to any required RTEMS libraries, resulting in an ELF file. The next step is to connect GDB to the STAR-Gate service and issue a GDB load command. GDB then parses the ELF file and sends a series of commands to the STAR-Gate service which are translated into RSP commands and sent to the LEON2-FT DSU via the UART connection. These commands write each section of the program to the correct location in memory. When the loading process is complete, the host is able to execute and debug the program using GDB commands, translated by STAR-Gate into further interactions with the DSU.

4.2.3 Creating the BSP

To create a BSP for a new SPARC architecture board, a directory must be created in the libbsp/sparc section of the RTEMS source tree. This new directory stores all of the configuration files, scripts and source files which make up the BSP.

4.2.3.1 Configuration Files

There are two configuration files required to support the automake and autoconf tools used in the RTEMS project to configure and build a BSP. In the LEON2-FT processor board BSP, the autoconf file is a simplified version of the file from the existing

LEON2 BSP, and the automake file is a modified version of the file from the existing LEON2 BSP. The automake file has been changed to refer to the source code files in the new BSP and add the “-mcpu=v8” compiler flag to enable the hardware multiply and division operations supported by the SPARC V8 instruction set.

4.2.3.2 Linker Command Script

A linker command script is used by the linker which, in this case, is the GNU linker (GNU Project 2015 A), to transform a number of compiled objects into an ELF file. This file describes where each section of the program should be placed in memory (GNU Project 2015 B).

The memory layout of the LEON2-FT processor board used in the SpaceWire-D Demonstrator differs from that of the existing LEON2 BSP board. In addition, the SpaceWire-D layer’s need for memory-mapped data structures required a new linker command script to be created.

The LEON2-FT processor board has two regions of memory. The first is 128 Kbytes of internal SRAM which is located at memory address 0x81000000 and the second is 256 Mbytes of external DDR3 located at memory address 0x60000000. The new linker command script instructs the linker that the executable code and initialised data should be loaded into the internal memory region. The memory-mapped data structures, uninitialized data, RTEMS workspace and the program stack should use the external memory region.

The external memory requires the LEON2-FT memory configuration registers to be set to specific values before it is enabled. Therefore, it is necessary to place the executable code and initialised data into the internal memory region because those sections require writing values to memory. The memory configuration registers have

their values set during execution of the board initialisation code. Therefore, the code is loaded into internal memory which does not require enabling via memory configuration registers.

The RTEMS operating system combined with the SpaceWire-D layer take up the majority of the 128 Kbytes of internal memory. After the external memory has been enabled during board initialisation, the remaining program sections are placed in the external memory.

The layout of an RTEMS program using this BSP is illustrated in Figure 4-3.

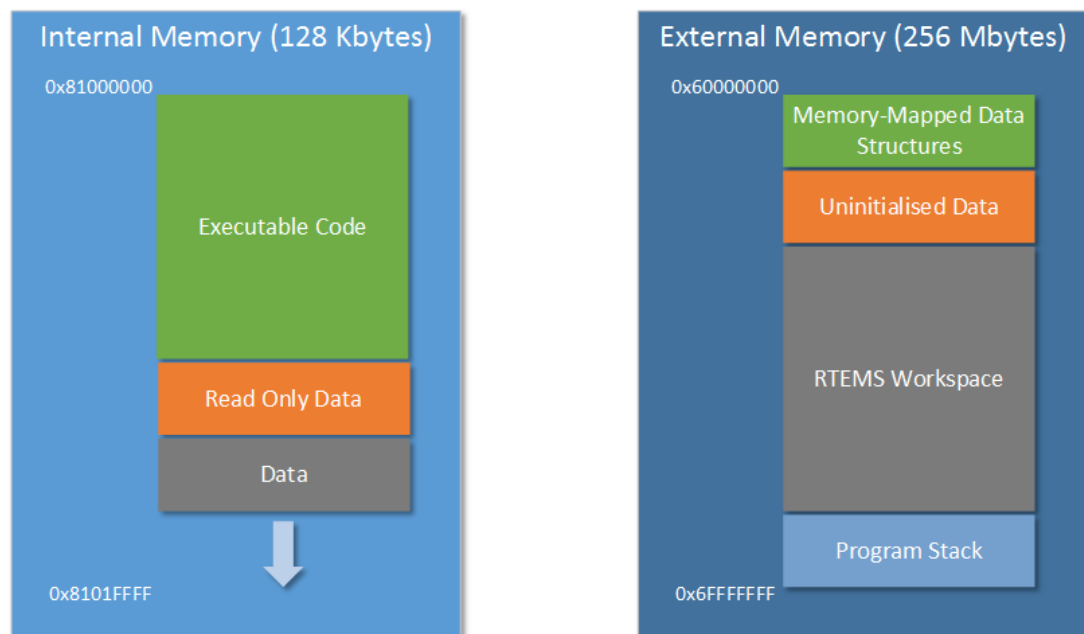


Figure 4-3: RTEMS Program Memory Layout

As shown in Figure 4-3, any program sections that require values to be written to memory i.e. the executable code and initialised data, are loaded into the 128 Kbytes of internal memory. Any unused space extends from the end of the initialised data section to the top of the internal memory region at address 0x8101FFFF. In the 256 Mbytes of external memory, the memory-mapped data structures are located at the bottom of the memory region at address 0x60000000. This is followed by any

uninitialized data, the RTEMS workspace and the program stack. The downward-growing program stack is allocated a 1 Mbyte region at the top of the external memory region and the RTEMS workspace, containing the operating system data structures and the heap, extends to fill any memory unused by the other sections.

4.2.3.3 Board Initialisation

The board initialisation process is the very first section of code that executes when a board is reset and is historically known as the `crt0` file, meaning the C runtime zero file. Within many of the BSPs in the RTEMS project it is called the `start.S` file. The `crt0` or `start.S` file is architecture dependent and it is usually written in assembly language. Its purpose is to configure hardware, initialise memory and perform any operating system initialisation that is required before the user application is called.

On the SPARC V8 architecture, the `start.S` file begins with a trap table which is a list of instructions, four for each type of trap. When an interrupt is raised or the processor detects an exceptional circumstance that must be handled by software, the program counter jumps to the trap's location in the trap table. The four instructions for each trap in the table might contain a function call to an interrupt vectoring routine if an interrupt is raised or, if the trap is an unrecoverable error, the instructions may cause the processor to halt.

The first trap is the board reset trap and tells the processor to jump to the instruction directly after the trap table where the initialisation code is located. The processor registers are then set to their initial values and any hardware that requires initialisation, such as the external memory, is configured. Programs written in C expect uninitialized data to be zeroed and because the initial value of memory may be undefined, the next step is to clear the entire uninitialized data section, referred to as the bss section.

Finally, the RTEMS initialisation function is called which initialises all of the operating system data structures, calls any device driver initialisation functions, creates the idle task then starts the user application.

The board initialisation file used in the LEON2-FT processor board BSP is a modified version of one provided by STAR-Dundee from the SPARC V8 Software Development Environment (STAR-Dundee Ltd 2015). The file was changed to configure the trap table to call the correct trap handlers, enable the external memory and call the RTEMS initialisation routine.

4.2.3.4 Interrupt Vectoring

When an interrupt is raised to the processor it causes the processor to trap into software. This makes the program counter jump to an entry in the trap table starting at the address indicated by the value of the SPARC Trap Base Register (TBR). In the LEON2-FT processor board BSP, the TBR is initialised to the value 0x81000000 which is the lowest address of the internal memory region.

Each of the interrupt trap entries in the trap table, from 0x11 to 0x1F, contain a function call to the RTEMS ISR handler. The ISR handler then looks up a table of ISRs and, if an ISR has been installed for a specific interrupt number, the ISR handler vectors the interrupt to the corresponding ISR. It is then the responsibility of the ISR to handle and clear the interrupt.

In the LEON2-FT processor board, each SpaceWire protocol engine shares a single processor level interrupt. In the RTEMS ISR handler, which is part of the operating system and doesn't know about specific peripherals, there is no way to discern if the SpaceWire protocol engine interrupt was raised by one of the DMA channels, the RMAP initiator or the RMAP target.

Once method of dealing with a shared interrupt is to demultiplex it by calling an ISR for each of the possible sources of the interrupt where each ISR only deals with their own interrupts. However, this results in code duplication in each ISR and wasted function calls if an ISR is called unnecessarily. A simple way to solve this problem is to install a demultiplexing ISR handler as the ISR for SpaceWire protocol engine interrupts. The demultiplexing ISR handler is responsible for checking which components of the engine are causing the interrupts and forwarding them to the corresponding drivers or second-level ISRs.

To do this, a SpaceWire ISR table is created which has entries for the DMA engines, RMAP initiators and RMAP targets for each of the SpaceWire protocol engines. The entries are pointers to second-level ISRs which can be installed or uninstalled. During RTEMS initialisation, the SpaceWire ISR handler is installed as the ISR for all three SpaceWire protocol engine interrupts. When one of the SpaceWire protocol engines raises an interrupt to the processor, the SpaceWire ISR handler is called which checks the source engine of the interrupt and which component of the engine it came from. The interrupt is then forwarded to the corresponding second-level ISR via the entry in the SpaceWire ISR table. If more than one component raises an interrupt at the same time, each is forwarded to the corresponding second-level ISR. The SpaceWire interrupt vectoring process is illustrated in Figure 4-4.

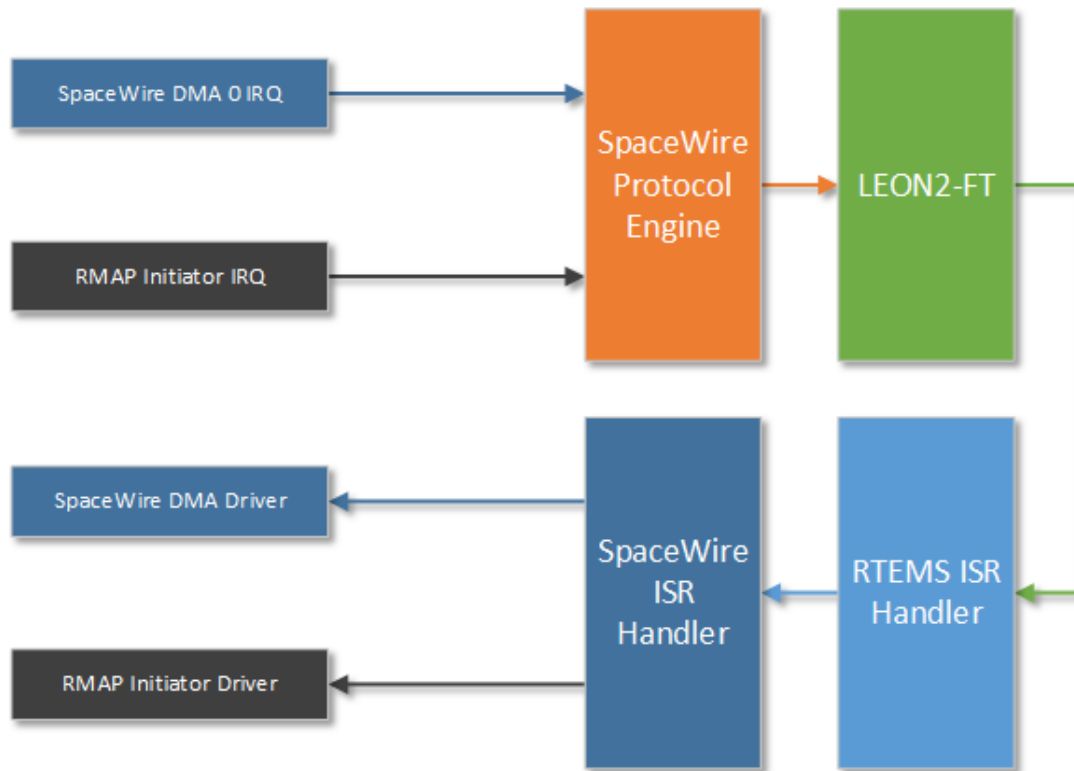


Figure 4-4: SpaceWire Protocol Engine Interrupt Vectoring

In Figure 4-4, two interrupts have been raised at the same time by separate components of the SpaceWire protocol engine: DMA channel 0 and the RMAP initiator. The two interrupts are combined into a single interrupt by the SpaceWire protocol engine which is then raised to the LEON2-FT processor. The initial interrupt is handled by the RTEMS ISR handler which calls the SpaceWire ISR handler. The possible sources of the interrupts are checked and forwarded to the corresponding second-level ISRs by calling the entries in the SpaceWire ISR table.

Handling the vectoring of SpaceWire protocol engine interrupts in this manner removes the need for the individual ISRs to check if they are the source of a shared interrupt. This results in the minimum number of ISR calls per shared interrupt rather than calls to the ISR of each possible source.

Using interrupts introduces problems such as interrupt overload, where interrupt handling starves tasks, resulting in missed deadlines. It also increases the difficulty of calculating the stack memory requirements due to nested interrupts. Furthermore, it increases the complexity of worst-case execution time (WCET) analysis because of the asynchronous and non-deterministic nature of interrupts (Regehr 2007). Interrupts are necessary if an event must be dealt with as soon as possible after it occurs and a state-monitoring task is not suitable (Kopetz, Real-Time Operating Systems 2011). However, in real-time systems, especially critical systems like spacecraft, non-essential interrupts should be used sparingly to minimise sources of non-determinism.

4.2.3.5 *Example: Ticker*

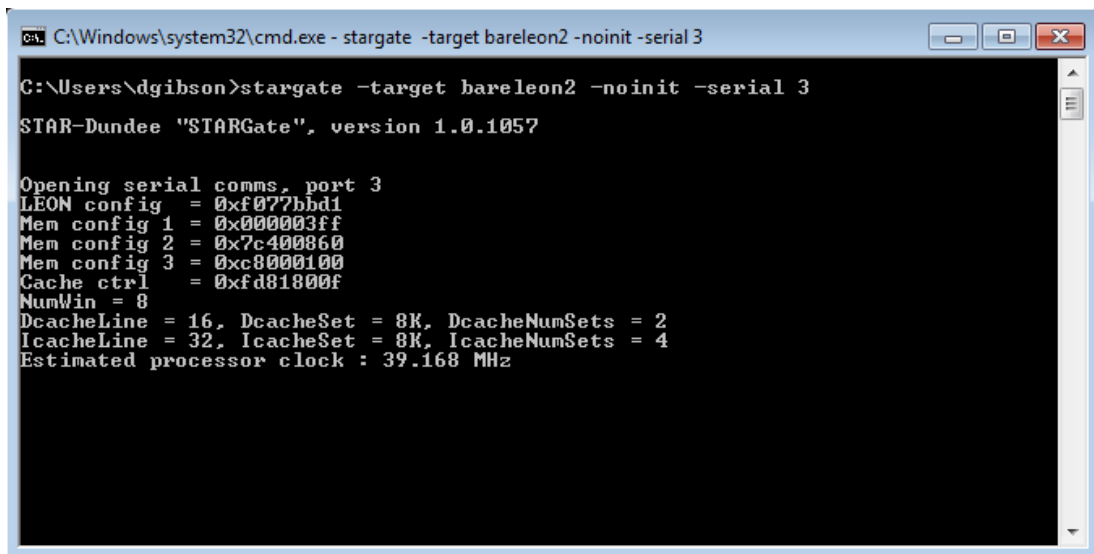
The RTEMS test suite includes a program called Ticker to test that the basics of a BSP are working. The program creates three tasks: TA1, TA2 and TA3 and schedules them to print a message every 5, 10 and 15 seconds respectively until 30 seconds have passed.

Running the Ticker program demonstrates that:

1. The cross-compiling toolchain has been setup correctly to:
 - a. Compile RTEMS and the BSP
 - b. Link applications with RTEMS and the BSP
 - c. Load a program to the board
 - d. Execute and debug a program
2. The board and RTEMS are initialised at the start of execution
3. The application is able to create and run tasks
4. The application can use the C standard library to print to standard output
5. Multiple tasks can be created and switched between

6. The LEON2-FT timer is generating interrupts which are being vectored to the RTEMS clock driver, allowing for a system-wide clock tick

After the operating system libraries, the BSP and the sample applications have been compiled, the first step in running the Ticker program is to connect to the LEON2-FT processor board with STAR-Gate so it can be used as the GDB RSP server, as shown in Figure 4-5.



```

C:\Windows\system32\cmd.exe - stargate -target bareleon2 -noinit -serial 3

C:\Users\dgibson>stargate -target bareleon2 -noinit -serial 3
STAR-Dundee "STARGate", version 1.0.1057

Opening serial comms, port 3
LEON config = 0xf077bbd1
Mem config 1 = 0x000003ff
Mem config 2 = 0x7c400860
Mem config 3 = 0xc8000100
Cache ctrl = 0xfd81800f
NumWin = 8
DcacheLine = 16, DcacheSet = 8K, DcacheNumSets = 2
IcacheLine = 32, IcacheSet = 8K, IcacheNumSets = 4
Estimated processor clock : 39.168 MHz

```

Figure 4-5: Using STAR-Gate to Connect to the LEON2-FT Processor Board

In Figure 4-5, a Windows terminal is used to issue a command to execute STAR-Gate, targeting a LEON2 board using serial port 3 with no default initialisation. The output received from this command shows that the connection was successful. In addition, it displays the values of some of the processor's registers, information about the register windows and caches, and the processor clock rate. Once executed, STAR-Gate runs as a service and waits for commands from GDB.

Once the STAR-Gate GDB RSP server is running, it can be connected to and a program can be loaded using GDB.

To view the output of the program, a serial port terminal can be used to receive any bytes transmitted over the UART. In this case, a simple freeware program called Terminate (CompuPhase 2015) is used, as shown in Figure 4-6.

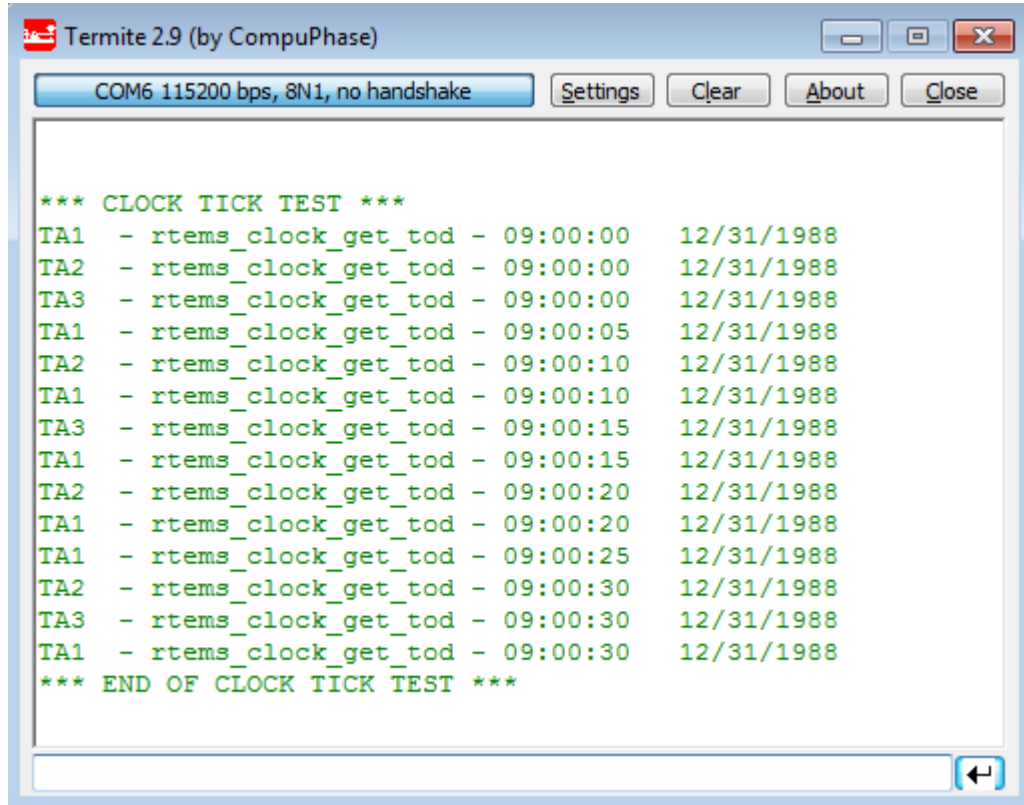


Figure 4-6: Ticker Program UART Output

In Figure 4-6, Terminate is used to connect to the LEON2-FT DSU UART using a baud rate of 115200, 8 data bits, no parity bits and 1 stop bit. After issuing an execution command in GDB, the output of the Ticker program is captured in Terminate and displayed as ASCII characters.

4.3 SpaceWire-D Layer

The SpaceWire-D layer consists of a number of modules written in the C language and running on top of the RTEMS real-time operating system and the LEON2-FT processor board BSP described in the previous sections. It handles the receiving of time-codes, the execution of time-slots and virtual buses, local-timer synchronisation,

error detection and reporting, user notifications. It also provides an API to the user application to open, load and close virtual buses as well as configure the network management parameters.

Determinism was of high-priority when designing the SpaceWire-D layer so the software uses no dynamic memory so that the memory footprint is well defined. In addition, after several iterations of the time-code receiving and time-slot execution process, no interrupts are used.

4.3.1 Architecture

A block diagram illustrating the top-level software architecture of the SpaceWire-D layer is shown in Figure 4-7.

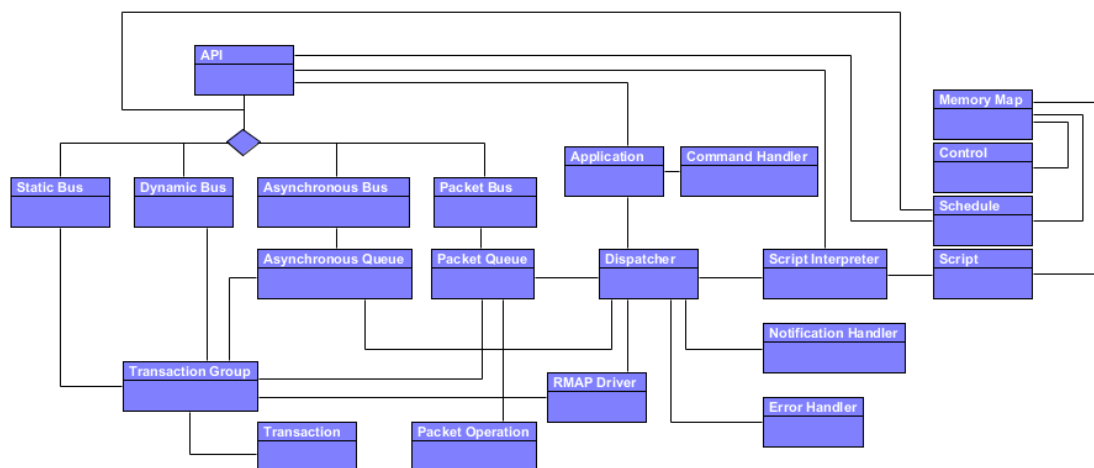


Figure 4-7: SpaceWire-D Layer Software Architecture

The following sections describe each block of Figure 4-7.

4.3.1.1 Application

The Application is responsible for initialising the SpaceWire-D layer and executing the control loop which is used to detect when a new time-slot should begin and when commands have been sent to the initiator by a configuration program, as described in Section 5.6.1. When a new time-slot is detected, either by a time-code or by local-

timer synchronisation, the dispatcher is told to execute the new time-slot. When a command is detected, it is passed to the command handler for execution.

4.3.1.2 API

The API provides functions for each service of the SpaceWire-D standard that can be called by a user application:

- Get/set parameters using the Management Service.
- Open/load/close static buses using the Static Bus Service.
- Open/load/close dynamic buses using the Dynamic Bus Service.
- Open/load/close asynchronous buses using the Asynchronous Bus Service.
- Open/close packet buses and receive/send packets packet buses using the Packet Bus Service.

4.3.1.3 Script Interpreter

The Script Interpreter executes time-code synchronised commands on behalf of the SpaceWire-D Demonstrator user, as described in Section 5.2.1.

4.3.1.4 Script

The Script is a data structure representing the time-triggered commands written by the SpaceWire-D Demonstrator user, as described in Section 5.2.1. It is divided into 64 different arrays of commands, one for each time-slot.

4.3.1.5 Command Handler

The command handler is responsible for executing commands sent by the SpaceWire-D Demonstrator user, as described in Section 5.6.1.

4.3.1.6 RMAP Driver

The RMAP Driver initialises the hardware RMAP initiator engine and is used by the Dispatcher to initiate groups of RMAP transactions.

4.3.1.7 Dispatcher

The Dispatcher is responsible for controlling the time-slot execution process, as described in Section 4.3.4.

4.3.1.8 Error Handler

The Error Handler holds a list of errors which are reported to the Network Manager at the end of each schedule epoch. Each error is a 32-bit value which contains the error type, the time-slot in which it occurred and a transaction ID if relevant.

4.3.1.9 Notification Handler

The Notification Handler holds an inbox of notifications which the user can use to read the latest events that have occurred.

4.3.1.10 Schedule

The Schedule data structure contains arrays of each type of virtual bus and it has an array of schedule entries which represent the virtual bus assigned to each time-slot.

4.3.1.11 Control

The Control data structure contains the initiator and target parameters which can be configured using the Management Service.

4.3.1.12 Memory Map

The Memory Map is used to lay out the Control and Schedule data structures in contiguous memory at a pre-determined address so that it can be read from and written to directly by a remote device.

4.3.1.13 Static Bus

The Static Bus data structure is used by the Static Bus API to create and assign Static Buses to the schedule. It can be loaded with a single transaction group which is executed in the next occurrence of its allocated slot and can be in one of two modes: single-shot or repeated.

4.3.1.14 Dynamic Bus

The Dynamic Bus data structure is used by the Dynamic Bus API to create and assign Dynamic Buses to the schedule. It can be loaded with two transaction groups; the current group is executed in the next allocated slot after which the next group becomes the current group.

4.3.1.15 Asynchronous Bus

The Asynchronous Bus data structure is used by the Asynchronous Bus API to create and assign Asynchronous Buses to the schedule. It can be loaded with single transactions which are held in a prioritised queue. In the time-slot preceding an allocated slot, a subset of transactions is pulled from the head of the queue and executed in the following slot.

4.3.1.16 Packet Bus

The Packet Bus data structure is used by the Packet Bus API to create and assign Packet Buses to the schedule. It can be loaded with requests to send or receive packets from a remote target device, which is implemented in three stages: first the Packet Bus checks if the relevant target's packet channel is ready to receive or send a packet, next the packet is transferred between the initiator and target in one or more segments and finally the Packet Bus writes an EOP to the target's packet channel to indicate that the operation is complete.

4.3.1.17 Asynchronous Queue

The Asynchronous Queue is used by the Asynchronous Bus to extract a transaction group for execution from the set of single loaded transactions. When an Asynchronous Bus is loaded with a transaction, the transaction is inserted into the queue with the requested priority level. In the time-slot preceding one allocated to the Asynchronous Bus, transactions are pulled from the head of the queue until their combined execution time fills the slot, forming the transaction group to be executed in the following slot.

4.3.1.18 Packet Queue

The Packet Queue is used by the Packet Bus to extract a transaction group for execution from the set of loaded packet transfer requests. When a Packet Bus is loaded with a packet transfer request, the request is inserted into the queue with the requested priority level. In the time-slot preceding one allocated to the Asynchronous Bus, a set of channel status requests, packet segment transactions or EOPs are pulled from the head of the queue until their combined execution time fills the slot, forming the transaction group to be executed in the following slot.

4.3.1.19 Packet Operation

The Packet Operation data structure describes a single Packet Bus packet receive/send request. The operation holds data such as the target logical address, the packet channel identifier and the current stage of the request which is used by the Packet Queue to form a transaction.

4.3.1.20 Transaction

The Transaction data structure describes a single RMAP transaction including the read/write buffer locations, notification buffer locations and the header values.

4.3.1.21 Transaction Group

The Transaction Group data structure contains an array of RMAP transactions.

4.3.2 Virtual Buses

There are four types of virtual bus: static, dynamic, asynchronous and packet. Each type of bus inherits from a shared virtual bus data structure containing an ID, an application handle, a 64-bit allocated time-slot vector and a 224-bit permitted target vector. The bit vectors are used in place of lists of bytes because they do not require any dynamically allocated memory and they require less static memory than equivalent lists of bytes.

Static buses extend the shared virtual bus data structure with a static bus mode value, which is set to either single-shot or repeating, and a pointer to an RMAP transaction group. Dynamic buses extend the shared virtual bus data structure with two pointers to RMAP transaction groups; one for the current group and one for the next group. Asynchronous and packet buses extend the shared virtual bus data structure with their own specialised priority queues containing either asynchronous bus transactions or packet bus operations.

The priority queues are implemented as min-heaps (Cormen, et al. 2009) with a fixed size, configurable at compile time, in order to keep a bounded memory footprint. Each queue item has a 32-bit label consisting of a 3-bit priority level and a 29-bit counter value which is incremented for each inserted item. This allows items to be extracted during the asynchronous or packet bus preparation processes and re-inserted at the same position, if required. In the time-slot preceding any allocated to an asynchronous or packet bus, a transaction group is prepared by processing the priority queue belonging to the bus.

For an asynchronous bus, the preparation process involves extracting the transactions from the head of the queue and checking if they will fit within the next time-slot's duration. If a transaction fits, it is added to the transaction group, otherwise it is ignored. When all of the queue's transactions have been checked, any ignored transactions are reinserted into the queue and the pre-extraction prioritised order is maintained because of the item label.

For a packet bus, the preparation process involves extracting the operations from the head of the queue and checking if the transaction related to the current stage of the operation will fit within the next time-slot's duration. A packet bus operation can be in one of three stages. Firstly, a transaction is formed to check the status of the relevant packet channel in the target. Secondly, one or more transactions are formed to transfer the packet in segments. Finally, in the third stage, a transaction is formed to write an EOP to the relevant packet channel in the target. When a packet bus operation is extracted from the queue, the relevant type of transaction is formed. If it fits within the next time-slot, the transaction is added to the transaction group. Otherwise, the packet bus operation is ignored and later re-inserted into the queue in the same way as the asynchronous bus preparation process.

4.3.3 Management

There are several SpaceWire-D management parameters that can control the operation of the initiator and are used to calculate RMAP transaction group execution times:

- **Activated:** indicates whether or not the initiator is currently executing its schedule
- **Logical address:** the logical address of the initiator

- **Active schedule:** switches between different schedules if there are more than one available
- **Processing time:** the overhead time required at the start of each time-slot
- **Post-processing:** the latency between an RMAP reply being received and the next RMAP command being transmitted
- **Time-slot duration:** the expected duration of a time-slot
- **Switching time:** the router switching time

In addition to the initiator parameters, each target has its own parameters:

- **Number of routers:** the number of routers in the path between the initiator and the target
- **Slowest link:** the slowest link speed in the path between the initiator and the target
- **Response time:** the latency between a command being received and a reply being sent by the target
- **Packet channels:** the location of the packet channel data structures in the target

For all parameters related to timing, the worst-case values should be used to make sure that the SpaceWire-D layer is accounting for the maximum transaction execution times.

4.3.4 Executing Time-Slots

The execution of a time-slot takes place after a time-code has been received by the initiator and handed to the dispatcher. The dispatcher then executes a time-slot in three stages. Firstly, the pre-execution stage reads and resets the time-code watchdog, checks for any outstanding transactions that need to be cancelled and copies the status

of any completed transactions into local buffers to perform error checking in the third stage. Secondly, the execution stage checks if there is a schedule entry for the current slot and if so, the corresponding virtual bus is executed. Finally, the post-execution stage checks the time-code watchdog to detect any early or late time-codes and performs error checking on the previous slot's transactions. Additionally, if there is an asynchronous or packet bus allocated to the following slot, this stage performs the bus preparation process.

4.3.4.1 Initiator Processing Time

The initiator processing time is the latency between a time-code being received by the initiator and the first RMAP command being transmitted in a time-slot. It is important to reduce this latency so that the maximum percentage of each time-slot can be used to execute RMAP transactions.

In the SpaceWire-D layer, the time-slot execution process went through several revisions in order to reduce the initiator processing time to a usable level. In order to measure the initiator processing time, a STAR-Dundee Link Analyser Mk2 (Scott and Parkes 2010) was placed between an initiator and an RMAP target. A STAR-Dundee Brick Mk2 (STAR-Dundee Ltd 2017) was used as the time-code master and the initiator schedule was configured to send RMAP commands in each time-slot which were captured using the Link Analyser. The initiator processing time was then measured by recording the worst observed latency, over several schedule epochs, between a time-code being received by the initiator and the first byte of the first transaction executed in each time-slot. The following section describes the optimisation of the initiator processing time.

4.3.4.2 *Optimisation*

In a traditional OS device driver framework, the OS provides the user with a common interface of system calls which, in turn, call device-specific driver functions. In POSIX environments, these functions include the open, close, read, write and ioctl (input/output control) calls and operate on a device file, allowing a user to interface with a device as if it was a normal file. This approach means that the user-space interface for drivers is standardised across all devices, allowing for a driver to be replaced by another more easily (Corbet, Rubini and Kroah-Hartman 2005).

The SpaceWire-D layer used a multi-task and device file driver approach for the first revision of the time-slot execution process. The full process was as follows:

1. Time-code IRQ handled by the router driver ISR
2. Router driver ISR sends the time-code event to the timer manager
3. Timer manager sends the next slot event to the dispatcher
4. Dispatcher opens the RMAP driver file
5. Dispatcher sends a queue command to the RMAP driver
6. RMAP driver copies the transactions into the driver buffers
7. Dispatcher sends a start command to the RMAP driver
8. RMAP driver executes the transaction group

The first revision of the time-slot execution process was a design with multiple tasks that had specific purposes and a kernel-space driver that used the POSIX style user-space interface. However, the software overhead of this design was extremely high, with an initiator processing time of ~2.7 ms for groups of 16 transactions and ~4.7 ms for groups of 32 transactions.

Although the SpaceWire-D standard does not define minimum and maximum time-slots, the protocol is intended to be able to operate at a minimum time-slot of 1 ms and a maximum time-slot of 1 second. Therefore, when measuring the efficiency of the initiator processing time, it is measured in the context of the worst-case in which a SpaceWire-D network is using 1 ms time-slots. Taking this into consideration, the initiator processing time required improvements before it would be suitable when using 1 ms time-slots.

To determine which sections of the time-slot execution process were causing the initiator processing time to be so high, support for logic analyser output was added to the LEON2-FT processor board by STAR-Dundee. A register was added that, when written to and enabled, outputs values to a logic analyser port. The embedded software was extended to write values to the logic analyser register at important points during the time-slot execution process. For example, at the start of the interrupt vectoring process; before and after context switching; before and after calling operating system functions; and before and after calling SpaceWire-D API functions. These measurements were then used to identify which sections of the time-slot execution process were consuming the most time.

To measure the overall initiator processing time, a STAR-Dundee Link Analyser Mk2 (Scott and Parkes 2010) was used. The Link Analyser allowed for traffic to be captured in both directions on a link before and after a configured trigger point. In this case, the Link Analyser was configured to trigger on the header bytes of any packets being transmitted out of the initiator. The initiator processing time was then defined as the time between a time-code being received by the initiator and the first byte of the first RMAP command transmitted by the initiator. This allowed the time-code interface and RMAP initiator hardware overheads to be included in the initiator processing time.

In order to reduce the initiator processing time, the first change that was made was to introduce a zero-copy version which used the user-supplied transaction buffers rather than copying the transactions into the driver buffers. The intention was that this would also keep the initiator processing time consistent no matter how many transactions were in a group because only a single pointer to the transaction group is copied. This change had a significant effect on the initiator processing time and reduced it to ~750 μs for all sizes of transaction groups up to the maximum of 32.

The second change that was made was to remove the device file system calls from the process. This optimisation was identified by the logic analyser output showing a substantial amount of processing time caused by the system calls. Normally, system calls are required to access driver functions because user applications aren't permitted to access restricted memory and functions such as those found in a driver. However, because RTEMS for the SPARC architecture operates with a flat address space, meaning a program can access all memory, there is no hard separation between user and kernel space. This allows the option to call the driver functions directly without switching to kernel space. The next revision of the time-slot execution process bypassed the system calls which further reduced the initiator processing time to ~389 μs .

Next, the event signalling was removed by using a callback function in the timer manager instead of issuing an event. The callback function then manually resumed the dispatch task which had suspended itself rather than waiting for the timer manager event, reducing the initiator processing time to ~201 μs . Again, this optimisation was identified using the timing information gathered using the logic analyser output.

The final version of the time-slot execution process made some additional optimisations. Firstly, in place of a time-code receive IRQ and timer manager ISR, the SpaceWire-D layer uses a control loop. This loop polls the time-code receive flag in one of the router registers as well as checks the local-timer synchronisation timer, as described in the following section. Secondly, the dispatch task was removed and the dispatching is now done in the main task, as soon as a time-code is received. Thirdly, the error checking and time-code watchdog was divided into a pre-execution stage, where the corresponding status values are saved to local variables, and a post-execution stage, where the status values are then checked. The final version using these optimisations reduced the initiator processing time to $\sim 90 \mu\text{s}$. These optimisations were identified firstly by the logic analyser timing output and secondly, during the architecture change to use polling instead of interrupts.

In the first version of the time-slot execution process, it was possible for three levels of nested interrupts. Interrupts could occur when time-codes were received, when transaction groups completed execution and when the operating system clock driver ticked. In the final version, the SpaceWire-D layer uses no interrupts in order to increase determinism. It polls the time-code status register to determine if time-codes have arrived, it uses status descriptors written to by the RMAP initiators rather than a completion interrupt and the system clock driver is disabled because the local-timer synchronisation mechanism uses polling.

4.3.5 Local-Timer Synchronisation

In order to provide redundancy in case time-codes are late or missing, SpaceWire-D allows an initiator to synchronise the receiving of time-codes with a local timer. This is used to automatically execute time-slots at the expected time-code intervals. The SpaceWire-D layer uses one of the LEON2-FT processor board's on-board timers to

perform the local-timer synchronisation. A rolling average of time-code intervals is calculated which is updated whenever a time-code is received. During the SpaceWire-D control cycle, the local-timer is checked for expiration and if it has expired, a missing time-code error is flagged and the next time-slot is executed in the same way as if the next time-code had been received.

4.3.6 Error Detection and Reporting

During the time-slot execution process, any transactions executed in the previous time-slot are checked for errors which may be categorised as encoder, decoder or incomplete transaction errors. If an RMAP command had erroneous values in its header parameters or an error occurs resulting in the command not being able to be sent, it is recorded as an encoder error. If an RMAP reply comes back with an error or if an error occurs resulting in the reply not being able to be processed, it is recorded as a decoder error. Finally, if a transaction is still outstanding when its time-slot ends, the transaction is cancelled and an incomplete transaction error is recorded. In addition to the RMAP errors, early or late time-code errors can be recorded by the time-code watchdog and missing time-code errors can be recorded by the local-timer synchronisation module.

When the errors are recorded, they are added to an error list located at a specific memory address in the initiator which is prepended by an error count value. At the end of each schedule epoch, this error list is reported to the network manager via an RMAP write transaction.

The SpaceWire-D layer is responsible for detecting and recording different types of errors and allowing them to be reported to the network manager but it doesn't perform

any error isolation or recovery. This is left to the system designer to add as an additional layer on top of SpaceWire-D or to handle in the application.

4.3.7 Notifications

The SpaceWire-D layer uses a notification queue to inform the user application of events that occur during the execution of the schedule. Whenever a virtual bus is executed, a transaction group is completed, a time-code error occurs or an RMAP error occurs, a notification is formed and added to the end of the notification queue. The user can then read the notifications at a suitable time and be updated on the status of the schedule and transactions.

4.3.8 Testing and Verification

In order to test and verify the SpaceWire-D layer, testing was undertaken in two separate activities. Firstly, during development of the SpaceWire-D layer, a suite of unit tests was created to verify that the embedded software continued to work correctly after each change. Secondly, a protocol verification activity was undertaken to ensure that the SpaceWire-D layer was fully and correctly implemented as specified in the SpaceWire-D standard (Parkes, Gibson and Ferrer 2015 B).

4.3.8.1 Unit Testing

A suite of unit tests was created using a minimal implementation of an xUnit style test framework (Meszaros 2007). These tests used the SpaceWire-D user and internal APIs to call functions with correct and incorrect parameters and validate the results. The unit test suite was helpful in confirming that optimisation or implementation changes in the embedded software did not introduce errors into the SpaceWire-D layer.

4.3.8.2 *Protocol Verification*

The SpaceWire-D layer was used by the initiators in the SpaceWire-D Demonstrator used to complete the ESA SpaceWire-D protocol verification activity. The aim of this activity was to map every clause of the SpaceWire-D standard that was possible to explicitly verify with a test using the SpaceWire-D layer. These tests were specified by the author in the SpaceWire-D Verification Test Specification Document (University of Dundee 2015) which was delivered to ESA as part of the SpaceWire-D project.

The tests consisted of multiple verification test set-ups each defined with a network architecture, initiator schedules, time-slot mode settings, a list of test procedures and a list of pass criteria. The test set-ups included single and multiple initiator tests, local-timer synchronisation, time-code watchdog, target initialisation tests, static bus, dynamic bus, asynchronous bus, packet bus tests, and fault prevention, detection and isolation tests.

Each clause of the standard was mapped to a specific test set-up and a description of how the pass criteria of the test verifies the clause was given. The tests were executed by the author and the results were gathered and presented in the SpaceWire-D Verification Report (University of Dundee 2016 A) which was then delivered to ESA as part of the SpaceWire-D project. This activity shows that the SpaceWire-D protocol is fully verified, possible to use and that the SpaceWire-D layer is compliant with it.

4.4 **Summary**

This chapter described the design of the SpaceWire-D software layer running on top of the RTEMS real-time operating system and used to allow the LEON2-FT based processor boards to act as SpaceWire-D initiators.

Firstly, the RTEMS real-time operating system was ported to the LEON2-FT processor board by creating a board-support package based on existing LEON support in the RTEMS source tree. The existing BSP was extended to support the different register layout, interrupt vectoring, peripherals and memory layout of the LEON2-FT processor board.

Next, a SpaceWire-D software layer was designed on top of the RTEMS BSP. The layer provides the virtual buses, network management, time-slot execution, local-time synchronisation, error detection and user notification functionality of a SpaceWire-D initiator. In addition, it provides an API to the user application to allow it to open, load and close virtual buses as well as configure network management parameters and receive notifications of SpaceWire-D related activity.

The SpaceWire-D layer was used in the SpaceWire-D protocol verification activity during the ESA SpaceWire-D project. This resulted in two deliverables to ESA: a verification test specification (University of Dundee 2015) and a verification report (University of Dundee 2016 A). The SpaceWire-D layer for the LEON2-FT processor board was the first system to implement the latest version of the standard as it was designed in collaboration with the drafting of the standard.

Chapter 5

SpaceWire-D Demonstrator

The SpaceWire-D Demonstrator is a system comprised of many different devices, housed within a 19-slot PXI (PXI Systems Alliance 2004) rack. It was used to complete the verification activity of the ESA SpaceWire-D project as well as demonstrate the features of SpaceWire-D.

Each device has a role, either as an initiator, a target, a router, the network manager, or the host PC running a suite of software used to configure, control and monitor the SpaceWire-D network. There are two initiator boards, twelve targets contained within three RMAP interface boards, two 8-port SpaceWire routers, one network manager and one PXI system controller acting as the host PC.

This chapter describes the design of the SpaceWire-D Demonstrator and the collaboration between the different devices contained within the Demonstrator and the software running on the host PC.

5.1 Overview

In order to verify the SpaceWire-D standard and demonstrate its features, a SpaceWire-D Demonstrator was required. It was specified in the SpaceWire-D D3 Demonstrator Specification Document (University of Dundee 2014) as part of the ESA SpaceWire-D project. The document described the requirements of the initiators,

targets and network manager as well as the required verification and validation software and validation test scenarios.

The SpaceWire-D Demonstrator consists of:

- Two PXI LEON2-FT processor boards acting as the initiators and controlling the execution of all RMAP transactions
- Three PXI RMAP interface boards each containing four individual RMAP targets with separate memory regions, resulting in a total of 12 RMAP targets
- One PXI RMAP interface board acting as the network manager used to receive and store statistics and error information reported by the initiators
- Two PXI 8-port SpaceWire routers providing the network between the devices
- One PXI system controller running Windows 7 acting as the host PC and running a suite of software used to configure, control and monitor the SpaceWire-D network

A photograph of the SpaceWire-D Demonstrator is shown in Figure 5-1.

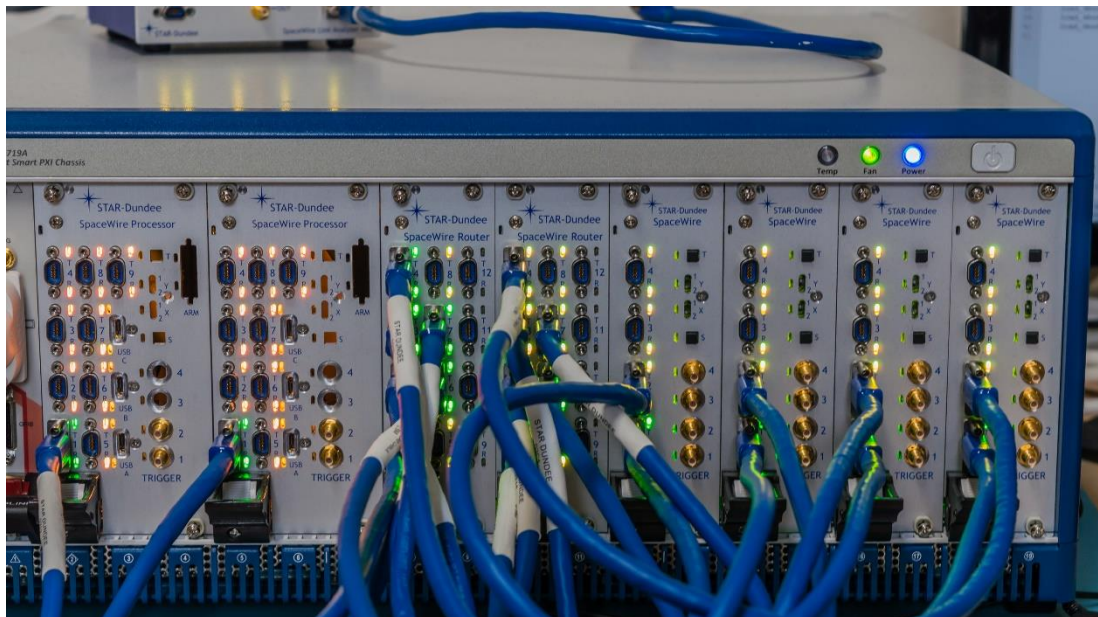


Figure 5-1: SpaceWire-D Demonstrator PXI Rack

In Figure 5-1, the PXI rack contains the following boards from left to right: Initiator 0, Initiator 1, Router 0, Router 1, Network Manager, Target Interface 0, Target Interface 1 and Target Interface 2. To the left of Initiator 0, partially in shot, is the Host PC.

There are 11 SpaceWire 0.5m cables providing the network between the initiators, targets, routers and network manager. The topology and logical addressing has been designed so that both initiators can communicate with targets on the same target interface board without sharing links. This allows, for example, Initiator 0 to communicate with two targets in Target Interface 0 and Initiator 1 to communicate with the other two targets within the same time-slot, without violating the no conflicting virtual buses rule of SpaceWire-D. A network topology diagram for the SpaceWire-D Demonstrator is shown in Figure 5-2.

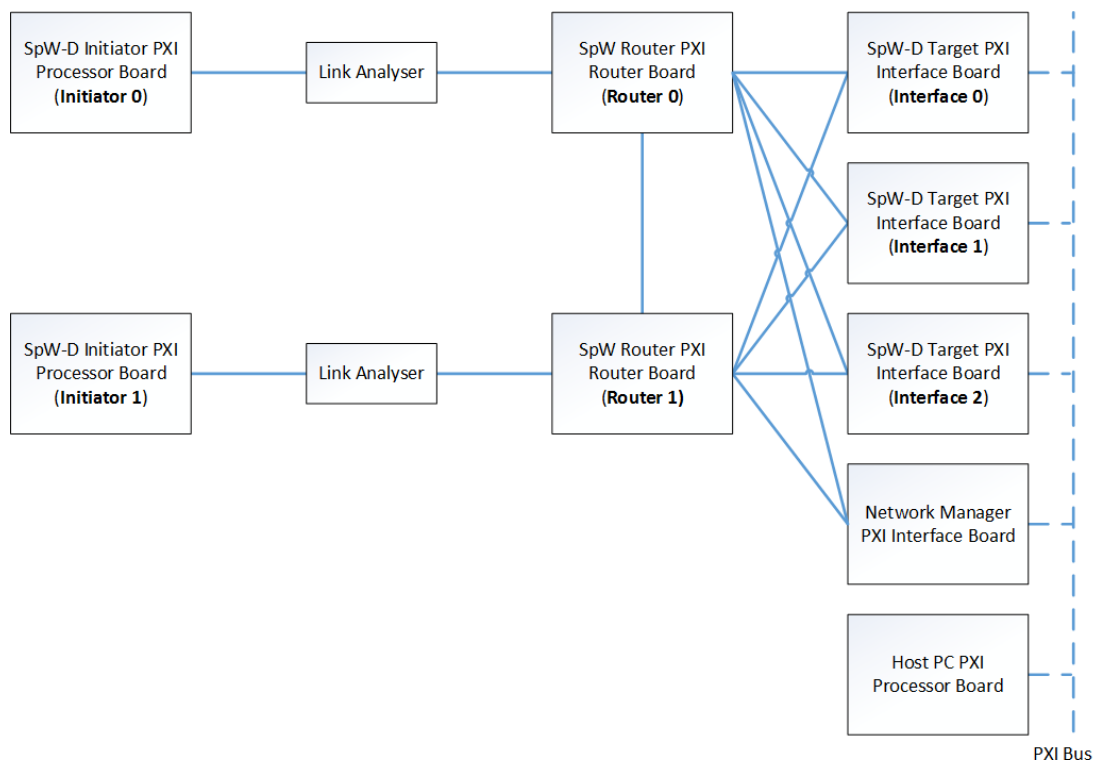


Figure 5-2: SpaceWire-D Demonstrator Network Topology

In Figure 5-2, the network topology shows that Initiator 0 is connected to Router 0 and Initiator 1 is connected to Router 1. If Initiator 0 wants to send an RMAP command to a target, the command is routed from Router 0 to SpaceWire port 1 of the relevant target interface board and if Initiator 1 wants to do the same, the command is routed from Router 1 to SpaceWire port 2 of the target interface board. Commands sent to the Network Manager from the Initiators are routed in a similar manner.

There are Link Analyser devices placed in between Initiator 0 and Router 0, and Initiator 1 and Router 1. These devices were used to capture the traffic flowing between the initiators and routers but they are passive and transparent. They were used only to capture the results presented in this thesis and were not part of the final SpaceWire-D Demonstrator deliverable.

Each of the target interface boards contains four individual RMAP targets with their own logical addresses and regions of memory. Targets 0-3, 4-7 and 8-11 are contained within Interface 0, Interface 1 and Interface 2, respectively. The Network Manager uses two of its targets. The first is dedicated to receiving Initiator 0's statistics and error reports and the second is dedicated to receiving the reports from Initiator 1.

The SpaceWire-D Demonstrator uses logical addressing throughout the network to route packets between nodes. Logical addressing was selected, rather than path addressing, because it can be used to allocate an explicit address to each device. It also increases the efficiency of the network by reducing the addressing overhead to a single logical address byte compared to multiple path address bytes. The logical addresses and the available memory regions of each device in the network are listed in Table 5-1.

Table 5-1: SpaceWire-D Demonstrator Address Scheme

Device	LA	Memory (Start)	Memory (End)
Initiator 0 (Initiator)	0x30	N/A	N/A
Initiator 0 (Target)	0x90	0x60000000	0x61000000
Initiator 1 (Initiator)	0x31	N/A	N/A
Initiator 1 (Target)	0x91	0x60000000	0x61000000
Target 0	0x40	0x00000000	0x10000000
Target 1	0x41	0x00000000	0x10000000
Target 2	0x42	0x00000000	0x10000000
Target 3	0x43	0x00000000	0x10000000
Target 4	0x50	0x00000000	0x10000000
Target 5	0x51	0x00000000	0x10000000
Target 6	0x52	0x00000000	0x10000000
Target 7	0x53	0x00000000	0x10000000
Target 8	0x60	0x00000000	0x10000000
Target 9	0x61	0x00000000	0x10000000
Target 10	0x62	0x00000000	0x10000000
Target 11	0x63	0x00000000	0x10000000
Net. Man. (Target 0)	0x70	0x00000000	0x10000000
Net. Man. (Target 1)	0x71	0x00000000	0x10000000

As listed in Table 5-1, each node has a logical address and, if the node is a target, a memory region. Each initiator board also contains a target with a 16 Mbyte region of memory starting at address 0x60000000. Each of the targets within the target interface boards have a 256 Mbyte region of memory starting at address 0x00000000.

Figure 5-2 also shows that the target interface boards, the Network Manager and the Host PC are connected to the PXI bus backplane. The backplane is used by the Host PC to read and write to target memory and receive RMAP command notifications from the targets, as described in Section 5.6.

The interactions between the different devices are illustrated in Figure 5-3.

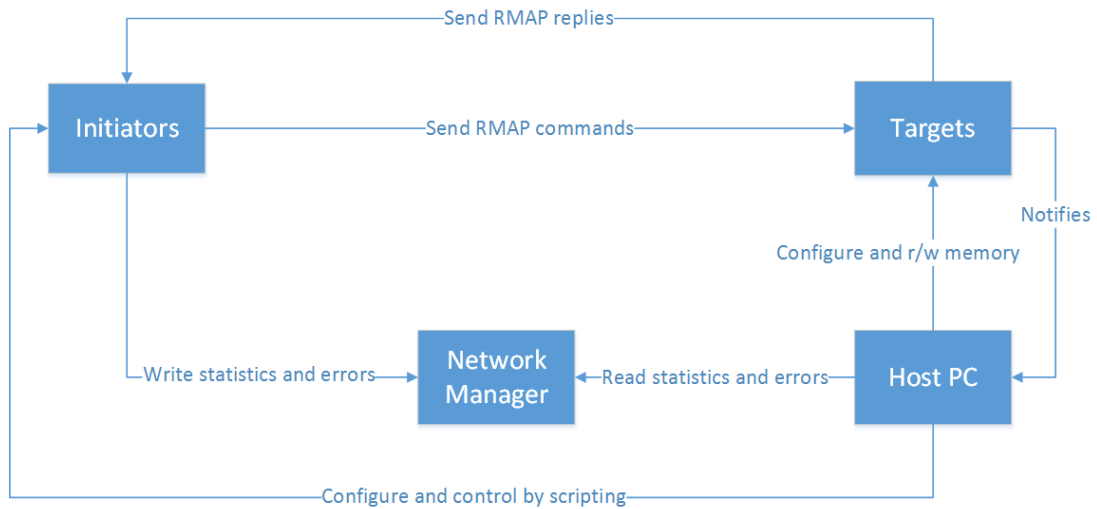


Figure 5-3: SpaceWire-D Demonstrator Interactions

As shown in Figure 5-3, each device interacts with one or more other devices in the SpaceWire-D Demonstrator. The initiators send RMAP commands to the targets and the targets send RMAP replies in response. The initiators report statistics and error information to the Network Manager, which is then read by the Host PC. The Host PC configures the initiators using RMAP commands and uploads automated test scripts to control their operation. The targets are configured by the Host PC using a combination of RMAP commands and memory reading/writing on the backplane. The targets notify the Host PC whenever they execute an RMAP command.

In the following sections, each type of device is described in more detail.

5.2 Initiators

The initiators are LEON2-FT based processor boards (Parkes, McClements and Mantelet, et al. 2013), with extensive SpaceWire support. Each board has a SpaceWire router with eight physical ports and three external ports. The external ports are connected to three independent SpaceWire protocol engines containing three DMA

controllers, an RMAP initiator and an RMAP target. The board's hardware is described in Appendix 1.

In addition to the embedded SpaceWire-D software layer running on the initiators, there is a demonstrator application. The application is responsible for interpreting scripted commands which are uploaded to the initiators by the Host PC in order to automate test scenarios.

5.2.1 Automated Test Scripting

To allow the SpaceWire-D Demonstrator to be more flexible and simplify the loading of test scenarios, an automated test scripting language was designed. The language allows a user to write a script as a text file containing transaction and transaction group information, packet bus operations and time-triggered commands to open, load and close different types of virtual buses. These commands are executed at specific times during the execution of the schedule.

The user can use the Host PC software to compile a script into a block of data and instructions, which is then uploaded into initiator memory. Throughout the execution of the SpaceWire-D Demonstrator, the initiators check the list of instructions and execute the appropriate commands as required.

5.2.1.1 Scripting Language

The requirements of the SpaceWire-D Demonstrator scripting language were very specific and simple. The language must allow RMAP transactions, transaction groups, packet bus operations and buffers to be defined. It must also allow virtual buses to be manipulated at specific times, synchronised with the receiving of SpaceWire time-codes by the initiator executing the script.

Creating a small scripting language to meet these requirements was simpler than using an existing general purpose scripting language such as Python, and has reduced memory requirements because of its very specific nature.

There are four stages in the automated test scripting process. Firstly, a user writes a script file. Secondly, the script file is compiled by a host PC application, using a simple text parser, and turned into a data structure describing the transactions, transaction groups, packet bus operations and time-slot triggered commands. Thirdly, a host PC application writes the data structure to a pre-determined location in an initiator and uses RMAP write transactions to configure any target data buffers as described in the script. Finally, the initiator begins executing the time-slot triggered commands, synchronised by the receiving of time-codes.

The data structure created by the host PC application and written to the initiator contains up to 64 arrays of commands, one for each time-slot. At the start of the data structure, a token is written so that the initiator can determine when a new script has been written to the script memory address. Following this are the 64 arrays of commands, each prefixed by a script header that describes the relevant time-slot and the number of commands contained in the array. Each command consists of a number of operands and an instruction describing the opcode to execute and its timing parameters, as described in Section 0.

Without an automated scripting language, it would be necessary for a user of the SpaceWire-D Demonstrator to write an RTEMS application for each test they want to run. This application would then need to be loaded and executed using GDB or a similar loader. Using the scripting language allows for scripts to be easily written

without knowledge of RTEMS or the embedded software, using the simple language then modified or extended, and loaded during run-time.

5.2.1.2 Examples

In the following sections, three example scripts are described. The first is a simple script which creates a schedule containing one static bus. The second creates a more complex script with multiple static buses. Finally, the third script creates a schedule containing all types of virtual buses. The scripts are executed and the results are captured and described in Section 0.

Example Script 1 – Simple Static Bus Schedule

The first example script, listed in Figure 5-4, creates four transactions and assigns them to two transaction groups. In time-slot 62, the initiator is instructed to open two static buses and load them with the transaction groups.

```

1  // Transactions
2  transaction(0, 64, read, 32, 0x00000000, 32, 0x60000000)
3  transaction(1, 80, write, 32, 0x00000000, 32, 0x60000020)
4  transaction(2, 96, read, 32, 0x00000000, 32, 0x60000040)
5
6  // Statistics/errors transaction
7  transaction(3, 112, write, 32, 0x00000000, 2048, 0x8101B800)
8
9  // Transaction groups
10 transaction_group(0, (0, 1, 2))
11
12 // Statistics/errors transaction group
13 transaction_group(1, (3))
14
15 slot(62):
16     open_sbus(1, 0, 0, 0, 1, (64, 80, 96))
17     open_sbus(1, 0, 0, 63, 1, (112))
18
19     load_sbus(1, 0, 0, 0, 0, repeat)
20     load_sbus(1, 1, 0, 63, 1, repeat)

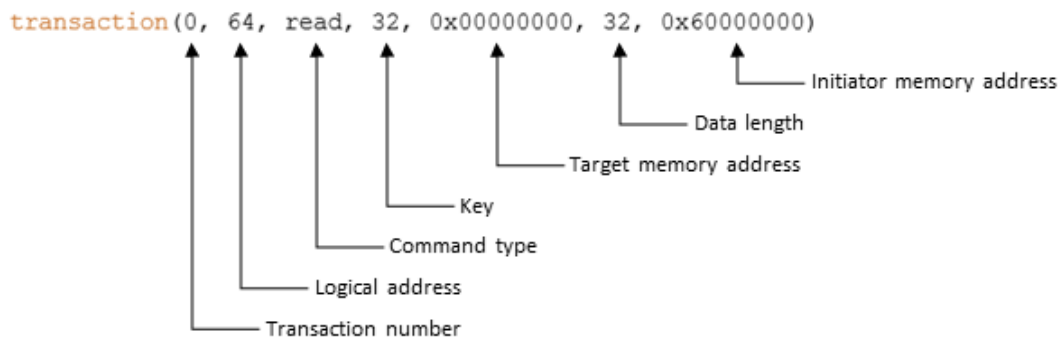
```

Figure 5-4: Example Script 1 – Simple Static Bus Schedule

In Figure 5-4, the script is divided into two sections. The first section, from lines 1 to 14, is the preamble which contains the transaction and transaction_group commands

used to create transactions and assign transactions to transaction groups. The second section, from lines 15 to 20, is the time-slot triggered commands section used to instruct the initiator to perform various actions at specific times.

The first command, on line 2, creates a transaction:



contains the statistics and error information that will be reported to the Network Manager at the end of each schedule epoch.

The command, on line 10, creates a transaction group:

```
transaction_group(0, (0, 1, 2))
```

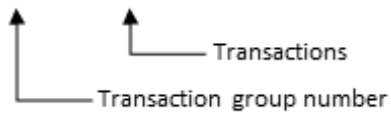


Figure 5-6: Transaction Group Command

In this line, the command is called “transaction_group” and it contains a parenthesised parameter list with two parameters. The first parameter is the transaction group number, used in other commands to reference this specific transaction group. The second parameter is a parenthesised list of transaction numbers, referencing any transactions created using “transaction” commands.

In this example, there are two transaction groups. The first, on line 10, contains transactions 0, 1 and 2. The second, on line 13, contains transaction 3.

The command, on line 15, starts a time-slot triggered command group:

```
slot(62):
```




Figure 5-7: Slot Command

In this line, the command is called “slot” and it contains a parenthesised parameter list with one parameter. The parameter is the number of the time-slot which triggers any commands contained in the group.

In this example, there is one time-slot triggered command group for time-slot 62 and it contains four commands from lines 16 to 20.

The command, on line 16, opens a static bus:

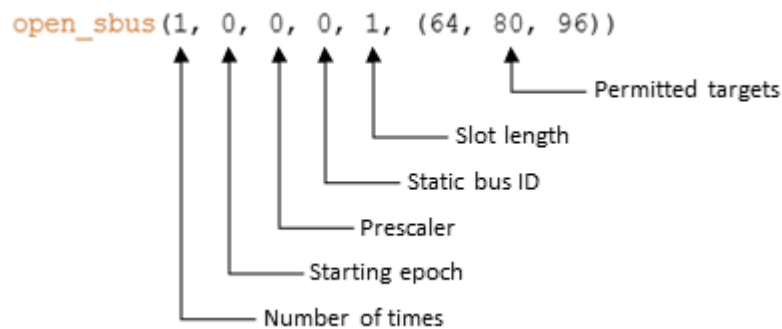


Figure 5-8: Open Static Bus Command

In this line, the command is called “open_sbus” and it contains a parenthesised parameter list containing six parameters. The first three parameters are the number of times the command will be executed, the starting epoch and the prescaler which will be described in detail below. The fourth parameter is the ID of the static bus, i.e. the number of the time-slot allocated to it. The fifth parameter is the length of the virtual bus’s slot, allowing for a slot containing multiple consecutive time-slots. Finally, the sixth parameter is a parenthesised list of targets that the static bus is permitted to send commands to.

The first three parameters allow for more flexibility with regards to how many times, and when, the command is executed rather than using only the time-slot. The first parameter is an 8-bit integer and controls how many times the command is executed. This value is decremented upon each execution and when the number of executions reaches zero, the command will be ignored. If the command is required to be executed for an infinite number of times, a special value of 255 can be used. The second parameter is an 8-bit counter that is decremented every time the relevant time-slot occurs. If the time-slot occurs and the counter is already zero, the command is executed and the counter is reloaded with the value of the third parameter, the 8-bit

prescaler. This allows a command to be executed in intervals greater than a schedule epoch or, by setting an initial counter value, defining a delay until the command's first execution.

Three examples of command execution parameter values are listed in Table 5-2.

Table 5-2: Example Command Execution Parameters

Number of Executions	Counter	Prescaler
2	0	0
255	0	0
255	1	1

If commands with these execution parameter values were assigned to time-slot 0 in a script, the resulting command executions would look like Figure 5-9.

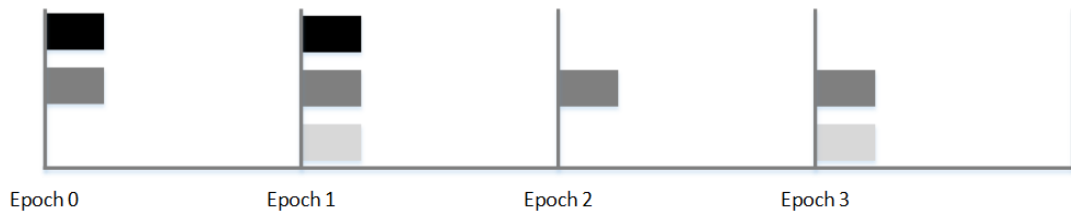


Figure 5-9: Controlling Command Execution

In Figure 5-9, the diagram shows four schedule epochs containing boxes where the commands would be executed. Each row of boxes represents the same row of command execution parameter values from Table 5-2.

The top row shows that the first command would be executed twice in the first two epochs and then ignored in future epochs because its execution count would be decremented twice until it was zero. The middle row shows that the second command, with the special execution count of 255, would be executed in every epoch. Finally, the bottom row shows that the command is first executed in the second epoch, because

it has an initial counter value of 1. After this, the command is executed in every second epoch due to its prescaler value being 1.

Controlling the execution of the commands using these parameters is advantageous because it reduces the need for an additional timer, with the corresponding interrupts, to time the execution of commands. Adding interrupts would result in additional software overhead to handle them. However, because the SpaceWire-D layer is already detecting when time-slots begin and end, the scripted commands can be executed at the end of the SpaceWire-D layer's time-slot execution process.

Additionally, the command execution parameters increase the flexibility of command execution compared to only using the time-slot. For example, consider a SpaceWire-D network where time-codes are being broadcast at a rate of 640 Hz. Using only time-codes, commands in one time-slot could be scheduled with 100 ms between executions. However, using the prescaler parameter, commands can be scheduled for execution with a minimum of 100 ms and maximum of 25.6 seconds between executions, in 100 ms intervals. Additionally, the counter parameter allows for an initial offset of between 0 ms and 25.6 seconds, again in 100 ms intervals.

In this example script, there are two “open_sbus” commands executed once in the first occurrence of time-slot 62. The first, on line 16, opens a static bus and allocates it time-slot 0 with a slot size of 1 and configures the bus to be permitted to send commands to targets 0x40, 0x50 and 0x60. The second “open_sbus” command, on line 17, opens a static bus and allocates it time-slot 63 with a slot size of 1 and configures the bus to be permitted to send commands to target 0x70. This is the target within the Network Manager dedicated to receiving Initiator 0's reported statistics and error information.

The command, on line 19, loads a static bus:

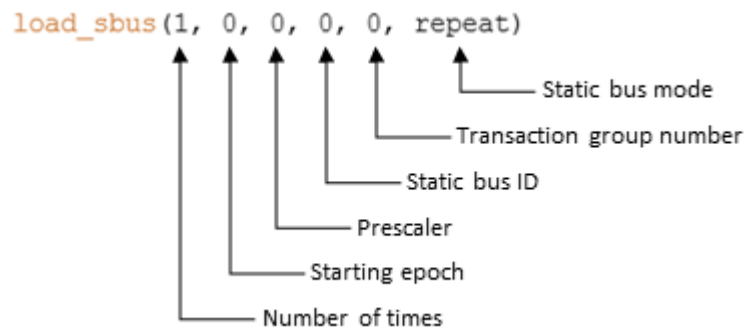


Figure 5-10: Load Static Bus Command

In this line, the command is called “load_sbust” and it contains a parenthesised parameter list containing six parameters. The first three parameters are command execution parameters as described above. The fourth parameter is the ID of the static bus, i.e. the first time-slot allocated to the static bus. The fifth parameter is the transaction group number, referencing a transaction group created using a “transaction_group” command. Finally, the sixth parameter is the static bus mode for the loaded transaction group, which may be “repeat” for a repeating transaction group or “single” for a single-shot transaction group.

In this example, there are two “load_sbust” commands. The first, on line 19, loads static bus 0 with transaction group 0 in a repeating mode. The second, on line 20, loads static bus 63 with transaction group 1, also in a repeating mode.

Example Script 2 – More Complex Static Bus Schedule

The next example script, listed in Figure 5-11, creates 13 transactions and assigns them to four transaction groups. In time-slot 62, the initiator is commanded to open five static buses and load them with the transaction groups.

```

1  // Transactions
2  transaction(0, 64, read, 32, 0x00000000, 32, 0x60000000)
3  transaction(1, 80, read, 32, 0x00000000, 32, 0x60000020)
4  transaction(2, 96, read, 32, 0x00000000, 32, 0x60000040)
5  transaction(3, 65, write, 32, 0x00000000, 8096, 0x60010000)
6  transaction(4, 81, write, 32, 0x00000000, 8096, 0x60012000)
7  transaction(5, 97, write, 32, 0x00000000, 8096, 0x60014000)
8  transaction(6, 66, write, 32, 0x00000000, 16384, 0x60020000)
9  transaction(7, 82, write, 32, 0x00000000, 16384, 0x60024000)
10 transaction(8, 98, write, 32, 0x00000000, 16384, 0x60028000)
11 transaction(9, 67, read, 32, 0x00000000, 256, 0x60030000)
12 transaction(10, 83, read, 32, 0x00000000, 256, 0x60030100)
13 transaction(11, 99, read, 32, 0x00000000, 256, 0x60030200)
14
15 // Statistics/errors transaction
16 transaction(12, 112, write, 32, 0x00000000, 2048, 0x8101B800)
17
18 // Transaction groups
19 transaction_group(0, (0, 1, 2))
20 transaction_group(1, (3, 4, 5))
21 transaction_group(2, (6, 7, 8))
22 transaction_group(3, (9, 10, 11))
23
24 // Statistics/errors transaction group
25 transaction_group(4, (12))
26
27 slot(62):
28     open_sbusr(1, 0, 0, 0, 1, (64, 80, 96))
29     open_sbusr(1, 0, 0, 16, 2, (65, 81, 97))
30     open_sbusr(1, 0, 0, 32, 4, (66, 82, 98))
31     open_sbusr(1, 0, 0, 48, 1, (67, 83, 99))
32     open_sbusr(1, 0, 0, 63, 1, (112))
33     load_sbusr(1, 0, 0, 0, 0, repeat)
34     load_sbusr(1, 0, 0, 16, 1, repeat)
35     load_sbusr(1, 0, 0, 32, 2, repeat)
36     load_sbusr(1, 0, 0, 48, 3, repeat)
37     load_sbusr(1, 1, 0, 63, 4, repeat)

```

Figure 5-11: Example Script 2 – More Complex Static Bus Schedule

In Figure 5-11, the script creates 13 transactions and 5 transaction groups in the preamble. The first 12 transactions, on lines 1 to 13, consist of RMAP write or read commands to targets on all of the target interface boards and with data lengths of 32 bytes, 256 bytes, 8 Kbytes, 16 Kbytes or 32 Kbytes. The 13th transaction, on line 16, contains the statistics and error information that will be reported to the Network Manager at the end of each schedule epoch. The first four transaction groups, on lines 19 to 22, each contain three transactions; one to a target on each of the three target

interface boards. For example, transaction group 0 contains transactions 0, 1 and 2 which are read transactions to target 64 on Target Interface 0, target 80 on Target Interface 1 and target 96 on Target Interface 2. The final command in the preamble, on line 25, creates the transaction group containing the statistics and error transaction.

In this example, there are 10 time-slot triggered commands, on lines 28 to 37, which are contained within the time-slot triggered command group for time-slot 62. These commands open five different static buses and load each of them with a different repeating transaction group.

The five static buses are opened in time-slots 0, 16, 32, 48 and 63 and each bus is permitted to execute transactions to three targets, one on each target interface board, except for static bus 63. This static bus is the statistics/error bus and executes transactions only with the Network Manager. Unlike the first example, where all buses had a length of 1, two of the static buses in this example are allocated multi-slots. Static bus 16 has a multi-slot with size 2, so it has been allocated time-slots 16 and 17. This means that static bus 16 can be loaded with a transaction group that has an execution time of up to two time-slots. Similarly, static bus 32 has a multi-slot with size 4, so it has been allocated time-slots 32 to 35 and can be loaded with a transaction group lasting up to 4 time-slots. The `open_sbus` commands are executed once in the first occurrence of time-slot 62 and then ignored for the remaining schedule epochs.

Each of the first four static buses are loaded with a transaction group containing three transactions, one to each of the permitted targets for each bus. The fifth, static bus 63, is loaded with a statistics/error information transaction as described in the previous example. Static bus 0 is loaded with a repeating transaction group containing three 32-byte read transactions to targets 64, 80 and 96. Static bus 16 is loaded with a repeating

transaction group containing three 8 Kbyte write transactions to targets 65, 81 and 97. Static bus 32 is loaded with a repeating transaction group containing three 16 Kbyte write transactions to targets 66, 82 and 98. Finally, static bus 48 is loaded with a repeating transaction group containing three 256 byte read transactions to targets 67, 83 and 99. The load_sbus commands are executed once, as they load repeating transaction groups, in the first occurrence of time-slot 62 and then ignored for the remaining schedule epochs.

Example Script 3 – Multiple Different Types of Buses

The last example script, listed in Figure 5-12, creates many transactions, transaction groups and packet bus operations. In time-slot 62, the initiator is commanded to open two static buses, one dynamic bus, one asynchronous bus and one packet bus. This is in addition to static bus 63 which is used to report statistics and error information to the network manager. During the execution of the script, the static buses are loaded once with a repeating transaction group, the dynamic bus is loaded repeatedly with three different transaction groups, the asynchronous bus is loaded every second with prioritised transactions and the packet bus is loaded every second with prioritised packet bus operations.

```

1  // Static bus transactions
2  transaction(0, 64, read, 32, 0x00000000, 1024, 0x60000000)
3  transaction(1, 65, read, 32, 0x00000000, 1024, 0x60000400)
4  transaction(2, 66, read, 32, 0x00000000, 1024, 0x60000800)
5  transaction(3, 67, write, 32, 0x00000000, 1024, 0x60000000)
6  transaction(4, 80, write, 32, 0x00000000, 1024, 0x60000400)
7  transaction(5, 81, write, 32, 0x00000000, 1024, 0x60000800)
8
9  // Dynamic bus transactions
10 transaction(6, 64, read, 32, 0x00010000, 1024, 0x60010000)
11 transaction(7, 65, read, 32, 0x00010000, 1024, 0x60010400)
12 transaction(8, 66, read, 32, 0x00010000, 1024, 0x60010800)
13 transaction(9, 67, write, 32, 0x00010000, 1024, 0x60010000)
14 transaction(10, 80, write, 32, 0x00010000, 1024, 0x60010400)
15 transaction(11, 81, write, 32, 0x00010000, 1024, 0x60010800)
16 transaction(12, 67, read, 32, 0x00010000, 1024, 0x60020000)
17 transaction(13, 80, read, 32, 0x00010000, 1024, 0x60020000)

```

```

18 transaction(14, 81, read, 32, 0x00010000, 1024, 0x60020000)
19
20 // Asynchronous bus transactions
21 transaction(15, 64, read, 32, 0x00020000, 1024, 0x60030000)
22 transaction(16, 65, read, 32, 0x00020000, 1024, 0x60030400)
23 transaction(17, 66, read, 32, 0x00020000, 1024, 0x60030800)
24 transaction(18, 67, write, 32, 0x00020000, 1024, 0x60030C00)
25 transaction(19, 80, write, 32, 0x00020000, 1024, 0x60031000)
26 transaction(20, 81, write, 32, 0x00020000, 1024, 0x60031400)
27
28 // Packet bus transactions
29 pbus_operation(0, 64, 0, 32, 1024, 0x60040000)
30 pbus_operation(1, 65, 0, 32, 1024, 0x60041000)
31 pbus_operation(2, 66, 0, 32, 1024, 0x60042000)
32 pbus_operation(3, 67, 0, 32, 1024, 0x60043000)
33 pbus_operation(4, 80, 0, 32, 1024, 0x60044000)
34 pbus_operation(5, 81, 0, 32, 1024, 0x60045000)
35
36 // Statistics/errors transaction
37 transaction(21, 112, write, 32, 0x00000000, 2048, 0x8101B800)
38
39 // Transaction groups
40 transaction_group(0, (0, 1, 2))
41 transaction_group(1, (3, 4, 5))
42 transaction_group(2, (6, 7, 8))
43 transaction_group(3, (9, 10, 11))
44 transaction_group(4, (12, 13, 14))
45
46 // Statistics/errors transaction group
47 transaction_group(5, (21))
48
49 slot(62):
50     open_sbus(1, 0, 0, 0, 1, (64,65,66,67,80,81))
51     open_sbus(1, 0, 0, 2, 1, (64,65,66,67,80,81))
52     open_dbus(1, 0, 0, (8, 10, 12), 1, (64,65,66,67,80,81))
53     open_abus(1, 0, 0, (16, 18, 20), 1, (64,65,66,67,80,81))
54     open_pbus(1, 0, 0, (32, 34, 36), 1, (64,65,66,67,80,81))
55
56     open_sbus(1, 0, 0, 63, 1, (112))
57
58     load_sbus(1, 0, 0, 0, 0, repeat)
59     load_sbus(1, 0, 0, 2, 1, repeat)
60
61     load_sbus(1, 1, 0, 63, 5, repeat)
62
63 slot(7):
64     load_dbus(255, 0, 0, 8, 2)
65
66 slot(9):
67     load_dbus(255, 0, 0, 8, 3)
68
69 slot(11):
70     load_dbus(255, 0, 0, 8, 4)
71
72 slot(14):
73     load_abus(255, 0, 9, 16, 15, 5)
74     load_abus(255, 0, 9, 16, 16, 3)
75     load_abus(255, 0, 9, 16, 17, 4)
76     load_abus(255, 0, 9, 16, 18, 2)
77     load_abus(255, 0, 9, 16, 19, 0)
78     load_abus(255, 0, 9, 16, 20, 1)

```

```

79
80 slot(30):
81     receive_pbus(255, 9, 9, 32, 0, 5)
82     receive_pbus(255, 9, 9, 32, 1, 3)
83     receive_pbus(255, 9, 9, 32, 2, 4)
84     send_pbus(255, 9, 9, 32, 3, 2)
85     send_pbus(255, 9, 9, 32, 4, 0)
86     send_pbus(255, 9, 9, 32, 5, 1)

```

Figure 5-12: Example Script 3 – Multiple Difference Types of Buses

In Figure 5-12, the script creates 22 transactions, 6 transaction groups and six packet bus operations in the preamble. The first 6 transactions, on lines 1 to 7, make up two transactions groups, defined on lines 40 and 41, which are loaded into the two static buses. The next 9 transactions, on lines 10 to 18, make up three transaction groups which are loaded into the dynamic bus. The next 6 transactions, on lines 15 to 20, are not contained within any transaction groups but instead, loaded into the asynchronous bus as single, prioritised transactions. The last transaction, on line 37, is the statistics and error transaction and it is assigned to the last transaction group on line 47.

On lines 29 to 34, the script creates 6 packet bus operations, which are used to load the packet bus with prioritised packet transfer requests between the initiator and a target's packet channel. The first packet bus operation is created on line 29:

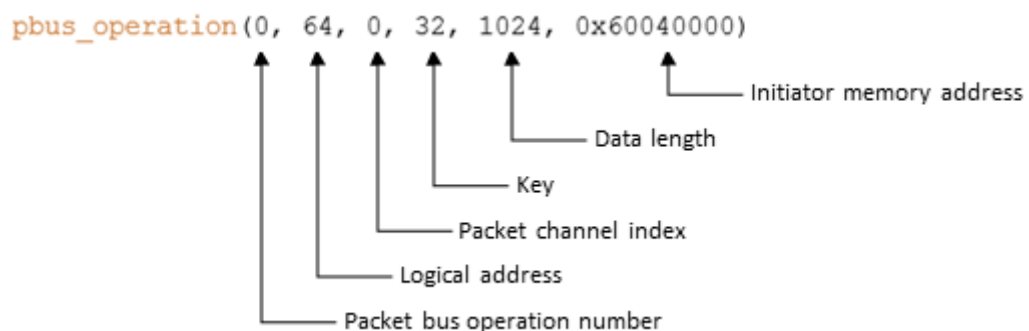


Figure 5-13: Packet Bus Operation Command

In this line, the command is called “pbus_operation” and it contains a parenthesised parameter list with six parameters. The first parameter is the packet bus operation

number, used in other commands to reference this specific packet bus operation. The second parameter is the logical address of the target which will receive or send a packet. The third parameter is the packet channel index, identifying a specific packet channel within a target. The fourth parameter is the RMAP key. The fifth parameter is the length, in bytes, of the buffer used to send or receive a packet. Finally, the sixth parameter is the initiator memory address i.e. the location of the buffer where the packet will be sent from or received to.

In this example, the 6 packet bus operations each have a 1 Kbyte buffer which is used by the initiator to send a packet to, or receive a packet from, packet channel 0 in targets 64, 65, 66, 67, 80 and 81 using an RMAP key with value 32.

Similar to the previous examples, there is a time-slot triggered command group for time-slot 62 that is used to open the virtual buses and load the static buses with repeating transaction groups. However, unlike the previous examples where this was the only time-slot triggered command group, this example contains multiple groups. Each group is used to load the different virtual buses at specific points throughout the execution of the schedule.

The static buses are opened on lines 50, 51 and 56 as described in the previous examples. Dynamic, asynchronous and packet buses are opened using the “open_dbus”, “open_abus” and “open_pbus” commands. The parameters of these commands are identical to the “open_sbus” command except for the fourth parameter. This parameter is a parenthesised list of allocated slot numbers rather than a single slot number. On lines 52, 53 and 54, a dynamic, asynchronous and packet bus is opened. The dynamic bus is allocated time-slots 8, 10 and 12; the asynchronous bus is allocated time-slots 16, 18 and 20; and the packet bus is allocated time-slots 32, 34 and 36. Each

of the buses have a slot length of 1 and is permitted, like all of the virtual buses in this example, to send commands to targets 64, 65, 66, 67, 80 and 81.

Following the time-triggered command group for time-slot 62 are three further commands groups for time-slots 7, 9 and 11. These command groups are used to load the dynamic bus allocated to time-slots 8, 10 and 12. The signature of the “load_dbus” command is identical to the “load_sbus” command except that there is no sixth parameter when loading a dynamic bus as the transaction group is implicitly a single-shot transaction group. In this example, dynamic bus 8 is repeatedly loaded with transaction groups 2, 3 and 4 in every occurrence of time-slots 7, 9 and 11, respectively.

Next, there are six “load_abus” commands assigned to time-slot 14 which are used to load the asynchronous bus allocated to time-slots 16, 18 and 20. The “load_abus” command is identical to the “load_sbus” command except for the fifth and sixth parameters. In place of a transaction group number, the “load_abus” command takes the number of an individual transaction which is loaded into the asynchronous bus with a priority level described in the sixth parameter. In this example, in every 10th schedule epoch, the asynchronous bus is loaded with transactions 15 to 20 with priority levels ranging from 0 to 5. These commands are assigned to time-slot 14 rather than 15 because asynchronous buses and packet buses are prepared in the time-slots prior to any assigned to them. This means that in order for the transactions to be considered in the preparation process, they must be loaded before the preparation time-slot begins.

Finally, there are three “receive_pbus” and three “send_pbus” commands assigned to time-slot 30. These commands are used to load a packet bus with prioritised requests to either receive a packet from, or send a packet to, a packet channel in a target. The

first four parameters in the signature of these commands are identical to the “load_sbus” command. The fifth parameter is the number of a packet bus operation, as defined in the script preamble. Finally, the sixth parameter is the priority level of the request. In this example, in every 10th schedule epoch after the first 10 epochs, the packet bus is loaded with requests to receive three packets using packet bus operations 0, 1 and 2, and send three packets using packet bus operations 3, 4 and 5. The priority level of the packet bus operations ranges from 0 to 5.

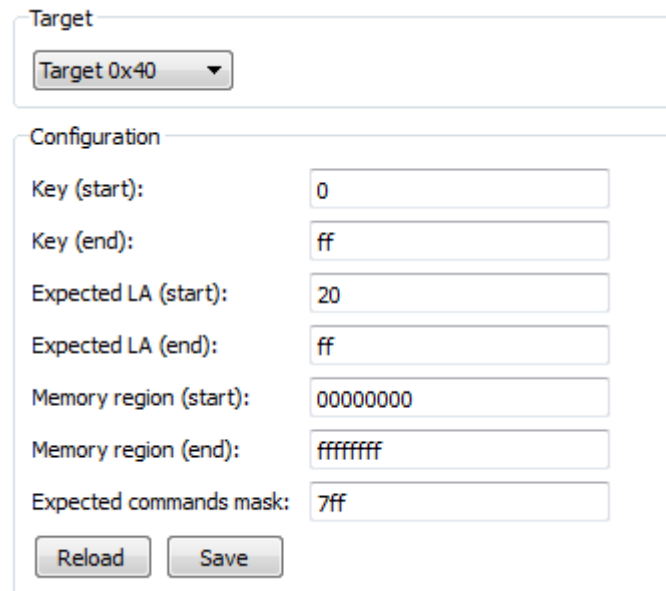
5.3 Targets

The targets are STAR-Dundee PXI RMAP interface boards (STAR-Dundee Ltd 2016) with a SpaceWire router containing four physical ports and four external ports, each connected to an individual RMAP target. The boards have 1 Gbyte of DDR3 memory which can be divided between the four targets as configured by the user. In the case of the SpaceWire-D Demonstrator, the targets are configured so that they each have access to 256 Mbytes of memory.

5.3.1 Command Authorisation

Each of the RMAP targets within the interface boards can be configured to operate in manual or automatic authorisation modes. The manual authorisation mode requires software to manually authorise or reject a command based on information stored in a series of registers accessible by the host application. To reduce the target response time i.e. the latency between a command being received and a reply being returned, the SpaceWire-D Demonstrator uses automatic authorisation mode for all targets. In this mode, the host application configures a series of registers with the permitted values for the command type, key, protocol ID, target logical address and memory address. The RMAP target hardware uses the values of these registers to automatically authorise or reject incoming commands.

The configuration registers are set to default values during the SpaceWire-D Demonstrator's initialisation process. A Target Configuration program was designed to allow a user to change the parameters as required. An image of the relevant section of the program is shown in Figure 5-14.



The screenshot shows a window titled "Target" with a dropdown menu set to "Target 0x40". Below this is a "Configuration" section with the following fields:

Key (start):	0
Key (end):	ff
Expected LA (start):	20
Expected LA (end):	ff
Memory region (start):	00000000
Memory region (end):	ffffffff
Expected commands mask:	7ff

At the bottom of the configuration section are two buttons: "Reload" and "Save".

Figure 5-14: Target Internal Authorisation Parameters

In Figure 5-14, the image shows that the configuration is done in three stages. First, the user selects the required target, by logical address, using the drop-down menu. Next, the required parameters are set using the text boxes. Lastly, the user uses the "Save" button to write the changes to the corresponding registers within the board's configuration address space.

5.3.2 Notifications

The target boards have the ability to notify a host application whenever certain events occur such as the execution of an RMAP command or a request for command authorisation. The notifications are sent as data structures contained within SpaceWire

packets to STAR-System channel 1 across the backplane. They can then be received by software using the STAR-System API (Mills and Parkes 2015).

Each notification contains the RMAP command header parameters as well as the value of the current time-code in the target board's router. The time-code can be used to determine in which time-slot the command was executed. In the SpaceWire-D Demonstrator, this information is extracted from the SpaceWire packets by the host PC's software so that it can be used to record and display the activity between the initiators and targets as described in Section 5.6.

5.4 Routers

The routers are STAR-Dundee PXI routers (STAR-Dundee Ltd 2016) with eight physical ports. They provide the network for the SpaceWire-D Demonstrator, allowing each initiator to be routed to each interface board without sharing any links.

5.5 Network Manager

The network manager is another STAR-Dundee PXI RMAP interface board (STAR-Dundee Ltd 2016) as described in Section 5.3. It is controlled by the host PC software to act as the time-code master for the SpaceWire-D network. It also receives statistics and error information reported by the initiators via RMAP write commands to two of the targets within the board.

Each initiator is assigned a separate RMAP target and memory address to write its statistics and error information into at the end of each schedule epoch. Initiators 0 and 1 are assigned address 0x00000000 within targets 0 and 1, respectively.

The host PC's Network Manager software is used to listen for RMAP event notifications coming from the board. The software parses the notifications and then

reads the statistics and error reports from address 0x00000000 in the corresponding target. The information is then parsed and displayed, as described in Section 5.6.

5.6 Host PC

The host PC is an ADLINK PXI-3950 system controller (ADLINK Technology Inc 2016) with an Intel Core2 Duo T7500 2.2 GHz processor and 4 Gbytes of 667 MHz DDR2 running the Windows 7 32-bit operating system. It is responsible for initialising the other devices within the SpaceWire-D Demonstrator and running a suite of Qt4.8 (The Qt Company Ltd 2016) based C++ applications used to configure and control the initiators, targets and network manager. It also displays the network activity reported to the network manager via RMAP commands by the initiators, and across the backplane by the targets.

5.6.1 Initiator Configuration

The Initiator Configuration program is used to configure and control each of the LEON2-FT processor boards acting as the initiators. It has the ability to read and write the network and target parameters, used by the initiators to calculate RMAP execution times. Different types of virtual buses can be created and assigned to the initiator's schedule. Automated test scripts can be parsed, compiled and uploaded. Finally, commands can be sent to the initiators to enable and disable the schedule and features like local-timer synchronisation. Figure 5-15 shows a screenshot of the Initiator Configuration program.

Target Address	Routers in Path	Slowest Link Speed (Mbit/s)	Response Time (us)	Packet Channels Address
0x40	3	200	2	ffffffc0
0x41	3	200	2	ffffffc0
0x42	3	200	2	ffffffc0
0x43	3	200	2	ffffffc0
0x44	0	0	0	0
0x45	0	0	0	0
0x46	0	0	0	0
0x47	0	0	0	0
0x48	0	0	0	0
0x49	0	0	0	0
0x4A	0	0	0	0
0x4B	0	0	0	0
0x4C	0	0	0	0
0x4D	0	0	0	0

Figure 5-15: Initiator Configuration Program

In Figure 5-15, the top left section allows the user to select which initiator they would like to configure, and provides buttons to read and write the schedule and upload the script. In the top right section, the user can send commands to the initiator to read in the script, enable and disable certain features. Below these two sections is a tab layout that contains seven separate tabs. The first tab contains fields to set the network parameters and a table to set the target parameters. Following this are four tabs, one for creating each type of virtual bus. Next is a tab that allows the user to allocate virtual buses to the schedule, if they are creating the schedule manually rather than using a script. Finally, there is a scripting tab which allows a user to open, parse and compile an automated test script file to be uploaded to the initiator.

5.6.2 Target Configuration

The Target Configuration program is used to configure and control each of the RMAP targets in the three PXI interface boards. It has the ability to read and write the automatic authorisation parameters. Packet channels can have their buffer locations and lengths configured. Data can be written to, and read from, the target memory. Finally, the target interface boards can be enabled as babbling nodes. Figure 5-16 shows a screenshot of the Target Configuration program.

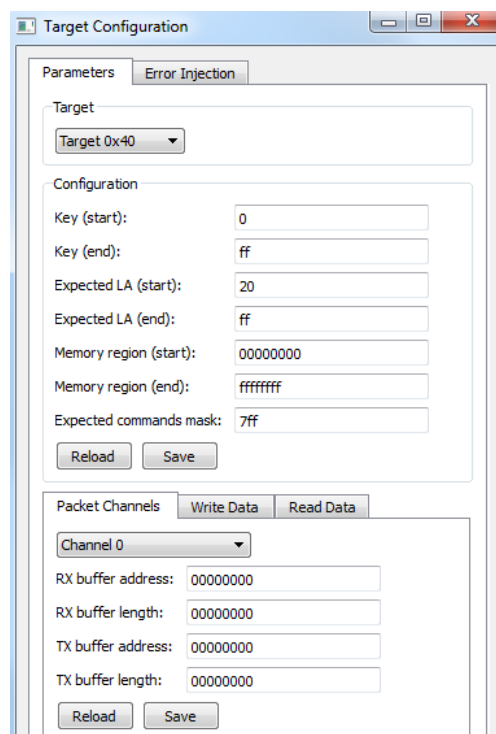


Figure 5-16: Target Configuration Program

In Figure 5-16, the top section allows the user to select which target they would like to configure. In the middle section, the automatic authorisation parameters can be set to define the valid key range, valid target logical address range, accessible memory region and permitted commands. In the bottom section is a tab layout with three separate tabs. The first tab contains a menu to select a packet channel and fields to set the location and length of the receive and transmit buffers used by the packet bus to

transfer packets between an initiator and the selected packet channel. The second and third tabs allow the user to write data to and read data from the target's memory. Finally, in the second main tab, the user can enable target interface boards as babbling nodes, which send out randomised transactions on the network.

5.6.3 Network Manager

The Network Manager program is used to configure the time-code master and receive and display statistics and error information reported to the network manager by the initiators. It has the ability to set the time-code rate and enable or disable the time-code master. Statistics reported by the initiator are displayed in a table, divided by type and time-slot, and errors are displayed in a list. Figure 5-17 shows a screenshot of the Network Manager program.

Time-Slot	Completed	Incomplete	RMAP Errors	Late Time-Codes	Early Time-Codes	Missing Time-Codes
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	1317	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0
11	0	0	0	0	0	0
12	0	0	0	0	0	0
13	0	0	0	0	0	0
14	0	0	0	0	0	0
15	0	0	0	0	0	0
16	0	0	0	0	0	0
17	0	0	0	0	0	0

Figure 5-17: Network Manager Program

In Figure 5-17, the top section allows the user to set the rate at which the network manager should broadcast time-codes and provides buttons to enable and disable the time-code master. The main left section, the statistics view, contains a table for each initiator. The table lists, for each time-slot, how many transactions have been

completed, how many were incomplete at the end of the time-slot, the number of errors, and the number of late, early and missing time-codes. The main right section, the error list view, contains a list for each initiator that provides the details of any errors that have occurred.

When the Network Manager program is initialised, it begins to listen for RMAP event notifications from the Network Manager target interface board. It does this by receiving SpaceWire packets on the backplane through STAR-System channel 1. When a notification is received, the software checks that the parameters of the RMAP command match those expected by an initiator statistics and error report. If so, the report is read from target memory and the statistics table for the relevant initiator is updated. In addition, any errors detected during the last schedule epoch are added to the initiator's error list.

5.6.4 Target Monitor

The Target Monitor program is used to display the network activity visually and statistically through a series of views. It has the ability to display activity in real-time, by updating a grid that shows if any of the targets were read from or written to during each time-slot. It shows the number of completed transactions, bytes read from and written to the target in total and per second, and it also breaks this information up for each time-slot. Finally, it shows a list of detailed information about all RMAP transactions taking place across all targets. Figure 5-18 shows a screenshot of the Target Monitor program.

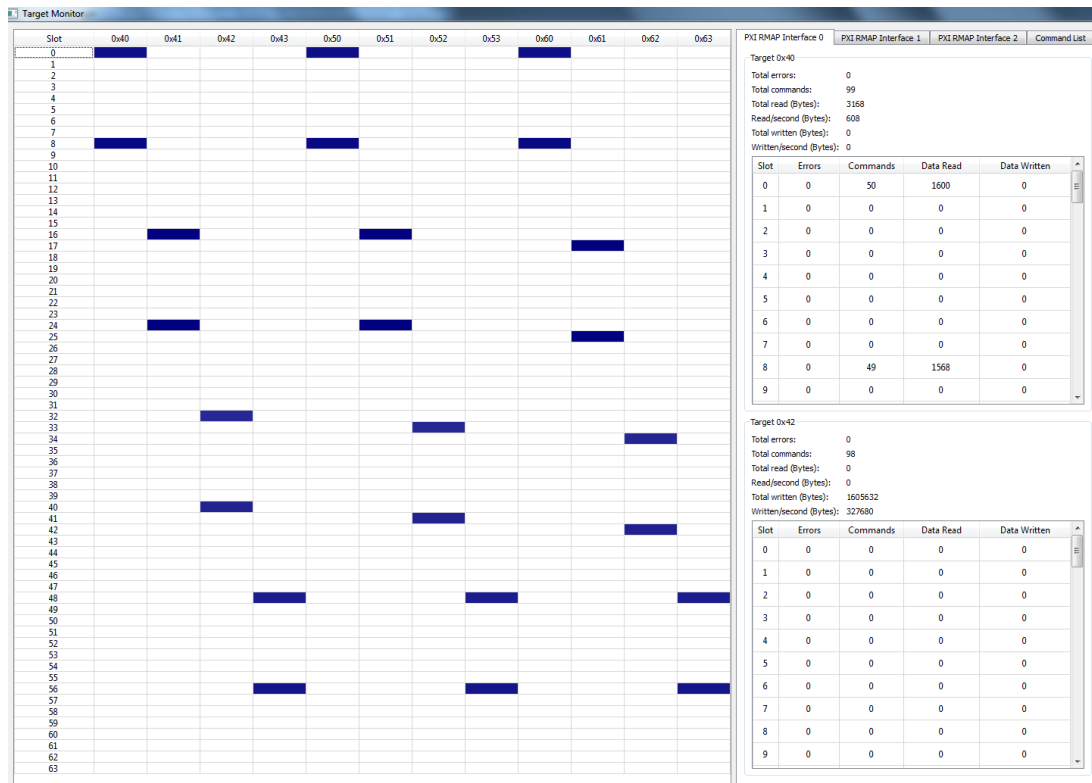


Figure 5-18: Target Monitor Program

In Figure 5-18, the Target Monitor program is divided, vertically, into two main sections. The left section contains a table, where each row represents a time-slot and each column represents an RMAP target. If a cell is shaded, it means that there was at least one RMAP command executed within that time-slot. The cell then fades over half a second, returning the cell to white. If another command is executed within the time-slot in a future schedule epoch before the cell is white, the colour is refreshed and the fade timer restarts. This view allows the user to see the network activity as it happens in a graphical manner. In the right section of the GUI, there is a tab layout with four individual tabs. The first three tabs represent a statistical view of the network activity for each of the RMAP interface boards. Each of these tabs are further divided into four sections; one for each individual RMAP target within the board. In order to see the interface more clearly, only two target sections are shown in Figure 5-18. The last tab displays a detailed list of commands that have been received by all targets.

Figure 5-19 shows an image of the left section, the schedule view, illustrating network activity within the first few rows of the table.

Slot	0x40	0x41	0x42	0x43	0x50	0x51	0x52	0x53	0x60	0x61	0x62	0x63
0												
1												
2												
3												
4												
5												
6												
7												
8												
9												

Figure 5-19: Target Monitor Schedule View

In Figure 5-19, the screenshot shows the first 10 rows of the schedule view, showing the network activity for time-slots 0 to 9 across targets 0x40-0x43, 0x50-0x53 and 0x60-0x63. The shaded cells show that there are six instances of network activity in this example. These cells, in time-slots 0 and 8, show that there was at least one RMAP command executed on targets 0x40, 0x50 and 0x60 within these time-slots.

After receiving feedback at the ESA SpaceWire-D project preliminary acceptance review, which took place at the University of Dundee on the 27th of April 2016, the Schedule View was updated to add two features to the GUI. Firstly, the shaded cells are coloured depending on the type of virtual bus that sent the RMAP commands. The virtual bus type is extracted from the transaction ID which contains the virtual bus ID and type. Secondly, the logical addresses of the initiators that sent the commands are displayed within each shaded cell. In order to make this clearer, the Target Monitor schedule view has been reproduced in black and white in Figure 5-20.

Slot	0x40	0x41	0x42	0x43	0x50	0x51
0	0x30	0x30	0x30	0x31	0x31	0x31
1						
2						
3						
4	0x30			0x31		
5		0x30			0x31	
6			0x30			0x31
7						
8	0x30	0x30	0x30	0x31	0x31	0x31
9						
10						
11						
12	0x30	0x30	0x30	0x31	0x31	0x31
13						
14	0x30	0x30	0x30	0x31	0x31	0x31
15						
16	0x30	0x30	0x30	0x31	0x31	0x31

Figure 5-20: Updated Target Monitor Schedule View

In Figure 5-20, the figure shows the schedule view of a SpaceWire-D network that contains two initiators. Each initiator is executing one static, dynamic, asynchronous and packet bus. The virtual buses in the first initiator are executing transactions with targets 0x40-0x42, taking up the first three target columns of the diagram. The second initiator's virtual buses are executing transactions with targets 0x43 and 0x50-0x51, shown on the next three target columns of the diagram. Each type of virtual bus is distinguished using a different shade of grey. There is one static bus executed by each initiator, shown as the lightest grey cells, and allocated to time-slot 0. The dynamic buses executed by each initiator, shown as the slightly darker grey cells, are allocated time-slots 4, 5 and 6. The asynchronous buses executed by each initiator, shown as dark grey cells, are allocated time-slot 8. Finally, the packet buses executed by each initiator, shown as black cells, are allocated time-slots 12, 14 and 16.

The target statistics view lists the number of errors, commands and bytes read/written in total, per second and divided by time-slot. Figure 5-21 shows an image of the target statistics view section for target 0x40.

Target 0x40

Total errors: 0
 Total commands: 99
 Total read (Bytes): 3168
 Read/second (Bytes): 608
 Total written (Bytes): 0
 Written/second (Bytes): 0

Slot	Errors	Commands	Data Read	Data Written
0	0	50	1600	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	49	1568	0
9	0	0	0	0

Figure 5-21: Target Monitor Target Statistics View

In Figure 5-21, the screenshot shows the target statistics view for target 0x40 during the execution of a schedule containing network activity in time-slots 0 and 8. The total and per second statistics are shown in the top section and the per time-slot statistics are shown in the scrollable table.

The final section of the Target Monitor program is the command list view, which displays a detailed description of every RMAP command received on all targets. An image of the command list view is shown in Figure 5-22.

PXI RMAP Interface 0		PXI RMAP Interface 1		PXI RMAP Interface 2		Command List			
0	0x60	0	0x30	0x0003	0x20	READ (I)	0x00000000	32	OK
0	0x40	0	0x30	0x0001	0x20	READ (I)	0x00000000	32	OK
0	0x50	0	0x30	0x0002	0x20	WRITE (IR)	0x00000000	32	OK
8	0x60	0	0x31	0x2003	0x20	READ (I)	0x00000000	32	OK
8	0x50	0	0x31	0x2002	0x20	WRITE (IR)	0x00000000	32	OK
8	0x40	0	0x31	0x2001	0x20	READ (I)	0x00000000	32	OK
16	0x41	1	0x30	0x4001	0x20	READ (I)	0x00000000	8096	OK
16	0x51	1	0x30	0x4002	0x20	WRITE (IR)	0x00000000	8096	OK
16	0x61	1	0x30	0x4003	0x20	READ (I)	0x00000000	8096	OK

Figure 5-22: Target Monitor Command List View

In Figure 5-22, the screenshot shows the start of the command list view during the execution of a schedule containing at least three static buses. The columns are: virtual bus ID, target logical address, target index, initiator logical address, transaction ID, RMAP key, command type, memory address, data length and status. In this case, there are nine transactions executed by static buses 0, 8 and 16. The first three, to targets 0x40, 0x50 and 0x60, and the last three, to targets 0x41, 0x51 and 0x61 are executed by initiator 0x30. The middle three, to targets 0x40, 0x50 and 0x60, are executed by static bus 8 in initiator 0x31.

The fifth column of data shows the transaction ID of each RMAP command, which encodes the virtual bus ID, virtual bus type and an 8-bit counter value. Looking at the counter value, the second byte in the ID, shows that the transactions are not listed in order for the first two groups of three transactions. This is due to the non-deterministic nature of the cPCI bus on which the RMAP notifications are sent by the RMAP interface boards and received by the Target Monitor program. Therefore, the Target Monitor program is used to display a real-time view of the network activity but in order to fully analyse the traffic, a device such as the STAR-Dundee Link Analyser Mk2 is required to capture the traffic flowing over the SpaceWire links.

5.7 Summary

This chapter has described the design of the SpaceWire-D Demonstrator which was created to facilitate the verification process of the SpaceWire-D standard and

demonstrate its features. The system consists of a PXI rack containing a host PC processor board, two LEON2-FT based processor boards acting as the initiators, three RMAP interface boards providing twelve RMAP targets, a network manager interface board and two 8-port routers.

A novel automated test scripting language and system was designed to simplify the execution of tests using the SpaceWire-D Demonstrator. The scripting system allows a user to describe RMAP transactions, transaction groups, packet bus operations and data buffers as a text file using a simple syntax. Time-slot triggered commands can then be used to open, load and close different types of virtual buses at specific times during the execution of the test. This system allows users to control the SpaceWire-D Demonstrator without knowledge of RTEMS or the embedded software and was used in all of the tests in the verification activity of the ESA SpaceWire-D project.

A suite of software programs was designed and created to run on the host PC to configure, control and monitor the other devices in the system. The Initiator Configuration program allows the user to set the network and target parameters, create virtual buses and assign them to the schedule, and upload automated test scripts. The Target Configuration program is used to set the RMAP authorisation parameters of each target, configure the packet channel buffers, and read and write to target memory. Statistics and errors reported to the network manager board are received and displayed by the Network Manager program. Finally, the Target Monitor program is used to visualise the network activity through a combination of a graphical network activity grid, a statistical view of each target and a list of RMAP commands received across all targets.

Chapter 6

Verification of the SpaceWire-D Demonstrator

In this chapter, each of the example automated test scripts described in Section 5.2.1 are executed on the SpaceWire-D Demonstrator. The results are captured using a combination of the Target Monitor program and a STAR-Dundee Link Analyser Mk2 device placed between Initiator 0 and Router 0. In each case, the Network Manager is broadcasting time-codes at a rate of approximately 640 Hz. Due to the microsecond resolution of the Network Manager's time-code master, this gives a time-slot duration of 1.562 ms and a schedule epoch of 999.68 ms.

These tests are intended to demonstrate that the SpaceWire-D protocol and Demonstrator is capable of running a network combining periodic traffic, aperiodic traffic, prioritised payload traffic and prioritised payload traffic with flow-control traffic. The first test shows the simplest SpaceWire-D network, with a single initiator executing periodic traffic. Next, the second test shows multiple initiators executing multiple instances of periodic traffic. Finally, the last test shows multiple initiators executing a combination of all types of traffic.

In addition to the tests described in this chapter, the SpaceWire-D Demonstrator was validated as part of the ESA SpaceWire-D project. This activity involved taking the initial validation goals listed in the SpaceWire-D Demonstrator Specification

Document (University of Dundee 2014) and designing a series of test scenarios, initiator schedules and transaction loading procedures used to meet the goals. The author detailed the validation activity and presented the results in the SpaceWire-D Validation Report (University of Dundee 2016 B), which was delivered to ESA. This activity showed that the SpaceWire-D Demonstrator was successful in satisfying the requirements of the project.

6.1 Running Example Script 1

In this test, there is a single initiator executing one static bus, in time-slot 0, in addition to the static bus used to report statistics and error information to the Network Manager at the end of each schedule epoch.

Slot	0x40	0x41	0x42	0x43	0x50	0x51	0x52	0x53	0x60	0x61	0x62	0x63
0												
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												
20												
21												
22												
23												
24												
25												
26												
27												
28												
29												
30												
31												

Figure 6-1: Example Script 1 – Target Monitor Schedule View

In Figure 6-1, the screenshot shows the Target Monitor schedule view during the execution of the script. This shows that in time-slot 0, there are three instances of network activity with targets 0x40, 0x50 and 0x60.

-1.61295 ms	1.56198 ms				TIMECODE [3F]		1.56198 ms
-50.930 µs	1.56202 ms	NULL		7.80996 ms	TIMECODE [00]		1.56202 ms
0 ns	50.930 µs	NCHAR [40]		50.930 µs	NULL		50.930 µs
40 ns	40 ns	NCHAR [01]		40 ns	NULL		40 ns
100 ns	60 ns	NCHAR [4C]		60 ns			

Figure 6-2: Example Script 1 – Time-Slot 0

In Figure 6-2, the Link Analyser screenshot shows the start of time-slot 0 as captured by the Link Analyser software. The time-slot is triggered by the arrival of time-code 0 at the initiator. After a short period of software overhead, about 51 µs in this case, to receive the time-code and perform some processing at the start of the time-slot, the initiator sends out the first RMAP command loaded into static bus 0.

0 ns		Header: RMAP Command			
0 ns		Target Address: 40			
100 ns	100 ns	Command: Read Command	100 ns		
100 ns		Acknowledge			
150 ns	60 ns	Key: 20	60 ns		
190 ns	40 ns	Initiator Address: 30	40 ns		
250 ns	60 ns	Transaction ID: 0001	60 ns		
340 ns	100 ns	Extended Address: 00	100 ns		
400 ns	60 ns	Address: 00000000	60 ns		
590 ns	190 ns	Data Length: 000020	190 ns		
740 ns	150 ns	Header CRC: A3	150 ns		
760 ns	20 ns	EOP	20 ns		
2.290 µs	1.520 µs			Header: RMAP Reply	
2.290 µs				Initiator Address: 30	
2.380 µs	100 ns			Command: Read Reply	100 ns
2.380 µs				Acknowledge	
2.420 µs	40 ns			Status: Success	40 ns
2.480 µs	60 ns			Target Address: 40	60 ns
2.530 µs	60 ns			Transaction ID: 0001	60 ns
2.690 µs	150 ns			Data Length: 000020	150 ns
2.820 µs	130 ns			Header CRC: 4D	130 ns
2.820 µs				Data CRC: 03	
2.880 µs	60 ns			Cargo Size: 32	60 ns
4.500 µs	1.620 µs			EOP	1.620 µs

Figure 6-3: Example Script 1 – Static Bus 0, Transaction 0

In Figure 6-3, the Link Analyser screenshot shows the first RMAP transaction executed by static bus 0 which is a command to read 32 bytes from address 0x00000000 in target 0x40 with a key value of 0x20 and a transaction ID of 0x0001.

12.270 µs	7.770 µs	Header: RMAP Command	11.500 µs		
12.270 µs		Target Address: 50			
12.360 µs	100 ns	Command: Write Command	100 ns		
12.360 µs		Data Not Verified			
12.360 µs		Acknowledge			
12.360 µs		Incrementing Address			
12.420 µs	60 ns	Key: 20	60 ns		
12.480 µs	60 ns	Initiator Address: 30	60 ns		
12.510 µs	40 ns	Transaction ID: 0002	40 ns		
12.610 µs	100 ns	Extended Address: 00	100 ns		
12.670 µs	60 ns	Address: 00000000	60 ns		
12.860 µs	190 ns	Data Length: 000020	190 ns		
13.010 µs	150 ns	Header CRC: 60	150 ns		
13.010 µs		Data CRC: 4C			
13.220 µs	210 ns	Cargo Size: 32	210 ns		
15.160 µs	1.940 µs	EOP	1.940 µs		
16.610 µs	1.450 µs			Header: RMAP Reply	12.110 µs
16.610 µs				Initiator Address: 30	
16.720 µs	110 ns			Command: Write Reply	110 ns
16.720 µs				Data Not Verified	
16.720 µs				Acknowledge	
16.720 µs				Incrementing Address	
16.760 µs	40 ns			Status: Success	40 ns
16.820 µs	60 ns			Target Address: 50	60 ns
16.860 µs	40 ns			Transaction ID: 0002	40 ns
16.970 µs	110 ns			Header CRC: 57	110 ns
16.990 µs	20 ns			EOP	20 ns

Figure 6-4: Example Script 1 – Static Bus 0, Transaction 1

In Figure 6-4, the Link Analyser screenshot shows the second RMAP transaction executed by static bus 0 which is a command to write 32 bytes to address 0x00000000 in target 0x50 with a key value of 0x20 and a transaction ID of 0x0002.

27.770 µs	10.780 µs	Header: RMAP Command	12.610 µs		
27.770 µs		Target Address: 60			
27.870 µs	100 ns	Command: Read Command	100 ns		
27.870 µs		Acknowledge			
27.900 µs	40 ns	Key: 20	40 ns		
27.960 µs	60 ns	Initiator Address: 30	60 ns		
28.020 µs	60 ns	Transaction ID: 0003	60 ns		
28.110 µs	100 ns	Extended Address: 00	100 ns		
28.170 µs	60 ns	Address: 00000000	60 ns		
28.360 µs	190 ns	Data Length: 000020	190 ns		
28.510 µs	150 ns	Header CRC: A0	150 ns		
28.530 µs	20 ns	EOP	20 ns		
29.960 µs	1.430 µs			Header: RMAP Reply	12.970 µs
29.960 µs				Initiator Address: 30	
30.060 µs	100 ns			Command: Read Reply	100 ns
30.060 µs				Acknowledge	
30.110 µs	60 ns			Status: Success	60 ns
30.170 µs	60 ns			Target Address: 60	60 ns
30.210 µs	40 ns			Transaction ID: 0003	40 ns
30.360 µs	150 ns			Data Length: 000020	150 ns
30.510 µs	150 ns			Header CRC: F0	150 ns
30.510 µs				Data CRC: C3	
30.570 µs	60 ns			Cargo Size: 32	60 ns
32.170 µs	1.600 µs			EOP	1.600 µs

Figure 6-5: Example Script 1 – Static Bus 0, Transaction 2

Finally, in Figure 6-5, the Link Analyser screenshot shows the third RMAP transaction executed by static bus 0 which is a command to read 32 bytes from address 0x00000000 in target 0x60 with a key value of 0x20 and a transaction ID of 0x0003.

In addition to these transactions being executed shortly after the arrival of time-code 0 at the initiator, the source of the commands can be further verified by looking at the

transaction IDs. In a SpaceWire-D transaction, the 16-bit transaction ID is divided into two bytes. The first byte contains an 8-bit counter and the second byte contains the virtual bus ID and the virtual bus type. The first two bits of the first byte indicate the virtual bus type, where 0x00 is a static bus, 0x01 is a dynamic bus, 0x10 is an asynchronous bus and 0x11 is a packet bus. The remaining six bits are the virtual bus ID i.e. the first time-slot of the first slot allocated to the virtual bus. In the three transactions captured in this example, the second byte of each transaction ID is 0x00. Therefore, the transactions were executed by a static bus with an ID of 0.

Initiator 0		Initiator 1				
Time-Slot	Completed	Incomplete	RMAP Errors	Late Time-Codes	Early Time-Codes	Missing Time-Codes
0	495	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0

Figure 6-6: Example Script 1 – Network Manager Statistics View

In Figure 6-6, the screenshot shows the Network Manager statistics view after a few seconds of executing the script. This shows that Initiator 0 is reporting that several RMAP transactions have been successfully completed in time-slot 0. In this screenshot, there has been 495 completed transactions reported so the initiator has been executing its schedule for 165 epochs.

This test has shown that the SpaceWire-D Demonstrator can run a SpaceWire-D network containing a single initiator which has a static bus loaded with multiple RMAP transactions. The transactions are sent to multiple targets in addition to the transaction sent by the static bus used for reporting statistics and error information to the network manager.

6.2 Running Example Script 2

In this test, there are two initiators executing four static buses each. The first initiator is executing its static buses in time-slots 0, 16, 32 and 48, as described in the example script listed in Figure 5-11. The second initiator is executing the same automated test script but its static buses are offset by eight time-slots so they are allocated to time-slots 8, 24, 40 and 56. Each of the initiators are executing their static buses in addition to the static buses used to report statistics and error information to the Network Manager at the end of each schedule epoch.



Figure 6-7: Example Script 2 – Target Monitor Schedule View

In Figure 6-7, the screenshot shows the Target Monitor schedule view during the execution of the script. In this figure, transaction groups that execute over multiple time-slots are surrounded by boxes. In time-slots 0 and 8, the initiators execute their first static buses as shown by the network activity to targets 0x40, 0x50 and 0x60. The second static buses, which are allocated a multi-slot with a length of 2, are executed in time-slots 16-17 for initiator 0 and 24-25 for initiator 1. These static buses execute their three 8 Kbyte transactions over two time-slots as they won't fit within a single time-slot in this case. The third static buses, which are allocated a multi-slot with a length of 4, are executed in time-slots 32-34 for initiator 0 and 40-42 for initiator 1. Again, these static buses execute their three 16 Kbyte transactions over three time-slots as they won't fit within a single time-slot. Finally, the last static buses are executed in time-slots 48 and 54 as they are loaded with small 256 byte transactions that fit within a single time-slot.

PXI RMAP Interface 0	PXI RMAP Interface 1	PXI RMAP Interface 2	Command List						
0	0x60	0	0x30	0x0003	0x20	READ (I)	0x00000000	32	OK
0	0x40	0	0x30	0x0001	0x20	READ (I)	0x00000000	32	OK
0	0x50	0	0x30	0x0002	0x20	WRITE (IR)	0x00000000	32	OK
8	0x60	0	0x31	0x2003	0x20	READ (I)	0x00000000	32	OK
8	0x50	0	0x31	0x2002	0x20	WRITE (IR)	0x00000000	32	OK
8	0x40	0	0x31	0x2001	0x20	READ (I)	0x00000000	32	OK
16	0x41	1	0x30	0x4001	0x20	READ (I)	0x00000000	8096	OK
16	0x51	1	0x30	0x4002	0x20	WRITE (IR)	0x00000000	8096	OK
16	0x61	1	0x30	0x4003	0x20	READ (I)	0x00000000	8096	OK
24	0x41	1	0x31	0x6001	0x20	READ (I)	0x00000000	8096	OK
24	0x51	1	0x31	0x6002	0x20	WRITE (IR)	0x00000000	8096	OK
24	0x61	1	0x31	0x6003	0x20	READ (I)	0x00000000	8096	OK
32	0x42	2	0x30	0x8001	0x20	READ (I)	0x00000000	16384	OK
32	0x52	2	0x30	0x8002	0x20	WRITE (IR)	0x00000000	16384	OK
32	0x62	2	0x30	0x8003	0x20	READ (I)	0x00000000	16384	OK
40	0x42	2	0x31	0xA001	0x20	READ (I)	0x00000000	16384	OK
40	0x52	2	0x31	0xA002	0x20	WRITE (IR)	0x00000000	16384	OK
40	0x62	2	0x31	0xA003	0x20	READ (I)	0x00000000	16384	OK
48	0x63	3	0x30	0xC003	0x20	READ (I)	0x00000000	256	OK
48	0x43	3	0x30	0xC001	0x20	READ (I)	0x00000000	256	OK
48	0x53	3	0x30	0xC002	0x20	WRITE (IR)	0x00000000	256	OK
56	0x63	3	0x31	0xE003	0x20	READ (I)	0x00000000	256	OK
56	0x53	3	0x31	0xE002	0x20	WRITE (IR)	0x00000000	256	OK
56	0x43	3	0x31	0xE001	0x20	READ (I)	0x00000000	256	OK

Figure 6-8: Example Script 2 – Target Monitor Command List View

In Figure 6-8, the screenshot shows the command list view for a single schedule epoch of example script 2. The transactions executed by static buses 0, 16, 32 and 48 are initiated by initiator 0x30, as listed in the fourth column. and those executed by static

buses 8, 24, 40 and 56 are initiated by initiator 0x31. This shows that the transactions match the automated test script listed in Figure 5-11.

During the execution of the script in which the results displayed in Figure 6-7 and Figure 6-8 were captured, the SpaceWire-D Demonstrator was running its SpaceWire network at 100 Mbit/s. However, shortly after this time, the network was changed to run at 200 Mbit/s to allow the maximum data-rate supported by the devices. The effect of this is that the transaction group executed by static bus 16 finishes its execution in a single time-slot rather than two, and the transaction group executed by static bus 32 finishes its execution in two time-slots rather than three. The full results for the execution of the schedule on initiator 0x30 with the network running at 200 Mbit/s can be found in Appendix 2.

The results in Appendix 2 show that the SpaceWire-D Demonstrator can run a SpaceWire-D network containing multiple initiators, each with many single or multi-slot static buses loaded with multiple RMAP transactions to different targets.

6.3 Running Example Script 3

In this test, there are two initiators executing five virtual buses each: two static buses, one dynamic bus, one asynchronous bus and one packet bus. The first initiator is executing its static buses in time-slots 0 and 2; its dynamic bus in time-slots 8, 10 and 12; its asynchronous bus in time-slot 16; and its packet bus in time-slots 32, 34 and 36, as described in the script listed in Figure 5-12. The second initiator is executing the same automated test script but using targets 0x52-0x53 and 0x60-0x63 instead of 0x40-0x43 and 0x50-0x51. Each of the initiators are executing their virtual buses in addition to the static buses used to report statistics and error information to the Network Manager at the end of each schedule epoch.

Slot	0x40	0x41	0x42	0x43	0x50	0x51	0x52	0x53	0x60	0x61	0x62	0x63
0												
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												
17												
18												
19												
20												
21												
22												
23												
24												
25												
26												
27												
28												
29												
30												
31												
32												
33												
34												
35												
36												
37												
38												

Figure 6-9: Example Script 3 – Target Monitor Schedule View

In Figure 6-9, the screenshot shows the Target Monitor schedule view during the execution of the script. In time-slots 0 and 2, the initiators execute their two static buses. In time-slots 8, 10 and 12, the initiators execute their dynamic bus which is repeatedly loaded with a transaction group. Once a second, each initiator loads its asynchronous bus with a group of prioritised transactions, which is executed in time-slot 16. Lastly, in time-slots 32, 34 and 36, the initiators execute their packet bus.

The network of the SpaceWire-D Demonstrator was designed so that there are non-conflicting paths between each initiator and the target interface boards. Therefore, because the initiators don't use any of the same targets within the same time-slot, they can operate in concurrent time-slots without packets being blocked in the routers.

In this example, the static buses operate similarly to the previous examples. The dynamic bus, with three allocated time-slots, is functionally similar to three individual

static buses that are loaded with single-shot transaction groups. Therefore, the remainder of this section will focus on the asynchronous and packet buses.

124.9279 ms	1.56202 ms	NULL	1.56202 ms	TIMECODE [0F]	1.56202 ms
126.48987 ms	1.56196 ms	NULL	1.56196 ms	TIMECODE [10]	1.56196 ms
126.53979 ms	49.920 µs	NCHAR [50]	49.920 µs		
126.53985 ms	60 ns	NCHAR [01]	60 ns	NULL	49.980 µs
126.5399 ms	60 ns	NCHAR [6C]	60 ns		

Figure 6-10: Example Script 3 – Time-Slot 16

In Figure 6-10, the Link Analyser screenshot shows the start of time-slot 16 which is triggered by the arrival of time-code 16 at the initiator.

126.53979 ms	6.09091 ms	Header: RMAP Command	6.2357 ms		
126.53979 ms		Target Address: 50			
126.5399 ms	110 ns	Command: Write Command	110 ns		
126.5399 ms		Data Not Verified			
126.5399 ms		Acknowledge			
126.5399 ms		Incrementing Address			
126.53994 ms	40 ns	Key: 20	40 ns		
126.540 ms	60 ns	Initiator Address: 30	60 ns		
126.54006 ms	60 ns	Transaction ID: 42EF	60 ns		
126.54015 ms	100 ns	Extended Address: 00	100 ns		
126.54019 ms	40 ns	Address: 00020000	40 ns		
126.5404 ms	210 ns	Data Length: 000400	210 ns		
126.54055 ms	150 ns	Header CRC: 01	150 ns		
126.54055 ms		Data CRC: B6			
126.54069 ms	130 ns	Cargo Size: 1024	130 ns		
126.60859 ms	67.900 µs	EOP	67.900 µs		
126.6101 ms	1.500 µs			Header: RMAP Reply	6.16122 ms
126.6101 ms				Initiator Address: 30	
126.61019 ms	100 ns			Command: Write Reply	100 ns
126.61019 ms				Data Not Verified	
126.61019 ms				Acknowledge	
126.61019 ms				Incrementing Address	
126.61025 ms	60 ns			Status: Success	60 ns
126.61029 ms	40 ns			Target Address: 50	40 ns
126.61034 ms	60 ns			Transaction ID: 42EF	60 ns
126.61044 ms	100 ns			Header CRC: E9	100 ns
126.61046 ms	20 ns			EOP	20 ns

Figure 6-11: Example Script 3 – Asynchronous Bus 16, Transaction 0

In Figure 6-11, the Link Analyser screenshot shows the first transaction executed by asynchronous bus 16 which is an RMAP write command to write 1 Kbyte to address 0x00020000 in target 0x50. This transaction was loaded into the asynchronous bus fifth but executed first as it has a priority level of 0.

126.61895 ms	8.500 µs	Header: RMAP Command	10.360 µs		
126.61895 ms		Target Address: 51			
126.61905 ms	100 ns	Command: Write Command	100 ns		
126.61905 ms		Data Not Verified			
126.61905 ms		Acknowledge			
126.61905 ms		Incrementing Address			
126.6191 ms	60 ns	Key: 20	60 ns		
126.61914 ms	40 ns	Initiator Address: 30	40 ns		
126.6192 ms	60 ns	Transaction ID: 42F0	60 ns		
126.6193 ms	100 ns	Extended Address: 00	100 ns		
126.61935 ms	60 ns	Address: 00020000	60 ns		
126.61954 ms	190 ns	Data Length: 000400	190 ns		
126.6197 ms	150 ns	Header CRC: 67	150 ns		
126.6197 ms		Data CRC: 57			
126.61987 ms	170 ns	Cargo Size: 1024	170 ns		
126.68787 ms	68.000 µs	EOP	68.000 µs		
126.68935 ms	1.490 µs			Header: RMAP Reply	78.900 µs
126.68935 ms				Initiator Address: 30	
126.68945 ms	100 ns			Command: Write Reply	100 ns
126.68945 ms				Data Not Verified	
126.68945 ms				Acknowledge	
126.68945 ms				Incrementing Address	
126.6895 ms	60 ns			Status: Success	60 ns
126.68956 ms	60 ns			Target Address: 51	60 ns
126.6896 ms	40 ns			Transaction ID: 42F0	40 ns
126.68971 ms	110 ns			Header CRC: 5E	110 ns
126.68973 ms	20 ns			EOP	20 ns

Figure 6-12: Example Script 3 – Asynchronous Bus 16, Transaction 1

In Figure 6-12, the Link Analyser screenshot shows the second transaction executed by asynchronous bus 16 which is an RMAP write command to write 1 Kbyte to address 0x00020000 in target 0x51. This transaction was loaded into the asynchronous bus sixth but executed second as it has a priority level of 1.

126.69823 ms	8.500 µs	Header: RMAP Command	10.360 µs		
126.69823 ms		Target Address: 43			
126.69832 ms	100 ns	Command: Write Command	100 ns		
126.69832 ms		Data Not Verified			
126.69832 ms		Acknowledge			
126.69832 ms		Incrementing Address			
126.69838 ms	60 ns	Key: 20	60 ns		
126.69842 ms	40 ns	Initiator Address: 30	40 ns		
126.69848 ms	60 ns	Transaction ID: 42F1	60 ns		
126.69857 ms	100 ns	Extended Address: 00	100 ns		
126.69863 ms	60 ns	Address: 00020000	60 ns		
126.69882 ms	190 ns	Data Length: 000400	190 ns		
126.69897 ms	150 ns	Header CRC: 47	150 ns		
126.69897 ms		Data CRC: 8D			
126.69914 ms	170 ns	Cargo Size: 1024	170 ns		
126.76709 ms	67.940 µs	EOP	67.940 µs		
126.76855 ms	1.470 µs			Header: RMAP Reply	78.820 µs
126.76855 ms				Initiator Address: 30	
126.76867 ms	110 ns			Command: Write Reply	110 ns
126.76867 ms				Data Not Verified	
126.76867 ms				Acknowledge	
126.76867 ms				Incrementing Address	
126.7687 ms	40 ns			Status: Success	40 ns
126.76876 ms	60 ns			Target Address: 43	60 ns
126.7688 ms	40 ns			Transaction ID: 42F1	40 ns
126.76891 ms	110 ns			Header CRC: 24	110 ns
126.76893 ms	20 ns			EOP	20 ns

Figure 6-13: Example Script 3 – Asynchronous Bus 16, Transaction 2

In Figure 6-13, the Link Analyser screenshot shows the third transaction executed by asynchronous bus 16 which is an RMAP write command to write 1 Kbyte to address

0x00020000 in target 0x43. This transaction was loaded into the asynchronous bus fourth but executed third as it has a priority level of 2.

126.77752 ms	8.590 µs	Header: RMAP Command	10.440 µs		
126.77752 ms		Target Address: 41			
126.77762 ms	100 ns	Command: Read Command	100 ns		
126.77762 ms		Acknowledge			
126.77768 ms	60 ns	Key: 20	60 ns		
126.77771 ms	40 ns	Initiator Address: 30	40 ns		
126.77777 ms	60 ns	Transaction ID: 42F2	60 ns		
126.77787 ms	100 ns	Extended Address: 00	100 ns		
126.77792 ms	60 ns	Address: 00020000	60 ns		
126.77811 ms	190 ns	Data Length: 000400	190 ns		
126.77827 ms	150 ns	Header CRC: 88	150 ns		
126.77829 ms	20 ns	BOP	20 ns		
126.77975 ms	1.470 µs			Header: RMAP Reply	10.820 µs
126.77975 ms				Initiator Address: 30	
126.77985 ms	100 ns			Command: Read Reply	100 ns
126.77985 ms				Acknowledge	
126.7799 ms	60 ns			Status: Success	60 ns
126.77996 ms	60 ns			Target Address: 41	60 ns
126.780 ms	40 ns			Transaction ID: 42F2	40 ns
126.78015 ms	150 ns			Data Length: 000400	150 ns
126.7803 ms	150 ns			Header CRC: 83	150 ns
126.7803 ms				Data CRC: EA	
126.78036 ms	60 ns			Cargo Size: 1024	60 ns
126.79063 ms	10.270 µs	Header: RMAP Command	12.340 µs		

Figure 6-14: Example Script 3 – Asynchronous Bus 16, Transaction 3

In Figure 6-14, the Link Analyser screenshot shows the fourth transaction executed by asynchronous bus 16 which is an RMAP read command to read 1 Kbyte from address 0x00020000 in target 0x41. This transaction was loaded into the asynchronous bus second but executed fourth as it has a priority level of 3.

126.79063 ms	10.270 µs	Header: RMAP Command	12.340 µs		
126.79063 ms		Target Address: 42			
126.79072 ms	100 ns	Command: Read Command	100 ns		
126.79072 ms		Acknowledge			
126.79078 ms	60 ns	Key: 20	60 ns		
126.79082 ms	40 ns	Initiator Address: 30	40 ns		
126.79088 ms	60 ns	Transaction ID: 42F3	60 ns		
126.79097 ms	100 ns	Extended Address: 00	100 ns		
126.79105 ms	80 ns	Address: 00020000	80 ns		
126.79124 ms	190 ns	Data Length: 000400	190 ns		
126.79141 ms	170 ns	Header CRC: E5	170 ns		
126.79143 ms	20 ns	BOP	20 ns		

Figure 6-15: Example Script 3 – Asynchronous Bus 16, Transaction 4

In Figure 6-15, the Link Analyser screenshot shows the fifth transaction executed by asynchronous bus 16 which is an RMAP read command to read 1 Kbyte from address 0x00020000 in target 0x42. This transaction was loaded into the asynchronous bus third but executed fifth as it has a priority level of 4.

126.80312 ms	11.700 µs	Header: RMAP Command	11.700 µs	
126.80312 ms		Target Address: 40		
126.80322 ms	100 ns	Command: Read Command	100 ns	
126.80322 ms		Acknowledge		
126.80328 ms	60 ns	Key: 20	60 ns	
126.80331 ms	40 ns	Initiator Address: 30	40 ns	
126.80337 ms	60 ns	Transaction ID: 42F4	60 ns	
126.80349 ms	110 ns	Extended Address: 00	110 ns	
126.80354 ms	60 ns	Address: 00020000	60 ns	
126.80373 ms	190 ns	Data Length: 000400	190 ns	
126.8039 ms	170 ns	Header CRC: E0	170 ns	
126.80392 ms	20 ns	EOP	20 ns	

Figure 6-16: Example Script 3 – Asynchronous Bus 16, Transaction 5

In Figure 6-16, the Link Analyser screenshot shows the sixth transaction executed by asynchronous bus 16 which is an RMAP read command to read 1 Kbyte from address 0x00020000 in target 0x40. This transaction was loaded into the asynchronous bus first but executed sixth as it has a priority level of 5.

As listed in Figure 5-12, the asynchronous bus transactions are loaded once a second within time-slot 14, in non-prioritised order. In time-slot 15, a transaction group is prepared by pulling transactions from the head of the prioritised queue for asynchronous bus 16. These transactions are then executed, in prioritised order, in time-slot 16.

35.88661 ms	1.56198 ms				TIMECODE [1F]	1.56198 ms
37.44859 ms	1.56198 ms				TIMECODE [20]	1.56198 ms
37.49783 ms	49.240 µs	NCHAR [50]		3.1732 ms	NULL	49.240 µs
37.49789 ms	60 ns	NCHAR [01]		60 ns	NULL	60 ns
37.49792 ms	40 ns	NCHAR [4C]		40 ns	NULL	40 ns

Figure 6-17: Example Script 3 – Time-Slot 32

In Figure 6-17, the Link Analyser screenshot shows the start of time-slot 32 which is triggered by the arrival of time-code 32 at the initiator. The transactions from time-slots 32, 34 and 36 were captured in the schedule epoch immediately after the Target Configuration program was used to configure packet channel 0 in target 0x40. The configuration indicated that there was a 1 Kbyte packet in a send packet buffer located at address 0x00000000. The targets have four packet channel data structures, each of which are 16 bytes long, located at address 0x0FFFFFFC0.

37.49783 ms	31.07945 ms	Header: RMAP Command	31.22396 ms		
37.49783 ms		Target Address: 50			
37.49792 ms	100 ns	Command: Read Command	100 ns		
37.49792 ms		Acknowledge			
37.49798 ms	60 ns	Key: 20	60 ns		
37.49804 ms	60 ns	Initiator Address: 30	60 ns		
37.49808 ms	40 ns	Transaction ID: 83B3	40 ns		
37.49819 ms	110 ns	Extended Address: 00	110 ns		
37.49823 ms	40 ns	Address: 0FFFFFFC0	40 ns		
37.49844 ms	210 ns	Data Length: 000010	210 ns		
37.49859 ms	150 ns	Header CRC: 05	150 ns		
37.49861 ms	20 ns	EOP	20 ns		
37.50008 ms	1.470 µs			Header: RMAP Reply	31.0817 ms
37.50008 ms				Initiator Address: 30	
37.50017 ms	100 ns			Command: Read Reply	100 ns
37.50017 ms				Acknowledge	
37.50023 ms	60 ns			Status: Success	60 ns
37.50029 ms	60 ns			Target Address: 50	60 ns
37.50032 ms	40 ns			Transaction ID: 83B3	40 ns
37.50048 ms	150 ns			Data Length: 000010	150 ns
37.50063 ms	150 ns			Header CRC: CF	150 ns
37.50063 ms				Data CRC: 00	
37.50069 ms	60 ns			Cargo Size: 16	60 ns
37.5015 ms	820 ns			EOP	820 ns

Figure 6-18: Example Script 3 – Packet Bus 32, Packet Channel Status Read 0

In Figure 6-18, the Link Analyser screenshot shows the first transaction executed by packet bus 32 which is an RMAP read command to read the 16-byte packet channel 0 status data structure located at address 0x0FFFFFFC0 in target 0x50. The packet transfer request related to this transaction was loaded into the packet bus fifth but executed first as it has a priority level of 0.

37.5089 ms	7.390 µs	Header: RMAP Command	10.290 µs		
37.5089 ms		Target Address: 51			
37.50899 ms	100 ns	Command: Read Command	100 ns		
37.50899 ms		Acknowledge			
37.50905 ms	60 ns	Key: 20	60 ns		
37.50909 ms	40 ns	Initiator Address: 30	40 ns		
37.50914 ms	60 ns	Transaction ID: 83B4	60 ns		
37.50924 ms	100 ns	Extended Address: 00	100 ns		
37.5093 ms	60 ns	Address: 0FFFFFFC0	60 ns		
37.50949 ms	190 ns	Data Length: 000010	190 ns		
37.50964 ms	150 ns	Header CRC: 41	150 ns		
37.50966 ms	20 ns	EOP	20 ns		
37.51112 ms	1.470 µs			Header: RMAP Reply	9.620 µs
37.51112 ms				Initiator Address: 30	
37.51122 ms	100 ns			Command: Read Reply	100 ns
37.51122 ms				Acknowledge	
37.51128 ms	60 ns			Status: Success	60 ns
37.51133 ms	60 ns			Target Address: 51	60 ns
37.51137 ms	40 ns			Transaction ID: 83B4	40 ns
37.51152 ms	150 ns			Data Length: 000010	150 ns
37.51168 ms	150 ns			Header CRC: 42	150 ns
37.51168 ms				Data CRC: 00	
37.51173 ms	60 ns			Cargo Size: 16	60 ns
37.51255 ms	820 ns			EOP	820 ns

Figure 6-19: Example Script 3 – Packet Bus 32, Packet Channel Status Read 1

In Figure 6-19, the Link Analyser screenshot shows the second transaction executed by packet bus 32 which is an RMAP read command to read the 16-byte packet channel 0 status data structure located at address 0x0FFFFFFC0 in target 0x51. The packet

transfer request related to this transaction was loaded into the packet bus sixth but executed second as it has a priority level of 1.

37.52232 ms	9.770 µs	Header: RMAP Command	12.670 µs		
37.52232 ms		Target Address: 43			
37.52242 ms	100 ns	Command: Read Command	100 ns		
37.52242 ms		Acknowledge			
37.52248 ms	60 ns	Key: 20	60 ns		
37.52251 ms	40 ns	Initiator Address: 30	40 ns		
37.52257 ms	60 ns	Transaction ID: 83B5	60 ns		
37.52267 ms	100 ns	Extended Address: 00	100 ns		
37.52272 ms	60 ns	Address: 0FFFFFFC0	60 ns		
37.52291 ms	190 ns	Data Length: 000010	190 ns		
37.52307 ms	150 ns	Header CRC: 61	150 ns		
37.52309 ms	20 ns	EOP	20 ns		
37.52455 ms	1.470 µs			Header: RMAP Reply	12.000 µs
37.52455 ms				Initiator Address: 30	
37.52465 ms	100 ns			Command: Read Reply	100 ns
37.52465 ms				Acknowledge	
37.5247 ms	60 ns			Status: Success	60 ns
37.52474 ms	40 ns			Target Address: 43	40 ns
37.5248 ms	60 ns			Transaction ID: 83B5	60 ns
37.52495 ms	150 ns			Data Length: 000010	150 ns
37.52509 ms	130 ns			Header CRC: 5E	130 ns
37.52509 ms				Data CRC: 00	
37.52514 ms	60 ns			Cargo Size: 16	60 ns
37.52596 ms	820 ns			EOP	820 ns

Figure 6-20: Example Script 3 – Packet Bus 32, Packet Channel Status Read 2

In Figure 6-20, the Link Analyser screenshot shows the third transaction executed by packet bus 32 which is an RMAP read command to read the 16-byte packet channel 0 status data structure located at address 0x0FFFFFFC0 in target 0x43. The packet transfer request related to this transaction was loaded into the packet bus fourth but executed third as it has a priority level of 2.

37.53328 ms	7.310 µs	Header: RMAP Command	10.190 µs		
37.53328 ms		Target Address: 41			
37.53337 ms	100 ns	Command: Read Command	100 ns		
37.53337 ms		Acknowledge			
37.53343 ms	60 ns	Key: 20	60 ns		
37.53347 ms	40 ns	Initiator Address: 30	40 ns		
37.53352 ms	60 ns	Transaction ID: 83B6	60 ns		
37.53362 ms	100 ns	Extended Address: 00	100 ns		
37.53368 ms	60 ns	Address: 0FFFFFFC0	60 ns		
37.53387 ms	190 ns	Data Length: 000010	190 ns		
37.53402 ms	150 ns	Header CRC: D4	150 ns		
37.53404 ms	20 ns	EOP	20 ns		
37.5355 ms	1.470 µs			Header: RMAP Reply	9.540 µs
37.5355 ms				Initiator Address: 30	
37.53562 ms	110 ns			Command: Read Reply	110 ns
37.53562 ms				Acknowledge	
37.53566 ms	40 ns			Status: Success	40 ns
37.53571 ms	60 ns			Target Address: 41	60 ns
37.53575 ms	40 ns			Transaction ID: 83B6	40 ns
37.5359 ms	150 ns			Data Length: 000010	150 ns
37.53606 ms	150 ns			Header CRC: 06	150 ns
37.53606 ms				Data CRC: 00	
37.53611 ms	60 ns			Cargo Size: 16	60 ns
37.53693 ms	820 ns			EOP	820 ns

Figure 6-21: Example Script 3 – Packet Bus 32, Packet Channel Status Read 3

In Figure 6-21, the Link Analyser screenshot shows the fourth transaction executed by packet bus 32 which is an RMAP read command to read the 16-byte packet channel 0 status data structure located at address 0x0FFFFFFC0 in target 0x41. The packet transfer request related to this transaction was loaded into the packet bus second but executed fourth as it has a priority level of 3.

37.54678 ms	9.850 µs	Header: RMAP Command	12.740 µs		
37.54678 ms		Target Address: 42			
37.54688 ms	100 ns	Command: Read Command	100 ns		
37.54688 ms		Acknowledge			
37.54693 ms	60 ns	Key: 20	60 ns		
37.54699 ms	60 ns	Initiator Address: 30	60 ns		
37.54703 ms	40 ns	Transaction ID: 83B7	40 ns		
37.54712 ms	100 ns	Extended Address: 00	100 ns		
37.54718 ms	60 ns	Address: 0FFFFFFC0	60 ns		
37.54739 ms	210 ns	Data Length: 000010	210 ns		
37.54752 ms	130 ns	Header CRC: B9	130 ns		
37.54756 ms	40 ns	EOP	40 ns		
37.54903 ms	1.470 µs			Header: RMAP Reply	12.100 µs
37.54903 ms				Initiator Address: 30	
37.54912 ms	100 ns			Command: Read Reply	100 ns
37.54912 ms				Acknowledge	
37.54918 ms	60 ns			Status: Success	60 ns
37.54924 ms	60 ns			Target Address: 42	60 ns
37.54928 ms	40 ns			Transaction ID: 83B7	40 ns
37.54943 ms	150 ns			Data Length: 000010	150 ns
37.54958 ms	150 ns			Header CRC: 1C	150 ns
37.54958 ms				Data CRC: 00	
37.54964 ms	60 ns			Cargo Size: 16	60 ns
37.55046 ms	820 ns			EOP	820 ns

Figure 6-22: Example Script 3 – Packet Bus 32, Packet Channel Status Read 4

In Figure 6-22, the Link Analyser screenshot shows the fifth transaction executed by packet bus 32 which is an RMAP read command to read the 16-byte packet channel 0 status data structure located at address 0x0FFFFFFC0 in target 0x42. The packet transfer request related to this transaction was loaded into the packet bus third but executed fifth as it has a priority level of 4.

37.5576 ms	7.140 µs	Header: RMAP Command	10.040 µs		
37.5576 ms		Target Address: 40			
37.5577 ms	100 ns	Command: Read Command	100 ns		
37.5577 ms		Acknowledge			
37.55775 ms	60 ns	Key: 20	60 ns		
37.55779 ms	40 ns	Initiator Address: 30	40 ns		
37.55785 ms	60 ns	Transaction ID: 83B8	60 ns		
37.55794 ms	100 ns	Extended Address: 00	100 ns		
37.558 ms	60 ns	Address: 0FFFFFFC0	60 ns		
37.55819 ms	190 ns	Data Length: 000010	190 ns		
37.55834 ms	150 ns	Header CRC: 1D	150 ns		
37.55836 ms	20 ns	EOP	20 ns		
37.55989 ms	1.520 µs			Header: RMAP Reply	9.430 µs
37.55989 ms				Initiator Address: 30	
37.55998 ms	100 ns			Command: Read Reply	100 ns
37.55998 ms				Acknowledge	
37.56002 ms	40 ns			Status: Success	40 ns
37.56008 ms	60 ns			Target Address: 40	60 ns
37.56013 ms	60 ns			Transaction ID: 83B8	60 ns
37.56029 ms	150 ns			Data Length: 000010	150 ns
37.56042 ms	130 ns			Header CRC: 2E	130 ns
37.56042 ms				Data CRC: 75	
37.56048 ms	60 ns			00 00 00 00 00 00 00 00	60 ns
37.56088 ms	400 ns			00 00 00 00 00 00 04 00	400 ns
37.5613 ms	420 ns			EOP	420 ns

Figure 6-23: Example Script 3 – Packet Bus 32, Packet Channel Status Read 5

In Figure 6-23, the Link Analyser screenshot shows the sixth transaction executed by packet bus 32 which is an RMAP read command to read the 16-byte packet channel 0 status data structure located at address 0x0FFFFFFC0 in target 0x40. The packet transfer request related to this transaction was loaded into the packet bus first but executed sixth as it has a priority level of 5.

The contents of the data section of the RMAP reply were expanded in this figure to show the values of the packet channel status data structure. The data structure contains four values in contiguous memory: the receive buffer memory address, the receive buffer length, the send buffer memory address and the send buffer length. In this example, the packet channel's status data structure indicates that there is no receive buffer ready, as its values are both 0x00000000, but there is a packet ready in the transmit buffer located at 0x00000000 with a length of 0x400 (1024) Bytes.

39.01055 ms	1.44926 ms				TIMECODE [21]	1.44926 ms
40.57257 ms	1.56202 ms				TIMECODE [22]	1.56202 ms
40.62145 ms	48.880 µs	NCHAR [50]		3.06015 ms		
40.6215 ms	60 ns	NCHAR [01]		60 ns	NULL	48.930 µs
40.62154 ms	40 ns	NCHAR [4C]		40 ns	NULL	40 ns

Figure 6-24: Example Script 3 – Time-Slot 34

In Figure 6-24, the Link Analyser screenshot shows the start of time-slot 34 which is triggered by the arrival of time-code 34 at the initiator. In this time-slot, all of the packet bus operations except the receive operation for target 0x40 remain in the first stage of the packet transfer process because the packet channel status data structures they read indicated that there was no relevant buffer ready for them. Therefore, they repeat their packet channel status read transactions in time-slot 32. However, the receive operation for target 0x40 is now in the second stage of the packet transfer process because the packet channel status read transaction indicated that there was a packet ready to send from the target to the initiator.

40.68109 ms	7.330 µs	Header: RMAP Command	10.230 µs		
40.68109 ms		Target Address: 40			
40.68118 ms	100 ns	Command: Read Command	100 ns		
40.68118 ms		Acknowledge			
40.68122 ms	40 ns	Key: 20	40 ns		
40.68128 ms	60 ns	Initiator Address: 30	60 ns		
40.68133 ms	60 ns	Transaction ID: 83BE	60 ns		
40.68143 ms	100 ns	Extended Address: 00	100 ns		
40.68149 ms	60 ns	Address: 00000000	60 ns		
40.68168 ms	190 ns	Data Length: 000400	190 ns		
40.68183 ms	150 ns	Header CRC: 14	150 ns		
40.68185 ms	20 ns	EOP	20 ns		
40.68331 ms	1.470 µs			Header: RMAP Reply	9.560 µs
40.68331 ms				Initiator Address: 30	
40.68341 ms	100 ns			Command: Read Reply	100 ns
40.68341 ms				Acknowledge	
40.68345 ms	40 ns			Status: Success	40 ns
40.6835 ms	60 ns			Target Address: 40	60 ns
40.68356 ms	60 ns			Transaction ID: 83BE	60 ns
40.68371 ms	150 ns			Data Length: 000400	150 ns
40.68385 ms	130 ns			Header CRC: 72	130 ns
40.68385 ms				Data CRC: 9E	
40.6839 ms	60 ns			Cargo Size: 1024	60 ns
40.73924 ms	55.330 µs			EOP	55.330 µs

Figure 6-25: Example Script 3 – Packet Segment Transfer

In Figure 6-25, the Link Analyser screenshot shows the transaction that is transferring a packet segment for the packet receive operation for target 0x40. This transaction is formed by using the packet channel data structure values received in the first stage of the packet transfer process. The operation is to receive a packet from the target, so it is implemented as an RMAP read command which is reading a 1 Kbyte packet from memory address 0x00000000 in target 0x40. As the packet is transferred in one segment, stage two of the packet transfer process is complete. If the packet was large

enough that it required multiple segments, the remaining segments would be transferred in one or more of the next allocated time-slots.

42.13455 ms	1.39531 ms	NULL		1.39533 ms	TIMECODE [23]		1.39531 ms
43.69653 ms	1.56198 ms				TIMECODE [24]		1.56198 ms
43.74724 ms	50.700 µs	NCHAR [50]		1.61269 ms			
43.7473 ms	60 ns	NCHAR [01]		60 ns	NULL		50.760 µs
43.74735 ms	60 ns	NCHAR [4C]		60 ns			

Figure 6-26: Example Script 3 – Time-Slot 36

In Figure 6-26, the Link Analyser screenshot shows the start of time-slot 36 which is triggered by the arrival of time-code 36 at the initiator. Similar to time-slot 34, five of the packet bus operations remain in stage one and the receive operation for target 0x40 moves on to the third and final stage of the packet transfer process.

43.80672 ms	7.390 µs	Header: RMAP Command	10.320 µs		
43.80672 ms		Target Address: 40			
43.80684 ms	110 ns	Command: Write Command	110 ns		
43.80684 ms		Data Not Verified			
43.80684 ms		Acknowledge			
43.80684 ms		Incrementing Address			
43.80688 ms	40 ns	Key: 20	40 ns		
43.80693 ms	60 ns	Initiator Address: 30	60 ns		
43.80699 ms	60 ns	Transaction ID: 83C4	60 ns		
43.80709 ms	100 ns	Extended Address: 00	100 ns		
43.80712 ms	40 ns	Address: 0FFFFFFC8	40 ns		
43.80733 ms	210 ns	Data Length: 000008	210 ns		
43.80749 ms	150 ns	Header CRC: 89	150 ns		
43.80749 ms		Data CRC: 00			
43.80762 ms	130 ns	00 00 00 00 00 00 00 00	130 ns		
43.80811 ms	500 ns	EOP	500 ns		
43.80947 ms	1.350 µs			Header: RMAP Reply	10.130 µs
43.80947 ms				Initiator Address: 30	
43.80958 ms	110 ns			Command: Write Reply	110 ns
43.80958 ms				Data Not Verified	
43.80958 ms				Acknowledge	
43.80958 ms				Incrementing Address	
43.80962 ms	40 ns			Status: Success	40 ns
43.80968 ms	60 ns			Target Address: 40	60 ns
43.80971 ms	40 ns			Transaction ID: 83C4	40 ns
43.80983 ms	110 ns			Header CRC: B6	110 ns
43.80985 ms	20 ns			EOP	20 ns

Figure 6-27: Example Script 3 – EOP Transaction

In Figure 6-27, the Link Analyser screenshot shows the transaction that completes the packet receive operation for target 0x40. This transaction uses the location of the packet channel status data structure in the target and clears the packet buffer values by writing zeroes to the relevant section of the data structure. After this transaction, the target application can read the packet channel data structure and see that it has been cleared and that it is ready for another packet to be sent.

6.4 Testing and Validation

As part of the ESA SpaceWire-D project, the author performed the validation activity which is detailed in the SpaceWire-D Validation Report (University of Dundee 2016 B). The activity consisted of taking the initial validation goals outlined in the SpaceWire-D Demonstrator Specification Document (University of Dundee 2014) and creating a series of test scenarios, initiator schedules and transaction loading procedures and gathering the results.

There were 11 test scenarios that were used to satisfy the validation requirements of the project. In addition to being reported in the SpaceWire-D Validation Report, the full validation and protocol verification test procedures were demonstrated at both the preliminary acceptance review and the final acceptance review of the SpaceWire-D ESA project and accepted successfully. The combination of the protocol verification activity and report; demonstrator validation activity and report; and the successful acceptance of the activities at the preliminary and final acceptance reviews was deemed extensive enough testing for the purposes of this project.

6.4.1 Test 1 - Single Static Bus

The aim of this test was to demonstrate that the SpaceWire-D Demonstrator could operate in a simple scenario, with a single initiator executing a single static bus involving several transactions. In this case, one of the initiators was configured to execute a repeating transaction group in static bus 0. In addition, in all of the tests described, each initiator executes a repeating transaction group in static bus 63 which is used to report statistics and error information to the network manager.

6.4.2 Test 2 – Multiple Static Buses

The aim of this test was to demonstrate that the SpaceWire-D Demonstrator could operate in a more advanced scenario, with both initiators executing multiple static buses involving several transactions. In this case, both initiators were configured to execute four repeating transaction groups in four separate static buses.

6.4.3 Test 3 - Multiple Different Buses

The aim of this test was to demonstrate that the SpaceWire-D Demonstrator could operate a network that uses multiple initiators and all types of virtual bus at the same time. For this test, both initiators execute two static buses, and one dynamic, asynchronous and packet bus each allocated multiple time-slots.

6.4.4 Test 4 – Slow Link

The aim of this test was to demonstrate the result of a link running at a rate too slow to handle the required traffic. In this case, one of the links was set to 10 Mbit/s instead of 200 Mbit/s and the initiators were configured with the same schedules and transaction loading procedures as in Test 3. This test showed that transactions that no longer fit within a time-slot due to the slow link were not executed because they failed the SpaceWire-D layer's execution time check. In addition, the asynchronous bus that initially executed its transactions in one slot used three slots instead.

6.4.5 Test 5 – Concurrent Slots

The aim of this test was to demonstrate that independent virtual buses can operate in concurrent slots without interfering with each other. In this case, both initiators were configured with the same transaction loading procedure as in Test 3 but allocated the exact same time-slots. This was a valid schedule because the virtual buses used by each initiator did not share links.

6.4.6 Test 6 – Common Link

The aim of this test was to demonstrate that virtual buses that share a link can over-utilise the link, causing packets to be blocked. In this case, both initiators were configured to share a link which resulted in transaction incomplete errors being reported to the network manager for the blocked transactions.

6.4.7 Test 7 – Multi-Slots

The aim of this test was to demonstrate that transaction groups could be executed over multiple consecutive time-slots in a multi-slot virtual bus. In this case, one of the initiators was configured to execute a static bus containing a multi-slot consisting of four consecutive time-slots.

6.4.8 Test 8 – Changing Schedules

The aim of this test was to demonstrate that the initiators could be commanded by a network manager to change their schedules. In this case, the network manager was used to toggle the schedules for both initiators between two different configurations.

6.4.9 Test 9 – Start and Stop Schedules

The aim of this test was to demonstrate that the initiators could be commanded by a network manager to start and stop the execution of their schedules. In this case, the network manager was used to start and stop the schedules for both initiators at different times.

6.4.10 Test 10 – Reset Initiator

The aim of this test was to demonstrate that the initiators could be reset during operation and recover successfully. In this case, the network manager was used to reset the initiators and capture the reset process.

6.4.11 Test 11 – Error Injection

The aim of this test was to demonstrate that the initiators could detect errors and report them to a network manager at the end of each schedule epoch. In this case, link disconnect errors were injected every 150 μ s which caused command and reply packets to be dropped, resulting in errors being detected and reported by the initiators.

6.5 Summary

In this chapter, three experiments were run using the automated test scripting system of the SpaceWire-D Demonstrator. Firstly, a simple test was run involving one initiator executing one static bus in addition to the static bus used to report statistics and errors at the end of each schedule epoch. Next, a more advanced test was run using two initiators each executing multiple static buses. Finally, the last test involved both initiators executing a schedule containing two static buses and one dynamic, asynchronous and packet bus. This demonstrated the prioritised execution of transactions in an asynchronous bus and the packet transfer process in a packet bus.

The results in this chapter demonstrate that the SpaceWire-D protocol was successfully operated in a real system using the RTEMS based SpaceWire-D software layer designed in Chapter 4. In addition, the final experiment showed that SpaceWire-D can be used to mix periodic repeating traffic, aperiodic traffic, prioritised traffic and prioritised traffic with end-to-end flow-control on a single network. Furthermore, after the prototype system was validated and used to complete the verification activity of the ESA SpaceWire-D project, it was delivered to ESA and installed at ESTEC.

Chapter 7

Scheduling SpaceWire-D Networks

The previous chapters have described the features of SpaceWire-D and the design and development of an efficient SpaceWire-D software layer and a SpaceWire-D Demonstrator. This chapter explores how SpaceWire-D schedules can be generated to allow the combination of payload and control traffic on a single network in order to satisfy the networking requirements of a mission.

To do this, the SpaceWire-D scheduling problem is first formalised by specifying its inputs and outputs. Following this, a two-stage scheduling strategy is described. In the first stage, paths between initiators and targets are selected. In the second stage, RMAP transactions are allocated using a combination of simple first-fit algorithms for periodic and aperiodic traffic, and a payload-data scheduling algorithm where the problem is modelled as a variation on the classic bin-packing problem. In the next chapter, the algorithm is evaluated using a number of randomised test cases and a case study of a real mission.

7.1 Problem Specification

In this section the SpaceWire-D scheduling problem is formalised by describing the different kinds of bandwidth requirements on a SpaceWire-D network, the network topology format, the network parameters required to calculate RMAP execution times and the format of a solution.

7.1.1 Bandwidth Requirements

The bandwidth requirements of a SpaceWire-D network are a list of each type of data-flow that takes place between the network's initiators and targets, where each entry is categorised into one of three classes: periodic, aperiodic or payload data.

7.1.1.1 Periodic Traffic

The periodic traffic class describes bandwidth requirements that are repeating transactions between initiators and targets at a set rate. For example, an OBC may read 32 bytes of housekeeping information from a star tracker 16 times a second. A static bus could be used to fulfil this bandwidth requirement as it repeats a transaction group at the same time during each schedule epoch. For example, the OBC could open a static bus in time-slot 0 with the star tracker as one of its targets. Using a time-slot frequency of 1024 Hz, giving a 16 Hz schedule epoch cycle, loading this static bus with the housekeeping transaction satisfies the periodic bandwidth requirement.

A periodic bandwidth requirement can be notated as $P_r = (i, t, o, s, r)$ where i is the initiator, t is the target, o is the type of RMAP operation, s is the size of the transaction in bytes and r is the rate in Hz i.e. how many times the RMAP operation must be executed per second. For example, $P_r^i = (OBC, STR1, read, 32, 16)$ lists the i^{th} periodic bandwidth requirement which is between the OBC initiator and a star tracker target named STR1 and consists of an RMAP read transaction of 32 bytes with a rate of 16 Hz.

7.1.1.2 Aperiodic Traffic

An aperiodic bandwidth requirement is a transaction between an initiator and a target that may require execution at any point in time and must complete execution within a deadline. For example, an OBC may want to send a command to an instrument

instructing it to capture some data after an event has occurred, with a maximum latency of 10 ms. In order to fulfil this bandwidth requirement, a dynamic bus could be used with slots allocated at intervals less than the maximum latency so that whenever a transaction is required, there is a time-slot available within the deadline. In this case, the OBC could open a dynamic bus in time-slots 0, 10, 20, 30, 40, 50 and 60 with the instrument as one of its targets. Using a time-slot frequency of 1024 Hz, giving a time-slot duration of 976.5625 μ s, each allocated time-slot is less than 10 ms apart.

Aperiodic bandwidth requirements can be notated as $A_r = (i, t, o, s, d)$ where i, t, o and s are the same as the parameters in the periodic bandwidth requirement and d is the deadline in ms. As an example, $A_r^i = (OBC, INSTR1, write, 64, 10)$ lists the i^{th} aperiodic bandwidth requirement which is between the OBC initiator and an instrument target named INSTR1 and consists of an RMAP write transaction of 64 bytes which must be completed before a 10 ms deadline.

7.1.1.3 Payload Data Traffic

The payload data traffic class specifies a bandwidth requirement as a data-rate required between an initiator and a target. For example, an instrument might generate 1 Mbyte/s of payload data in 4 Kbyte packets, giving 256 packets per second, which need to be stored in a solid-state mass-memory (SSMM) device. An asynchronous bus with enough time-slots allocated to transfer the required number of packets can be used to fulfil this requirement, for example, the instrument could open an asynchronous bus and allocate it time-slots 0, 16, 32 and 48 with the SSMM as one of its targets. Using a schedule epoch cycle of 16 Hz, the instrument could send 4 packets in each time-slot, resulting in $4 * 4 * 16 = 256$ packets per second, satisfying the bandwidth requirement.

A payload data bandwidth requirement can be notated as $D_r = (i, t, o, s, n)$ where i, t, o and s are the same as the parameters in the periodic and aperiodic bandwidth requirements and n is the maximum number of packets that may be sent per second. For example, $D_r^i = (INSTR1, SSMM, write, 4096, 256)$ lists the i^{th} payload data bandwidth requirement which is between an instrument initiator named INSTR1 and the SSMM target and consists of 256 RMAP write transactions of 4096 bytes a second with an overall data-rate requirement of 1 Mbyte/s. The data-rate requirement does not include the RMAP protocol overhead, which must be taken into account by the scheduling algorithm.

7.1.2 Network Topology

The network topology can be defined as a list of bidirectional edges, representing SpaceWire links between SpaceWire devices. There may be more than one SpaceWire link between two routers so the edges can appear multiple times in the list, forming a multigraph.

7.1.3 Network Parameters

To accurately calculate the execution time of RMAP transactions and check if transaction groups fit within a slot, there are a number of parameters that must be defined at the initiator, target and system levels. These parameters and the RMAP execution time formula were introduced in (Parkes, Ferrer, et al. 2010) and also used in a simulated annealing approach to scheduling for an early version of SpaceWire-D (Chen, et al. 2013).

The initiator parameters are listed in Table 7-1.

Table 7-1: Initiator Parameters

Parameter	Symbol	Unit	Description
Processing time	I_p	μs	Latency between an initiator receiving a time-code and the first byte of the first RMAP command leaving the initiator.
Post-processing time	I_r	μs	Latency between an initiator receiving an RMAP reply and the first byte of the next RMAP command leaving the initiator in the worst-case scenario where all RMAP commands are non-posted.

The target parameters, which are defined on a per initiator basis because paths between different initiators and targets will differ, are listed in Table 7-2. Note that if the routing tables are selected dynamically during the execution of the scheduling algorithm, the T_n and T_l parameters will be calculated then, rather than pre-defined, as they are dependent on the paths between the initiators and targets.

Table 7-2: Target Parameters

Parameter	Symbol	Unit	Description
Response time	T_r	μs	Latency between a target receiving an RMAP command and the first byte of the corresponding RMAP reply leaving the target.
Routers in path	T_n	Integer	Number of routers in the path between the initiator and the target.
Slowest link speed	T_l	Mbit/s	Slowest link speed in the path between the initiator and the target.

The system parameters are listed in Table 7-3.

Table 7-3: System Parameters

Parameter	Symbol	Unit	Description
Switching time	S_w	μs	Packet switching latency.
Time-slot duration	S_t	μs	Expected interval between time-codes being sent by the time-code master.

Using the parameters listed in the previous tables and the size and type of an RMAP transaction, the worst-case execution time (WCET) of transactions can be calculated.

7.1.4 RMAP Execution Time

To calculate the execution time of a single RMAP transaction, the round-trip time from the command being sent by the initiator, processed by the target and the reply returned and processed by the initiator is measured. In order to calculate the network propagation time, the size of the RMAP command and reply packets must be known, which differ depending on the type of operation. Table 7-4 shows the functions used to calculate the size of a command and reply packet for write, read and read-modify-write RMAP operations.

Table 7-4: RMAP Command and Reply Packet Sizes

RMAP Operation	Command (Bytes)	Reply (Bytes)
Write	$size(C_i) = D_i + 17$	$size(R_i) = 8$
Read	$size(C_i) = 16$	$size(R_i) = D_i + 13$
Read-modify-write	$size(C_i) = 2D_i + 17$	$size(R_i) = D_i + 13$

In Table 7-4, a function is defined, named *size*, that takes either a command C_i , meaning the command data structure for the i^{th} RMAP transaction, or a reply R_i and

returns the size of the packet in bytes, which differs depending on the type of operation and D_i , the length of the data section of the transaction.

Using the network parameters defined in Section 7.1.3, the functions defined in Table 7-4 and the formula introduced in (Parkes, Ferrer, et al. 2010), the WCET of an RMAP transaction can be calculated as follows:

$$W_t(N_i) = 10 \frac{\text{size}(C_i) + \text{size}(R_i)}{1000000T_l^{i,j}} + T_n^{i,j}S_w + T_r^{i,j} + I_r^i$$

In the above formula, a function is defined, named W_t , that takes a transaction N_i and returns the WCET measured in microseconds. The target parameter terms, with a superscript of i,j , mean to use the target parameters of initiator i and the relevant target j for the transaction. The initiator parameter, with a superscript of i , means to use the relevant initiator i . The command and reply packet sizes, measured in bytes, are multiplied by 10 because SpaceWire's character level defines a data character as a 10-bit character containing a parity bit, data-control flag and a byte of data (ECSS 2008 A).

Using the function W_t , the WCET of a transaction group can be calculated as follows:

$$W_g(G) = I_p^i + \sum_{i=0}^{n-1} W_t(G_i)$$

In the above formula, a function is defined, named W_g , that takes a transaction group G containing n transactions and returns the WCET of the group in microseconds by summing the WCET of each individual transaction and adding the IPT. The function

takes into account the worst-case scenario where each transaction is non-posted i.e. each transaction is fully executed before the next transaction begins.

7.1.5 Solution Format

The output from the SpaceWire-D network scheduling algorithm should describe everything required to build a SpaceWire network and implement a schedule that satisfies the bandwidth requirements of a mission.

Firstly, if the routing tables are not already pre-determined by the system engineer, they are selected by the algorithm during the path selection phase. Secondly, for the periodic and aperiodic bandwidth requirements, a list of allocated slots for each requirement needs to be output. Lastly, for payload data requirements, a list of allocated slots for each requirement needs to be output as well as a number to indicate how many transactions are allocated in each slot.

7.2 Solving the Problem

The SpaceWire-D scheduling problem can be solved in two stages. Firstly, given the network topology and required initiator/target pairs, the paths between the initiators and the targets need selected. Secondly, the periodic, aperiodic and payload data bandwidth requirements need their transactions allocated to time-slots until the requirements are satisfied.

An overview of the solving of the problem is illustrated in Figure 7-1.

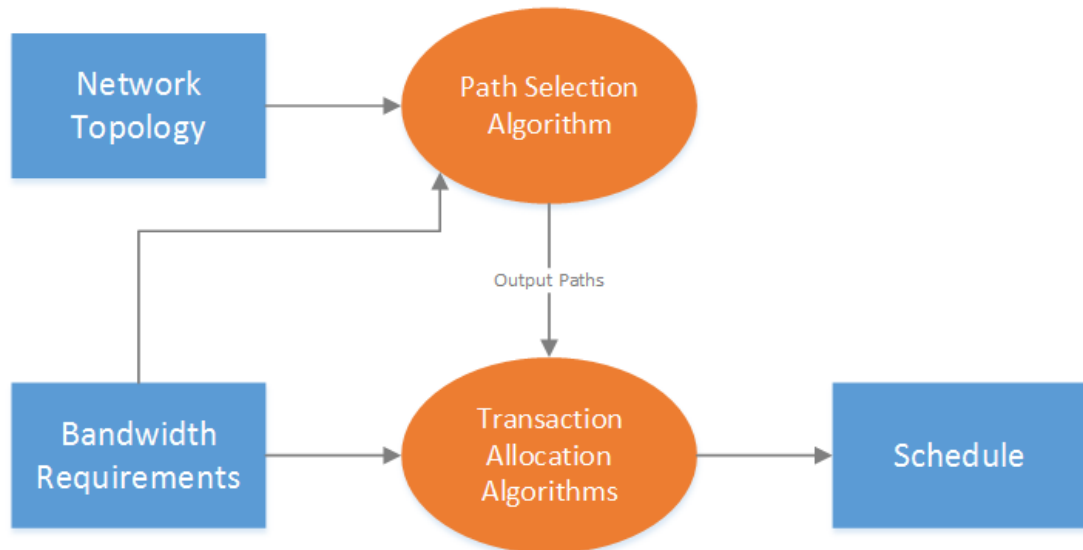


Figure 7-1: Problem Solving Overview

As shown in Figure 7-1, there are two stages to the problem solving. Firstly, the network topology and the bandwidth requirements are input into the path selection algorithm. This results in a set of initiator to target paths that is then passed to the transaction allocation algorithms along with the bandwidth requirements. These algorithms then attempt to generate the initiator schedules which satisfy the problem specification.

Therefore, there are two questions that must be answered when solving the SpaceWire-D scheduling problem:

1. How should the paths between the initiators and the targets be selected?
2. How can transactions be scheduled in order to satisfy the bandwidth requirements of the mission?

The following sections describe methods to answer these questions.

7.2.1 Selecting Paths

The simplest way to build the network's routing tables is for the system engineer to hard-code them. However, if they are not pre-determined, the scheduling algorithm can select the paths using a heuristic to attempt to improve the available bandwidth by reducing the number of potential collisions between transactions.

7.2.1.1 Breadth-First Search

The breadth-first search (BFS) algorithm (Moore 1959) (Lee 1961) can be used to select the shortest path, measured in the number of SpaceWire links, between an initiator and a target. This algorithm was investigated first because it is simple and the shortest paths will result in the minimum latency, assuming the link speed is uniform across the network. However, in a SpaceWire-D network, this is not always the best choice because it may result in collisions that could otherwise be avoided. An example topology in which this would be the case is illustrated in Figure 7-2.

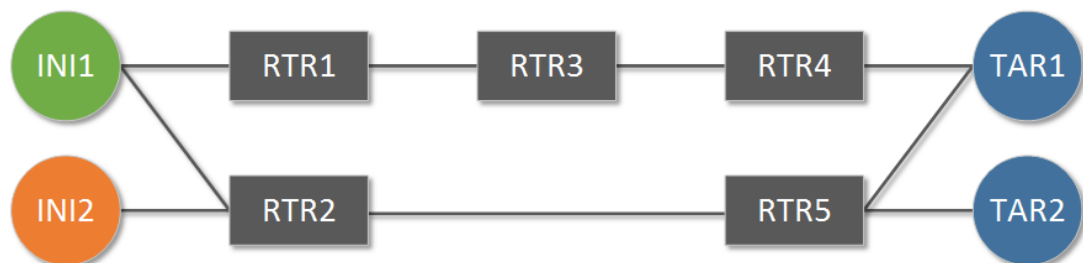


Figure 7-2: Network Topology with Potential Collisions

In Figure 7-2, there are two initiators and two targets, with a SpaceWire network in between them. Consider two bandwidth requirements, the first between INI1 and TAR1, and the second between INI2 and TAR2. The shortest path between these pairs is shown in Figure 7-3.

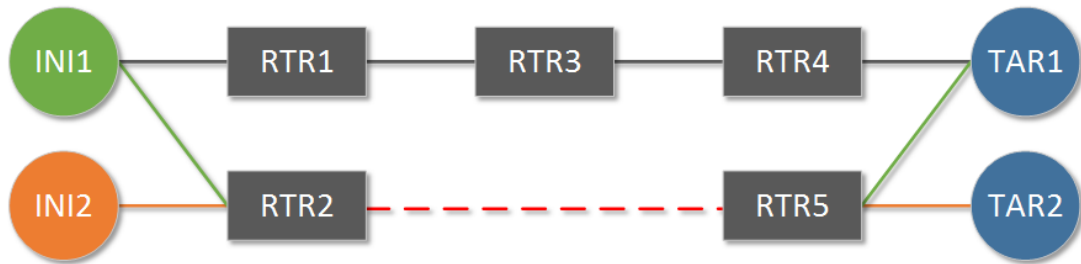


Figure 7-3: Shortest Path Selection with Collisions

As shown in Figure 7-3, if the path is selected by a breadth-first-search algorithm, the paths between INI1 to TAR1 and INI2 to TAR2 will both share the link between RTR2 and RTR5, highlighted as a dashed line, resulting in possible packet blocking. In this topology, there is a longer path available between INI1 and TAR1 as shown in Figure 7-4.

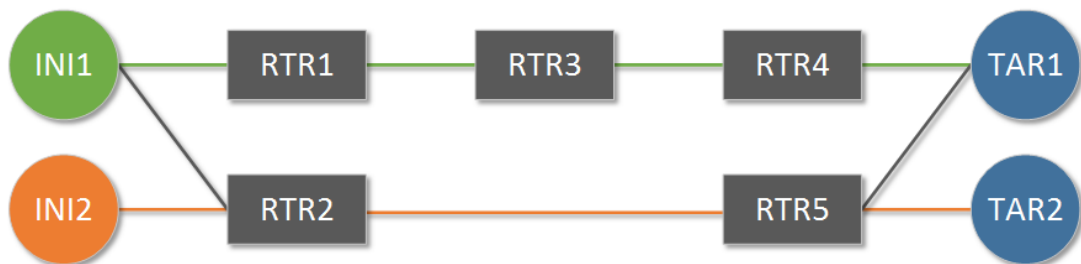


Figure 7-4: Path Selection with No Collisions

The path between INI1 and TAR1 in Figure 7-4 is one SpaceWire link longer than the shortest path but it now allows for both paths to be active within the same time-slot. In order to select these better paths computationally, a heuristic can be used to penalise the use of links that are already used in other paths.

Using the BFS algorithm to select paths does not take into account that the network topology may be a multi-graph, i.e. there are multiple links between two routers, because the BFS algorithm does not assign a value to the links and therefore, shows no preference between multiple links.

7.2.1.2 Weighted Search

A weighted search algorithm like Dijkstra's algorithm (Dijkstra 1959) uses a metric other than the number of edges to determine the cost of a path. Each edge is allocated a cost and the algorithm finds the path from a start node to an end node that minimises the combined cost of all edges in the path. Dijkstra's algorithm was investigated second because, like BFS, it is a relatively simple algorithm but it allows for more intelligent heuristics to be used when selecting paths between the initiators and targets.

For the SpaceWire-D scheduling problem, the cost of a shared link can be dynamically increased every time it is selected in the path between an initiator and a target. When using Dijkstra's algorithm, this discourages the selection of used links in favour of unused links when building a path between an initiator and a target. An example of a network topology with initially uniform edge costs is shown in Figure 7-5.

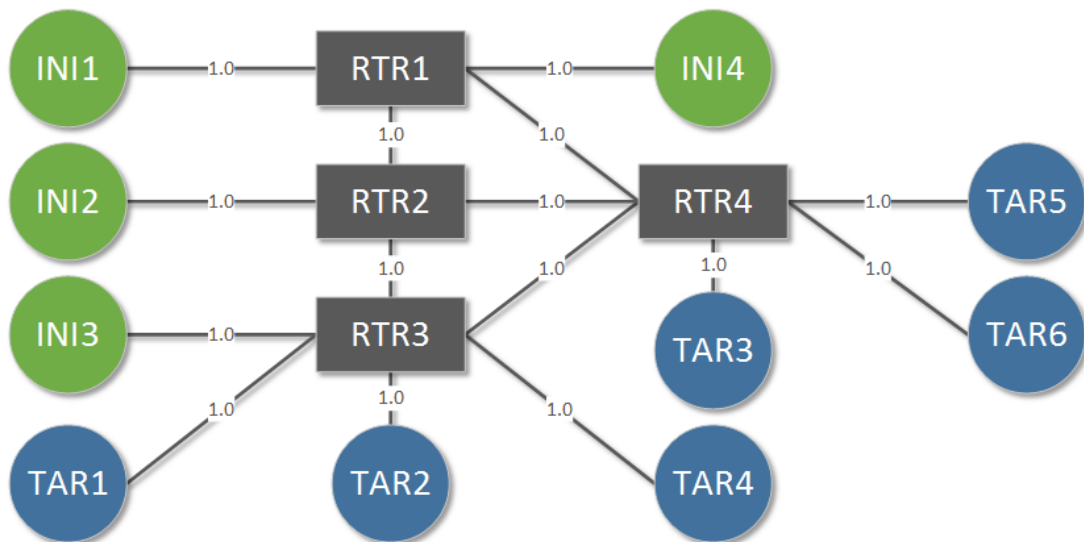


Figure 7-5: Weighted Search Initial Costs

In Figure 7-5, the cost of each edge is initially set to 1.0. Currently, if Dijkstra's algorithm was run over this network with uniform edge costs, the path selected

between an initiator and a target would be the same as if the BFS algorithm was used i.e. the path with the minimum number of edges.

As an example, a number of required initiator/target pairs are listed in Table 7-5 along with the paths that are selected when using the BFS algorithm.

Table 7-5: Paths Selected with the BFS Algorithm

Initiator	Target	Path	Total Conflicts
INI1	TAR5	INI1 → RTR1 → RTR4 → TAR5	0
INI4	TAR6	INI4 → RTR1 → RTR4 → TAR6	1
INI2	TAR4	INI2 → RTR2 → RTR3 → TAR4	1
INI4	TAR1	INI4 → RTR1 → RTR2 → RTR3 → TAR1	2

Similarly, the same initiator/target pairs and the paths selected when using Dijkstra's algorithm with a dynamic penalty of 3.0 whenever a link is used by a path are listed in Table 7-6.

Table 7-6: Paths Selected with Dijkstra's Algorithm

Initiator	Target	Path	Total Conflicts
INI1	TAR5	INI1 → RTR1 → RTR4 → TAR5	0
INI4	TAR6	INI4 → RTR1 → RTR2 → RTR4 → TAR6	0
INI2	TAR4	INI2 → RTR2 → RTR3 → TAR4	0
INI4	TAR1	INI4 → RTR1 → RTR4 → RTR3 → TAR1	1

In Table 7-5 and Table 7-6, the fourth column shows the cumulative number of shared links as each path is selected from top to bottom in the tables. When using the BFS algorithm, the RTR1 to RTR4 link is used by the first and second initiator/target pairs and the RTR2 to RTR3 link is shared by the third and fourth. This means that there are two pairs of conflicting paths that can't be scheduled within the same time-slot due to the shared links.

When using Dijkstra's algorithm with dynamic penalties, the second initiator/target pair's path avoids the RTR1 to RTR4 link because it is no longer the path with the lowest cost as shown in Figure 7-6.

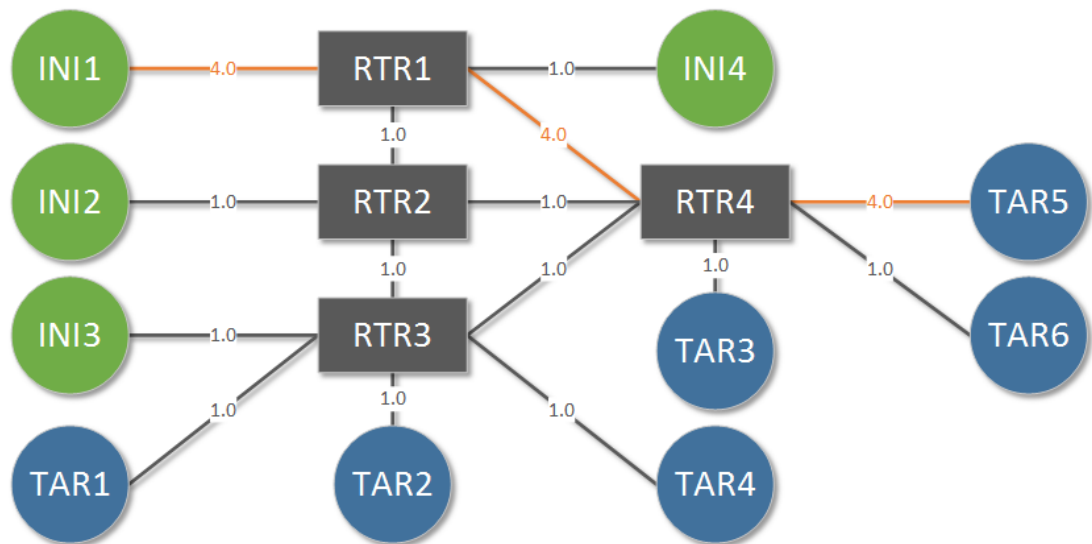


Figure 7-6: Weighted Search Costs after First Path Selected

In Figure 7-6, the router to router link used by the first initiator/target pair, RTR1 to RTR4, has had a penalty cost of 3.0 added. When the algorithm searched for the cheapest path for the next initiator/target pair, INI4 to TAR6, it avoided the high-cost edge by selecting the longer but cheaper path from RTR1 to RTR2 to RTR4 before ending at TAR6.

In comparison to the paths selected using the BFS algorithm which had two pairs or conflicting paths, using Dijkstra's algorithm with dynamic penalties results in just one pair of paths that can't be scheduled within the same time-slot.

For completeness, the edge costs after all four paths have been selected are shown in Figure 7-7.

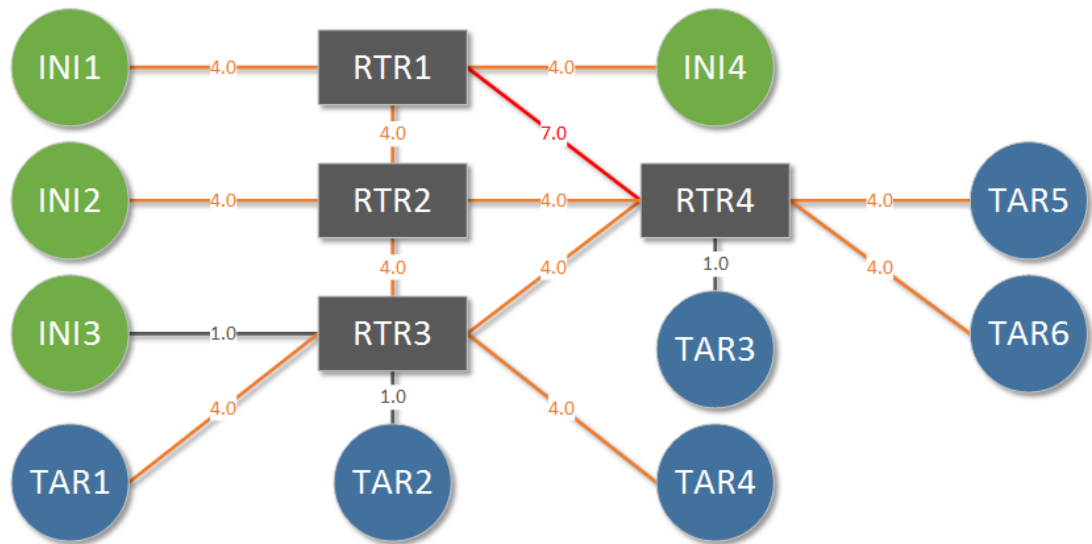


Figure 7-7: Weighted Search Costs after All Paths Selected

As shown in Figure 7-7, several links now have a cost of 4.0 meaning that they have been used in one path, and the link between RTR1 and RTR4 has a cost of 7 because it has been used in two paths.

In this implementation of Dijkstra's algorithm, it is extended it to take into account that the network graph may be a multi-graph. By giving each duplicate edge a label, a separate cost can be maintained for each edge and it is then the lowest cost edge from the set of duplicate edges that is selected during the execution of the algorithm. When a path is selected, the dynamic penalty is only added to the edges that have been chosen from the sets of duplicate edges.

7.2.2 Scheduling Transactions

This section describes how the scheduling of transactions for the SpaceWire-D scheduling problem is modelled by transforming it into a combination of simple allocation algorithms and a variation on the classic bin-packing problem.

7.2.2.1 Conflict Graph

A conflict graph is used to describe if the initiator/target pairs share any SpaceWire links and therefore are unable to be scheduled within the same time-slot. The vertices in the conflict graph represent each of the initiator/target pairs and there is an edge between two initiator/target pairs if their paths share a link. An adjacency matrix representation of the conflict graph for the initiator/target pairs from Table 7-5 is shown in Figure 7-8.

	(INI1, TAR5)	(INI1, TAR6)	(INI2, TAR4)	(INI4, TAR1)
(INI1, TAR5)	0	1	0	0
(INI1, TAR6)	1	0	0	0
(INI2, TAR4)	0	0	0	1
(INI4, TAR1)	0	0	1	0

Figure 7-8: Example Conflict Graph

An adjacency matrix M is a two-dimensional binary array representation of a graph $G = (V, E)$ where each vertex, or node, in V is represented by a row and column with the same index, and the value of a cell $M_{i,j}$ is equal to 1 if the edge (V_i, V_j) exists. In this case, the conflict graph is bidirectional so the adjacency matrix is mirrored along the rising diagonal axis. As shown in Figure 7-8, there are two conflicts between the (INI1, TAR5) and (INI1, TAR6) initiator/target pair, and the (INI2, TAR4) and (INI4, TAR1) initiator/target pair, indicated by the value of 1 in the relevant cells.

Using a conflict graph provides a simple data structure for the algorithm to check if two initiator/target pairs can be scheduled within the same time-slot.

7.2.2.2 Periodic Traffic

A periodic traffic bandwidth requirement $P_r = (i, t, o, s, r)$, as described in Section 7.1.1.1, is a cyclic communication between an initiator and a target at a rate r . The value of r should be a power of two multiple of the system's control cycle i.e. the

number of schedule epochs per second. This is so that the transactions can be placed at even intervals within the schedule. For example, if the system's control cycle is 16 Hz i.e. 1024 time-codes per second, a 16 Hz transaction could be implemented as a static bus in time-slot 0 and a 32 Hz transaction could be implemented as two static buses in time-slots 0 and 32. This is a current limitation of the scheduling algorithm implementation which should be updated in the future to allow for any value of r .

On a spacecraft, the source of periodic traffic could be one or more OBCs reading or receiving housekeeping information from other devices on the network. It is likely that there will be few initiators requiring periodic traffic but if there is more than one initiator, there must be a check to make sure that the initiator/target pair's path does not conflict with any paths used by other initiator's static buses operating in the same time-slot.

Additionally, a periodic transaction should only be added to an existing static bus if the combined execution time of the existing transactions and the new transaction fits within the time-slot.

Once a non-conflicting time-slot that has enough room for the transaction has been found, if the rate of the periodic bandwidth requirement is a multiple of the control cycle it will require additional transactions at uniform intervals throughout the schedule. The algorithm must check that the first suitable slot is early enough in the schedule that additional transactions can be scheduled before the end of the schedule, otherwise the bandwidth requirement is not schedulable.

The periodic bandwidth requirement scheduling algorithm, written in Python, is listed in Figure 7-9.

```

1  def schedule_periodic(reqs):
2      for req in reqs:
3          nslots = req.r / CTRL_CYCLE
4          ivl = 64 / nslots
5          slot = 0
6          while conflict(slot, req) or not fit(slot, req):
7              slot = slot + 1
8          if slot > 63 or slot > (ivl - 1):
9              return False
10         for i in range(0, nslots):
11             next = slot + i * ivl
12             if not is_slot_good(next, req):
13                 return False
14             add_to_sbus(next, req)
15     return True

```

Figure 7-9: Periodic Bandwidth Scheduling Algorithm

In Figure 7-9, a function is defined, called `schedule_periodic`, which takes a list of periodic bandwidth requirements and attempts to allocate the transactions for each requirement to one or more static buses. In line 3, the requirement's rate is divided by the control cycle in order to find the number of slots required for static buses and in line 4, the schedule length of 64 is divided by the number of slots to get the interval between the static buses. In lines 5 to 7 the time-slots are iterated starting at time-slot 0 until a slot is found that doesn't contain any conflicting initiator/target pairs and has room for the requirement's transaction, using the `conflict` and `fit` functions. Lines 8 and 9 check if the slots needed by the requirement fit within the remainder of the schedule, starting at `slot`. Finally, lines 10 to 14 check each slot to make sure it is suitable then extends the static buses in the required slots to add the new requirement by calling the `add_to_sbus` function.

A block diagram of the algorithm used to schedule a periodic bandwidth requirement is shown in Figure 7-10.

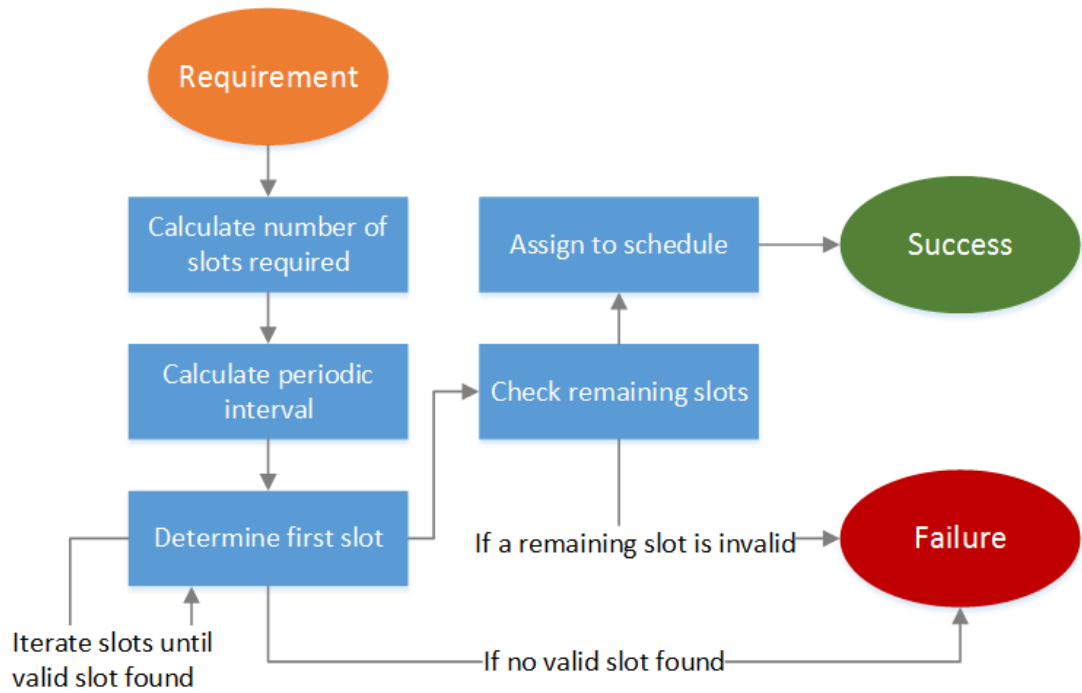


Figure 7-10: Periodic Bandwidth Scheduling Algorithm Block Diagram

As shown in Figure 7-10, the algorithm starts with a periodic requirement. Next, the number of slots required is calculated, then the periodic interval. The first valid slot is then found by iterating over the slots. If no valid slot is found, the algorithm fails. If a valid slot is found, the remaining required slots are checked for validity. If any of the remaining required slots are invalid, the algorithm fails. If all of the required slots are valid, the requirement is assigned to the schedule and the algorithm succeeds.

To demonstrate periodic traffic scheduling, an example of scheduling a number of periodic bandwidth requirements from two initiators with some conflicting initiator/target pairs is described next. The periodic bandwidth requirements used in the example are shown in Table 7-7

Table 7-7: Example Periodic Bandwidth Requirements

Initiator	Target	Operation	Size	Rate
INI1	TAR1	Read	128	16 Hz
INI1	TAR2	Read	128	32 Hz
INI2	TAR1	Read	256	16 Hz
INI2	TAR3	Read	128	64 Hz

The conflict graph for the example, with one conflict between (INI1, TAR1) and (INI2, TAR1) is shown in Figure 7-11.

	(INI1, TAR1)	(INI1, TAR2)	(INI2, TAR1)	(INI2, TAR3)
(INI1, TAR1)	0	0	1	0
(INI1, TAR2)	0	0	0	0
(INI2, TAR1)	1	0	0	0
(INI2, TAR3)	0	0	0	0

Figure 7-11: Example Periodic Bandwidth Requirements Conflict Graph

Using the periodic bandwidth requirement scheduling algorithm from Figure 7-9 and a control cycle of 16 Hz, the resulting static buses are listed in Table 7-8

Table 7-8: Example Periodic Bandwidth Requirements Static Buses

Initiator	Targets	Slot	Size
INI1	TAR1, TAR2	0	1
INI1	TAR2	32	1
INI2	TAR1	1	1
INI2	TAR3	0	1
INI2	TAR3	16	1
INI2	TAR3	32	1
INI2	TAR3	48	1

As shown in Table 7-8, seven static buses have been created to satisfy the bandwidth requirements listed in Table 7-7. The first requirement has been added to static bus 0, the second has been added to static buses 0 and 32, the third has been added to static bus 1 because it cannot be added to static bus 0 due to the (INI1, TAR1) and (INI2,

TAR1) paths conflicting and the fourth requirement has been added to static buses 0, 16, 32 and 48.

7.2.2.3 *Aperiodic Traffic*

An aperiodic traffic bandwidth requirement $A_r = (i, t, o, s, d)$, as described in Section 7.1.1.2, is a transaction between an initiator and a target that can be loaded at any time and must be completed within a deadline d . A dynamic bus can be used for aperiodic traffic with time-slots placed at intervals less than the deadline so that no matter when a transaction is loaded, an allocated time-slot will be available to execute the transaction before the deadline expires.

On a spacecraft, the source of aperiodic traffic could be an OBC sending commands to another device after receiving confirmation of an event occurring. For example, if a sensor on the spacecraft detects that a change in the environment warrants recording, it could notify the OBC which would then need to send a command to an instrument to capture the event, which must reach the instrument within a deadline.

When allocating time-slots to a dynamic bus, the minimum number of time-slots which satisfy the aperiodic requirement should be used so that bandwidth is not wasted by over-allocating time-slots which could be used for other virtual buses.

The aperiodic bandwidth requirement scheduling algorithm, written in Python, is listed in Figure 7-12.

```

1  def schedule_aperiodic(reqs):
2      for req in reqs:
3          max_ivl = (req.dl * 1000 / SLOT_LENGTH) - 1
4          first = first_good_slot(req)
5          slots = [first]
6          while not is_allocated(slots, req):
7              if first_slot > 63 or first > (max_ivl - 1):
8                  return False
9              reset = False
10             next = min(slots[-1] + max_ivl, 63)
11             while not is_slot_good(next, req):
12                 next -= 1
13                 if next <= slots[-1]:
14                     reset = True
15                     break
16             if reset:
17                 first += 1
18                 slots = [first]
19             else:
20                 slots.append(next)
21             for slot in slots:
22                 add_to_dbus(slot, req)
23         return True

```

Figure 7-12: Aperiodic Bandwidth Requirement Scheduling Algorithm

In Figure 7-12, a function is defined, named `schedule_aperiodic`, which takes a list of aperiodic bandwidth requirements and attempts to allocate the transactions for each requirement to a dynamic bus. Line 3 calculates the maximum interval between time-slots in order to satisfy the aperiodic requirement. Line 4 finds the first suitable slot that is available for the requirement and line 5 adds the slot to a list. Line 6 begins a loop that runs until the algorithm completes or fails. Line 7 checks if there is no suitable first slot or if the first suitable slot is too far into the schedule to meet the interval requirements, and if so returns false on line 8. Line 9 defines a flag that is used to reset the process and increment the first slot, in lines 16-18, to retry scheduling from another first slot. Lines 10-15 attempt to find the next suitable slot between the last selected slot and the maximum interval. If a suitable next slot is found, it is added to

the list in line 20. Finally, lines 21 and 22 extend the dynamic buses in the required slots to add the new requirement by calling the `add_to_dbus` function.

A block diagram of the algorithm used to schedule an aperiodic bandwidth requirement is shown in Figure 7-13.

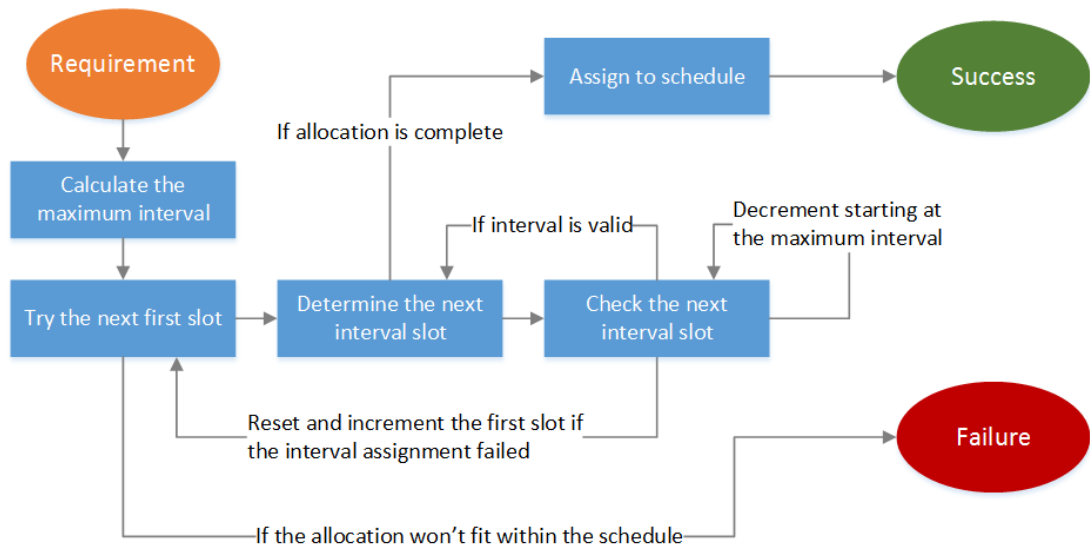


Figure 7-13: Aperiodic Bandwidth Scheduling Algorithm Block Diagram

As shown in Figure 7-13, the algorithm starts with an aperiodic requirement. Next, the maximum interval is calculated. The allocation loop is then entered which attempts to schedule the requirement starting at the first valid slot. If, at this point, the allocation won't fit within the schedule based on the first slot, the algorithm fails. The slot at the next maximum interval is then determined. This slot is checked for validity, and decremented if it is invalid. If the slot is decremented so that the assignment of the interval slot fails, the allocation loop is reset starting at an incremented first slot. If the assignment of the interval slot succeeds, the next interval slot is tried or, if the allocation is complete at this point, the requirement is assigned to the schedule and the algorithm succeeds.

7.2.2.4 *Payload Data Traffic*

Once the periodic and aperiodic traffic bandwidth requirements have been satisfied, the remaining non-conflicting or free time-slots can be used for payload data traffic. A payload data traffic bandwidth requirement $D_r = (i, t, o, s, n)$, as described in Section 7.1.1.3, is a data-flow between an initiator and a target that must be allocated enough space within the schedule to read or write n packets of data with size s per second. One or more asynchronous buses can be used for payload data traffic with enough time-slots allocated throughout the schedule to transfer the required number of packets per second between the initiator and the target.

On a spacecraft, the source of payload data traffic is one or more scientific instruments that generate a number of data packets a second that must be stored in a mass-memory device before being transmitted back to Earth.

The objective of the payload data traffic scheduling algorithm is to allocate transactions to the minimum number of time-slots possible. It is possible for the algorithms to allocate to more than 64 time-slots during their execution. This is so that its performance can be measured to determine how far away it is from an acceptable solution, but a solution is not acceptable unless it fits within 64 time-slots.

Bin Packing

Scheduling payload data traffic can be modelled as a variation on the bin-packing problem (BPP) which uses conflicts (Epstein and Levin 2006) and multi-capacity bins (Leinberger, Karypis and Kumar 1999). The BPP is an NP-hard combinatorial optimisation problem where the aim is to place a finite number of items with varying sizes into a minimum number of bins with limited capacities.

In the model, there is a number of payload data packets that must be encapsulated within RMAP transactions and executed within the 64 time-slots. The number of packets that must be executed across a schedule epoch is determined by the payload data requirement's number of packets divided by the number of schedule epochs per second. Each transaction has a size which is its execution time and each time-slot is a bin that has multiple sub-bins, one for each initiator, where each sub-bin's capacity is the time-slot duration minus the initiator processing time. The conflict graph determines if transactions from two different initiators can be placed in the same time-slot and any conflicts from the periodic and aperiodic transactions already scheduled must also be taken into account.

As an example, consider the following four payload data traffic requirements:

Table 7-9: Example Payload Data Bandwidth Requirements

Initiator	Target	Operation	Size	Packets/Second
INI1	TAR1	Write	4096	4
INI2	TAR2	Write	4096	6
INI3	TAR1	Write	4096	8
INI4	TAR2	Write	4096	2

With the following conflict graph:

	(INI1, TAR1)	(INI2, TAR2)	(INI3, TAR1)	(INI4, TAR2)
(INI1, TAR1)	0	0	1	0
(INI2, TAR2)	0	0	0	1
(INI3, TAR1)	1	0	0	0
(INI4, TAR2)	0	1	0	0

Figure 7-14: Example Payload Data Conflict Graph

As shown in Table 7-9 and Figure 7-14, there are four payload data bandwidth requirements with a combined 20 packets per second that need to be scheduled and

there is a conflict between (INI1, TAR1) and (INI3, TAR1) as well as between (INI2, TAR2) and (INI4, TAR2).

Assuming a simplified schedule with two time-slots, each of which has the capacity to execute eight RMAP transactions with a data size of 4096 bytes, and a control cycle of 1 Hz, a possible allocation is shown in Figure 7-15.

Time-Slot 0						
(INI1, TAR1)	(INI1, TAR1)	(INI1, TAR1)	(INI1, TAR1)	Free		
(INI2, TAR2)	(INI2, TAR2)	(INI2, TAR2)	(INI2, TAR2)	(INI2, TAR2)	(INI2, TAR2)	Free
Time-Slot 1						
(INI3, TAR1)	(INI3, TAR1)	(INI3, TAR1)	(INI3, TAR1)	(INI3, TAR1)	(INI3, TAR1)	(INI3, TAR1)
(INI4, TAR2)	(INI4, TAR2)	Free				

Figure 7-15: Example Payload Data Allocation

As shown in Figure 7-15, there are two time-slots which have been allocated a number of RMAP transactions and each row of transactions represents an initiator sub-bin. Transactions with the same shade are conflicting and have not been allocated within the same time-slot. In the slots used by the (INI1, TAR1), (INI2, TAR2) and (INI4, TAR2) initiator/target pairs, there is some additional space at the end of the slots for any additional transactions sent by the initiator because the time-slots haven't been completed consumed.

In a more complex scenario with additional bandwidth requirements, it may be possible to use some of the free space in one of the initiator sub-bins. Selecting which

time-slot to use can be done heuristically using different criteria to choose an allocation of transactions.

First-Fit Heuristic

The first-fit (FF) heuristic (Coffman, et al. 2013) is a simple algorithm that begins looking at the first bin and determines if there is enough remaining capacity for the item to fit. If there is enough room, the item is placed in the bin, otherwise the algorithm moves to the next bin and so on until the item is allocated. If there are no bins with enough capacity for the item, a new bin is opened and the item is allocated there.

For each payload data requirement in this model, the FF heuristic scans the schedule and finds the first time-slot which has enough remaining capacity for at least one transaction from a payload data bandwidth requirement and inserts as many as possible until the requirement has been satisfied or no more will fit in the time-slot. The heuristic then repeats this process until all transactions for the payload data requirement have been allocated.

The intention of this heuristic is to allocate transactions using the simplest possible method so that it can be used as a benchmark against which other heuristics can be measured.

A block diagram of the algorithm used to schedule a payload data bandwidth requirement using the first-fit heuristic is shown in Figure 7-16.

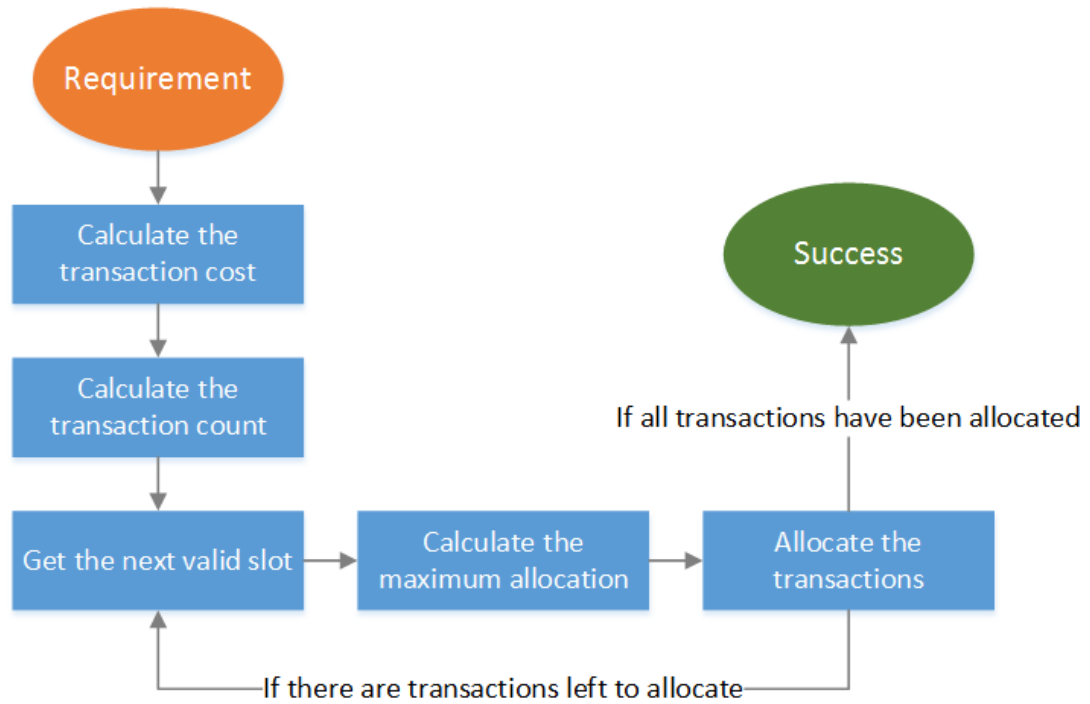


Figure 7-16: First-Fit Heuristic Block Diagram

As shown in Figure 7-16, the algorithm starts with a payload data requirement. Next, the cost, i.e. the execution time, of a single RMAP transaction from this requirement is calculated. Following this, the transaction count is calculated which determines how many transactions are required to be scheduled in each schedule epoch. Finally, the algorithm enters a loop which allocates the maximum possible number of transactions to the first-found valid slots until the requirement is satisfied.

Best-Fit Heuristic

The best-fit (BF) heuristic (Coffman, et al. 2013) looks at every open bin and finds the bin that has the most remaining capacity in which to allocate the item. If there are no bins with sufficient capacity for the item, a new bin is opened and the item is placed there.

For each payload data requirement in this model, the BF heuristic finds the time-slot that has the most space available in the initiator sub-bin for the requirement and inserts

as many transactions as possible until the requirement has been satisfied or no more will fit in the time-slot. The heuristic then repeats this process until all transactions for the payload data requirement have been allocated.

Using the FF heuristic, it's possible for transactions to be split across many nearly full time-slots. For example, consider a payload data requirement that requires five large transactions to be allocated to the schedule. In this example, the first four slots are partially full and can only fit one of the transactions but the fifth slot is completely empty. The FF heuristic will first allocate one transaction to each of the first four slots and the last transaction to the empty fifth slot, as shown in Figure 7-17.

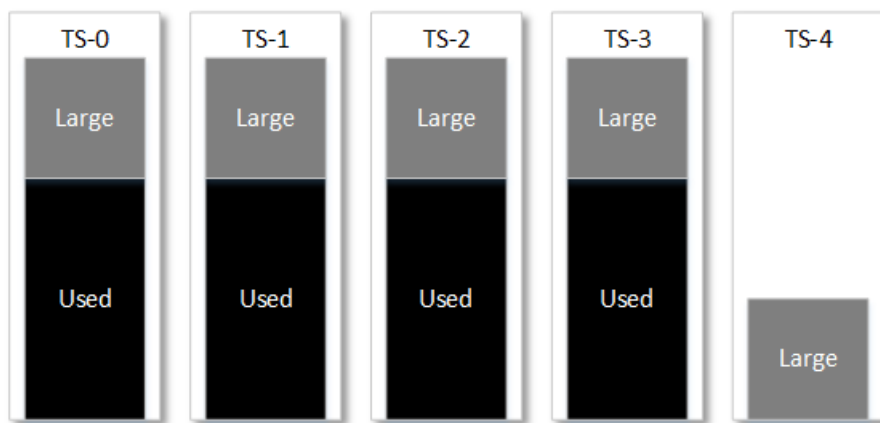


Figure 7-17: First-Fit Heuristic Allocation

If there is another payload data requirement that must be allocated eight small transactions which conflict with the large transactions, they can't be allocated to any of the first five slots. However, if the algorithm had allocated more transactions to the empty fifth slot, it would allow the small transactions to be allocated to one or more of the first four slots as shown in Figure 7-18.

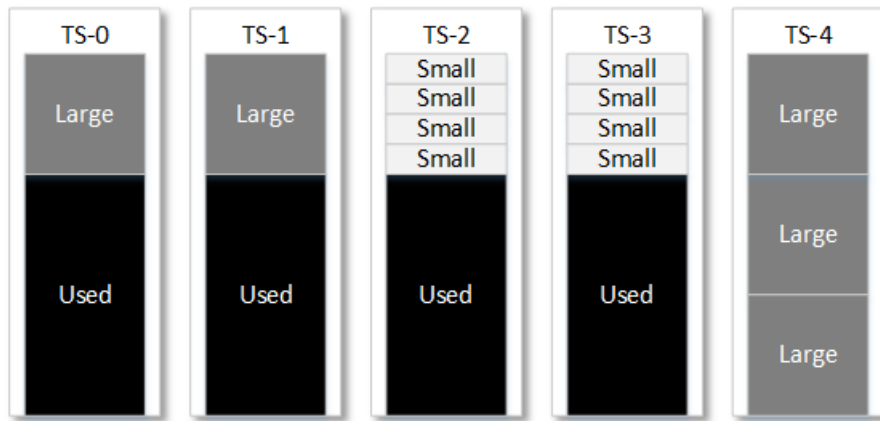


Figure 7-18: Best-Fit Heuristic Allocation

The intention of the BF heuristic is to insert transactions into as few time-slots as possible in order to maximise the number of time-slots available for other conflicting payload data requirement transactions.

A block diagram of the algorithm used to schedule a payload data bandwidth requirement using the best-fit heuristic is shown in Figure 7-19.

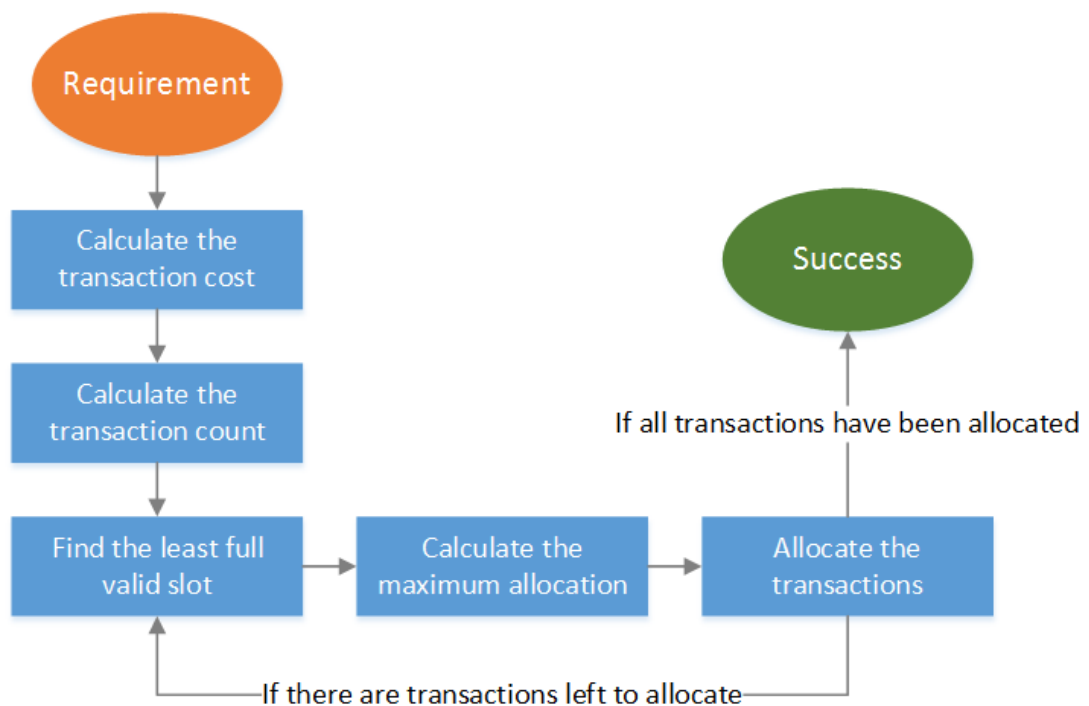


Figure 7-19: Best-Fit Heuristic Block Diagram

As shown in Figure 7-19, the algorithm starts with a payload data requirement. Next, the transaction cost and transaction count are calculated in the same way as in the first-fit heuristic. Following this, the algorithm enters a loop that finds the least full slot and allocates the maximum number of transactions to it. This loop repeats until the bandwidth requirement is satisfied.

Least-Conflicting Heuristic

The least-conflicting (LC) heuristic searches for the bin which will introduce the least additional conflicts with other potential items within the same bin if the item is allocated within it. If there are multiple least-conflicting bins, the item is inserted into one of them by using the BF heuristic and if there are no open bins with enough space for the item, a new bin is opened and the item is placed there.

For each payload data requirement in this model, the LC heuristic finds the time-slot that, when a transaction is allocated to it, the minimum number of conflicts with other transactions are added. For example, consider a payload data requirement that must be allocated transactions that conflict with three other transactions in the network. There is a schedule with two slots: the first is completely free while the second is partially full and there have been some transactions allocated which share two of the three conflicts with the new requirement. If the new transactions are allocated to the completely free first slot, three conflicts will be added to the time-slot. However, if the transactions are allocated to the second slot, only one conflict will be introduced into the slot as the other two are already present.

The intention of the LC heuristic is to allocate transactions to the schedule while introducing the minimum number of potential conflicts, therefore allowing more conflicting transactions to be allocated to other slots.

A block diagram of the algorithm used to schedule a payload data bandwidth requirement using the least-conflicting heuristic is shown in.

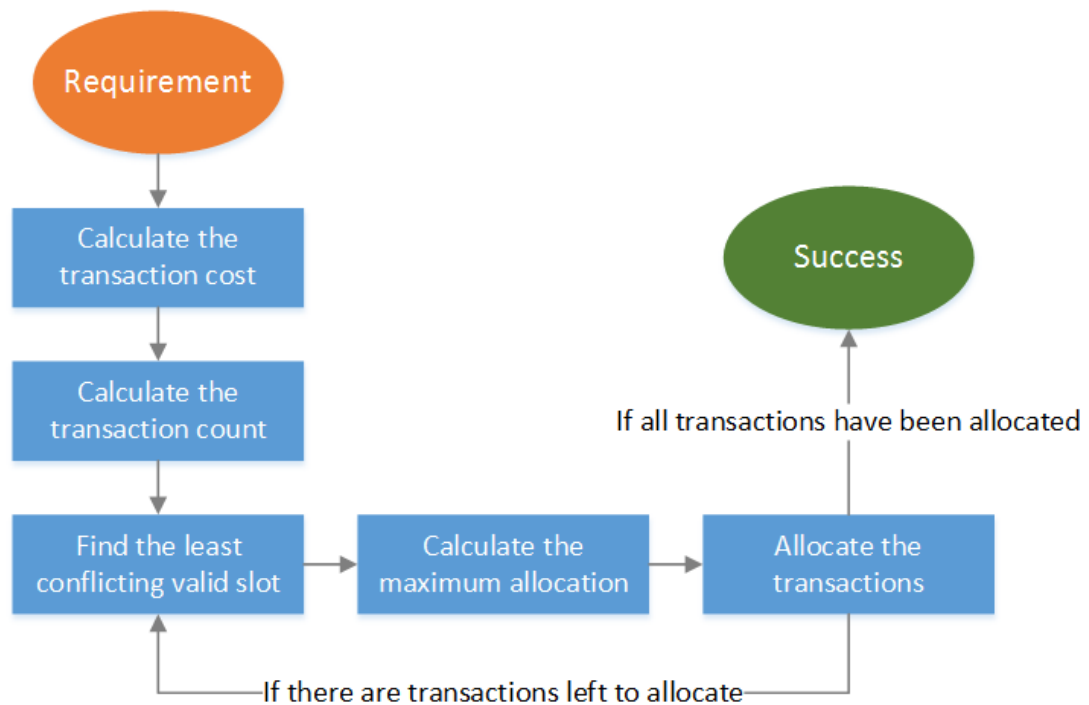


Figure 7-20: Least-Conflicting Heuristic Block Diagram

As shown in Figure 7-20, the algorithm is very similar to the first-fit and best-fit heuristics. The only difference is the block that determines the most appropriate slot measures the number of conflicts and selects the slot with the most conflicts shared with the bandwidth requirement.

7.2.3 Complete Algorithm

The algorithm for scheduling a SpaceWire-D network based on a list of bandwidth requirements is a combination of path selection, periodic, aperiodic and payload data traffic scheduling as shown in Figure 7-21.

```

1  def schedule_spwd_network(preqs, areqs, pdreqs):
2      select_paths(preqs, areqs, pdreqs)
3      schedule_periodic(preqs)
4      schedule_aperiodic(areqs)
5      schedule_payload_data(pdreqs)

```

Figure 7-21: Complete SpaceWire-D Scheduling Algorithm

Figure 7-21 shows an example of the strategy design pattern where the complete algorithm consists of four steps which may be implemented in various ways by selecting the different heuristics.

The top-level design of the scheduling algorithm is illustrated in Figure 7-1. There is a network topology and a list of bandwidth requirements that are input into the path selection and transaction allocation stages of the algorithm. The path selection strategies are described in Section 7.2.1 and the transaction allocation strategies are described in Section 7.2.2. Each type of bandwidth requirement is allocated using a different algorithm. Periodic bandwidth requirements are allocated using the algorithm listed in Figure 7-9 and illustrated in Figure 7-10; aperiodic bandwidth requirements are allocated using the algorithm listed in Figure 7-12 and illustrated in Figure 7-13; and finally, payload data bandwidth requirements are allocated using a bin-packing algorithm and the heuristics illustrated in Figure 7-16, Figure 7-19 and Figure 7-20.

This scheduling strategy uses a combination of approximation algorithms to attempt to find a solution that satisfies the problem specification. This means that if a solution is found, it is not guaranteed that it is the optimal solution. In future work, it may be necessary to explore exhaustive algorithms to find optimal solutions to the SpaceWire-D scheduling problem. However, as described in Chapter 8, the scheduling strategy presented here generates good results for both the randomised test cases and a real-life test case.

7.3 Summary

In this chapter, a novel strategy for scheduling SpaceWire-D networks was proposed. The scheduling algorithm takes a network topology and a list of bandwidth requirements as input and generates a set of transaction to time-slot allocations as output. The allocations attempt to satisfy the bandwidth requirements whilst adhering to the rules of SpaceWire-D.

First, the problem was specified in terms of periodic, aperiodic and payload data bandwidth requirements. The network topology and parameters were also specified which are required to model the performance of the SpaceWire-D network with regards to the initiators, targets, routers and links.

Next, a number of algorithms were described to perform path selection and transaction scheduling. The simplest path selection algorithm involved choosing the shortest paths using BFS. The second path selection algorithm was a weighted search path selection technique using a modified version of Dijkstra's algorithm. This algorithm dynamically penalised the cost of links if they were shared between multiple initiator/target paths, with the intention of reducing conflicts in the network. For transaction scheduling, two simple algorithms were described for allocating periodic and aperiodic transactions and the payload data transaction allocation process was modelled as a variation of the classic bin-packing problem.

Chapter 8

Evaluating the Scheduling Strategy

In the previous chapter, a two-stage scheduling strategy was described for generating SpaceWire-D initiator schedules. In this chapter, the strategy is evaluated by using it to schedule a variety of randomised test cases consisting of network topologies and bandwidth requirement lists. Following this evaluation, the scheduling strategy is used in a case study of a real space mission, the JUperiter ICy moons Explorer (European Space Agency 2011).

8.1 Test Cases

The process of generating test cases was done in two steps: firstly, a semi-randomised network topology is built to represent the SpaceWire network for the mission; and secondly, a bandwidth requirement list is generated consisting of periodic, aperiodic and payload data requirements. There are five parameters input into the test case generation algorithm: the number of nodes, routers, periodic, aperiodic and payload data requirements.

8.1.1 Generating Network Topologies

The network topology of a test case is semi-randomised because it creates randomised links but adheres to some structural rules. The first rule is that the network graph must be connected i.e. it is possible to route a packet from any node to any other node on

the network through one or more routers. Secondly, each node is connected to a single router but routers may be connected to multiple nodes or other routers.

Generating the network topology is done in a number of stages. The first stage connects each of the nodes to a random router. The second stage connects some routers together by adding in random router to router links. At this point it's possible but not guaranteed that the network graph is connected, so there is another stage which merges each of the connected components until the entire graph is connected.

Firstly, the list of connected components is determined by doing a series of graph traversals. Starting at node 0, an exhaustive search is done to note which nodes were discovered during this process, forming the first connected component. Next, the same is done for the first undiscovered node to form another connected component. This process is continued until all the nodes have been discovered and a list of the connected components is created.

To merge the connected components into a single connected graph, the process starts at the first component and adds a random router to router link to the second connected component in order to form a new, larger connected component. A random router to router link is then added between the new connected component and the third connected component. This process is repeated until all of the connected components have been merged into one connected graph.

This network topology generation process is illustrated in Figure 8-1.

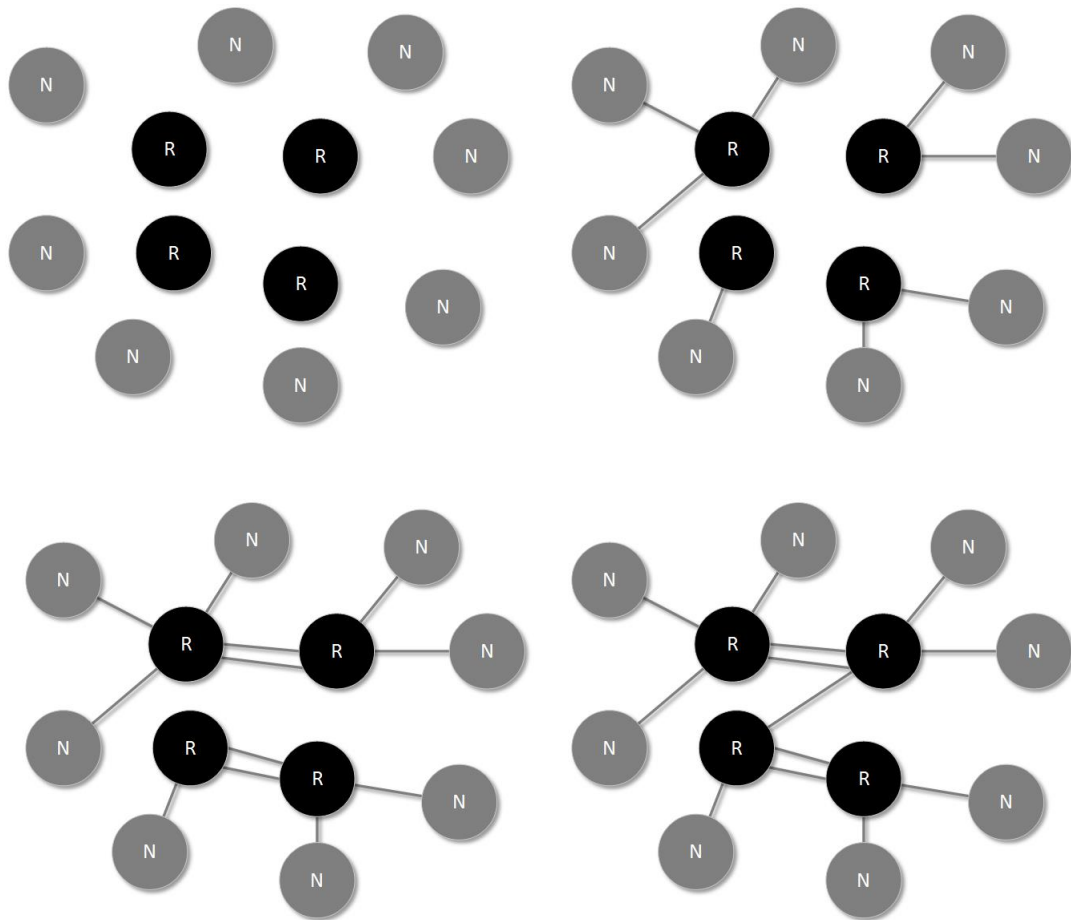


Figure 8-1: Generating Network Topology Stages

In Figure 8-1, there is a graph, in the top left diagram, consisting of eight nodes and four routers with no links. In the second stage, in the top right diagram, each node is connected to one of the routers. Next, each router is connected to one of the other routers. In this example, the top two and bottom two routers are mutually connected to each other which forms two connected components. In the final stage, the top connected component is linked to the bottom connected component by adding a link between the top right and bottom left router.

While the third stage could be left out and the algorithm could just merge the connected components each consisting of a router plus one or more nodes, the third stage adds the possibility of multiple links being created between routers. These are

likely to occur in a SpaceWire network in order to reduce the bottleneck of a single link and increase the available bandwidth between routers.

8.1.2 Generating Bandwidth Requirements

The generation of bandwidth requirements is relatively simple in comparison to the network topology. The initiator and target are selected at random from a subset of nodes, depending on the type of requirement. Periodic and aperiodic requirements select an initiator from the first two nodes and a random target from any other node, mimicking the housekeeping or command transactions between one or two on-board computers and other devices on the network. Payload data transactions randomly select an initiator and target from any node on the network.

RMAP operation types are randomly set to a read or write operation and the size of the transaction is dependent on the type of bandwidth requirement. Periodic and aperiodic requirements select a random size from the following values: 32, 64, 128 or 256 bytes. Payload data requirements select a random size from the following values: 512, 1024, 2048 or 4096 bytes and a packet count randomly selected from the multiples of 16 between 64 and 256.

Periodic bandwidth requirement rates are selected based on a multiple of the control cycle, which is assumed is 16 Hz i.e. 1024 SpaceWire time-codes per second and a time-slot interval of 976.5625 ms for these test cases. The periodic requirement rates are randomly selected from the following values: 16, 32 or 64 Hz. Each aperiodic bandwidth requirement has a deadline which is randomly selected from the following values: 10, 15, 20, 25 or 30 ms.

8.1.3 Test Case Generation Algorithm

There are two stages to the test case generation algorithm: firstly, the random network topology is generated using the process described in Section 8.1.1; and secondly, the random periodic, aperiodic and payload data bandwidth requirements are generated using the processes described in Section 8.1.2.

The random network topology generation algorithm is illustrated in Figure 8-2.

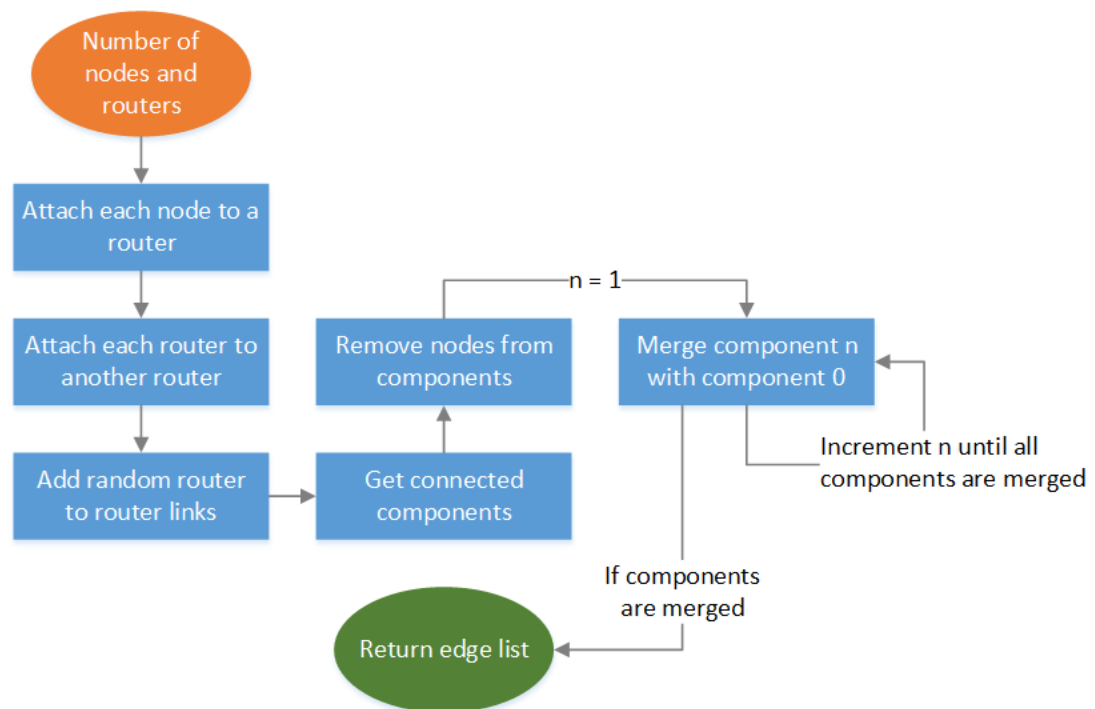


Figure 8-2: Network Topology Generation Algorithm Block Diagram

As shown in Figure 8-2, the algorithm begins by taking a number of nodes and routers as input. Next, each node is attached to a router then each router is attached to another router. Following this, random router to router links are added. At this point, the network may be connected but it's more likely to have multiple connected components so a process begins to merge them into a single connected network. Firstly, the nodes are removed from the components so that only router to router links are added in the next stage. Components are merged starting at the second component by adding a

random link between it and the first component. This process repeats until all connected components are merged into a connected network at which point the algorithm returns a list of edges to the application. Figure 8-1 illustrates an example of the algorithm generating a network with 8 nodes and 4 routers.

The random bandwidth requirement generation algorithm is illustrated in Figure 8-3.

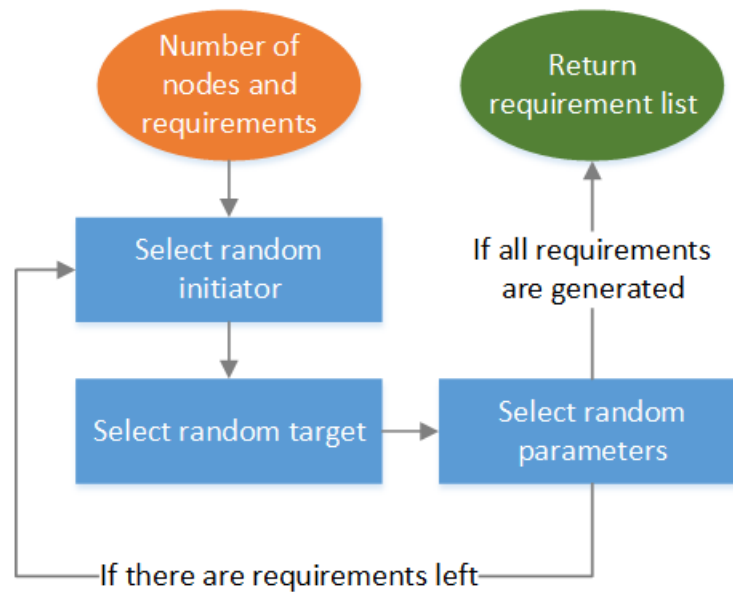


Figure 8-3: Requirement Generation Algorithm Block Diagram

As shown in Figure 8-3, the algorithm begins by taking the number of nodes and requirements as input. A random initiator and target are then selected based on the constraints for each bandwidth requirement as described in Section 8.1.2. For example, periodic and aperiodic constrain the initiators to the first 2 nodes and the targets to the remaining nodes to emulate OBCs as initiators and other devices as targets. Payload data bandwidth requirements select any pair of nodes as the initiator and target. Next, the requirement's parameters are randomly selected based on the constraints described in Section 8.1.2. Additional requirements are generated until the required number is met, then the list is returned to the application.

The test case generation algorithm software was implemented in Python 3 using the open-source PyCharm Community Edition IDE from JetBrains (JetBrains 2015). The software is a command-line based script that takes a list of parameters which are used to generate the test cases. The script takes the following 7 parameters: the number of test cases to generate; the number of nodes; the number of routers; the number of periodic, aperiodic and payload data bandwidth requirements; and a prefix for the test case filenames. These parameters are then passed to the relevant topology and bandwidth requirement generating algorithms.

An overview of the test case generation software flow is shown in Figure 8-4. In this figure, Block A refers to the topology generation algorithm illustrated in Figure 8-2 and Blocks B, C and D refer to the bandwidth requirement generation algorithm illustrated in Figure 8-3 for periodic, aperiodic and payload data requirements, respectively.

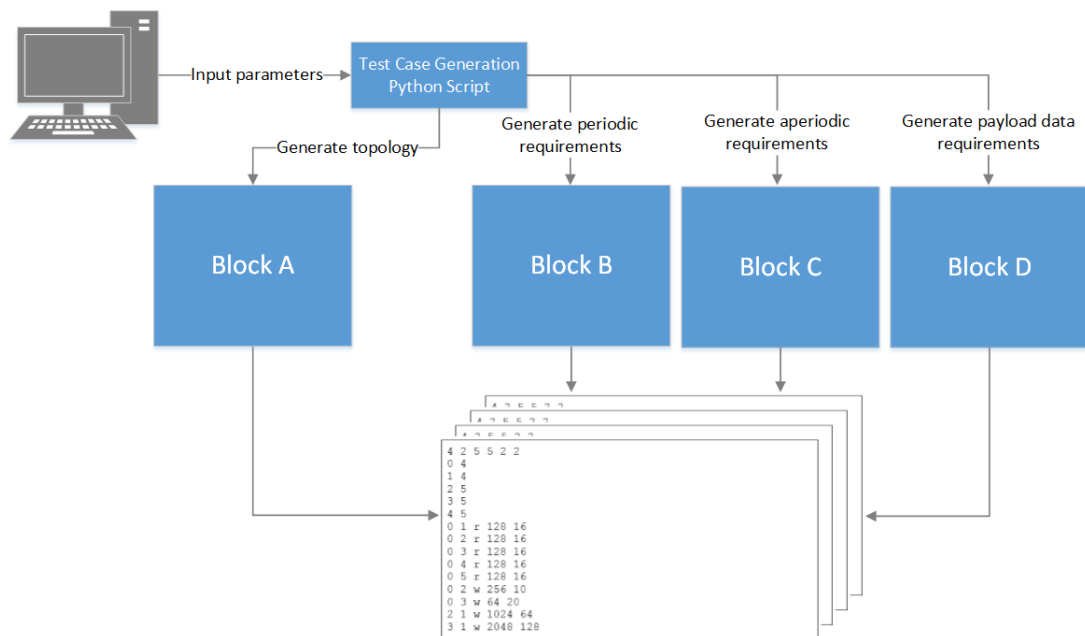


Figure 8-4: Test Case Generation Software Flow

As shown in Figure 8-4, the test case generation software flow begins with a user executing the test case generation Python script with a list of parameters. These parameters are then parsed by the script and passed to each of the algorithms which generate the topology and the periodic, aperiodic and payload data bandwidth requirements. The topology and the requirements are then written to text files, describing each test case in the format described in the following section.

8.1.4 File Format

When the test case generation algorithm is complete, the test case is stored as a plain text file so that it can be easily parsed by the scheduling algorithm. The test case file format is shown in Figure 8-5.

```

nodes routers links periodic aperiodic payload
link_0
...
link_n
periodic_0
...
periodic_n
aperiodic_0
...
aperiodic_n
payload_0
...
payload_n

```

Figure 8-5: Test Case File Format

As shown in Figure 8-5, the file begins with six integers describing the number of nodes, routers, links, periodic, aperiodic and payload requirements. Following the first line, the links and requirements are listed. Each link is a pair of integers describing a bidirectional SpaceWire connection between two numbered devices. The bandwidth requirements consist of five parameters where the first four are common across all types: the initiator, target, RMAP operation and size in bytes. The fifth parameter is

the rate (Hz) for periodic requirements, the deadline (ms) for aperiodic requirements and the packet count for payload data requirements. All parameters are integers except for the RMAP operation with may be “r” or “w” meaning read or write respectively.

An example of a small test case file is shown in Figure 8-6.

```

4 2 5 5 2 2
0 4
1 4
2 5
3 5
4 5
0 1 r 128 16
0 2 r 128 16
0 3 r 128 16
0 4 r 128 16
0 5 r 128 16
0 2 w 256 10
0 3 w 64 20
2 1 w 1024 64
3 1 w 2048 128

```

Figure 8-6: Example Test Case File

In Figure 8-6, the first line indicates that there are four nodes, two routers, five links, five periodic requirements, two aperiodic requirements and two payload data requirements. The nodes are always indexed first, followed by the routers so in this case the four nodes are numbered from 0 to 3 and the two routers are numbered 4 and 5. There are links between each node and one of the routers and the routers are also connected to each other. The corresponding network architecture for the example test case is shown in Figure 8-7.

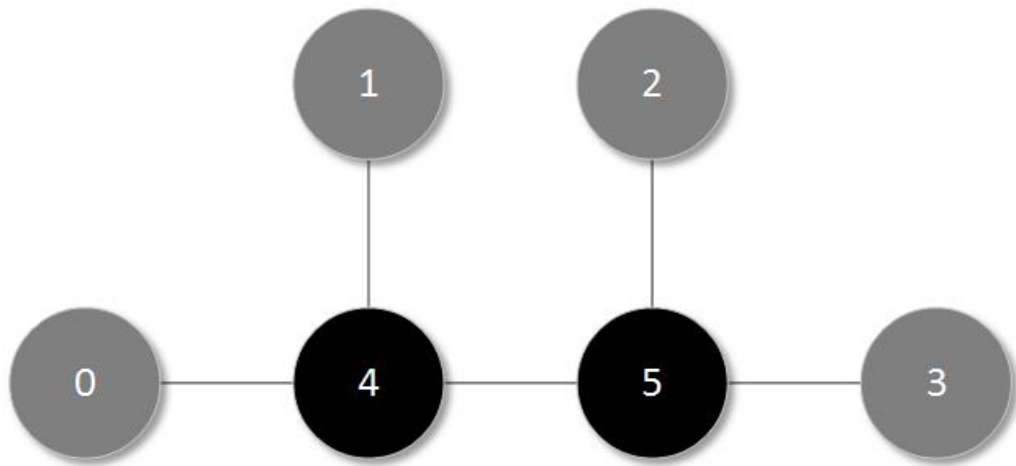


Figure 8-7: Example Test Case Architecture

As shown in Figure 8-7, there are four nodes represented as grey circles and two routers represented as black circles. The five SpaceWire links match those described in lines 2 to 6 of the example test case file in Figure 8-6.

The periodic requirements show that node 0 is reading 128 bytes of housekeeping information from the other devices, including the routers, at a rate of 16 Hz. Node 0 also has two aperiodic requirements to nodes 2 and 3; the first is writing 256 bytes of data with a 10 ms deadline and the second is writing 64 bytes of data with a 20 ms deadline. Finally, there are two payload data requirements; the first is between nodes 2 and 1 and requires 64 packets of 1024 bytes per second and the second, between nodes 3 and 1, requires 128 packets of 2048 bytes per second.

Using a standard file format to describe each test case allows the ability to easily add additional test cases based on real or proposed missions rather than random test cases.

8.2 Experimental Setup

The experiments in this chapter were run on a Lenovo ThinkPad laptop with an Intel Pentium B960 processor running at 2.2 GHz, with 4 Gbytes of RAM on top of the Windows 7 64-bit operating system. The test generation algorithm and the scheduling

algorithms were implemented using the Python 3 programming language and the open-source PyCharm Community Edition IDE from JetBrains (JetBrains 2015).

Test cases were generated assuming that the SpaceWire link speed is 200 Mbit/s across the entire network and that the control cycle is 16 Hz. There are three groups of test cases: small, medium and large, with ten random test cases in each group. The input parameters for the test case generation algorithm for each class are listed in Table 8-1.

Table 8-1: Test Case Classes

Class	Nodes	Routers	Periodic	Aperiodic	Payload
Small	16	6	16	8	16
Medium	32	12	32	16	32
Large	64	24	64	32	64

In each class, ten test cases were randomly generated with randomised links between the nodes and routers, as described in Section 8.1.1. Periodic, aperiodic and payload requirements were then randomly generated between randomly selected nodes in the network, as described in Section 8.1.2. Each experiment ran their scheduling strategies on all test cases and the results were written to plain text files.

8.3 Results

This section describes each experiment and provides the results from scheduling the test cases using the algorithms and techniques described in the previous sections.

8.3.1 Experiment 1: BFS Path Selection

In this experiment, paths are selected for each bandwidth requirement using the BFS algorithm to determine the shortest path, based on number of links, between each required initiator/target pair. Once the paths have been selected, the cost of each bandwidth requirement's transactions are determined by calculating the worst-case

RMAP transaction execution time using the path information. Next, the conflict graph is built by pair-wise checking each bandwidth requirement and adding a conflict if two requirements have different initiators and share one or more SpaceWire links. After this initial processing, the periodic and aperiodic transactions are allocated first, using the algorithms from Sections 7.2.2.2 and 7.2.2.3. Finally, the payload data bandwidth requirements are translated into bin packing items by using the RMAP execution time as the cost and the packet count as the number of instances of each item. The items are then allocated using the bin packing heuristics as described in Section 7.2.2.4.

8.3.1.1 Results

The results from running the first experiment on all 30 test cases using the FF, BF and LC heuristics are listed in Table 8-2.

Table 8-2: Experiment 1: BFS Path Selection Results

Test Case	Conflicts	Total Slots Used			Payload Slots		
		FF	BF	LC	FF	BF	LC
small_000	107	(60)	61	61	(19)	20	19
small_001	94	(61)	61	61	(12)	12	12
small_002	131	(61)	61	61	(21)	22	21
small_003	207	(61)	61	61	(13)	13	13
small_004	87	(64)	64	64	(14)	14	14
small_005	117	(64)	64	64	(15)	16	15
small_006	76	(64)	64	64	(14)	15	14
small_007	135	(64)	64	64	(19)	20	19
small_008	114	(61)	63	63	(14)	15	14
small_009	126	(61)	61	61	22	(21)	22
medium_000	550	(64)	66	66	(42)	44	44
medium_001	470	(64)	69	69	(32)	36	36
medium_002	371	(64)	65	65	24	25	(23)
medium_003	389	(64)	64	64	(38)	41	38
medium_004	386	(64)	64	64	(33)	33	33

Test Case	Conflicts	Total Slots Used			Payload Slots		
		FF	BF	LC	FF	BF	LC
medium_005	454	(61)	61	61	(42)	43	42
medium_006	508	(65)	66	65	(49)	50	49
medium_007	403	(64)	65	65	(29)	29	29
medium_008	367	(61)	61	61	(19)	21	19
medium_009	501	(64)	64	64	(31)	31	31
large_000	1388	(64)	64	64	(39)	42	39
large_001	1509	(64)	64	64	37	38	(36)
large_002	1076	(64)	68	68	(36)	39	39
large_003	1433	(64)	66	66	(36)	37	38
large_004	1559	78	(77)	78	64	(61)	63
large_005	964	(64)	64	64	(30)	33	32
large_006	1220	(64)	71	71	(43)	49	49
large_007	1247	(68)	68	68	64	(63)	63
large_008	1350	(64)	73	71	(41)	49	47
large_009	1102	(64)	66	66	(43)	43	43

In Table 8-2, each row describes the results of the experiment on the test case listed in the first column. The second column lists the number of edges in the conflict graph which describes if different initiator/target pairs share one or more SpaceWire links and therefore their transactions can't be allocated within the same time-slot. The third column is a group of three columns, one for each bin packing heuristic, which lists the total number of slots required to schedule all of the bandwidth requirements. Finally, the fourth column lists, for each heuristic, the number of slots containing one or more payload data transactions. The parenthesised results show the best result for the total number of slots and the number of payload slots.

8.3.1.2 Analysis

The number of conflicts increases as the instances grow larger, as expected due to the larger number of initiator/target pairs used by the periodic, aperiodic and payload data

bandwidth requirements. This figure is included in the results to compare how the path selection affects the number of conflicts between the BFS path selection algorithm and the weighted search path selection algorithm in the next experiment. If the number of conflicts can be reduced then the number of initiator/target pairs that can be scheduled at the same time can be increased, which may increase the quality of the schedule by reducing the number of slots required.

Looking at the total slots used columns shows that there are several test cases in which one or more heuristic didn't find a suitable solution using the BFS path selection within 64 time-slots or less. Additionally, there are three test cases in which no heuristic found a suitable solution using the BFS path selection: medium_006, large_004 and large_007.

Although the total number of slots used is a useful metric to determine if a suitable solution is found, it doesn't describe much about the quality of the scheduled transactions, most of which are payload data transactions. For example, if a test case contained one aperiodic bandwidth requirement with a tight deadline, the schedule may use many slots but most of which would contain only one transaction. Therefore, the number of slots that contain one or more payload transactions is included as an additional quantitative measure of the quality of the schedule.

Looking at the best results for total slots used in each test case, as indicated by the parenthesised values or those that match the value, shows that the heuristics perform similarly in many cases in this experiment with the FF heuristic being able to find the best total slots used result in most of the cases, although it is matched by at least one of the other heuristics in 18 out of 29 cases and outperformed by the BF heuristic in the large_004 test case.

The best results for the number of payload slots used shows that, again, the FF heuristic manages to find the best result in many of the cases, being matched by the BF or LC heuristic in 18 out of 25 cases and outperformed by the BF or LC heuristic in the small_009, medium_002, large_001, large_004 and large_007 cases.

8.3.1.3 Detailed Result Description

This section provides a more detailed description of this experiment's results for one of the test cases, small_000, which contains 16 nodes, 6 routers, 28 links, 16 periodic requirements, 8 aperiodic requirements and 16 payload data requirements. The topology of the network is shown in Figure 8-8.

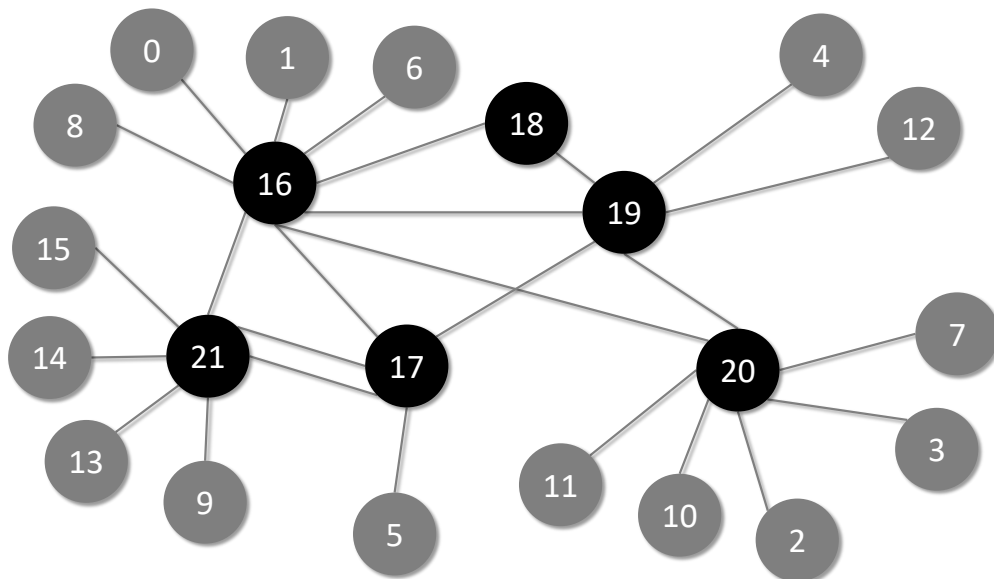


Figure 8-8: Network Topology for Test Case (small_001)

After running the BFS path selection, the paths for each initiator/target pair present in the complete list of bandwidth requirements are listed in Table 8-3.

Table 8-3: Paths for Test Case (small_000)

Initiator	Target	Path
2	16	$2 \rightarrow 20 \rightarrow 16$
1	3	$1 \rightarrow 16 \rightarrow 20 \rightarrow 3$
7	1	$7 \rightarrow 20 \rightarrow 16 \rightarrow 1$
1	13	$1 \rightarrow 16 \rightarrow 21 \rightarrow 13$
15	5	$15 \rightarrow 21 \rightarrow 17 \rightarrow 15$
0	7	$0 \rightarrow 16 \rightarrow 20 \rightarrow 7$
0	12	$0 \rightarrow 16 \rightarrow 19 \rightarrow 12$
11	0	$11 \rightarrow 20 \rightarrow 16 \rightarrow 0$
0	16	$0 \rightarrow 16$
1	4	$1 \rightarrow 16 \rightarrow 19 \rightarrow 4$
6	12	$6 \rightarrow 16 \rightarrow 19 \rightarrow 12$
0	14	$0 \rightarrow 16 \rightarrow 21 \rightarrow 14$
7	11	$7 \rightarrow 20 \rightarrow 11$
6	4	$6 \rightarrow 16 \rightarrow 19 \rightarrow 4$
1	11	$1 \rightarrow 16 \rightarrow 20 \rightarrow 11$
1	2	$1 \rightarrow 16 \rightarrow 20 \rightarrow 2$
0	8	$0 \rightarrow 16 \rightarrow 8$
0	15	$0 \rightarrow 16 \rightarrow 21 \rightarrow 15$
1	12	$1 \rightarrow 16 \rightarrow 19 \rightarrow 12$
16	3	$16 \rightarrow 20 \rightarrow 3$
1	14	$1 \rightarrow 16 \rightarrow 21 \rightarrow 14$
0	11	$0 \rightarrow 16 \rightarrow 20 \rightarrow 11$
0	2	$0 \rightarrow 16 \rightarrow 20 \rightarrow 2$
0	6	$0 \rightarrow 16 \rightarrow 6$
1	8	$1 \rightarrow 16 \rightarrow 8$
1	15	$1 \rightarrow 16 \rightarrow 21 \rightarrow 15$
5	15	$5 \rightarrow 17 \rightarrow 21 \rightarrow 15$
1	9	$1 \rightarrow 16 \rightarrow 21 \rightarrow 9$
0	4	$0 \rightarrow 16 \rightarrow 19 \rightarrow 4$
13	9	$13 \rightarrow 21 \rightarrow 9$
7	6	$7 \rightarrow 20 \rightarrow 16 \rightarrow 6$
0	10	$0 \rightarrow 16 \rightarrow 20 \rightarrow 10$

As listed in Table 8-3, the paths for initiator/target pairs are described as a sequence of nodes from the initiator to the target. Many of the initiator/target pairs share the (16, 20) edge between the two routers, preventing them from being allocated within the same time-slot. As the BFS path selection uses the lowest cost path with respect to the number of links, the other possible routes between routers 16 and 20, such as (16, 18, 19, 20), (16, 19, 20) and (16, 17, 19, 20) are not used even though they would allow concurrent initiator/target node pairs that require a path between routers 16 and 20. Similar situations arise between other initiator/target pair paths with a shared router to router link, such as the (16, 21) or (16, 19) links. The BFS path selection algorithm also does not utilise multiple links between routers as it does not assign a value to the links. The two initiator/target pairs that use the (17, 21) link in their paths share the same link even though they wouldn't need to if they each used one of the two links between routers 17 and 21.

The bandwidth requirement list along with the transaction allocations found using the FF heuristic, which was the best result for this test case, is shown in Table 8-4.

Table 8-4: Schedule for Test Case (small_000)

Req.	I	T	Size	R/D/C	Allocations
Per. 0	0	16	32	32	(0, 1), (32, 1)
Per. 1	1	8	256	64	(0, 1), (16, 1), (32, 1), (48, 1)
Per. 2	1	9	32	16	(1, 1)
Per. 3	0	10	32	64	(0, 1), (16, 1), (32, 1), (48, 1)
Per. 4	1	2	32	64	(1, 1), (17, 1), (33, 1), (49, 1)
Per. 5	0	15	128	32	(0, 1), (32, 1)
Per. 6	1	4	32	32	(0, 1), (32, 1)
Per. 7	0	14	256	64	(0, 1), (16, 1), (32, 1), (48, 1)
Per. 8	0	7	256	64	(0, 1), (16, 1), (32, 1), (48, 1)
Per. 9	1	11	64	64	(1, 1), (17, 1), (33, 1), (49, 1)
Per. 10	1	14	64	32	(1, 1), (17, 1)

Req.	I	T	Size	R/D/C	Allocations
Per. 11	0	12	256	32	(1, 1), (17, 1)
Per. 12	0	11	32	64	(0, 1), (16, 1), (32, 1), (48, 1)
Per. 13	1	12	256	16	(0, 1)
Per. 14	0	6	128	64	(0, 1), (16, 1), (32, 1), (48, 1)
Per. 15	0	4	128	16	(1, 1)
Ape. 0	1	4	64	15	(3, 1), (15, 1), (28, 1), (42, 1), (55, 1)
Ape. 1	1	3	32	10	(2, 1), (11, 1), (20, 1), (29, 1), (38, 1), (47, 1), (56, 1)
Ape. 2	1	13	64	20	(2, 1), (21, 1), (40, 1), (59, 1)
Ape. 3	0	4	32	15	(2, 1), (14, 1), (27, 1), (41, 1), (54, 1)
Ape. 4	1	12	32	15	(3, 1), (15, 1), (28, 1), (42, 1), (55, 1)
Ape. 5	0	6	64	20	(2, 1), (21, 1), (40, 1), (59, 1)
Ape. 6	0	8	32	15	(2, 1), (15, 1), (29, 1), (43, 1), (57, 1)
Ape. 7	0	12	128	10	(2, 1), (11, 1), (20, 1), (29, 1), (38, 1), (47, 1), (56, 1)
Pay. 0	0	2	512	512	(3, 23), (4, 9)
Pay. 1	13	9	512	304	(0, 19)
Pay. 2	7	11	1024	496	(2, 14), (3, 14), (4, 3)
Pay. 3	2	16	1024	112	(5, 7)
Pay. 4	7	6	4096	256	(6, 4), (7, 4), (8, 4), (9, 4)
Pay. 5	1	2	4096	240	(10, 4), (12, 4), (13, 4), (14, 3)
Pay. 6	5	15	2048	224	(1, 7), (2, 7)
Pay. 7	0	14	4096	368	(4, 2), (5, 4), (6, 4), (7, 4), (8, 4), (9, 4), (10, 1)
Pay. 8	6	12	1024	256	(4, 14), (5, 2)
Pay. 9	16	3	1024	416	(15, 14), (18, 12)
Pay. 10	15	5	4096	192	(3, 4), (4, 4), (5, 4)
Pay. 11	1	15	1024	352	(14, 4), (18, 14), (19, 4)
Pay. 12	11	0	512	384	(19, 23), (22, 1)
Pay. 13	1	13	1024	128	(19, 8)
Pay. 14	7	1	512	336	(6, 1), (7, 1), (8, 1), (9, 1), (23, 17)
Pay. 15	6	4	2048	368	(5, 6), (10, 7), (12, 7), (13, 3)

In Table 8-4, each row lists a bandwidth requirement name, either a periodic, aperiodic or payload requirement; the initiator/target pair involved; the size of the transaction in

bytes; the rate, deadline, or packet count; and a list of transaction allocations. The allocations are in the format (S, N) , where S is the time-slot and N is the number of transactions allocated.

As the control cycle is 16 Hz, each bandwidth requirement needs to have $1/16^{\text{th}}$ of its required transactions scheduled during each epoch. For example, the periodic requirements with a rate of 64 Hz need to have four transactions evenly spaced throughout the schedule and the payload requirements need to have $1/16^{\text{th}}$ of their transactions allocated to any slots within the schedule. Consider payload requirement 0, which has a packet count of 512 packets per second. It has been allocated 23 of its packets to time-slot 3 and 9 of its packets to time-slot 4, giving a total of 32 packets per schedule epoch. Multiplied by the number of schedule epochs per second, 16, this results in 512 payload data packets per second, satisfying the payload data bandwidth requirement.

8.3.2 Experiment 2: Weighted Search Path Selection

In this experiment, paths are selected using a modified version of Dijkstra's algorithm to find the lowest cost paths between initiators and targets. The algorithm has been modified to take into account that the network topology might be a multigraph i.e. there may be multiple links between two routers. Each link is given a unique label and a separate weight value and when the cheapest edge between two vertices is being determined, if there are multiple edges, the edge with the lowest cost is always selected. The other modification to the algorithm is the addition of dynamic penalties, added to the weight of edges that are selected in a path between an initiator and a target. This encourages the selection of non-conflicting, but possibly longer, paths rather than shorter paths that may share one or more links. After the paths are selected, the experiment executes in the same manner as the first experiment.

8.3.2.1 Results

The results from running the second experiment on all 30 test cases using the FF, BF and LC heuristics and a dynamic penalty value of 0.25 are shown in Table 8-5.

Table 8-5: Experiment 2: Weighted Search Path Selection Results

Test Case	Conflicts	Total Slots Used			Payload Slots		
		FF	BF	LC	FF	BF	LC
small_000	95	(60)	61	61	(15)	16	15
small_001	90	(61)	62	62	(11)	11	11
small_002	102	(61)	61	61	(25)	25	25
small_003	148	(61)	63	63	(13)	14	14
small_004	78	(64)	64	64	(14)	14	14
small_005	107	(64)	64	64	14	(13)	14
small_006	68	(61)	61	61	(13)	14	13
small_007	114	(64)	64	64	(19)	20	19
small_008	89	(61)	61	61	(13)	14	13
small_009	113	(61)	61	61	22	(16)	22
medium_000	412	(64)	66	66	(31)	33	33
medium_001	362	(64)	71	71	(30)	36	36
medium_002	356	(64)	65	65	(21)	21	21
medium_003	342	(64)	64	64	(23)	24	23
medium_004	377	(64)	64	64	(31)	33	31
medium_005	366	(64)	64	64	(34)	34	34
medium_006	509	(64)	70	70	(50)	55	55
medium_007	352	(64)	65	65	30	30	(29)
medium_008	348	(64)	64	64	(22)	23	22
medium_009	422	(64)	64	64	32	(31)	31
large_000	1072	(64)	64	64	37	37	(36)
large_001	1085	(64)	64	64	29	(28)	29
large_002	868	(64)	64	64	(25)	27	26
large_003	1022	(64)	64	64	(28)	29	28
large_004	1193	(64)	64	64	(40)	40	41
large_005	824	(64)	64	64	(31)	32	31
large_006	959	(64)	75	75	(31)	42	41

Test Case	Conflicts	Total Slots Used			Payload Slots		
		FF	BF	LC	FF	BF	LC
large_007	1014	(64)	70	69	(43)	48	47
large_008	1126	(64)	67	66	(41)	43	42
large_009	980	(64)	64	64	38	(37)	38

8.3.2.2 Analysis

It can be seen, by looking at the total used slots columns, that there is an acceptable solution found for the test cases that the first experiment failed on: medium_006, large_004 and large_007. There are two test cases in which the second experiment found a solution requiring more total used slots than the first: medium_005 and medium_008.

The number of conflicts has been significantly reduced in many cases due to the weighted search path selection. The ratios of the number of conflicts between the results of the two experiments are shown in Figure 8-9.

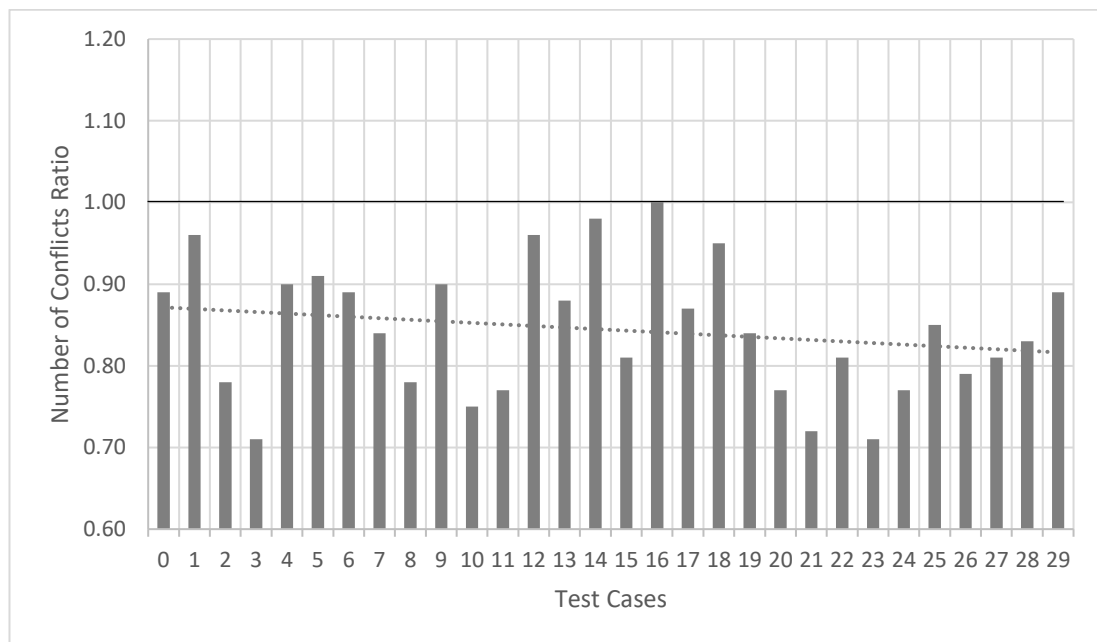


Figure 8-9: Number of Conflicts Ratios

In Figure 8-9, a bar chart shows the performance of the second experiment compared to the first in terms of conflicts. The horizontal line at the y-axis value of 1.00 shows the reference line for the number of conflicts from the first experiment, which used the BFS path selection algorithm. Each test case is then represented as a vertical bar showing the number of conflicts compared to the first experiment. The horizontal axis labels represent the test case number, going from small_000 as label 0 to large_009 as label 29.

Given the number of conflicts in a test case from experiment one c_1 and the number of conflicts from a test case in experiment two c_2 , the value of each bar is defined as $\frac{c_2}{c_1}$. For example, small_000 (labelled as 0) has just under 90% of the number of conflicts compared to the first experiment, and small_003 (labelled as 3) has just over 70%.

The dotted line shows the linear trend line as the test cases go from small to large. In all but one test case, the number of conflicts is reduced, and in over half of the test cases, the weighted search path selection algorithm produces 85% or less conflicts compared to BFS path selection.

Due to the reduction in the number of conflicts between initiator/target pairs, more transactions may be able to be scheduled within the same time-slots, reducing the number of time-slots required for payload data transactions. The ratios of the number of payload slots required between the best results of the two experiments are shown in Figure 8-10.

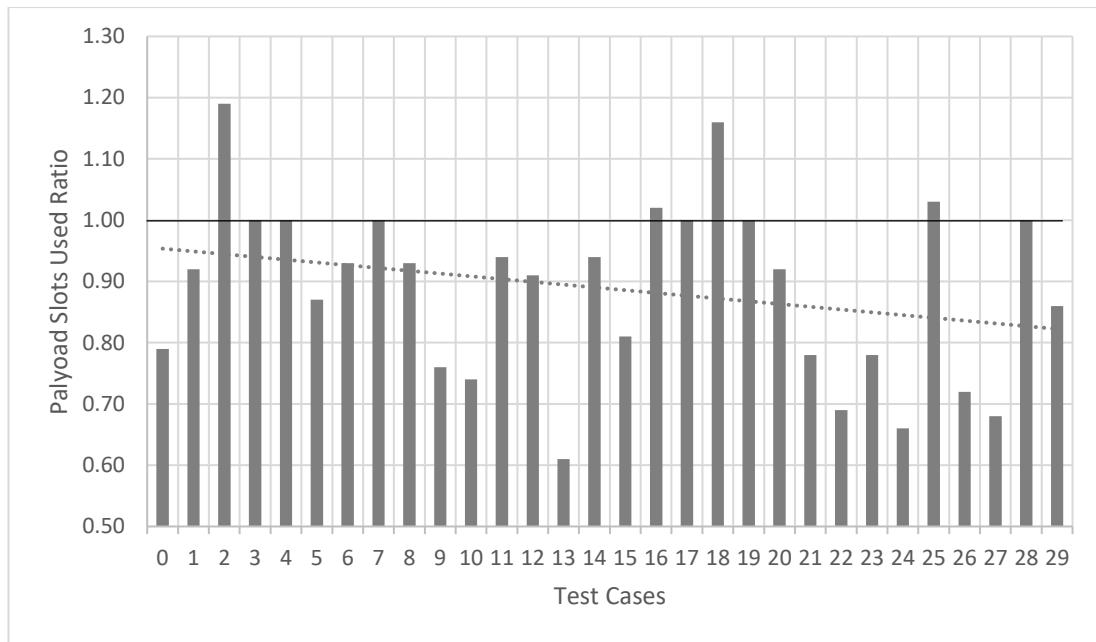


Figure 8-10: Payload Slots Used Ratios

As shown in Figure 8-10, the number of slots containing one or more payload data transactions has been reduced in 20 out of 30 test cases. There are four test cases in which experiment two performed worse than the first experiment, with regards to payload slots used, and six cases in which the number of payload slots used are the same for both experiments. The dotted line shows the linear trend line as the test cases go from small to large.

To explain why the four test cases performed worse than in the first experiment, the results of the third test case, which had the worst results compared to the first experiment, can be analysed. This test case required 25 payload slots in the second experiment compared to 21 in the first. In the first experiment, the paths used by payload data bandwidth requirements 2 and 3, which have high data rates, do not conflict. However, although the total number of conflicts is lower in the second experiment, the paths used by these requirements in the second experiment do conflict. In the first experiment, payload data bandwidth requirements 2 and 3 use time-slots 5

to 11 and 7 to 14, respectively. However, in the second experiment, requirement 2 uses time-slots 5 to 11 and requirement 3 uses time-slots 12 to 14 and 17 to 21. In addition, the paths used by payload bandwidth requirements 3 and 8, which also have high data rates, do not conflict in the first experiment but do in the second. These requirements have an effect on the schedule because they are high data rate requirements that have many allocated time-slots. This increases the number of slots required to satisfy these high data rate requirements, resulting in a schedule that requires a higher overall number of time-slots allocated to payload data bandwidth requirements.

These results imply that, in some cases, a more intelligent path selection strategy is required. A load balancing path selection heuristic which attempts to meet this requirement by taking into account the data rates of bandwidth requirements when adding dynamic penalties to links is explored in Section 8.4.4.3.

The dynamic penalty value used when executing the weighted search path selection algorithm can be experimented with to measure its effects. The results above use a dynamic penalty value of 0.25, which adds a small penalty to each shared link whenever it is selected in the path between an initiator and a target. To more heavily penalise the selection of shared links, higher dynamic penalty values can be used. Figure 8-11 shows the payload slots used ratios for penalty values of 0.25, 3 and 10.

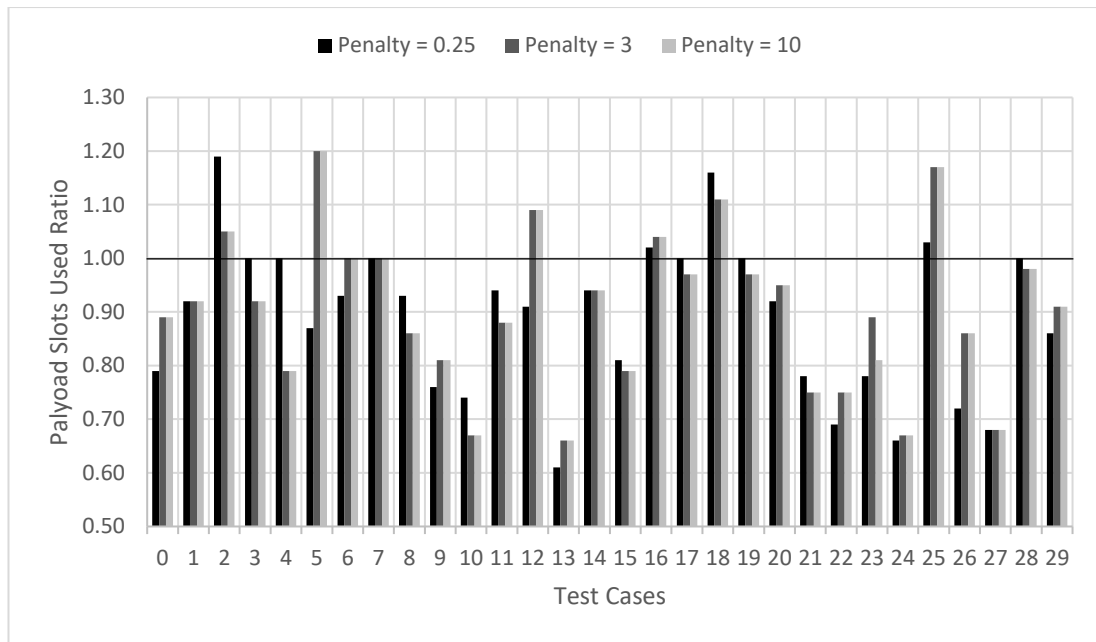


Figure 8-11: Experimenting with Dynamic Penalty Values

In Figure 8-11, the black series shows the values from the original execution of experiment two, and the dark-grey and light-grey series show the results when using dynamic penalty values of 3 and 10, respectively. Using the new penalty values does not provide consistent improvements in the results but they do find new best results in 10 out of the 30 test cases. Additionally, in test cases 3, 4, 17, 19 and 28, where the original dynamic penalty value of 0.25 was not able to improve upon experiment one's results, the new values found better solutions.

Using different dynamic penalty values changes how the algorithm selects paths. If a low penalty value, e.g. 0.25, is added to a link then the path selection algorithm will favour a path of non-penalised links with a length of up to one SpaceWire link longer than a path including the penalised link. This is advantageous if, for example, there are multiple links between two routers which can be used to split traffic. Each path selected will toggle between using each of the links. This is exactly what occurs in the case study of the JUICE mission in Section 8.4, where there are two links between a

router and the SSMM. If a high penalty value, e.g. 3, is added to a link then the path selection algorithm will favour a path of non-penalised links with a length of up to three SpaceWire links longer than a path including the penalised link. This may be advantageous in some cases, as shown by the results in Figure 8-11, but it is dependent on the network architecture. In other cases, the selection of a longer path may introduce more conflicts between initiator/target pairs which will have a detrimental effect on the resulting schedule. Therefore, either the network architecture should be analysed to determine a good dynamic penalty value or a range of dynamic penalty values should be experimented with in order to find the best result for the specific network architecture.

8.4 Case Study: JUPiter ICy moons Explorer (JUICE)

The JUPiter ICy moons Explorer, known as JUICE, is an ESA mission under the Cosmic Vision Program that aims to study Jupiter and three of its icy moons: Ganymede, Callisto and Europa (Grasset, et al. 2013). The scientific purpose of the mission is to investigate if the moons are able to support life. The mission has a proposed launch date of 2022 and would enter orbit around Ganymede in 2033.

8.4.1 Network Topology

JUICE uses SpaceWire to connect the payload instruments, OBC, SSMM and downlink telemetry and the network handles telecommands, telemetry and payload data (Airbus Defence & Space 2015). The architecture of the SpaceWire network used in the JUICE mission is illustrated in Figure 2-15.

As shown in Figure 2-15, the SpaceWire network consists of ten instruments, an OBC with a redundant processor, SSMM and downlink telemetry, connected via a SpaceWire router. Each instrument is connected to the router with a pair of SpaceWire

links, one nominal and one redundant, except for the PEP instrument that has two pairs of SpaceWire links. For simplicity, the redundant links are not shown in the network architecture in Figure 2-15. There is a pair of links between each OBC processor and the router, and two pairs of links between the SSMM and the router. The SSMM is connected to the two sets of X and Ka band downlink telemetry transfer frame generators via pairs of point-to-point links.

The link speeds are not uniform across the network, with the OBC and SSMM running their links at 128 Mbit/s, the JANUS, MAJIS and SWI instruments running their links at 100 Mbit/s and the remaining instruments running theirs at 40 Mbit/s.

8.4.2 Time-Slot Duration

Each instrument has the capability to buffer up to 100 ms worth of their peak payload data generation. Therefore, the schedule epoch needs to be less than or equal to 100 ms in order for an instrument to be able to buffer a full epoch's worth of data. This is due to the possibility that the scheduler may allocate an instrument's transactions in a number of time-slots in close locality, requiring the instrument to buffer payload data until the next available slot in which it can send. This is a current limitation of the scheduling algorithm implementation which should be updated in the future to attempt to spread the payload data bandwidth requirement's allocated time-slots throughout the schedule in order to minimise the buffering requirements.

A time-slot also needs to be long enough to execute the longest transaction without overlapping with the next slot. In this case, as explained in the next section on payload data bandwidth requirements, the longest transaction is a 4096 byte RMAP write command over a path where the slowest link speed is 40 Mbit/s. The initiator processing time, target response times and router switching times are not available in

the JUICE communications scenario document (Airbus Defence & Space 2015) so the same values as the SpaceWire-D Demonstrator are used. This gives a longest transaction execution time of 1043.05 μs over the 40 Mbit/s paths and 424.3 μs when using the 100 Mbit/s paths. A time-slot interval of 976.5625 μs , as in the previous experiments, is no longer suitable because the longest transaction won't fit within a time-slot.

Therefore, a time-slot interval of 1562.5 μs was chosen, giving a schedule epoch of 100 ms and 10 schedule epochs per second. This allows the instrument buffering limitation to be satisfied and the time-slot duration is long enough to fit one of the largest transactions at 40 Mbit/s or three at 100 Mbit/s.

8.4.3 Bandwidth Requirements

In this section, the periodic, aperiodic and payload data bandwidth requirements for the JUICE mission are described as listed in the JUICE communications scenario document (Airbus Defence & Space 2015).

8.4.3.1 Periodic

Each instrument, except for RADEM, sends housekeeping telemetry to the OBC at a rate of two 128 byte packets per second. As the schedule is executing at a rate of 10 Hz, this means the bandwidth requirements must be over-allocated and use one time-slot in each schedule epoch which would allow up to 10 housekeeping packets per second. In this case, as the periodic rate is low compared to the number of schedule epochs per second, this results in wasted bandwidth in the 8 out of 10 schedule epochs per second in which the instruments don't send housekeeping telemetry to the OBC. The periodic bandwidth requirements for the JUICE network are listed in Table 8-6.

Table 8-6: JUICE Periodic Bandwidth Requirements

Initiator	Target	Operation	Size (B)	Rate (Hz)
JANUS	OBC	Write	128	10
GALA	OBC	Write	128	10
J-MAG	OBC	Write	128	10
RIME	OBC	Write	128	10
SWI	OBC	Write	128	10
UVS	OBC	Write	128	10
MAJIS	OBC	Write	128	10
RPWI	OBC	Write	128	10
PEP-NU/ZU	OBC	Write	128	10

As shown in Table 8-6, there are nine periodic bandwidth requirements between the instruments and the OBC with the same size of 128 bytes and a rate of 10 Hz.

8.4.3.2 Aperiodic

Telecommands are sent by the OBC to the instruments at a peak rate of five commands per second. Again, the aperiodic requirements must be over-allocated due to the schedule rate of 10 Hz, resulting in wasted bandwidth in the 5 out of 10 schedule epochs per second in which the OBC doesn't send commands to the instruments. Aperiodic bandwidth requirements are normally suited to dynamic buses; however, because there is only one time-slot required for each of the OBC to instrument aperiodic bandwidth requirements, the aperiodic requirements can be simplified using static buses. Therefore, each aperiodic bandwidth requirement can be transformed into a periodic bandwidth requirement with a rate of 10 Hz as listed in Table 8-7.

Table 8-7: JUICE Transformed Aperiodic Bandwidth Requirements

Initiator	Target	Operation	Size (B)	Rate (Hz)
OBC	JANUS	Write	128	10
OBC	GALA	Write	128	10
OBC	J-MAG	Write	128	10
OBC	RIME	Write	128	10
OBC	SWI	Write	128	10
OBC	UVS	Write	128	10
OBC	MAJIS	Write	128	10
OBC	RPWI	Write	128	10
OBC	PEP-NU/ZU	Write	128	10
OBC	RADEM	Write	128	10

As shown in Table 8-7, there are ten aperiodic bandwidth requirements between the instruments and the OBC. There is no information about the size of the telecommand packets within the communication scenario document so the same size as the telemetry packets, 128 bytes, is used with a rate of 10 Hz.

8.4.3.3 Payload Data

There are nine payload data bandwidth requirements between the instruments and the SSMM. Each requirement consists of RMAP write transactions with a data length of 4096 bytes and a packet count between 10.6 and 310 packets per second. As the schedule is executing at a rate of 10 Hz, the packet count of each payload data bandwidth requirement is increased to the closest multiple of 10. The JUICE communications scenario document lists the average and peak throughput of the instrument to SSMM payload telemetry. In a SpaceWire-D network, the schedule must satisfy the bandwidth requirements in the worst-case, which would be the peak throughput. The payload data bandwidth requirements for the worst-case throughput of the JUICE mission are listed in Table 8-8.

Table 8-8: JUICE Peak Throughput Payload Data Bandwidth Requirements

Initiator	Target	Operation	Size (B)	Packet Count
JANUS	SSMM	Write	4096	310
GALA	SSMM	Write	4096	20
JMAG	SSMM	Write	4096	70
RIME	SSMM	Write	4096	310
SWI	SSMM	Write	4096	20
UVS	SSMM	Write	4096	20
MAJIS	SSMM	Write	4096	1500
RPWI	SSMM	Write	4096	20
PEP-NU/ZU	SSMM	Write	4096	40

As shown in Table 8-8, there are nine payload data bandwidth requirements, most of which have relatively low packet counts except for the JANUS, MAJIS and RIME instruments which have higher packet counts of 310, 1500 and 310 packets per second, respectively.

8.4.4 Scheduling

This section describes the process of scheduling the JUICE mission's bandwidth requirements using the results from the experiments described earlier in the chapter.

8.4.4.1 Path Selection

There are two paths between the router and the SSMM, and as the links between the instruments and the router are not shared by any other initiator/target pairs this allows for the scheduling of two different bandwidth requirements within the same time-slot if they each use one of the router to SSMM links. Therefore, it makes sense to use the weighted search path selection algorithm so that both links will be considered. When writing the instance file to input to the scheduling algorithm, a single link is used to represent a redundant pair of links in the network topology. This is because, at any time, there will be only a single link active between each node, whether that be the

nominal or redundant link. This prevents the scheduling algorithm from using both the nominal and redundant links at the same time.

Using different dynamic penalty values will have no effect in this network topology because there is no possibility of a longer path being selected for any of the initiator/target pairs as there is only a single router. However, using a dynamic penalty value will still split traffic between the two links connecting the router and the SSMM, so the dynamic penalty value is left at the default of 0.25.

8.4.4.2 Initial Results

The best resulting schedule from running the algorithm on the JUICE network topology and bandwidth requirements, which in this case was found using the FF heuristic, is shown in Table 8-9.

Table 8-9: Initial Result for JUICE Schedule

Type	I	T	Size	R/D/C	Allocations
Periodic	JANUS	OBC	128	10	(0, 1)
Periodic	GALA	OBC	128	10	(1, 1)
Periodic	J-MAG	OBC	128	10	(2, 1)
Periodic	RIME	OBC	128	10	(3, 1)
Periodic	SWI	OBC	128	10	(4, 1)
Periodic	UVS	OBC	128	10	(5, 1)
Periodic	MAJIS	OBC	128	10	(6, 1)
Periodic	RPWI	OBC	128	10	(7, 1)
Periodic	PEP	OBC	128	10	(8, 1)
Periodic	OBC	JANUS	128	10	(9, 1)
Periodic	OBC	GALA	128	10	(9, 1)
Periodic	OBC	J-MAG	128	10	(9, 1)
Periodic	OBC	RIME	128	10	(9, 1)
Periodic	OBC	SWI	128	10	(9, 1)
Periodic	OBC	UVS	128	10	(9, 1)
Periodic	OBC	MAJIS	128	10	(9, 1)

Type	I	T	Size	R/D/C	Allocations
Periodic	OBC	RPWI	128	10	(9, 1)
Periodic	OBC	PEP	128	10	(9, 1)
Payload	JANUS	SSMM	4096	310	(1-8, 3), (10-11, 3), (12, 1)
Payload	GALA	SSMM	4096	20	(0, 1), (2, 1)
Payload	J-MAG	SSMM	4096	70	(0, 1), (13-18, 1)
Payload	RIME	SSMM	4096	310	(1, 1), (4-8, 1), (10-34, 1)
Payload	SWI	SSMM	4096	20	(19, 2)
Payload	UVS	SSMM	4096	20	(3, 1), (35, 1)
Payload	MAJIS	SSMM	4096	1500	(20-69, 3)
Payload	RPWI	SSMM	4096	20	(36, 1), (37, 1)
Payload	PEP	SSMM	4096	40	(70-73, 1)

Table 8-9 shows the time-slots in which each of the transactions for the bandwidth requirements are allocated. In the allocations column, an allocation in the form (A-B, N) means that there are N transactions allocated in every time-slot from A to B inclusive. Looking at the payload data bandwidth requirement transaction allocations shows that this is not a suitable solution as it would need 74 time-slots in which to execute transactions, as shown by the allocation of (70-73, 1) for the PEP to SSMM payload data bandwidth requirement. Therefore, a different strategy is required to successfully schedule the JUICE bandwidth requirements, as described in the following section.

8.4.4.3 Load Balancing Dynamic Penalties

To investigate how the scheduling approach can be improved, the network topology and the bandwidth requirements can be analysed to determine which transactions need to be scheduled differently to generate a valid schedule.

There is a chokepoint in the network topology of the JUICE mission, illustrated in Figure 2-15, between the router and the SSMM. There are two SpaceWire links

available which must be shared between all instrument to SSMM payload data bandwidth requirements.

The most demanding payload data bandwidth requirement is between the MAJIS instrument and the SSMM which requires 1500 transactions per second, each containing 4096 bytes of data. Ideally, this requirement should conflict with as few of the other bandwidth requirements as possible as it requires 150 transactions per schedule epoch spread over at least 50 time-slots. The initiator/target pair paths that conflict with the MAJIS to SSMM pair are listed in Table 8-10.

Table 8-10: MAJIS to SSMM Conflicting Paths

Initiator	Target	Path
MAJIS	SSMM	MAJIS → ROUTER(A) → SSMM(A)
JANUS	SSMM	JANUS → ROUTER(A) → SSMM(A)
J-MAG	SSMM	J-MAG → ROUTER(A) → SSMM(A)
SWI	SSMM	SWI → ROUTER(A) → SSMM(A)
PEP	SSMM	PEP → ROUTER(A) → SSMM(A)

In Table 8-10, a link in the form ROUTER(A) → SSMM(A) means link A connecting the SSMM to the router. There are four other payload bandwidth requirements that share the same router to SSMM link, meaning the other four are using the ROUTER(B) → SSMM(B) link. Although the path selection algorithm attempts to balance the use of the two links evenly based on the number of bandwidth requirements using them, it doesn't take into account the link utilisation of the requirements. This means that the level of traffic going across the two links is very uneven because the MAJIS to SSMM payload data bandwidth requirement, which has a packet count of 1500, is much more demanding than the other requirements. This results in 1940 transactions per second allocated across link A and 370 per second across link B.

To combat this, rather than using a constant dynamic penalty value of 0.25, the packet count can be used as the dynamic penalty value. The effect this has is to increase the cost of a link used by a high demand requirement much more than a low demand requirement. The payload bandwidth requirements can also be reordered by descending packet count so that the MAJIS to SSMM requirement immediately adds to the cost of link A. Using the load balancing dynamic penalties, the new payload data allocations, which were found using the FF heuristic, are listed in Table 8-11.

Table 8-11: JUICE Schedule with Load Balancing Dynamic Penalties

Type	I	T	Size	R/D/C	Allocations
Payload	JANUS	SSMM	4096	310	(1-8, 3), (10-11, 3), (12, 1)
Payload	GALA	SSMM	4096	20	(54-55, 1)
Payload	J-MAG	SSMM	4096	70	(43-49, 1)
Payload	RIME	SSMM	4096	310	(0, 1), (13-42, 1)
Payload	SWI	SSMM	4096	20	(56, 2)
Payload	UVS	SSMM	4096	20	(57-58, 1)
Payload	MAJIS	SSMM	4096	1500	(0-5, 3), (7-8, 3), (10-51, 3)
Payload	RPWI	SSMM	4096	20	(59-60, 1)
Payload	PEP	SSMM	4096	40	(50-53, 1)

As shown in Table 8-11, the algorithm now finds a suitable solution requiring 61 time-slots, compared to 74 time-slots in the previous attempt. It also shows that during the execution of the MAJIS to SSMM transactions in slots 0-5, 7-8 and 10-51, there are more transactions being executed over link B. There are now 1500 transactions using link A and 810 using link B, compared to 1940 and 370 without using load balancing dynamic penalties.

In the previous sections, the scheduling strategy has been shown to successfully generate initiator schedules for randomised test cases of various sizes, showing that it scales to networks of different sizes. It has now been extended to successfully work

for a real mission, showing that it is applicable to real-world scenarios. If a mission can be specified using the format described in Section 7.1, the scheduling tool described in this chapter can be used to attempt to generate initiator schedules. Future work should look at other real missions with a range of architectures to analyse the generality of the tool.

8.5 Load Balancing Dynamic Penalties

The test cases of the experiments described earlier in the chapter can be revisited with the new load balancing dynamic penalties and requirement ordering strategies that were found to give good results in the JUICE case study. Figure 8-12 shows the results when using the new strategies compared to the best results from the second experiment.

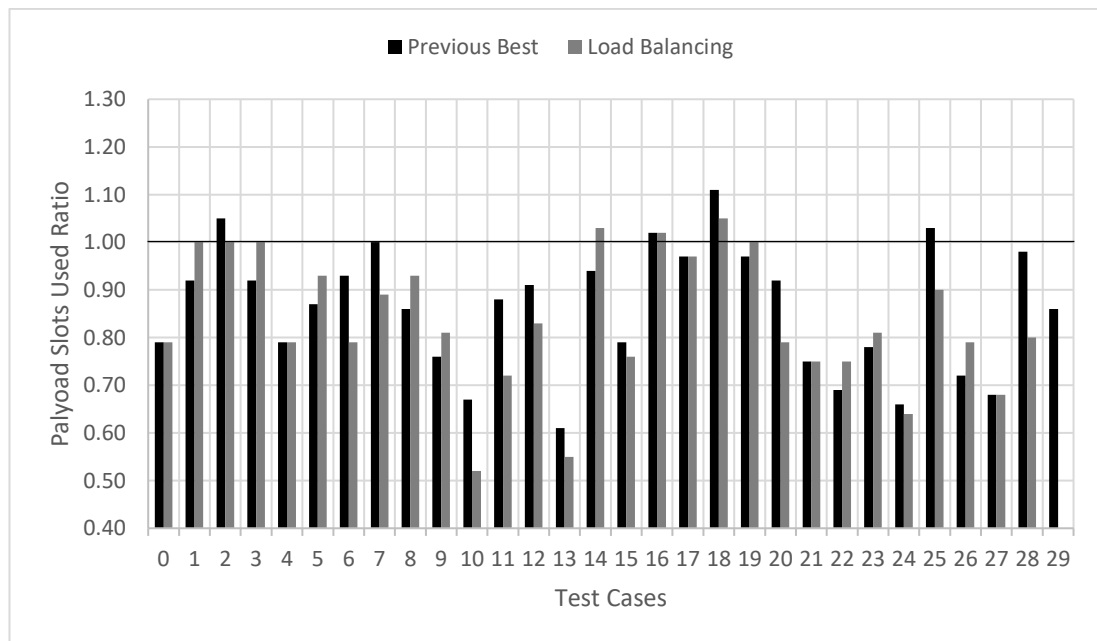


Figure 8-12: Load Balancing Dynamic Penalties

As shown in Figure 8-12, where the black series is the best results found previously from experiment two, using load balancing dynamic penalties finds a new best result

for the number of payload slots used in 11 out of 30 cases. No suitable solution was found in 64 or less time-slots for test case 29, large_009, as it required 65 time-slots.

8.6 Summary

In this chapter, two experiments were performed using the path selection algorithms and the transaction scheduling algorithms described in Chapter 7. The algorithms were run on 30 test cases in three different size classes: small, medium and large and the results were presented and analysed.

Following the experiments, the JUICE mission was used as a case study for the scheduling algorithms. The bandwidth requirements and network architecture were taken from the JUICE communications scenario document (Airbus Defence & Space 2015). During the case study, a new strategy was explored that re-ordered the payload data bandwidth requirements by decreasing data-rate and used a load balancing heuristic when performing the path selection process. This allowed the scheduling algorithm to find a suitable solution which satisfied the bandwidth requirements of the JUICE mission.

Finally, the original test cases were revisited using the load balancing heuristic found during the JUICE case study and the results were presented which performed better than the previous heuristics in 11 of the 30 test cases.

This chapter has proven that the novel scheduling strategy described in Chapter 7 can successfully schedule SpaceWire-D networks. The strategy has been shown to be scalable by performing experiments using randomised test cases of increasing size classes. In addition, the applicability of the scheduling strategy to real-world scenarios has been demonstrated by extending it with a new heuristic to successfully schedule the bandwidth requirements of the JUICE mission.

Chapter 9

Future Work

In this chapter, several avenues for future work are described. These avenues include improvements to the efficiency of the SpaceWire-D software layer described in Chapter 4, a possible hardware implementation of a SpaceWire-D controller, additional FDIR capabilities for SpaceWire-D and research into scheduling mechanisms for SpaceFibre, the next generation of on-board data handling.

9.1 SpaceWire-D Efficiency Improvements

Although the software overhead at the start of each time-slot in the SpaceWire-D layer was reduced to a usable level, it is possible to make it more efficient, depending on the SpaceWire hardware available to the initiator.

The initiators described in this thesis are based on a LEON2-FT based processor board which contains three individual SpaceWire protocol engines, giving a total of three RMAP initiators. In the SpaceWire-D layer described in Chapter 4, only one of the initiators is used to execute the RMAP transactions at the start of each time-slot.

As described in Section 4.3.4, there are three stages to the RMAP dispatching process: the pre-execution stage which checks for any outstanding transactions that need to be cancelled and saves the status values of the previous time-slots transactions, the execution stage which hands a group of RMAP transactions to the initiator for execution and the post-execution stage which checks the status values for errors and

prepares any asynchronous or packet buses allocated to the subsequent time-slot. The RMAP initiator used in the LEON2-FT based processor board updates the status of each executed transaction in an encoder and decoder descriptor. These descriptors are read at the start of each time-slot to determine if each transaction had an encoder or decoder error. If there is no decoder error but the transaction is still incomplete, it must be cancelled by the initiator before the next group of transactions can be executed.

If more than one RMAP initiators are used, the first can check for any errors or outstanding transactions which need to be cancelled after the second initiator has already started executing its group of transactions. This would reduce the software overhead to the minimum by removing the pre-execution stage completely and moving it to the post-execution stage. However, this would also mean that each initiator device would require multiple logical addresses in order to de-multiplex RMAP reply packets and filter them to the correct initiator. This would slightly increase the complexity of the network and would be a trade-off with the potential performance gain.

9.2 SpaceWire-D Hardware Controller

Another possibility for improving the efficiency of the SpaceWire-D layer is to design a hardware SpaceWire-D controller in order to remove the software overhead completely. With a hardware controller, the user application would have the same interface to the SpaceWire-D API functions such as opening, loading and closing virtual buses. However, the implementation of these functions would interact with registers used to configure and control the SpaceWire-D controller rather than use data structures and algorithms in software. The hardware would need to be signalled when a time-code arrives at the board containing the SpaceWire-D controller, which would then execute any transactions loaded into the virtual bus, if any, allocated to the time-

slot. Research could be done to investigate how the SpaceWire-D controller could be designed in hardware and how it compares to a software implementation.

9.3 SpaceWire-D FDIR

The SpaceWire-D standard defines the detection of errors such as RMAP reply errors or transaction incomplete errors. The standard says that these errors should be reported to the network manager for handling, the meaning of which is mission-specific and left to the system designer. In the SpaceWire-D Demonstrator, the errors are compiled into a list of 32-bit error descriptors and sent to the network manager in time-slot 63 of each schedule epoch via an RMAP write to a specific address in one of the network manager's RMAP targets. The errors are displayed to the user but no further action is taken to attempt to isolate and recover from the errors.

A possible path for future work would be to research methods for adding an FDIR layer on top of, or integrated into, SpaceWire-D in order to isolate and recover from encoder, decoder, incomplete transaction or other types of errors.

9.4 SpaceWire-D On-Board Scheduler

In Chapter 7, scheduling mechanisms were described to transform a list of mission bandwidth requirements, a network topology and network parameters into a network schedule. However, this was not prototyped in the SpaceWire-D Demonstrator as a scheduling layer on top of the SpaceWire-D layer.

Future work could be undertaken to build a scheduling layer that takes the schedule information and uses it to configure the schedule and load transactions into the correct virtual buses at the correct times. The user application would then simply hand transactions to the scheduling layer without having to concern itself with which virtual buses to use.

9.5 SpaceFibre Scheduling

SpaceFibre is the next generation of on-board data handling networks. It provides high data-rate payload data-handling, built-in QoS and FDIR and runs over electrical and fibre-optic cables and transceivers (Parkes, McClements and McLaren, et al. 2015). The built-in QoS includes scheduling of virtual channels (VC), prioritised traffic and bandwidth reservation. It provides all of the features of SpaceWire-D built into the protocol and hardware directly without the software overhead and it is backwards compatible with existing SpaceWire networks.

The built-in QoS allows a lower priority VC to use bandwidth allocated to a higher priority VC if the higher priority VC has nothing to transmit. This means that unused pre-allocated bandwidth isn't wasted like in a SpaceWire-D network. The traffic is also split up into frames so that if a high-priority VC does require access to the link, it is only delayed by the time it takes to finish transmitting the current lower-priority frame rather than waiting for an arbitrary length packet to finish transmission.

Research could be done into scheduling mechanisms and algorithms for SpaceFibre applications to combine payload and critical traffic on the same network using the QoS features of SpaceFibre and comparing it to SpaceWire-D and the scheduling mechanisms presented in Chapter 7.

9.6 Summary

This chapter has described some areas for future work to improve the efficiency of the SpaceWire-D layer, to investigate a possible hardware version of a SpaceWire-D controller, to research more advanced FDIR capabilities and also to look at scheduling mechanisms for SpaceFibre, the next generation of on-board data handling networks.

Chapter 10

Conclusions

In this chapter, this thesis is concluded by summarising the results gathered whilst answering the research questions and describing the novel contributions.

10.1 Research Summary

The following sections summarise how each research question was answered.

10.1.1 Designing a SpaceWire-D Software Layer

Question 1: How can an efficient SpaceWire-D software layer be designed on top of existing SpaceWire devices?

Chapter 4 answers this question by describing the design of an embedded SpaceWire-D software layer built for a LEON2-FT processor board and executing on top of the RTEMS real-time operating system.

In order to run the RTEMS real-time operating system on the LEON2-FT processor board, a board-support package was required. The BSP was based on the existing LEON support in the RTEMS source tree but extended to support the different register layout, interrupt scheme, peripherals and memory layout of the LEON2-FT processor board. The STAR-Dundee STAR-Gate remote debugging and loading program was then used to debug the BSP until the basic RTEMS test programs could run successfully.

A SpaceWire-D layer was then designed on top of the RTEMS BSP to allow the processor board to act as a SpaceWire-D initiator. The SpaceWire-D layer contains the virtual buses, network management, time-slot execution, local-timer synchronisation, error detection and user notification functionality of a SpaceWire-D initiator. It also provides an API to the user application to allow it to open, load and close virtual buses as well as configure network management parameters and receive notifications of SpaceWire-D related activity.

The SpaceWire-D layer's time-slot execution process initially had a relatively high software overhead which prevented it being used in a SpaceWire-D network using the minimum time-slot duration of 1 ms. The time-slot execution process was then optimised over several revisions until a software overhead of less than 100 μ s was achieved.

10.1.2 Designing a SpaceWire-D Demonstrator

Question 2: How can a system using the SpaceWire-D protocol be prototyped in order to demonstrate the standard?

Chapters 5 and 6 answer this question by describing the design of a prototype SpaceWire-D system that was used to demonstrate and verify the standard. The system consists of a PXI rack containing two LEON2-FT processor boards, three RMAP interface boards each containing four targets, a network manager, two routers and a host PC. The network topology was designed to allow non-conflicting paths between each initiator and each target interface board which allows both initiators to access separate targets within the same board in the same time-slot without blocking. The initiator boards use the SpaceWire-D software layer running on top of the RTEMS real-time operating system and they also include a demonstrator application. The

demonstrator application is responsible for interpreting and executing automated test commands uploaded to the initiators by the user using a program on the host PC.

An automated test scripting language was designed to simplify the process of creating and running tests using the SpaceWire-D Demonstrator. The language allows a user to define a list of transactions, transaction groups and packet bus operations as a text file and use time-slot triggered commands to open, load and close virtual buses at specific times during the execution of the schedule.

Three example scripts are described starting with a simple static bus test, a more complex static bus test then a test that uses all four types of virtual bus. These scripts are run using the SpaceWire-D Demonstrator as well as the suite of programs running on the host PC used to configure, control and monitor the network. The results of the tests were gathered using a combination of the host PC programs and a STAR-Dundee Link Analyser Mk2 placed between one of the initiators and a router in order to capture the RMAP commands and replies. These results showed that the SpaceWire-D network was operating correctly and that the protocol can be used to combine periodic traffic, aperiodic traffic, prioritised best-effort traffic and prioritised best-effort traffic with flow-control on a single SpaceWire-D network

The SpaceWire-D Demonstrator was used to complete the verification activity of the ESA SpaceWire-D project and was delivered to ESA and installed at ESTEC.

10.1.3 Scheduling SpaceWire-D Networks

Question 3: How can a SpaceWire-D mission's bandwidth requirements be represented and satisfied computationally?

Chapters 7 and 8 answer this question by first describing how the SpaceWire-D scheduling problem can be represented as a list of different bandwidth requirements,

a network topology and a list of network parameters. Next, the problem-solving strategy was described using a multi-stage algorithm that first selects the paths between the initiators and the targets then allocates transactions into time-slots to create the network schedule.

The problem representation was specified by first grouping bandwidth requirements into three categories: periodic bandwidth requirements consist of RMAP transactions between initiators and targets that are repeated at regular intervals; aperiodic bandwidth requirements consist of RMAP transactions between initiators and targets that are sporadic but must be completed within a certain deadline once they are active; and payload data bandwidth requirements consist of data-rate requirements between initiators and targets. Secondly, the network topology is defined as a bi-directional multigraph where the vertices represent the initiators, targets and routers of the network and the edges represent the SpaceWire links. Lastly, the network parameters such as worst-case routing latency and response times are described which are required in order to calculate the execution times of the RMAP transactions.

After the problem representation was specified, it was divided into two algorithmic stages. Firstly, the path selection methods were described in order to select the sets of SpaceWire links between the initiators and the targets. Two algorithms were used in the path selection stage: the BFS algorithm and a variation of Dijkstra's algorithm that dynamically increases the cost of a link every time it is used in a path to attempt to reduce the number of potential conflicts. Secondly, the periodic, aperiodic and payload data bandwidth requirements are scheduled. Periodic and aperiodic bandwidth requirements are allocated using simple algorithms which allocate their transactions to the first suitable time-slots. To allocate the payload data bandwidth requirements, the problem is transformed into a variation of the classic bin-packing problem where

the items are the transactions and the bins are the time-slots, divided into several sub-bins, one for each initiator. A conflict graph, where the nodes are initiator/target pairs and the edges represent a collision between two initiator/target pairs, determines if two transactions can be allocated within the same time-slot. Three heuristics were used in the bin-packing algorithm: first-fit, best-fit and least-conflicting. The resulting solution's quality is measured firstly by the overall number of time-slots required to satisfy all bandwidth requirements, and secondly by the number of time-slots required for payload data bandwidth requirements. If a network has an aperiodic bandwidth requirement with a short deadline, it will require many time-slots most of which may contain just a single transaction, so the second metric is used to gain additional insight into the quality of a schedule.

The algorithm was evaluated in two experiments: the first used the BFS path selection and the second used Dijkstra's algorithm with the dynamic penalty variation. There were 30 randomised test cases consisting of 10 small, 10 medium and 10 large networks. Each test case had different periodic, aperiodic and payload data bandwidth requirements and the network was assumed to have its SpaceWire links running at 200 Mbit/s with a 1024 Hz time-code rate.

Using the BFS path selection, the algorithm found solutions within 64 time-slots for 27 of the 30 test cases, with the first-fit heuristic performing best or equal with another heuristic in most cases. With the dynamic penalty weighted search path selection, the algorithm found solutions within 64 time-slots for the remaining 3 test cases and reduced the number of conflicts in 29 of the 30 test cases. In over half of the test cases, the number of conflicts was reduced to at least 85% of the conflicts when using the BFS path selection. With reduced conflicts, this allowed the algorithm to find better solutions for 20 of the 30 test cases, worse solutions for 4 and equal solutions for the

remaining 6. Additionally, the experiment was redone with different dynamic penalty values which resulted in new best solutions for 10 of the 30 test cases.

Finally, the scheduling algorithm using weighted search path selection was used in a case study of the ESA JUICE mission. The JUICE network consists of 10 instruments, a pair of OBCs, an SSMM and a downlink telemetry module all connected together with a SpaceWire router. Each link is redundant, with two redundant links between the router and SSMM and the links run at variable speeds. There were 9 periodic, 10 aperiodic and 9 payload data bandwidth requirements to be scheduled.

Using the weighted search path selection did not find a suitable solution for this mission, with the best result, using the FF heuristic, finding a solution that required 74 time-slots. Therefore, a new path selection heuristic was designed to perform load balancing when selecting links rather than applying a constant penalty value when links were selected in paths. Using the load balancing path selection, the algorithm found a suitable solution in 61 time-slots. The original test cases were then revisited using the load balancing path selection which found new best results in 11 of the 30 test cases but failed to find a suitable solution for one.

10.2 Contributions

This thesis contributes a novel scheduling strategy for SpaceWire-D networks. It allows a mission to be specified as a network topology and a list of periodic, aperiodic and payload data bandwidth requirements. The system then uses a combination of path selection algorithms and transaction allocation algorithms to attempt to find a list of network paths and transaction allocations that satisfy the bandwidth requirements.

An efficient embedded SpaceWire-D software layer, built on top of the RTEMS real-time operating system, was also contributed. This software layer was the first system

to implement the latest draft of the SpaceWire-D protocol and was used to verify the protocol during the ESA SpaceWire-D project.

The SpaceWire-D software layer was also used in combination with multiple other target boards, routers and a host PC running a suite of applications to create a SpaceWire-D Demonstrator. This system contributed a novel automated test scripting language for SpaceWire-D, as well as a traffic visualisation program used to display SpaceWire-D traffic in real-time. The SpaceWire-D Demonstrator was validated as part of the SpaceWire-D project and the results of this activity were delivered to ESA.

The combination of the embedded SpaceWire-D layer, the SpaceWire-D Demonstrator, the results of the protocol verification activity and the results of the SpaceWire-D Demonstrator validation activity show that a deterministic SpaceWire network was contributed which adhered to the new SpaceWire-D specification and the requirements of the project.

10.3 Outcomes

As the outcomes of this research, three papers have been published with the author as the primary author and three papers with the author as a co-author. In addition, the SpaceWire-D initiator software layer described in Chapter 4 and the SpaceWire-D Demonstrator described in Chapter 5 was used to complete the verification activity of the ESA SpaceWire-D project. The SpaceWire-D Demonstrator was then delivered to ESA and installed at ESTEC.

Furthermore, the author has presented work related to this research at two SpaceWire Working Group meetings at ESTEC and gave an invited tutorial on SpaceWire-D at the 7th International SpaceWire Conference in Yokohama, Japan, 2016.

Bibliography

ADLINK Technology Inc. 2016. *PXI-3950*. Accessed August 2, 2016.
[http://www.adlinktech.com/PD/web/PD_detail.php?cKind=&pid=797&seq=](http://www.adlinktech.com/PD/web/PD_detail.php?cKind=&pid=797&seq=&id=&sid=&utm_source=#)
[&id=&sid=&utm_source=#](http://www.adlinktech.com/PD/web/PD_detail.php?cKind=&pid=797&seq=&id=&sid=&utm_source=#).

Airbus Defence & Space. 2015. *JUICE SpaceWire Application Protocol Specification*.
 Airbus Defence & Space.

Andersson, Jan, Jiri Gaisler, and Roland Weigand. 2010. “Next Generation Multipurpose Microprocessor.” *Proceedings of Data Systems in Aerospace (DASIA) Conference*. Budapest, Hungary. 83-86.

Bracknell, Derek. 1988. “Introduction to the MIL-STD-1553B Serial Multiplex Data Bus.” *Microprocessors and Microsystems 12.1* (Royal Aerospace Establishment), 3-12.

CAN in Automation. 2011. *CiA Standard 301 Version 4.2.0: CANopen application layer and communication profile*. Standard, CiA.

CCSDS. 2013. *Spacecraft Onboard Interface Services*. CCSDS Information Report, Washington, DC, USA: Consultative Committee for Space Data Systems.

Chen, Yang, Mitsutaka Takada, Ryo Kurachi, and Hiroaki Takada. 2013. “A Scheduling Method of RMAP Packets for SpaceWire-D.” *Proceedings of 5th International SpaceWire Conference*. Gothenburg, Sweden. 205-208.

Coffman, Edward G, János Csirik, Gábor Galambos, Silvano Martello, and Daniele Vigo. 2013. “Bin Packing Approximation Algorithms: Survey and

- Classification.” In *Handbook of Combinatorial Optimization*, by Panos M Pardalos, Ding-Zhu Du and Ronald Graham, 455-531. Springer.
- CompuPhase. 2015. *Termite: A Simple RS232 Terminal*. 18 August. Accessed April 6, 2017. http://www.compuphase.com/software_termite.htm.
- Corbet, Jonathan, Alessandro Rubini, and Greg Kroah-Hartman. 2005. “An Introduction to Device Drivers.” In *Linux Device Drivers*, by Jonathan Corbet, Alessandro Rubini and Greg Kroah-Hartman, 1-14. O'Reilly Media, Inc.
- Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. “Priority Queues.” In *Introduction to Algorithms*, 162-164. The MIT Press.
- Dijkstra, Edsger W. 1959. “A note on two problems in connexion with graphs.” *Numerische mathematik 1.1*, 269-171.
- ECSS. 2003. *ECSS-E-ST-50-12A: SpaceWire - Links, nodes, routers and networks*. Standard, European Cooperation for Space Standardization.
- ECSS. 2008 A. *ECSS-E-ST-50-12C: SpaceWire - Links, nodes, routers and networks*. Standard, European Cooperation for Space Standardization.
- ECSS. 2008 B. *ECSS-E-ST-50-13C: Interface and communication protocol for MIL-STD-1553B data bus onboard spacecraft*. Standard, European Cooperation for Space Standardization.
- ECSS. 2013. *ECSS-E-ST-50-15C DIR1: CANBus extension protocol*. Standard, European Cooperation for Space Standardization.
- ECSS. 2010 A. *ECSS-E-ST-50-51C: SpaceWire Protocol Identification*. Standard, European Cooperation for Space Standardisation.

ECSS. 2010 B. *ECSS-E-ST-50-52C: SpaceWire - Remote memory access protocol*.

Standard, European Cooperation for Space Standardization.

Epstein, Leah, and Asaf Levin. 2006. "On Bin Packing with Conflicts." *Proceedings of the International Workshop on Approximation and Online Algorithms*. Zurich, Switzerland. 160-173.

European Space Agency. 2014 A. *Architectures of Onboard Data Systems*. 1 May.

Accessed 1 6, 2017.

http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Architectures_of_Onboard_Data_Systems.

European Space Agency. 2014 B. *First Copernicus Satellite Now Operational*. 6

October. Accessed April 6, 2017.

http://www.esa.int/Our_Activities/Observing_the_Earth/Copernicus/Sentinel-1/First_Copernicus_satellite_now_operational.

European Space Agency. 2015 A. *ESA Experimental Spaceplane Completes Research*

Flight. 11 February. Accessed April 6, 2017.

http://www.esa.int/Our_Activities/Launchers/IXV/ESA_experimental_spaceplane_completes_research_flight.

European Space Agency. 2015 B. *SpaceWire Missions*. 4 December. Accessed April

6, 2017. <http://www.spacewire.esa.int/content/Missions/Missions.php>.

European Space Agency. 2011. *JUICE: Exploring the emergence of habitable worlds around gas giants*. Assessment Study Report, European Space Agency.

European Space Agency. 2009. *Seeing Stars, PROBA-2 Platform Passes Its First*

Health Check. 11 November. Accessed April 6, 2017.

http://www.esa.int/Our_Activities/Observing_the_Earth/SMOS/Two_new_ESA_satellites_successfully_lofted_into_orbit.

Fuller, Sam. 2005. *RapidIO: The Embedded System Interconnect*. John Wiley & Sons.

Gatliff, Bill. 1999 A. "Embedding with GNU: GNU Debugger." *Embedded Systems Programming* 12, 80-95.

Gatliff, Bill. 1999 B. "Embedding with GNU: the gdb Remote Serial Protocol." *Embedded Systems Programming* 12, 108-113.

Gibson, David, Steve Parkes, and Karen Petrie. 2013. "Modeling Deterministic Spacecraft Networks with Constraint Programming." *Proceedings of 19th International Conference on Principles and Practices of Constraint Programming (Doctoral Program)*. Uppsala, Sweden. 49-54.

Gibson, David, Steve Parkes, Chris McClements, and Stuart Mills. 2016. "SpaceWire-D Prototype and Demonstration System." *Proceedings of 7th International SpaceWire Conference*. Yokohama, Japan. 275-281.

Gibson, David, Steve Parkes, Chris McClements, Stuart Mills, and David Paterson. 2014. "SpaceWire-D on the Castor Spaceflight Processor." *Proceedings of 6th International SpaceWire Conference*. Athens, Greece. 197-203.

GNU Project. 2015 A. *GNU Linker Documentation*. 9 September. Accessed April 6, 2017. <https://sourceware.org/binutils/docs/ld/index.html#Top>.

GNU Project. 2015 B. *Linker Scripts*. 9 September. Accessed April 6, 2017. <https://sourceware.org/binutils/docs/ld/Scripts.html#Scripts>.

Grasset, O, M K Dougherty, A Coustenis, E J Bunce, C Erd, D Titov, M Blanc, et al. 2013. "JUperiter ICy moons Explorer (JUICE): An ESA mission to orbit

Ganymede and to characterise the Jupiter system.” *Planetary and Space Science* 78, 1-21.

Guasch, J R, S Parkes, and A Christen. 1999. “From IEEE 1355 High-Speed Serial Links to SpaceWire (Programmable Aspects and Applications).” *Proceedings of Data Systems in Aerospace (DASIA) Conference*. Lisbon, Portugal. 117-122.

Hult, Torbjörn, and Steve Parkes. 2014. “On-Board Data Systems.” In *The International Handbook of Space Technology*, by Malcolm Macdonald and Viorel Badescu, 441-470. Springer.

Hutchinson, Michael, Les Griffiths, Adrian Smith, and Sam Doody. 2012. “The Sentinel-1 Radar Electronics Subsystem Development.” *Proceedings of European Conference on Synthetic Aperture Radar*. Nuremberg. 170-173.

IEEE Computer Society. 1996 A. *IEEE-1355-1995: IEEE Standard for Heterogeneous Interconnect (HIC), (Low Cost, Low-Latency Scalable Serial Interconnect for Parallel System Construction)*. Standard, IEEE Computer Society.

IEEE Computer Society. 1996 B. *IEEE-1596.3-1996: IEEE Standard for Low-Voltage Differential Signals (LVDS) for Scalable Coherent Interface (SCI)*. Standard, IEEE Computer Society.

JAXA. 2016. “Supplemental Handout on the Operation Plan of the X-ray Astronomy Satellite ASTRO-H (Hitomi).” *JAXA / X-ray Astronomy Satellite "Hitomi" (ASTRO-H)*. 28 April. Accessed November 11, 2016. http://global.jaxa.jp/projects/sat/astro_h/topics.html.

JetBrains. 2015. *Python IDE & Django IDE*. 29 October. Accessed April 6, 2017.
<https://www.jetbrains.com/pycharm/>.

Klar, Robert A, Scott A Miller, Michael L Brysch, and R Allison Bertrand. 2013.
 “Performance of the Magnetospheric Multiscale Central Instrument Data
 Handling.” *Proceedings of IEEE Aerospace Conference*. Big Sky, MT, USA:
 IEEE. 1-7.

Kopetz, Hermann. 2011. “Real-Time Operating Systems.” In *Real-Time Systems:
 Design Principles for Distributed Embedded Applications*, by Hermann
 Kopetz, 215-238. Springer Science & Business Media.

Kopetz, Hermann, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. 2005. “The
 Time-Triggered Ethernet (TTE) Design.” *Proceedings of Eighth IEEE
 International Symposium on Object-Oriented Real-Time Distributed
 Computing*. Seattle, WA, USA: IEEE. 22-33.

Lee, Chin Yang. 1961. “An algorithm for path connections and its applications.” *IRE
 Transactions on Electronic Computers* 3, 346-365.

Leinberger, William, George Karypis, and Vipin Kumar. 1999. “Multi-Capacity Bin
 Packing Algorithms with Applications to Job Scheduling Under Multiple
 Constrains.” *Proceedings of International Conference on Parallel Processing*.
 Aizu-Wakamatsu City, Japan: IEEE. 404-412.

McComas, David. 2012. “NASA/GSFC's Flight Software Core Flight System.” *Flight
 Software Workshop*. San Antonio, TX, USA.

Meszaros, Gerard. 2007. *xUnit Test Patterns: Refactoring Test Code*. Pearson
 Education.

- Mills, Stuart, and Steve Parkes. 2015. "A Software Suite for Testing SpaceWire Devices and Networks." *Proceedings of Data Systems in Aerospace (DASIA) Conference*. Barcelona, Spain.
- MinGW. 2015. *MinGW Minimalist GNU for Windows*. 8 September. Accessed April 6, 2017. <http://www.mingw.org/>.
- Moore, Edward F. 1959. "The shortest path through a maze." *Proceedings of the International Symposium on the Theory of Switching*. Harvard University Press. 285-292.
- NASA GSFC. 2015. *The Magnetospheric Multiscale Mission*. 4 December. Accessed April 6, 2017. http://mms.gsfc.nasa.gov/about_mms.html.
- OAR Corporation. 2011. "RTEMS C User's Guide." *RTEMS 4.10.2 On-Line Library*. 13 December. Accessed April 6, 2017. https://docs.rtems.org/releases/rtemsdocs-4.10.2/share/rtems/pdf/c_user.pdf.
- Ozaki, Masanobu, Tadayuki Takahashi, Motohide Kokubun, and Takeshi Takashima. 2010. "SpaceWire Driven Architecture for the ASTRO-H Satellite." *Proceedings of 3rd International SpaceWire Conference*. St. Petersburg, Russia. 445-451.
- Parkes, S. 1999. "SpaceWire: The Standard." *Proceedings of Data Systems in Aerospace (DASIA) Conference*. Lisbon, Portugal. 111-116.
- Parkes, Steve, Albert Ferrer, Stuart Mills, and Alex Mason. 2010. "SpaceWire-D: Deterministic Data Delivery with SpaceWire." *Proceedings of 3rd International SpaceWire Conference*. St Petersburg, Russia. 31-40.

- Parkes, Steve, Chris McClements, David McLaren, Albert Ferrer Florit, and Alberto Gonzalez Villafranca. 2015. "SpaceFibre: A Multi-Gigabit/s Interconnect for Spacecraft Onboard Data Handling." *Proceedings of IEEE Aerospace Conference*. Big Sky, MT, USA: IEEE. 1-13.
- Parkes, Steve, Chris McClements, Gerald Kempf, Stephan Fischer, Pierre Fabry, and Agustin Leon. 2007. "SpaceWire Router ASIC." *Proceedings of 1st International SpaceWire Conference*. Dundee, UK. 301-306.
- Parkes, Steve, Chris McClements, Guy Mantelet, and Nicolas Ganry. 2013. "The Next Generation of Spaceflight Processors: Low Power, High Performance, With Integrated SpaceWire Router and Protocol Engines." *Proceedings of International Astronautical Congress*. Beijing, China. 7807-7814.
- Parkes, Steve, David Gibson, and Albert Ferrer. 2015 A. "Experimental Results for SpaceWire-D." *Proceedings of Data Systems in Aerospace (DASIA) Conference*. Barcelona, Spain.
- Parkes, Steve, David Gibson, and Albert Ferrer. 2015 B. *SpaceWire-D Standard Draft E*. Standard, Dundee, UK: University of Dundee.
- Parkes, Steve, David Gibson, and Albert Ferrer. 2014. "SpaceWire-D: Deterministic Data Delivery over SpaceWire." *Proceedings of Data Systems in Aerospace (DASIA) Conference*. Warsaw, Poland.
- Paterson, David, David Gibson, and Steve Parkes. 2014. "An RTEMS Port for the AT6981 SpaceWire-Enabled Processor: Features and Performance." *Proceedings of 6th International SpaceWire Conference*. Athens, Greece. 234-237.

- PXI Systems Alliance. 2004. *PXI Hardware Specification Revision 2.2*. Specification, PXI Systems Alliance.
- Raphael, David, Robert F Stone, Damaris L Guevara, and James E Fraction. 2014. "Command & Data Handling for the Magnetospheric Multiscale Mission." *Proceedings of IEEE Aerospace Conference*. Big Sky, MT, USA: IEEE. 1-12.
- Regehr, John. 2007. "Safe and Structured Use of Interrupts in Real-Time and Embedded Software." In *Handbook of Real-Time and Embedded Systems*, by Insuo Lee, Joseph Y-T Leung and Sang H Son, 16.1-16.12. Chapman & Hall/CRC.
- Robert Bosch GmbH. 1991. *CAN Specification Version 2.0*. Standard, Robert Bosch GmbH.
- Rossignol, Alain, and Jacques Seronie-Vivien. 2012. "ASTRIUM Satellites Experiment About RTEMS On-Board Software Product: From an Open Source Software to an Operational Satellite Real-Time Operating System." *L'Open Source Pour Les Systèmes Embarqués Temps Réel*. Toulouse, France.
- Santandrea, S, K Gantois, K Strauch, F Teston, E Tilmans, C Baijot, D Gerrits, A De Groof, G Schwehm, and J Zender. 2013. "PROBA2: Mission and Spacecraft Overview." *Solar Physics*, 5-19.
- Scott, Pete, and Steve Parkes. 2010. "SpaceWire Link Analyser Mk2: A New Analysis Device for SpaceWire Systems." *Proceedings of 3rd International SpaceWire Conference*. St. Petersburg, Russia. 67-71.
- SPARC International, Inc. 2015. *Specifications Download*. 4 September. Accessed April 6, 2017. <http://sparc.org/technical-documents/specifications/#ARCH>.

- Stallman, Richard. 2001. "Using and Porting the GNU Compiler Collection." *M.I.T. Artificial Intelligence Laboratory*. Boston, MA, USA.
- Stankovic, John A, and R Rajkumar. 2004. "Real-Time Operating Systems." *Real-Time Systems*, 237-253.
- STAR-Dundee Ltd. 2016. *SpaceWire PXI*. Accessed August 1, 2016. <https://www.star-dundee.com/products/spacewire-pxi>.
- STAR-Dundee Ltd. 2015. *SPARC V8 Software Development Environment*. 9 September. Accessed April 6, 2017. <https://www.star-dundee.com/products/sparcv8-software-development-environment>.
- STAR-Dundee Ltd. 2017. *STAR-Dundee Products*. 6 April. Accessed April 6, 2017. <https://www.star-dundee.com/products>.
- Takahashi, Tadayuki, Kazuhisa Mitsuda, Richard Kelley, and Henri Aarts. 2012. "The ASTRO-H X-Ray Observatory." *Astronomical Telescopes + Instrumentation*. Amsterdam, Netherlands: International Society for Optics and Photonics. 84431Z-84431Z.
- Telecommunications Industry Association. 2012. *TIA-644-A: Electrical Characteristics of Low Voltage Differential Signalling (LVDS) Interface Circuits, Revision A*. Standard, Telecommunications Industry Association.
- The Qt Company Ltd. 2016. *Qt 4.8*. Accessed August 3, 2016. <http://doc.qt.io/qt-4.8/>.
- The RTEMS Project. 2015. *Historical Timeline*. 6 January. Accessed April 6, 2017. <https://devel.rtems.org/wiki/History/Timeline>.
- The RTEMS Project. 2013. *RTEMS Applications*. 30 December. Accessed April 6, 2017.

<https://devel.rtems.org/wiki/TBR/UserApp/RTEMSApplications#SpaceandAviation>.

The RTEMS Project. 2017. *RTEMS Real Time Operating System (RTOS)*. 5 April.

Accessed April 6, 2017. <https://www.rtems.org/>.

Torres, Ramon, Paul Snoeij, Dirk Geudtner, David Bibby, Malcolm Davidson, Evert

Attema, Pierre Potin, et al. 2012. "GMES Sentinel-1 Mission." *Remote Sensing of Environment*, 9-24.

University of Dundee. 2014. "SpaceWire-D D3 Demonstrator Specification Document." ESA Project Document.

University of Dundee. 2015. "SpaceWire-D D4 Verification Test Specification Document." ESA Deliverable.

University of Dundee. 2016 A. "SpaceWire-D D6 Verification Report." ESA Deliverable.

University of Dundee. 2016 B. "SpaceWire-D D7 Validation Report." ESA Deliverable.

US Department of Defense. 1978. *Aircraft Internal Time Division Command/Response Multiplex Data Bus*. Standard, Washington D.C., USA: US Department of Defense.

Vladimirova, Tanya, Christopher P Bridges, George Prassinis, Xiaofeng Wu, Kawsu Sidibeh, David J Barnhart, Abdul-Halim Jallad, et al. 2007. "Characterising Wireless Sensor Motes for Space Applications." *Proceedings of Second NASA/ESA Conference on Adaptive Hardware and Systems*. Edinburgh, UK: IEEE. 43-50.

Wind River. 2016. *VxWorks*. Accessed April 2, 2016.
<https://windriver.com/products/vxworks/>.

Yuasa, Takayuki, Tadayuki Takahashi, Masanobu Ozaki, and Motohide Kokubun. 2011. "A Deterministic SpaceWire Network Onboard the ASTRO-H Space X-Ray Telescope." *Proceedings of 4th International SpaceWire Conference*. San Antonio, TX, USA. 333-336.

Zaccagnino, E, G Malucchi, V Marco, A Drocco, S Dussy, and J Préaud. 2011. "Intermediate eXperimental Vehicle (IXV), the ESA Re-entry Demonstrator." *Proceedings of AIAA Guidance, Navigation, and Control Conference*. Portland, OR, USA. 6340-6353.

Appendix 1

LEON2-FT Processor Board

Each LEON2-FT processor board combines a LEON2-FT processor with on-chip memory, DDR and integrated peripherals including SpaceWire DMA engines, RMAP engines and an embedded SpaceWire router (Parkes, McClements and Mantelet, et al. 2013). The LEON2-FT processor board's RMAP engines include an RMAP initiator implemented in hardware which reduces the software overhead required to transmit RMAP commands and process RMAP replies. This makes it attractive for used in SpaceWire-D systems, where reducing the initiator processing time as much as possible is essential.

The LEON2-FT processor is based on the SPARC V8 architecture (SPARC International, Inc. 2015) and the FPGA implementation used in the SpaceWire-D Demonstrator runs at a clock rate of 40 MHz. The processor includes 128 KB of on-chip memory, two on-chip UARTs, a debug support unit as well as standard processor elements such as caches and an interrupt controller. A block diagram illustrating the LEON2-FT processor board's architecture is shown in Figure 12-1.

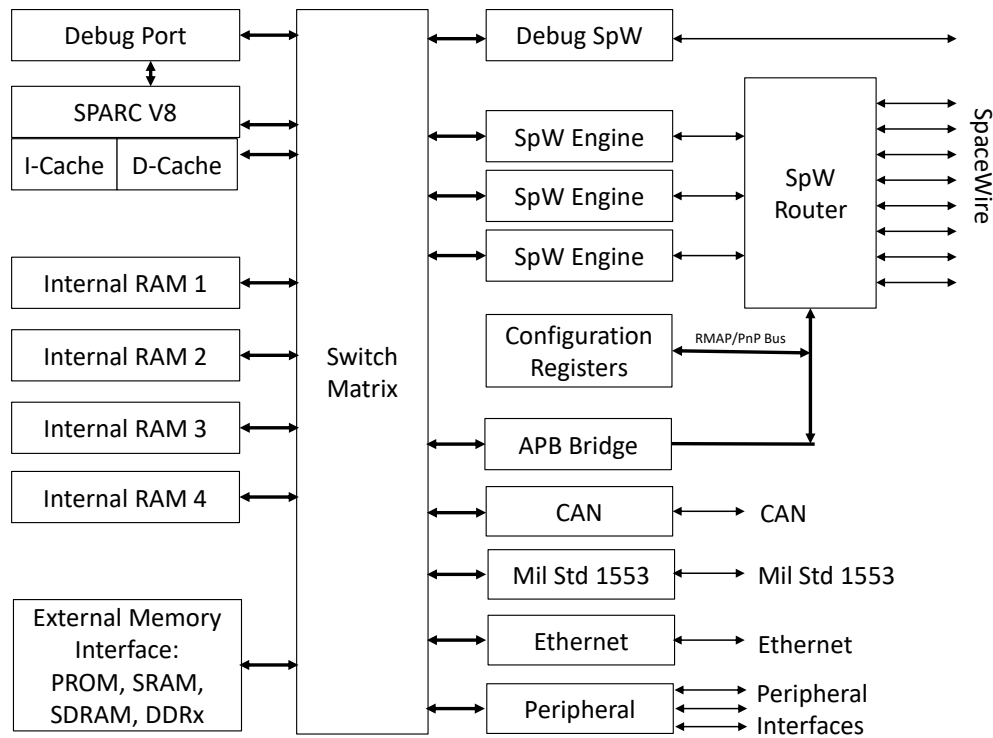


Figure 12-1: LEON2-FT Processor Board Architecture (Parkes, McClements and Mantelet, et al. 2013)

In Figure 12-1, the full version of the LEON2-FT processor board is shown, which includes many peripherals connected through a centralised switch matrix. In addition to the processor and memory described above, the FPGA version of the LEON2-FT processor board used for the work in this thesis contains 8 SpaceWire interfaces, a 12-port SpaceWire router, 3 SpaceWire protocol engines, a debug SpaceWire port and the configuration registers. The full version of the board may also support the CAN, MIL-STD-1553B and Ethernet communication networks.

RMAP Engines

The LEON2-FT processor board contains three SpaceWire protocol engines, each containing SpaceWire DMA channels, an RMAP initiator and an RMAP target. The RMAP initiator is responsible for transmitting commands and handling replies, minimising the amount of work that the processor is required to perform.

Each RMAP initiator contains a set of configuration and control registers which the software uses to set up a group of one or more transactions for execution. Each transaction is split into two data structures: the first contains the RMAP command header parameters and the second is a descriptor that indicates the command's settings and the location of the header and buffers.

In Figure 12-2, the transaction descriptor array is illustrated, showing an array of N transaction descriptors in the left column. The arrows pointing from the first descriptor into the diagram on the right show the pointers into memory contained within the descriptor data structure.

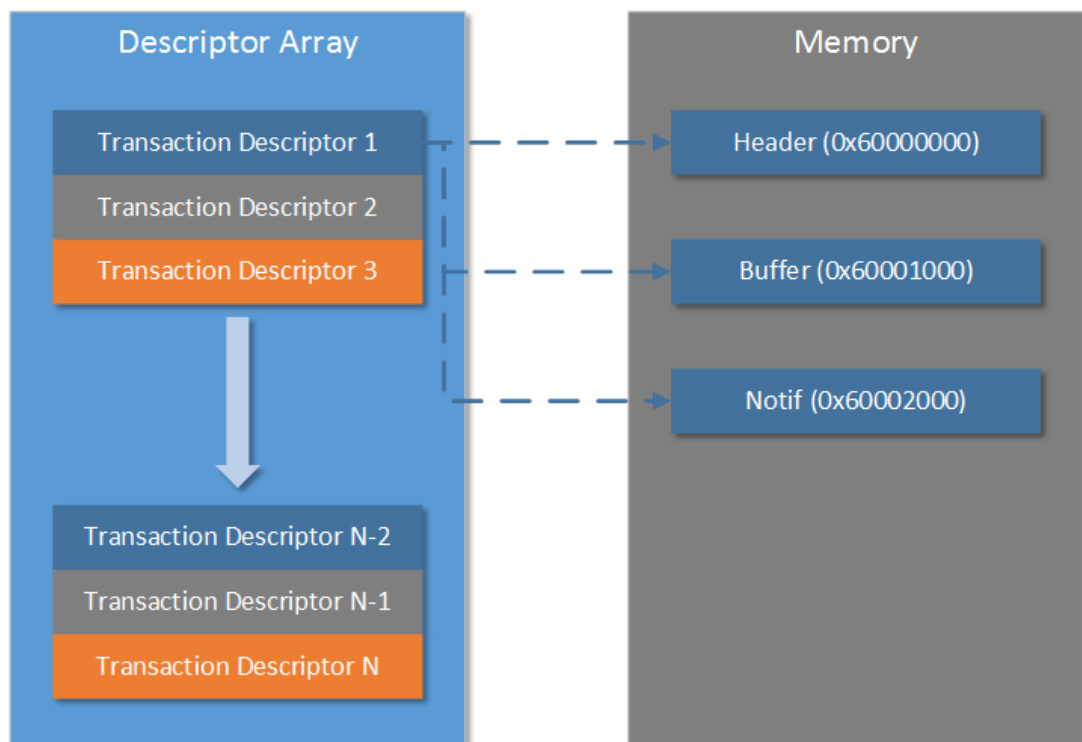


Figure 12-2: RMAP Initiator Descriptor Array

When the processor wants to execute a transaction group, it creates an array of transaction descriptors and tells the RMAP initiator to begin execution. The initiator uses the transaction descriptors to form a series of RMAP command packets which

are sent out of the SpaceWire router. If any RMAP replies are received, the initiator matches the replies to the corresponding commands stored in a descriptor table. The initiator then performs any additional work such as writing received data to a buffer or updating a notification data structure.

SpaceWire DMA Channels

To handle transmitting and receiving regular SpaceWire traffic, there are a total of nine SpaceWire DMA channels available, with three on each of the SpaceWire protocol engines.

The DMA controllers allow a transmitted packet to be defined as a list of chunks at addresses defined by the user. This list is then used by the controller to construct the transmitted packet, offloading most of the work to the dedicated hardware and reducing the CPU overhead. When receiving packets, the DMA controllers write the received data to memory addresses defined by the user and update a descriptor with the packet information.

By offloading most of the SpaceWire packet processing to dedicated DMA controllers, this allows the LEON2-FT processor board to operate three simultaneous SpaceWire links at the maximum data-rate of 200 Mbits/s.

Although all of the SpaceWire traffic used in the SpaceWire-D layer described in this thesis use the RMAP engines exclusively, the SpaceWire DMA channels could also be used for this purpose. However, the benefit of using the RMAP engines is the decreased overhead because the protocol is implemented in dedicated hardware. If the SpaceWire-D layer was to use the SpaceWire DMA channels, software would be required to generate RMAP commands and process RMAP replies.

Embedded SpaceWire Router

The LEON2-FT processor board contains an embedded SpaceWire router, based on the SpW-10X router ASIC (Parkes, McClements and Kempf, et al. 2007). The router has eight physical SpaceWire ports allowing connections to other devices, three external ports each connected to one of the SpaceWire protocol engines, a configuration port and a physical SpaceWire debug port.

Packet Demultiplexer

When a SpaceWire packet is switched into one of the external ports connected to the SpaceWire protocol engines, the router uses a demultiplexer to determine if the packet should be forwarded on to the RMAP target, the RMAP initiator or one of the three DMA channels.

The demultiplexer matches up to four bytes at the head of the packet against patterns and masks configured in each of the SpaceWire protocol engine's configuration registers. For example, RMAP packets conform to the SpaceWire Protocol Identification standard (ECSS 2010 A) which ensures that each packet is pre-pended by a one-byte logical address and a protocol identifier. The protocol identifier is nominally one byte but may be extended to three bytes with the first byte set to zero to indicate an extended protocol identifier. The demultiplexer configuration to filter reply packets from an RMAP target with a logical address of 0x20 is shown in Table 12-1.

Table 12-1: Example RMAP Reply Demultiplexer Configuration

Register	Byte 0 (Logical Address)	Byte 1 (Protocol ID)	Byte 2 (Instruction)	Byte 3 (Key)
Pattern	0x20	0x01	0x00	0x00
Mask	0x00	0x00	0xBF	0xFF

In Table 12-1, the demultiplexer is configured to filter packets if the packet contains a first byte with the value of the expected logical address (0x20), a second byte with the value of the RMAP protocol ID (0x01) and a third byte with the seventh bit cleared, to indicate that the packet is an RMAP reply. The demultiplexer mask uses a clear bit to indicate that the bit should be used in the matching process. In this case, the mask is configured to match against the entire first and second bytes, using the value 0x00, and the seventh bit of the third byte, using the value 0xBF (0b10111111). Any packets arriving at the SpaceWire protocol engine and matching the pattern and mask of the RMAP initiator demultiplexer configuration will now be forwarded on to the RMAP initiator.

Similarly, the demultiplexer configuration to filter command packets from an RMAP initiator with a logical address of 0x21 is shown in Table 12-2.

Table 12-2: Example RMAP Command Demultiplexer Configuration

Register	Byte 0 (Logical Address)	Byte 1 (Protocol ID)	Byte 2 (Instruction)	Byte 3 (Key)
Pattern	0x21	0x01	0x40	0x00
Mask	0x00	0x00	0xBF	0xFF

In Table 12-2, the only difference, in comparison to the initiator's demultiplexer configuration, is the logical address and the seventh bit of the third pattern byte is set to indicate that the packet is an RMAP command.

LEON2-FT in Space

The first two generations of the LEON processor family, LEON1 and LEON2, were developed at the European Space Agency (ESA). Fault tolerance was added to the original LEON2 resulting in the LEON2-FT. This version added protection against single event upsets, when charged particles hit memory causing bit values to be

flipped. Two further generations, LEON3 and LEON4, have been developed at Gaisler Research (Andersson, Gaisler and Weigand 2010).

Currently, the LEON2-FT processor is being used in multiple active spacecraft in a variety of roles such as controlling an avionics subsystem, controlling a radar instrument and acting as the main on-board computer.

The Intermediate eXperimental Vehicle (IXV) is an ESA mission to demonstrate an atmospheric re-entry system. In the IXV, the LEON2-FT processor, running at 50 MHz, is used in the avionics on-board computer. It is in charge of executing guidance, navigation and control (GNC) tasks; issuing commands to actuators; and encoding, storing and initiating the transmission of GNC and housekeeping telemetry (Zaccagnino, et al. 2011). The IXV successfully completed its first flight in February 2015 (European Space Agency 2015 A).

Sentinel-1 is an ESA earth observation mission and is the first in a series of two-satellite constellations (Torres, et al. 2012) and contains a synthetic aperture radar (SAR) as its payload. In Sentinel-1, the LEON2-FT processor is used within the SAR electronics subsystem for instrument control (Hutchinson, et al. 2012). Sentinel-1 was launched in April 2014 and is currently active (European Space Agency 2014 B).

PROBA2 is the second in the Project for On-Board Autonomy (PROBA) series of ESA missions to demonstrate in-orbit autonomous technology (Santandrea, et al. 2013). In PROBA2, the LEON2-FT processor is used for all computing requirements on the spacecraft, excluding the instruments, such as GNC, AOCS, payload data handling and power management. PROBA2 was launched in November 2009 and is currently active until its expected mission end in 2018 (European Space Agency 2009).

As described above, the LEON2-FT processor is a popular ESA technology and is being used in multiple missions, making it an ideal and realistic platform to be used in the design of the SpaceWire-D software layer and prototype system.

Appendix 2

Multiple Initiators and Static Buses Results

This section describes the results of running the test described in Section 6.2 using the SpaceWire-D Demonstrator.

-1.60461 ms	1.56196 ms				TIMECODE [3F]	1.56196 ms
-42.630 µs	1.56198 ms				TIMECODE [00]	1.56198 ms
0 ns	42.630 µs	NCHAR [40]		3.16657 ms	NULL	42.630 µs
40 ns	40 ns	NCHAR [01]		40 ns	NULL	40 ns
100 ns	60 ns	NCHAR [4C]		60 ns		

Figure 13-1: Example Script 2 – Time-Slot 0

In Figure 13-1, the Link Analyser screenshot shows the start of time-slot 0 which is triggered by the arrival of time-code 0 at the initiator.

0 ns		Header: RMAP Command		
0 ns		Target Address: 40		
100 ns	100 ns	Command: Read Command	100 ns	
100 ns		Acknowledge		
130 ns	40 ns	Key: 20	40 ns	
190 ns	60 ns	Initiator Address: 30	60 ns	
250 ns	60 ns	Transaction ID: 0001	60 ns	
340 ns	100 ns	Extended Address: 00	100 ns	
400 ns	60 ns	Address: 00000000	60 ns	
590 ns	190 ns	Data Length: 000020	190 ns	
740 ns	150 ns	Header CRC: A3	150 ns	
760 ns	20 ns	EOP	20 ns	
2.190 µs	1.430 µs			Header: RMAP Reply
2.190 µs				Initiator Address: 30
2.300 µs	110 ns			Command: Read Reply
2.300 µs				Acknowledge
2.340 µs	40 ns			Status: Success
2.400 µs	60 ns			Target Address: 40
2.460 µs	60 ns			Transaction ID: 0001
2.590 µs	130 ns			Data Length: 000020
2.740 µs	150 ns			Header CRC: 4D
2.740 µs				Data CRC: 03
2.800 µs	60 ns			Cargo Size: 32
4.420 µs	1.620 µs			EOP

Figure 13-2: Example Script 2 – Static Bus 0, Transaction 0

In Figure 13-2, the Link Analyser screenshot shows the first transaction executed by static bus 0 which is an RMAP read command to read 32 bytes from address 0x00000000 in target 0x40.

12.000 µs	7.580 µs	Header: RMAP Command	11.240 µs		
12.000 µs		Target Address: 50			
12.100 µs	100 ns	Command: Write Command	100 ns		
12.100 µs		Data Not Verified			
12.100 µs		Acknowledge			
12.100 µs		Incrementing Address			
12.150 µs	60 ns	Key: 20	60 ns		
12.210 µs	60 ns	Initiator Address: 30	60 ns		
12.250 µs	40 ns	Transaction ID: 0002	40 ns		
12.360 µs	110 ns	Extended Address: 00	110 ns		
12.400 µs	40 ns	Address: 00000000	40 ns		
12.610 µs	210 ns	Data Length: 000020	210 ns		
12.760 µs	150 ns	Header CRC: 60	150 ns		
12.760 µs		Data CRC: 4C			
12.910 µs	150 ns	Cargo Size: 32	150 ns		
14.780 µs	1.870 µs	EOP	1.870 µs		
16.190 µs	1.410 µs			Header: RMAP Reply	11.770 µs
16.190 µs				Initiator Address: 30	
16.300 µs	110 ns			Command: Write Reply	110 ns
16.300 µs				Data Not Verified	
16.300 µs				Acknowledge	
16.300 µs				Incrementing Address	
16.340 µs	40 ns			Status: Success	40 ns
16.400 µs	60 ns			Target Address: 50	60 ns
16.460 µs	60 ns			Transaction ID: 0002	60 ns
16.550 µs	100 ns			Header CRC: 57	100 ns
16.570 µs	20 ns			EOP	20 ns

Figure 13-3: Example Script 2 – Static Bus 0, Transaction 1

In Figure 13-3, the Link Analyser screenshot shows the second transaction executed by static bus 0 which is an RMAP write command to write 32 bytes to address 0x00000000 in target 0x50.

27.370 µs	10.800 µs	Header: RMAP Command	12.590 µs		
27.370 µs		Target Address: 60			
27.490 µs	110 ns	Command: Read Command	110 ns		
27.490 µs		Acknowledge			
27.520 µs	40 ns	Key: 20	40 ns		
27.580 µs	60 ns	Initiator Address: 30	60 ns		
27.640 µs	60 ns	Transaction ID: 0003	60 ns		
27.730 µs	100 ns	Extended Address: 00	100 ns		
27.770 µs	40 ns	Address: 00000000	40 ns		
27.980 µs	210 ns	Data Length: 000020	210 ns		
28.130 µs	150 ns	Header CRC: A0	150 ns		
28.150 µs	20 ns	EOP	20 ns		
29.640 µs	1.490 µs			Header: RMAP Reply	13.070 µs
29.640 µs				Initiator Address: 30	
29.730 µs	100 ns			Command: Read Reply	100 ns
29.730 µs				Acknowledge	
29.770 µs	40 ns			Status: Success	40 ns
29.830 µs	60 ns			Target Address: 60	60 ns
29.890 µs	60 ns			Transaction ID: 0003	60 ns
30.020 µs	130 ns			Data Length: 000020	130 ns
30.170 µs	150 ns			Header CRC: F0	150 ns
30.170 µs				Data CRC: C3	
30.230 µs	60 ns			Cargo Size: 32	60 ns
31.850 µs	1.620 µs			EOP	1.620 µs

Figure 13-4: Example Script 2 – Static Bus 0, Transaction 2

In Figure 13-4, the Link Analyser screenshot shows the third transaction executed by static bus 0 which is an RMAP read command to read 32 bytes from address 0x00000000 in target 0x60.

23.38718 ms	1.56196 ms	NULL	3.12398 ms	TIMECODE [0F]	1.56196 ms
24.94916 ms	1.56198 ms	NULL	1.56198 ms	TIMECODE [10]	1.56198 ms
24.98922 ms	40.060 µs	NCHAR [41]	40.060 µs		
24.98926 ms	40 ns	NCHAR [01]	40 ns		
24.98931 ms	60 ns	NCHAR [4C]	60 ns	NULL	40.150 µs

Figure 13-5: Example Script 2 – Time-Slot 16

In Figure 13-5, the Link Analyser screenshot shows the start of time-slot 16 which is triggered by the arrival of time-code 16 at the initiator.

24.98922 ms	24.95737 ms	Header: RMAP Command	24.96107 ms	
24.98922 ms		Target Address: 41		
24.98931 ms	100 ns	Command: Read Command	100 ns	
24.98931 ms		Acknowledge		
24.98935 ms	40 ns	Key: 20	40 ns	
24.98941 ms	60 ns	Initiator Address: 30	60 ns	
24.98947 ms	60 ns	Transaction ID: 4001	60 ns	
24.98956 ms	100 ns	Extended Address: 00	100 ns	
24.98962 ms	60 ns	Address: 00000000	60 ns	
24.98981 ms	190 ns	Data Length: 001FA0	190 ns	
24.98996 ms	150 ns	Header CRC: F5	150 ns	
24.98998 ms	20 ns	BOP	20 ns	
24.99147 ms	1.490 µs			Header: RMAP Reply 24.95962 ms
24.99147 ms				Initiator Address: 30
24.99156 ms	100 ns			Command: Read Reply 100 ns
24.99156 ms				Acknowledge
24.9916 ms	40 ns			Status: Success 40 ns
24.99166 ms	60 ns			Target Address: 41 60 ns
24.99171 ms	60 ns			Transaction ID: 4001 60 ns
24.99185 ms	130 ns			Data Length: 001FA0 130 ns
24.992 ms	150 ns			Header CRC: 8B 150 ns
24.992 ms				Data CRC: EC
24.99206 ms	60 ns			Cargo Size: 8096 60 ns
25.00215 ms	10.100 µs	Header: RMAP Command	12.170 µs	

Figure 13-6: Example Script 2 – Static Bus 16, Transaction 0

In Figure 13-6, the Link Analyser screenshot shows the first transaction executed by static bus 16 which is an RMAP read command to read 8 Kbytes from address 0x00000000 in target 0x41.

25.00215 ms	10.100 µs	Header: RMAP Command	12.170 µs	
25.00215 ms		Target Address: 51		
25.00227 ms	110 ns	Command: Write Command	110 ns	
25.00227 ms		Data Not Verified		
25.00227 ms		Acknowledge		
25.00227 ms		Incrementing Address		
25.0023 ms	40 ns	Key: 20	40 ns	
25.00236 ms	60 ns	Initiator Address: 30	60 ns	
25.00242 ms	60 ns	Transaction ID: 4002	60 ns	
25.00253 ms	110 ns	Extended Address: 00	110 ns	
25.00259 ms	60 ns	Address: 00000000	60 ns	
25.00278 ms	190 ns	Data Length: 001FA0	190 ns	
25.00295 ms	170 ns	Header CRC: 36	170 ns	
25.00295 ms		Data CRC: C4		
25.00366 ms	700 ns	Cargo Size: 8096	700 ns	
25.80446 ms	800.800 µs			EOP
25.82282 ms	18.360 µs	EOP	819.160 µs	812.400 µs
25.82427 ms	1.450 µs			Header: RMAP Reply
25.82427 ms				Initiator Address: 30
25.82436 ms	100 ns			Command: Write Reply
25.82436 ms				Data Not Verified
25.82436 ms				Acknowledge
25.82436 ms				Incrementing Address
25.82442 ms	60 ns			Status: Success
25.82446 ms	40 ns			Target Address: 51
25.82451 ms	60 ns			Transaction ID: 4002
25.82461 ms	100 ns			Header CRC: D3
25.82463 ms	20 ns			EOP

Figure 13-7: Example Script 2 – Static Bus 16, Transaction 1

In Figure 13-7, the Link Analyser screenshot shows the second transaction executed by static bus 16 which is an RMAP write command to write 8 Kbytes to address 0x00000000 in target 0x51.

25.83747 ms	12.840 µs	Header: RMAP Command	14.650 µs	
25.83747 ms		Target Address: 61		
25.83756 ms	100 ns	Command: Read Command	100 ns	
25.83756 ms		Acknowledge		
25.83762 ms	60 ns	Key: 20	60 ns	
25.83766 ms	40 ns	Initiator Address: 30	40 ns	
25.83771 ms	60 ns	Transaction ID: 4003	60 ns	
25.83781 ms	100 ns	Extended Address: 00	100 ns	
25.83787 ms	60 ns	Address: 00000000	60 ns	
25.83806 ms	190 ns	Data Length: 001FA0	190 ns	
25.83821 ms	150 ns	Header CRC: F6	150 ns	
25.83823 ms	20 ns	EOP	20 ns	
25.83973 ms	1.500 µs			Header: RMAP Reply
25.83973 ms				Initiator Address: 30
25.83983 ms	100 ns			Command: Read Reply
25.83983 ms				Acknowledge
25.83989 ms	60 ns			Status: Success
25.83992 ms	40 ns			Target Address: 61
25.83998 ms	60 ns			Transaction ID: 4003
25.84013 ms	150 ns			Data Length: 001FA0
25.84029 ms	150 ns			Header CRC: 36
25.84029 ms				Data CRC: DC
25.84032 ms	40 ns			Cargo Size: 8096
26.24855 ms	408.230 µs			EOP

Figure 13-8: Example Script 2 – Static Bus 16, Transaction 2

In Figure 13-8, the Link Analyser screenshot shows the third transaction executed by static bus 16 which is an RMAP read command to read 8 Kbytes from address 0x00000000 in target 0x61.

48.37891 ms	1.562 ms				TIMECODE [1F]	1.562 ms
49.9409 ms	1.56198 ms				TIMECODE [20]	1.56198 ms
49.97752 ms	36.630 µs	NCHAR [42]		3.16061 ms	NULL	36.630 µs
49.97758 ms	60 ns	NCHAR [01]		60 ns	NULL	60 ns
49.97764 ms	60 ns	NCHAR [4C]		60 ns		

Figure 13-9: Example Script 2 – Time-Slot 32

In Figure 13-9, the Link Analyser screenshot shows the start of time-slot 32 which is triggered by the arrival of time-code 32 at the initiator.

49.97752 ms	23.72897 ms	Header: RMAP Command	24.1393 ms		
49.97752 ms		Target Address: 42			
49.97764 ms	110 ns	Command: Read Command	110 ns		
49.97764 ms		Acknowledge			
49.97768 ms	40 ns	Key: 20	40 ns		
49.97773 ms	60 ns	Initiator Address: 30	60 ns		
49.97777 ms	40 ns	Transaction ID: 8001	40 ns		
49.97789 ms	110 ns	Extended Address: 00	110 ns		
49.97792 ms	40 ns	Address: 00000000	40 ns		
49.97813 ms	210 ns	Data Length: 004000	210 ns		
49.97829 ms	150 ns	Header CRC: 18	150 ns		
49.9783 ms	20 ns	EOP	20 ns		
49.97973 ms	1.430 µs			Header: RMAP Reply	23.73118 ms
49.97973 ms				Initiator Address: 30	
49.97983 ms	100 ns			Command: Read Reply	100 ns
49.97983 ms				Acknowledge	
49.97989 ms	60 ns			Status: Success	60 ns
49.97994 ms	60 ns			Target Address: 42	60 ns
49.97998 ms	40 ns			Transaction ID: 8001	40 ns
49.98013 ms	150 ns			Data Length: 004000	150 ns
49.98029 ms	150 ns			Header CRC: 17	150 ns
49.98029 ms				Data CRC: 4E	
49.98034 ms	60 ns			Cargo Size: 16384	60 ns
49.99269 ms	12.340 µs	Header: RMAP Command	14.380 µs		

Figure 13-10: Example Script 2 – Static Bus 32, Transaction 0

In Figure 13-10, the Link Analyser screenshot shows the first transaction executed by static bus 32 which is an RMAP read command to read 16 Kbytes from address 0x00000000 in target 0x42.

49.99269 ms	12.340 µs	Header: RMAP Command	14.380 µs	
49.99269 ms		Target Address: 52		
49.99278 ms	100 ns	Command: Write Command	100 ns	
49.99278 ms		Data Not Verified		
49.99278 ms		Acknowledge		
49.99278 ms		Incrementing Address		
49.99282 ms	40 ns	Key: 20	40 ns	
49.99288 ms	60 ns	Initiator Address: 30	60 ns	
49.99293 ms	60 ns	Transaction ID: 8002	60 ns	
49.99303 ms	100 ns	Extended Address: 00	100 ns	
49.9931 ms	80 ns	Address: 00000000	80 ns	
49.9933 ms	190 ns	Data Length: 004000	190 ns	
49.99345 ms	150 ns	Header CRC: DB	150 ns	
49.99345 ms		Data CRC: C8		
49.9944 ms	950 ns	Cargo Size: 16384	950 ns	
51.64011 ms	1.64571 ms			EOP
51.6581 ms	17.980 µs	EOP	1.6637 ms	1.65977 ms
51.65956 ms	1.470 µs			Header: RMAP Reply
51.65956 ms				Initiator Address: 30
51.65956 ms	100 ns			Command: Write Reply
51.65956 ms				Data Not Verified
51.65956 ms				Acknowledge
51.65956 ms				Incrementing Address
51.65971 ms	60 ns			Status: Success
51.65975 ms	40 ns			Target Address: 52
51.65981 ms	60 ns			Transaction ID: 8002
51.6599 ms	100 ns			Header CRC: 9E
51.65992 ms	20 ns			EOP

Figure 13-11: Example Script 2 – Static Bus 32, Transaction 1

In Figure 13-11, the Link Analyser screenshot shows the second transaction executed by static bus 32 which is an RMAP write command to write 16 Kbytes to address 0x00000000 in target 0x52.

51.6685 ms	8.570 µs	Header: RMAP Command	10.400 µs	
51.6685 ms		Target Address: 62		
51.66861 ms	110 ns	Command: Read Command	110 ns	
51.66861 ms		Acknowledge		
51.66865 ms	40 ns	Key: 20	40 ns	
51.6687 ms	60 ns	Initiator Address: 30	60 ns	
51.66874 ms	40 ns	Transaction ID: 8003	40 ns	
51.66886 ms	110 ns	Extended Address: 00	110 ns	
51.6689 ms	40 ns	Address: 00000000	40 ns	
51.6691 ms	210 ns	Data Length: 004000	210 ns	
51.66926 ms	150 ns	Header CRC: 1B	150 ns	
51.66928 ms	20 ns	EOP	20 ns	
51.6707 ms	1.430 µs			Header: RMAP Reply
51.6707 ms				Initiator Address: 30
51.67082 ms	110 ns			Command: Read Reply
51.67082 ms				Acknowledge
51.67086 ms	40 ns			Status: Success
51.67091 ms	60 ns			Target Address: 62
51.67097 ms	60 ns			Transaction ID: 8003
51.6711 ms	130 ns			Data Length: 004000
51.67126 ms	150 ns			Header CRC: AA
51.67126 ms				Data CRC: 1B
51.67131 ms	60 ns			Cargo Size: 16384
52.49093 ms	819.620 µs			EOP

Figure 13-12: Example Script 2 – Static Bus 32, Transaction 2

In Figure 13-12, the Link Analyser screenshot shows the third transaction executed by static bus 32 which is an RMAP read command to read 16 Kbytes from address 0x00000000 in target 0x62.

Measuring the time between the first character of the first transaction and the last character of the third transaction gives a total execution time of 2.513 ms. As this is more than the time-slot duration of 1.562 ms, this shows that static bus 32 is executing its transaction group over a multi-slot.

51.5028 ms	60 ns	NCHAR [53]		60 ns			
51.50284 ms	40 ns	NCHAR [DB]		40 ns			
51.5029 ms	60 ns	NCHAR [55]		60 ns	TIMECODE [21]		150 ns
51.50299 ms	100 ns	NCHAR [11]		100 ns	NULL		100 ns
51.50303 ms	40 ns	NCHAR [60]		40 ns	NULL		40 ns

Figure 13-13: Example Script 2 – Multi-Slot Time-Code

In Figure 13-13, the Link Analyser screenshot shows the point at which the transaction group crosses over the boundary between time-slots 32 and 33. Time-code 33 is received by the initiator but it doesn't trigger the start of a new time-slot because the multi-slot of static bus 32 is still executing.

73.37063 ms	1.56196 ms	NULL		3.12394 ms	TIMECODE [2F]		1.56196 ms
74.93265 ms	1.56202 ms				TIMECODE [30]		1.56202 ms
74.97067 ms	38.020 µs	NCHAR [43]		1.60004 ms			
74.9707 ms	40 ns	NCHAR [01]		40 ns			
74.97076 ms	60 ns	NCHAR [4C]		60 ns	NULL		38.110 µs

Figure 13-14: Example Script 2 – Time-Slot 48

In Figure 13-14, the Link Analyser screenshot shows the start of time-slot 48 which is triggered by the arrival of time-code 48 at the initiator.

74.97067 ms	22.47973 ms	Header: RMAP Command	23.30139 ms		
74.97067 ms		Target Address: 43			
74.97076 ms	100 ns	Command: Read Command	100 ns		
74.97076 ms		Acknowledge			
74.97082 ms	60 ns	Key: 20	60 ns		
74.97086 ms	40 ns	Initiator Address: 30	40 ns		
74.97091 ms	60 ns	Transaction ID: C001	60 ns		
74.97101 ms	100 ns	Extended Address: 00	100 ns		
74.97107 ms	60 ns	Address: 00000000	60 ns		
74.97126 ms	190 ns	Data Length: 000100	190 ns		
74.97141 ms	150 ns	Header CRC: AA	150 ns		
74.97143 ms	20 ns	EOP	20 ns		
74.97291 ms	1.490 µs			Header: RMAP Reply	22.48198 ms
74.97291 ms				Initiator Address: 30	
74.97301 ms	100 ns			Command: Read Reply	100 ns
74.97301 ms				Acknowledge	
74.97305 ms	40 ns			Status: Success	40 ns
74.9731 ms	60 ns			Target Address: 43	60 ns
74.97316 ms	60 ns			Transaction ID: C001	60 ns
74.97331 ms	150 ns			Data Length: 000100	150 ns
74.97345 ms	130 ns			Header CRC: 35	130 ns
74.97345 ms				Data CRC: 44	
74.9735 ms	60 ns			Cargo Size: 256	60 ns
74.98362 ms	10.110 µs	Header: RMAP Command	12.190 µs		

Figure 13-15: Example Script 2 – Static Bus 48, Transaction 0

In Figure 13-15, the Link Analyser screenshot shows the first transaction executed by static bus 48 which is an RMAP read command to read 256 bytes from address 0x00000000 in target 0x43.

74.98362 ms	10.110 µs	Header: RMAP Command	12.190 µs	
74.98362 ms		Target Address: 53		
74.98371 ms	100 ns	Command: Write Command	100 ns	
74.98371 ms		Data Not Verified		
74.98371 ms		Acknowledge		
74.98371 ms		Incrementing Address		
74.98377 ms	60 ns	Key: 20	60 ns	
74.98381 ms	40 ns	Initiator Address: 30	40 ns	
74.98389 ms	80 ns	Transaction ID: C002	80 ns	
74.98398 ms	100 ns	Extended Address: 00	100 ns	
74.98404 ms	60 ns	Address: 00000000	60 ns	
74.98423 ms	190 ns	Data Length: 000100	190 ns	
74.9844 ms	170 ns	Header CRC: 69	170 ns	
74.9844 ms		Data CRC: 5D		
74.98522 ms	820 ns	Cargo Size: 256	820 ns	
74.98989 ms	4.670 µs			EOP
75.00598 ms	16.100 µs	EOP	20.760 µs	
75.00747 ms	1.490 µs			Header: RMAP Reply
75.00747 ms				Initiator Address: 30
75.00756 ms	100 ns			Command: Write Reply
75.00756 ms				Data Not Verified
75.00756 ms				Acknowledge
75.00756 ms				Incrementing Address
75.0076 ms	40 ns			Status: Success
75.00766 ms	60 ns			Target Address: 53
75.00771 ms	60 ns			Transaction ID: C002
75.00781 ms	100 ns			Header CRC: 1A
75.00783 ms	20 ns			EOP

Figure 13-16: Example Script 2 – Static Bus 48, Transaction 1

In Figure 13-16, the Link Analyser screenshot shows the second transaction executed by static bus 48 which is an RMAP write command to write 256 bytes to address 0x00000000 in target 0x53.

75.01855 ms	10.720 µs	Header: RMAP Command	12.570 µs	
75.01855 ms		Target Address: 63		
75.01865 ms	100 ns	Command: Read Command	100 ns	
75.01865 ms		Acknowledge		
75.01869 ms	40 ns	Key: 20	40 ns	
75.01874 ms	60 ns	Initiator Address: 30	60 ns	
75.0188 ms	60 ns	Transaction ID: C003	60 ns	
75.0189 ms	100 ns	Extended Address: 00	100 ns	
75.01895 ms	60 ns	Address: 00000000	60 ns	
75.01914 ms	190 ns	Data Length: 000100	190 ns	
75.0193 ms	150 ns	Header CRC: A9	150 ns	
75.01931 ms	20 ns	EOP	20 ns	
75.02076 ms	1.450 µs			Header: RMAP Reply
75.02076 ms				Initiator Address: 30
75.02088 ms	110 ns			Command: Read Reply
75.02088 ms				Acknowledge
75.02091 ms	40 ns			Status: Success
75.02097 ms	60 ns			Target Address: 63
75.02101 ms	40 ns			Transaction ID: C003
75.02116 ms	150 ns			Data Length: 000100
75.02131 ms	150 ns			Header CRC: 88
75.02131 ms				Data CRC: F7
75.02137 ms	60 ns			Cargo Size: 256
75.03851 ms	17.140 µs			EOP

Figure 13-17: Example Script 2 – Static Bus 48, Transaction 2

In Figure 13-17, the Link Analyser screenshot shows the third transaction executed by static bus 48 which is an RMAP read command to read 256 bytes from address 0x00000000 in target 0x63.