# SpaceWire-D on the Castor Spaceflight Processor

SpaceWire Networks and Protocols, Long Paper

David Gibson, Steve Parkes
Space Technology Centre
University of Dundee
Dundee, Scotland
{dzgibson, smparkes}@dundee.ac.uk

Chris McClements, Stuart Mills, David Paterson
STAR-Dundee
Dundee, Scotland

***Abstract*—SpaceWire-D is a deterministic extension to the SpaceWire protocol designed to satisfy hard real-time constraints on a SpaceWire network. This allows a single SpaceWire network to be used for both control applications and payload data-handling.**

**The Atmel AT6981 Castor device is a LEON2-FT based system-on-chip with multiple integrated peripherals including an eight-port SpaceWire router and three internal SpaceWire engines each containing three DMA channels, an RMAP initiator, and an RMAP target.**

**This paper describes the SpaceWire-D protocol; the design of RTEMS networking software to test the protocol using the AT6981 system-on-chip; and the results of those tests.**

***Index Terms*— SpaceWire, SpaceWire-D, deterministic networks, spacecraft onboard processing, AT6981**

## I. Introduction

SpaceWire-D is a deterministic extension to the SpaceWire on-board data handling network [1] being designed by the University of Dundee for ESA [2] [3]. To provide a deterministic capability, SpaceWire-D uses time-division multiplexing and slices network time into a series of time-slots in which RMAP [4] transactions are executed. These transactions are grouped into a virtual bus system, where each bus consists of an initiator node, one or more target nodes, and the set of links that make up the paths between the nodes.

Figure 1 shows an example of a virtual bus with an initiator, three targets, and five links. The semi-transparent nodes and links are not part of the virtual bus.

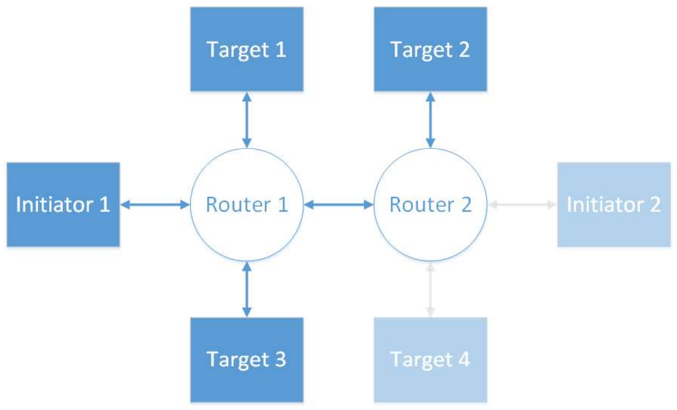


*Fig. 1. Example of a virtual bus with three targets and five links*

Due to the wormhole routing used by SpaceWire enabled routers, if there are multiple data-flows in a SpaceWire network there is a possibility of a packet being blocked if one of the SpaceWire links it requires is already in use. There may be more than one initiator operating in a SpaceWire-D network at the same time, so a set of initiator schedules is required to constrain traffic such that no two virtual buses are active in the same slot if there is a chance that they could have a colliding transaction i.e. if they have any shared links.

## II. SpaceWire-D

The following subsections briefly describe the features of SpaceWire-D. For more in-depth coverage, see the standard draft [2] and [3].

### *A. Time-Division Multiplexing*

In a SpaceWire-D network, the end of the current time-slot and the beginning of the next time-slot is signaled by the arrival of the next valid time-code. SpaceWire time-codes contain a 6-bit time value, so there are 64 slots in a SpaceWire-D schedule beginning at slot 0 and ending at slot 63. Additionally, a local timer can be used to synchronise with the arriving time-codes to provide redundancy in case a time-code fails to arrive.

Each time-slot can be assigned a single virtual bus. However, this is not a symmetric relationship because depending on the type of virtual bus, a bus may be assigned to multiple time-slots or adjacent sequences of time-slots called multi-slots, as described in the following sections.

When a new time-slot begins, if there is a virtual bus assigned to the time-slot, the group of transactions associated with the virtual bus is executed.

### *B. Static Bus*

The SpaceWire-D protocol provides services to open, load, execute, and close four different types of virtual buses. The first and simplest virtual bus is the static bus.

Each static bus is assigned to a single time-slot or single multi-slot. Once opened, the user application can then load the static bus with a group of RMAP transactions. During the loading operation, the transaction group's worst-case execution time (WCET) is checked before the transaction group is accepted into the static bus. If the WCET of the transaction group exceeds the duration of the time-slot or multi-slot it may interfere with the next slot's transactions, so the transaction group is not loaded and an erroneous response is sent to the user application.

A transaction group can be loaded as a repeating group in which case it is repeated every time the bus's time slot occurs until the bus is reloaded or closed, or as a single shot group where the transaction group is unloaded after a single execution.

### *C. Dynamic Bus*

A dynamic bus can be assigned to multiple time-slots or multi-slots. When a transaction group is loaded, its WCET is checked, like the static bus, before it is accepted. If a transaction group is accepted and loaded into a dynamic bus, it is executed in the next time-slot or multi-slot assigned to the bus. This results in less predictability than a static bus because a transaction group could be executed in one of multiple time-slots.

### *D. Asynchronous Bus*

As with a dynamic bus, an asynchronous bus can be assigned to multiple time-slots or multi-slots.

However, unlike the static bus and dynamic bus, which are based around loading groups of transactions, the asynchronous bus works on a single transaction basis. When a user application loads an asynchronous bus, it sends a data structure describing a single transaction along with the transaction's priority. The asynchronous bus maintains a prioritised queue of transactions, and in each available time-slot or multi-slot assigned to the bus, a subset of the highest priority transactions is removed from the queue and executed. The subset of transactions to be executed in the next available time-slot or multi-slot is updated whenever the user application loads a new transaction.

### *E. Packet Bus*

The packet bus is a bi-directional channel between an initiator node and a target node. Receiving packets from and sending packets to a target are controlled by the initiator via RMAP read and write operations, respectively.

An initiator node can open multiple channels to targets and a target can open multiple channels to initiators. When the channel has been opened on both the initiator and target side, the packet bus is ready to handle RMAP transactions between the two nodes.

When a packet bus's time-slot or multi-slot begins, the status of all channels is checked to make sure a channel is not busy before it is used by the packet bus. This allows multiple initiators to open a channel to the same target and reserve it for exclusive use.

Optionally, the packet bus can use segmentation to split the transmission or receiving of a large packet over multiple time-slots or multi-slots.

### *F. Schedules*

The source of unpredictability in a SpaceWire network is the possibility of packets being blocked by wormhole routing. Wormhole routing enables a packet to be switched from an input port to an output port quickly, but only if the output port is not already in use. If it is in use, the packet is blocked until the output port is released.

In order for traffic in a SpaceWire-D network to be deterministic, the possibility of blocking must be removed. This is done by ensuring that in each time-slot, the set of links used by an initiator's virtual bus is distinct from the set used by every other initiator's virtual bus operating within the same time-slot. If no link is used by two buses at the same time, then the blocking of SpaceWire packets cannot occur. This means that for each initiator, a schedule must be created that simultaneously satisfies this constraint and meets the bandwidth demands of a mission.

Research into the configuration of schedules for SpaceWire-D networks is ongoing at the University of Dundee and elsewhere [5] [6].

Figure 2 shows an example schedule for a single initiator with 64 time-slots and a combination of different virtual bus types [3].

| Time-Slot | Bus |
|---|---|
| 0 | Static 0 |
| 1 | Dynamic 1 |
| 2 | Static 2 |
| 3 | Async 3 |
| 4 | Static 4 |
| 5 | Async 5 |
| 6 | Async 5 |
| 7 | Dynamic 7 |
| 8 | Empty |
| 9 | Dynamic 1 |
| 10 | Dynamic 7 |
| 11 | Packet 11 |
| 12 | Packet 11 |
| 13 | Packet 11 |
| 14 | Packet 11 |
| … | |
| 61 | Static 61 |
| 62 | Dynamic 7 |
| 63 | Static 63 |

*Fig. 2. Schedule for a single initiator with 64 time-slots [3]*

## III. AT6981 Castor System-On-Chip

The Atmel AT6981 Castor system-on-chip [7] is a LEON2-FT (SPARC V8 ISA) based flight processor with multiple integrated peripherals including extensive SpaceWire support.

Figure 3 shows a photo of the cPCI variant of the prototype AT6981 board, with three SpaceWire cables connected to the router

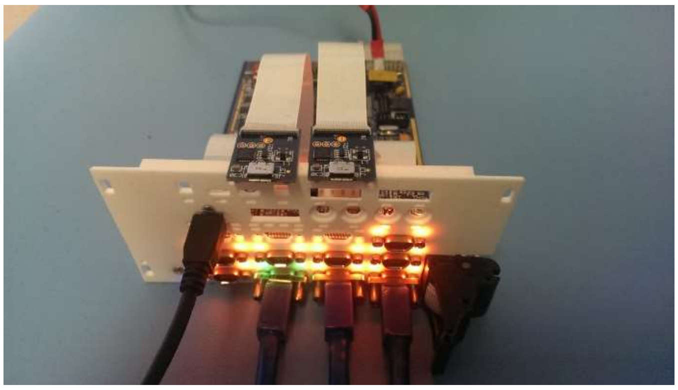


*Fig. 3. cPCI variant of the prototype AT6981 board*

The following subsections briefly describe the relevant SpaceWire peripherals, and features in the prototype board used for this research.

### *A. SpaceWire Router*

The SpaceWire front-end for the AT6981 board is a SpaceWire router with eight external ports and three internal ports connected to the SpaceWire engines. The internal ports have physical addresses 9, 10, and 11 which connect to SpaceWire engines 1, 2, and 3 respectively.

### *B. SpaceWire Engines*

Connected to the SpaceWire router, the three SpaceWire engines each contain three DMA channels, an RMAP initiator, and an RMAP target. The SpaceWire-D tests described in this paper use only the RMAP functionality in the engines.

In order to allow a SpaceWire packet to address individual DMA channels, RMAP initiator, or RMAP target, the SpaceWire engines use a de-multiplexer. The de-multiplexer matches up to four bytes of the incoming packet against a pattern and mask configured by the user in the engine's registers. It then uses this matching to filter the packet into the correct DMA channel, RMAP initiator, or RMAP target. This allows the RMAP initiator and target to have their own logical addresses.

The execution of RMAP commands are offloaded to the SpaceWire engines, reducing the demands on the LEON2-FT processor. The user application holds a list of data structures in memory describing the required RMAP commands and then writes the memory address of the list to the RMAP initiator's registers. Consequently, if the list of commands is unchanging over time as in the case of a static bus with a repeating transaction group, the processing required to begin executing the transactions is minimal.

### *C. Memory and Processor*

The prototype AT6981 board has 128Kbyte of SRAM and 256MByte of DRAM and the LEON2-FT processor clock rate is 33MHz, while the production board will run at 200MHz.

### *D. Debug Support Unit*

Loading and debugging a program is done via the LEON2-FT debug support unit (DSU). The DSU provides a simple protocol to read and write to memory on the board directly through hardware. This allows software running on the development machine to load a program directly into the AT6981's memory without the requirement of a bootloader. To debug a program, a STAR-Dundee software module on the development machine acts as a GDB remote protocol server and translates GDB commands into interactions with the DSU, allowing a simple method for debugging. The AT6981 prototype board provides a USB to UART bridge for connecting a computer to the DSU.

## IV. RTEMS SUPPORT

The tests described in this paper use version 4.10.2 of the RTEMS real-time operating system, which is an open-source project being used in many space applications as well as in other industries [8]. The following subsections describe our use of RTEMS and its relevant features.

### *A. Board Support Package*

A board support package (BSP) was designed to port RTEMS to the AT6981 board [9]. The basic BSP consists of the minimum requirements to run the basic RTEMS tests and examples. This includes the board initialisation code, a UART console driver, a clock driver, and support files such as a linker script file. There exists a BSP for an existing LEON2 device in the RTEMS source tree, however, the AT6981 is sufficiently different that it requires a separate BSP.

The AT6981 BSP uses the LEON2-FT's on-chip UARTs and timers with slightly modified drivers from the existing LEON2 BSP. Like the DSU UART, the LEON2-FT on-chip UARTs are accessible through USB to UART bridges on the prototype board.

As the AT6981 shares an interrupt between SpaceWire DMA and RMAP engines in the primary interrupt controller, the interrupt handling has been extended to allow an interrupt service routine (ISR) to be registered for either DMA or RMAP interrupts. When an interrupt is raised on the primary controller, the interrupt handler then filters it to the relevant ISR. This allows for separate device drivers for DMA and RMAP engines.

### *B. RTEMS Features*

RTEMS is a real-time multi-task operating system with a unified address space. It provides features common in most operating systems such as tasks, interrupt handling, inter-process communication, synchronisation, standard data structures, and a device driver framework. RTEMS also provides in-depth compile time customisation.

A real-time operating system is designed to value predictability above other features [10]. As RTEMS is a real-time operating system, it provides task scheduling algorithms relevant to a real-time environment. In our case, we are using the default priority based pre-emptive scheduler which will switch context to a higher priority task if one becomes available at any time.

## V. Networking Software

RTEMS based networking software is responsible for providing the SpaceWire-D API to the user application, managing the virtual buses, managing the transition between time-slots, and dispatching RMAP commands.

The following subsections describe the different modules of the SpaceWire-D test software.

### *A. SpaceWire-D API*

The SpaceWire-D API provides a public interface to the user application and enables an application to initialise the other SpaceWire-D modules, open a virtual bus, load a virtual bus, and close a virtual bus. During initialisation, the API creates tasks for the other modules as well as a task for itself and uses the RTEMS message queue manager in order to listen for requests from user applications. These requests are then handled by the virtual bus manager.

### *B. Virtual Bus Manager*

All functionality related to the opening, loading, and closing of virtual buses is controlled by the virtual bus manager. It also contains the data structures describing the parameters of a virtual bus and its transactions.

### *C. Time Manager*

The time manager is responsible for transitioning between time-slots, based on the arrival of valid time-codes. In this version, we are using only time-codes to signal the beginning and end of time-slots. However, the standard also describes the use of local timers to synchronise with the arrival of time-codes for redundancy in case a time-code fails to arrive.

During initialisation of the time manager, we enable interrupts for the receiving of time-codes using a simple device driver for the AT6981 SpaceWire router. We then install a callback function which is called during the router driver's ISR. The callback function uses the RTEMS event manager to send an event to the transaction dispatcher, signalling the start of the next time-slot.

### *D. Transaction Dispatcher*

When the SpaceWire-D API initialises the other modules, a task is created for the transaction dispatcher. This task begins and then blocks, waiting for an event to be received from the time manager. The task wakes up when the event is received and, if there is a virtual bus assigned to the time-slot, executes the virtual bus. For example, if there is a static bus assigned to the time-slot, the bus's transaction group will be executed, assuming one is loaded.

A simple RMAP driver was designed to provide three features: the first is a function to start a group of RMAP transactions, the second is an ISR to handle RMAP initiator interrupts, and the third is a function to initialise one of the AT6981's RMAP targets to act as a target node for the purpose of the experiments.

## VI. Experimental Setup

For our experiments we used two different network architectures. The following subsections describe and illustrate both architectures, and the additional supporting hardware and software used.

### *A. Single Initiator Architecture*

The single initiator architecture as shown in Figure 4, uses a single AT6981 board as an RMAP initiator and RMAP target. The AT6981 is connected to a development machine for loading and debugging programs. The board's router loops back to itself with a STAR-Dundee SpaceWire Link Analyser Mk2 in the middle, to view the transactions and time-codes flowing through the links. A STAR-Dundee SpaceWire-USB Brick Mk2 is connected to the AT6981 and acts as the time-code master. The Link Analyser Mk2 and the Brick Mk2 are connected to a second laptop for ease of use. The Brick's time-code generation is controlled via STAR-Dundee's STAR-System software [11]. In this architecture, all SpaceWire links are operating at 100Mbps.

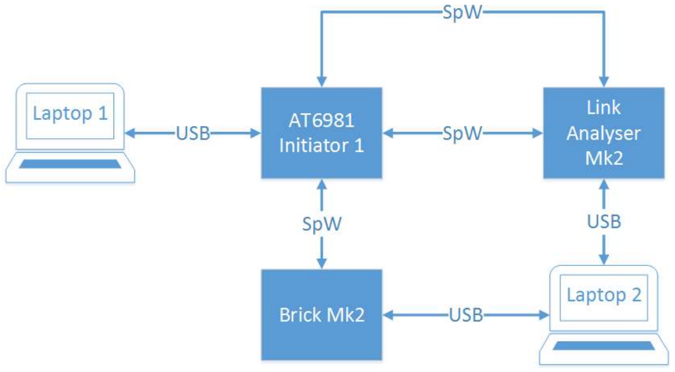


*Fig. 4. Single initiator architecture*

## VII. Multiple Initiator Architecture

The multiple initiator architecture shown in Figure 5 is similar to the single initiator architecture shown in Figure 4 with the exception that the loopback through the Link Analyser is removed and replaced by a link between both AT6981 boards, again through a Link Analyser. In this architecture, the first AT6981 board's SpaceWire links are operating at 100Mbps and the second board's links are running at 50Mbps.

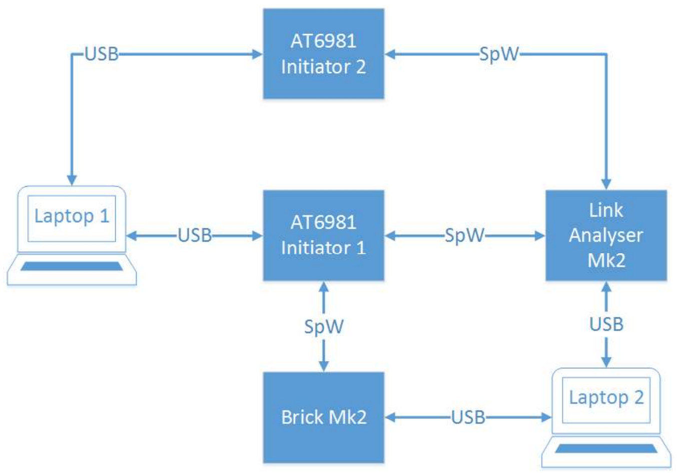


*Fig. 5. Multiple initiator architecture*

Figure 6 shows a photo of the hardware used in the multiple initiator architecture setup. From left to right, the hardware is the first AT6981 prototype board, a STAR-Dundee SpaceWire Link Analyser Mk2, the second AT6981 prototype board, and a STAR-Dundee SpaceWire-USB Brick Mk2

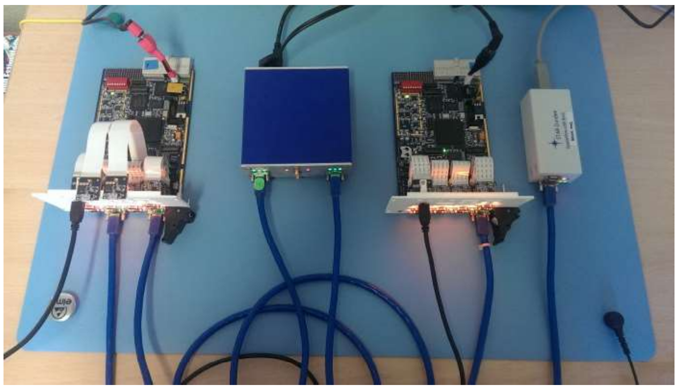


*Fig. 6. Photo of the multiple initiator architecture*

## VIII. Experimental Results

The following subsections describe the experiments carried out to test the SpaceWire-D static bus with both single and multiple initiator architectures, and presents the results obtained.

### A. RMAP Driver

The first iteration of the RMAP driver used by the transaction dispatcher utilised the UNIX-like device file driver framework provided by RTEMS. Within this framework, every device is treated as a file and a driver provides initialise, open, close, read, write, and ioctl functions to be used with standard system calls.

After performing some initial tests and measuring the performance of the driver, it was found that the overhead required by opening a device file and using an ioctl system call when dispatching a transaction group was too expensive. By allowing the transaction dispatcher to call the driver functions directly instead of through the ioctl system call interface, the processing time between a time-code being received and the first RMAP transaction leaving the router was reduced from 741µs to 389µs.

Further optimisation was introduced by simplifying the event handling when a time-code is received. Originally, the time manager would receive an event from the router ISR, then forward the event to the transaction dispatcher. By sending the event directly from the time manager's callback function, the processing time was further reduced from 389µs to 201µs.

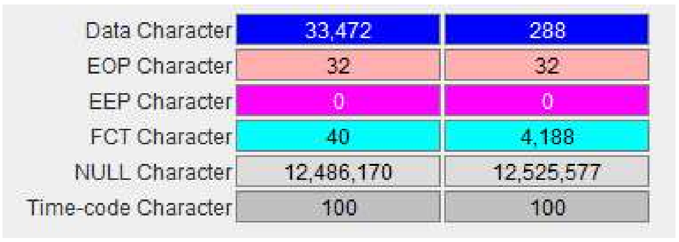
Reducing the initiator processing time between a time-code being received and the first RMAP transaction leaving the router from 741µs to 201µs allows the SpaceWire-D network to run at the minimum slot duration of 1ms. With the production version of the AT6981 running at 200MHz, compared to the prototype's 33MHz, and with additional software optimisations, the initiator processing time should be further reduced.

### B. Single Initiator Experiments

In the single initiator architecture, the AT6981 board acts as both the RMAP initiator and the RMAP target. During the test setup, the SpaceWire-D API is initialised, the RMAP target is initialised, and the test static buses are opened and loaded with a transaction group.

The first experiment involved opening a single static bus in slot 0 and loading it with a transaction group containing 32 RMAP write-with-reply commands with a data size of 1KB. The Brick is generating time-codes every 10ms.

Figure 7 shows a screenshot of the Link Analyser status counter display interface for the first experiment. In this interface, the number of various types of characters received per second are displayed. The first column is the Link Analyser port that the RMAP headers are transmitting through, and the second column is the RMAP replies. We can see that there are 32 RMAP transactions being executed by viewing the number of EOP characters and confirming that the commands were executed successfully by viewing the packet display interface within the Link Analyser software. The number of data characters being transmitted per second can be verified by calculating the size of the RMAP headers and replies. For the header, there is 1 physical address at the head of the packet, a 21 byte RMAP header, and 1024 bytes of data. This results in 1046 data characters multiplied by 32 which is 33472 data characters per second. The reply has 1 physical address at the head of the packet, and an 8 byte reply, which results in 9 data characters multiplied by 32, giving 288 data characters per second.

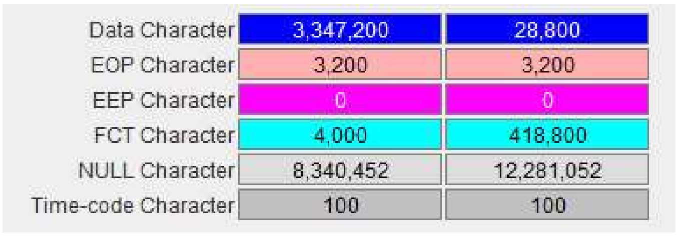


| | | |
|---|---|---|
| Data Character | 33,472 | 288 |
| EOP Character | 32 | 32 |
| EEP Character | 0 | 0 |
| FCT Character | 40 | 4,188 |
| NULL Character | 12,486,170 | 12,525,577 |
| Time-code Character | 100 | 100 |

*Fig. 7. Link Analyser output for a single slot schedule*

Next, we opened a static bus on all 64 slots and loaded them with the same transaction group as the previous experiment.

Figure 8 shows the results from opening a static bus on all 64 slots. Again, the number of data characters transmitted can be verified by multiplying 1046 data characters by 3200 in this case, which is 3,347,200 data characters per second. Similarly the data characters per second for the replies is calculated by multiplying 9 data characters by 3200 which is 28800.



| | | |
|---|---|---|
| Data Character | 3,347,200 | 28,800 |
| EOP Character | 3,200 | 3,200 |
| EEP Character | 0 | 0 |
| FCT Character | 4,000 | 418,800 |
| NULL Character | 8,340,452 | 12,281,052 |
| Time-code Character | 100 | 100 |

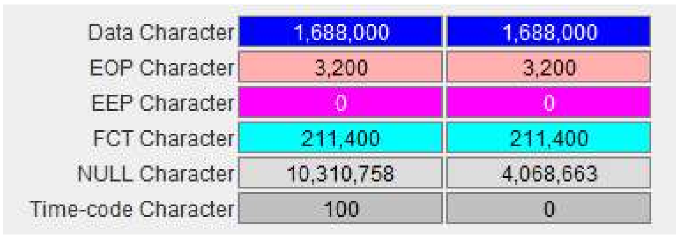
*Fig. 8. Link Analyser output for a 64 slot schedule*

In both cases of the single slot and the 64 slot schedule, the observed WCET of the transaction group is 4182µs. The observed worst-case processing time of 201µs can be added to this to give a total static bus execution time of 4383µs.

## *C. Multiple Initiator Experiments*

In the multiple initiator architecture, there are two AT6981 both acting as initiators and as targets for each other. The schedule in this experiment is split between the two boards. In all of the even numbered time-slots, the first board opens a static bus. The second board opens a static bus in all of the odd numbered time-slots. Each static bus is loaded with the same transaction group as the single initiator experiments, 32 RMAP write-with-reply transactions with a data size of 1KB.

Figure 9 shows a screenshot of the Link Analyser status counter display for the multiple initiator architecture experiment. In this case, both RMAP headers and RMAP replies are travelling bidirectionally through both links. To verify the number of data characters being transmitted per second for each side of the Link Analyser, we can add the data characters for both the RMAP headers and the replies. In this case, there are 1600 transactions being executed every second by each initiator. This results in 1600 multiplied by 1046 data characters for the RMAP headers, which is 1,673,600 data characters per second. For the RMAP replies, there is 1600 multiplied by 9 data characters, which is 14400 data characters per second. Summing the two gives 1,688,000 as supported by the screenshot.



| | | |
|---|---|---|
| Data Character | 1,688,000 | 1,688,000 |
| EOP Character | 3,200 | 3,200 |
| EEP Character | 0 | 0 |
| FCT Character | 211,400 | 211,400 |
| NULL Character | 10,310,758 | 4,068,663 |
| Time-code Character | 100 | 0 |

*Fig. 9. Link Analyser output for the multiple initiator architecture*

## IX. Future Work

The experiments described in this paper were focused on parts of the static bus of SpaceWire-D networks. Further work is required to test the remaining features of the static bus such as transaction group execution time calculation and multi-slot buses. Additionally, the remaining virtual bus types: the dynamic bus, the asynchronous bus, and the packet bus require similar experimentation and testing. Future research will be carried out to fulfil these goals.

As mentioned in Section 2, the schedulability of SpaceWire-D networks is an important problem. Research is being carried out to investigate scheduling methods for the latest draft of the standard.

## X. Conclusions

This paper has briefly described the latest version of SpaceWire-D [3] and presented the results from experiments using the AT6981 [7] prototype board, an RTEMS port for the AT6981 [9], and RTEMS based networking software to test the static bus functionality of SpaceWire-D.

The results show that the AT6981 prototype board can be used to operate a SpaceWire-D network using the static bus with schedules utilising single slots and all 64 slots. An experiment was successfully carried out to test a SpaceWire-D network with two AT6981 boards acting as RMAP initiators operating in alternating time-slots.

## Acknowledgments

The research leading to these results has received funding from the European Space Agency under ESA contract number 4000107346/12/NL/LvH/fe. We would also like to thank David Jameux the ESA project manager for the SpaceWire-D related activity.

## References

[1] ECSS Standard ECSS-E-ST-50-12C, “SpaceWire – Links, nodes, routers and networks”, Issue 2, European Cooperation for Space Standardization, 31 July 2008, available from http://www.ecss.nl

[2] S. Parkes, A. Ferrer Florit, A. Gonzalez Villafranca, C. McClements, “SpaceWire-D Deterministic Control and Data Delivery Over SpaceWire Networks”, Draft C, April 2012, available from http://spacewire.esa.int/WG/SpaceWire

[3] S. Parkes, D. Gibson, A. Ferrer, “SpaceWire-D: Deterministic Data Delivery over SpaceWire”, Data Systems in Aerospace, Warsaw, Poland, June 2014

[4] ECSS Standard ECSS-E-ST-50-52C, “SpaceWire - Remote memory access protocol”, Issue 1, European Cooperation for Space Standardization, 5 February 2010, available from http://www.ecss.nl

[5] D. Raszhivin, Y. Sheynin, A. Abramov, “Deterministic Scheduling of SpaceWire Data Streams”, International SpaceWire Conference, Gothenburg, Sweden, June 2013

[6] Y. Chen, M. Takada, R. Kurachi, H. Takada, “A Scheduling Method of RMAP Packets for SpaceWire-D”, International SpaceWire Conference, Gothenburg, Sweden, June 2013

[7] S. Parkes, C. McClements, G. Mantelet, N. Ganry, “The Next Generation of Spaceflight Processors: Low Power, High Performance, with Integrated SpaceWire Router and Protocol Engines”, International Astronautical Congress, Beijing, China, September 2013

[8] RTEMS, “RTEMS Applications”, Available at: http://rtems.org/wiki/index.php/RTEMSApplications

[9] D. Paterson, D. Gibson, S. Parkes, “An RTEMS Port for the AT6981 SpaceWire-Enabled Processor: Features and Performance”, International SpaceWire Conference, Athens, Greece, September 2014

[10] J. Stankovic, R. Rajkumar, “Real-Time Operating Systems”, Real-Time Systems, vol 28.2-3, pp. 237-253, 2004

[11] S. Mills, A. Mason, C. McClements, D. Paterson, I. Martin, S. Parkes, “Developing SpaceWire Devices with STAR-Dundee Test and Development Equipment”, International SpaceWire Conference, Gothenburg, Sweden, June 2013