Deterministic SpaceWire Network Stack for ARINC 653 RTOS

Vladislav Aleinik
Ivannikov Institute for System Programming
of the Russian Academy of Sciences
Moscow, Russia
valeinik@ispras.ru

Vitaly Cheptsov
Ivannikov Institute for System Programming
of the Russian Academy of Sciences
Moscow, Russia
cheptsov@ispras.ru

***Abstract*—Modern Integrated Modular Avionics (IMA) systems require a Real-Time Operating System (RTOS) for robust resource partitioning. In the same manner, IMA systems require a deterministic network to guarantee timely communication between modules of an onboard system. However, unlike RTOS, the interface and behaviour of which is well-documented in the ARINC 653 standard, determinstic network architecture is much less standardized and depends on the RTOS implementation and the underlying protocol stack. In this paper we present the network stack architecture for deterministic SpaceWire network implemented in ARINC 653 RTOS. The design decisions presented in the paper were implemented and verified on different types of space equipment.**

***Index Terms*—IMA, deterministic networking, real-time operating systems, ARINC 653, SpaceWire**

I. Introduction

Modern approach to development of airbourne hard real-time systems — called Integared Modular Avionics (IMA) — is to execute application software in an environment of Real-Time Operating System (RTOS) that ensures robust partitioning of CPU time, memory and devices. IMA allows to simplify system verification by reducing it to seperate verification of isolated applications and static analysis of compatibility of application requirements.

According to IMA control modules, sensors and actuators are composed into a single deterministic network. Similarly, network interaction composability is required: seperate verification of network interaction of two hosts and static analysis of compatibility of different network interactions must be enough to guarantee robust operation of system as a whole.

ARINC 653 [1] standardizes the interface given to system integrator and application developer, however network stack architecture is not standardized and is thus invented by the RTOS developer.

Network stack architecture may be dictated by the transport layer protocol, as it is done ARINC 664P7 (AFDX) [2]. As for SpaceWire protocol [3] used in space missions, there are several competing architectures of deterministic networks, however not all of them are compatible with ARINC 653.

In this paper we list and specify higher-level requirements to the deterministic SpaceWire network stack implemented in ARINC 653 RTOS (section II) and describe the algorithm of developing a driver under ARINC 653 (section III). Proposed network stack architecture (section IV) has numerous benefits:

- Unified architecture allows to simplify the process of porting SpaceWire drivers to new platforms.
- Network scheduling allows to meet the requirements of deterministic packet delivery.
- Coordinated packet buffering in network stack allows to eliminate packets being dropped on transmission and to decrease transmission latency.
- Driver Finite State Machine allows to handle incoming and outgoing DMA transfers with bounded processing time.

Proposed network stack architecture is implemented in CLOS, an RTOS developed at ISP RAS, and tested on hardware platforms used in space industry (section V).

II. Requirement Analysis

For Spacewire network stack architecture there are generally four sources of requriements:

1) **Abstract model of deterministic network**.
2) **ARINC 653 standard for RTOS interface**.
3) **Deterministic SpaceWire protocols**.
4) **Considerations of user-friendliness to system integrator**.

A. DetNet requirements

A thorough analysis of requirements for deterministic network and the resulting architecture can be found in RFC 8655 [4]. Quality of Sevice (QoS) is defined in terms of parameters:

- Minimum and maximum end-to-end latency from source to destination.
- Packet delay variation (jitter).
- Packet loss ratio.

To optimize these parameters a set of techniques is proposed:

1) **Explicit static route allocation**. Dynamic routing is inapplicable to deterministic networks as it adds routing convergence time to the worst case of packet delivery latency.
2) **Static allocation of buffer space and bandwidth along the flow**. A simplest way to introduce packet loss or

additional packet delay is a configuration of two data flows colliding in a switch. The approach of dynamic congestion control used in TCP is inaplicable to deterministic networks as it is based on packet loss in the first place.

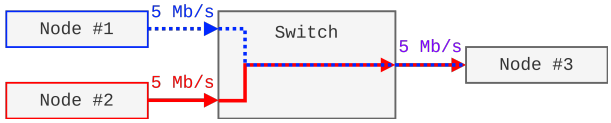

Figure 1: Introduction of packet loss or packet delay during switching

3) **Fixed time of packed delivery**. Static periodic scheduling of application software allows to seperate time of packet reception from the network and time of it's delivery to user application, decreasing network jitter to RTOS scheduler jitter.

B. ARINC 653 requirements

According to ARINC 653 applications are executed in partitions. Each partition has isolated address space, it's own set of OS objects (file descriptors, synchronization primitives, etc.), a set of executed processes and their respective time windows in the periodic schedule.

In order to minimize the amound of RTOS code excuted with elevated privilege, network stacks (and file systems, and device drivers) are placed in dedicated system partitions. System partitions resemble user partitions, however they are allowed to use non-standard extensions to OS kernel interface.

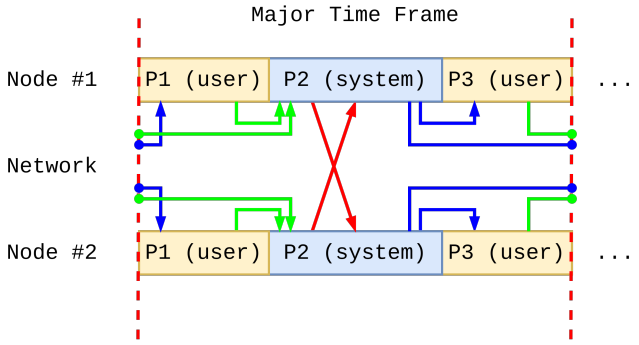

Figure 2: Network communication model of ARINC 653

The model of inter-module communication [5] is as follows:

1) User partition puts outgoing packet to ARINC Queuing port or ARINC Sampling port.
2) System partition during it's time window gets the packet and sends it to the NIC and further to the network. Received packets are handled as well: every packet is dispatched by it's destination address and is stored into corresponding queuing/sampling port.
3) In their next time window user partitions process packets received from the network.

C. Deterministic SpaceWire

It is possible to distinguish two dominating approaches to organization of deterministic SpaceWire network.

Approach of static bandwidth allocation is achieved by adaptation of AFDX over SpaceWire networks [6]. AFDX standard builts upon IEEE 802.3 (Ethernet) [7] and views network as a set of unidirectional unicast routes called Virtual Links (VLs). Each Virtual Link is assigned a value of it's maximum used bandwidth (measured in bit/s). Sum of all VL's bandwidths for each physical link must not exceed it's capacity. End nodes and switches must guarantee that for each link the flow does not exceed VL's bandwidth value and that there is enough resources to process worst case requested bandwidth. In theory, this approach achieves robust partitioning of network interactions.

Approach of time-division multiplexing (TDM) is achieved by SpaceWire-D [8] and STP-ISS-14 [9] protocols. With TDM, time is divided into time slots of specified size. Several slots are assigned to each destination address — each sender transmits packets only in allocated slots. End node is notified of next time slot by means of SpaceWire time markers — their delivery is prioritized over data traffic on Data Link Layer and is rapid.

Second approach is better covered in research on deterministic SpaceWire, and an approximation of worst case time of end-to-end packet delivery may be found [10].

D. User-friendliness of system integration

In order to ease the burden of system integration a simple and intuitive model for deterministic SpaceWire network is required. A desired property of the model — minimal number of preconditions for correct functioning of the network.

For instance, to simplify system integration we:

- Use SpaceWire logical addressing instead of path addressing.
- Abandon broadcast and multicast messaging in favour of unicast messaging.
- Unify network stack architecture and configuration for all hardware platforms.
- Keep port enumeration used in the configuration consistent with that used on transceiver exterior.

III. Developing drivers for ARINC 653 RTOS

RTOS developers take into account that OS will be repeatedly ported to various hardware platforms by different people. As a result, the development of driver code and configuration is simplified and algorithmized.

A. Component-based Model

A component-based model [11] algorithmizes development of system partition drivers. System partition is viewed as composition of seperate components, each of which serves it's special purpose (examples of components: NIC card driver, user partition connector).

In component-based model development of a new component (yet another NIC driver for already existing network stack) is done as follows:

1) Driver developer writes a component configuration file and declares: input and output component ports, configuration fields availible to system integrator and their respective types.
2) Code generator transforms component configuration file into a number of C files with declarations of functions that driver developer needs to implement.
3) Driver developer implements required functions: component initialization and it's main activity.
4) Driver developer writes a Python script to preprocess component configuration before inserting it into the loaded OS image.

Once all components are ready, the project builds as follows:

1) Component composition is turned into a C file with a sequence of function calls: component initialization and connection of links. Each function was previously defined by driver developer.
2) ARINC 653 process is created for each requested sequence of component activity functions.

In ARINC 653 network stack component activity has a special role. ARINC 653 limits use of device interrupts as they may break partition isolation: first partition's interrupt may be handled during the second partition execution and intoduce unwanted delay. Component activity can substitute interrupt handling by busy-polling the NIC hardware registers.

B. Driver configuration

In the development and integration of IMA system [12] there are four general participants:

- **Hardware platform supplier** provides RTOS supplier with hardware suitable for implementation of basic RTOS functionality (isolation of memory, CPU time, platform devices) and capable of meeting IMA system requirements.
- **RTOS supplier** implements a robustly partitioned execution environment.
- **Application (user partition) supplier** develops application software and mission-specific devices and corresponding drivers.
- **System integrator** develops RTOS and application software configuration, mathes requirements of integrated system parts.

Drivers may be developed either by RTOS supplier, or by application suppiler. To simplify system integration configuration parameters are divided by their owner to driver developer parameters and system integrator parameters. Driver developer parameters have a complex or unobvious effect on driver operation or have an unclear domain, and should be either fixed in driver code or precomputed from other parameters at build time. System integrator parameters, on the contrary, have a clear and traceable effect on driver operation and simple domain that is verified by the build system at build time, and thus must be easily changed by system integrator.

```
- name: spw0_driver
  type: SPACEWIRE_DEVICE_DRIVER
  configuration:
    max_packet_size: 16
    tx_queue_size: 16
    rx_queue_size: 16
    tx_speed: 100
    log_level: important
  device:
    device_name: spw0
```

Figure 3: Component configuration (system integrator view)

The algorithm of component configuration in component-based model is as follows:

1) System integrator writes a component composition file of a system partition, in which he lists all required components and their configuration. For each component there is a unique name, a component type, a list of requested devices,
2) Build system verifies configuration types and permissions of access to requested devices.
3) Component build script handles the configuration and forms a configuration tree that is inserted into RTOS boot image.
4) In RTOS runtime driver initialization code parses a configuration tree and configures hardware.

C. Hardware access in RTOS environment

According to ARINC 653 concept of robust partitioning, access to device hardware must be controlled by the OS. As most of the devices are availible as memory-mapped registers (MMIO), the approach of robust resource partitioning [13] is mostly sufficient to restrict access to devices.

Thus, the algorithm of controlled access to hardware is as follows:

1) In the component composition system integrator lists all devices, requiested by his system partition.
2) Each device requests the mapping of physical memory block with it's registers to partition address space.
3) At build time all of the memory mapping requests are analyzed for compatibility and the MMU configuration is constructed.
4) At runtime CPU accesses controller registers by their virtual addresses.

IV. Deterministic SpaceWire network stack

A. Unified network stack architecture

The main problem in network stack unification is the variety of supported architecture. Two supported SpaceWire

controllers (based on MIPS32 R4000 and PowerPC 476FP) have two major distinctions.

Firstly, on one platform data exchange is performed with the NIC directly, while on the other — only through the hardware switch. For unification purposes we provided a software switch component with interface and operation equivalent to the harware switch. Thus, DMA transfers are managed either in the hardware switch component or in the NIC driver.

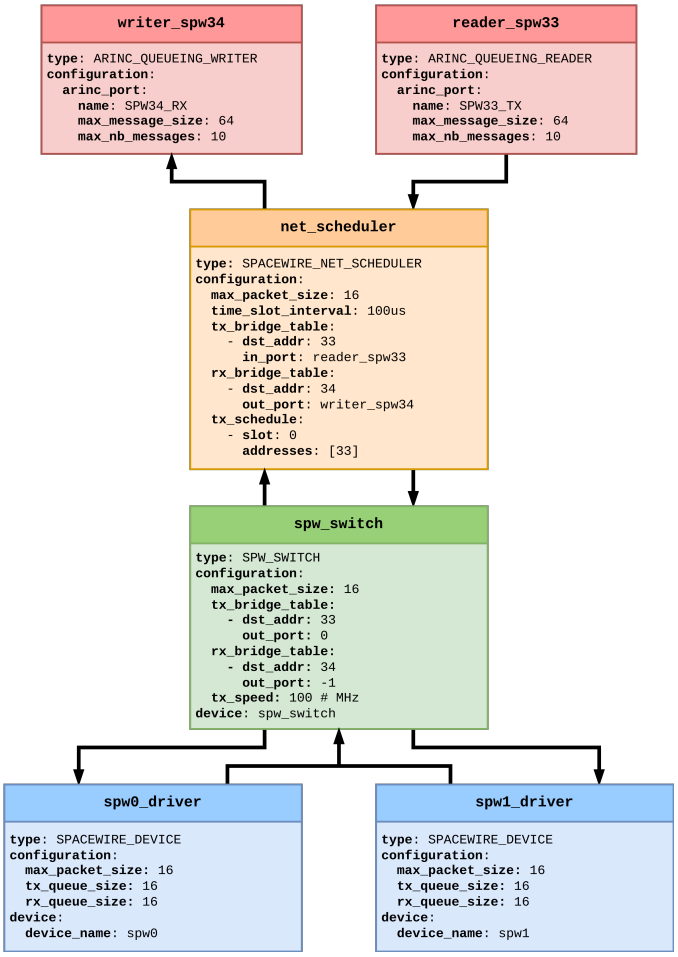

Figure 4: Unified SpaceWire network stack

Secondly, one controller sends time markers periodically with specified period, while the other transfers markers on write to a register. On-demand mechanism turned out to be more convinient to organize a network scheduler, and we had to emulate it with the periodic mechanizm.

Unified SpaceWire network stack consists of four layers:

- **Network drivers** handle monitor state, manage packet queues and perform DMA list transfers to/from SpaceWire controller.
- **Switch** redirects incoming and outgoing packets to a specified outgoing port based on destination logical address.
- **Network scheduler** broadcasts time matkers over the network and enforces network schedule (according to STP-ISS-14 [9]), passes packets allowed for transmission in current schedule slot.
- **ARINC 653 port handlers** perform system calls and exhange packets with buffers in the RTOS kernel.

B. Network scheduler

In the presented architecture network scheduler is of special interest.

In time-master mode network scheduler periodically broadcasts time markers, notifying the start of a next schedule slot. The time to send a time-marker is determined by RTOS system timer: if enough RTOS major time frames have passed from previous marker sendout, a next marker is broadcasted.

In time-slave mode network scheduler handles received markers, updates current slot number and sends packets allowed for transmission down the stack.

To measure time slot size required for a given network configuration we must study the worst case data transfer scenario in detail:

1) Schedule slot starts with system partition window. Time master broadcasts time marker into the network. In the worst case time marker is sent right in the end of system partition.
2) As nodes in the network are not synchronized, slave node is late with time-marker handling for the time, equal to it's major time frame.
3) Slave node starts packet transmission at the end of it's system partition time window in the worst case.
4) The worst case time of packet tranmission can be estimated with multiplication of queue size by maximum packet size, divided by baudrate of the network slowest channel.

Schedule slot duration satisfies the following:

$$T_{slot} >= T_{sys,1} + T_{sys,2} + T_2 + T_{onwire}$$

$$T_{slot} = N \cdot T_1, N \in \mathbb{Z}$$

$$T_{onwire} \approx \frac{TxQueueSize \cdot PacketSize}{SpwFrequency}$$

where T_{slot} — schedule slot duration, T_i — i-th module major time frame, $T_{sys,i}$ — i-th module system partition period, T_{onwire} — end-to-end time to transfer a packet.

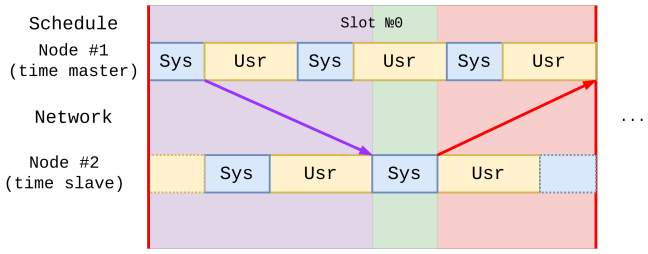

Figure 5: Worst case packet transmission scenario

However, this approach is only applicable when RTOS major time frame is several times smaller than the schedule

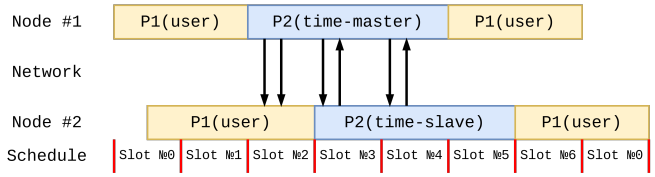

Figure 6: Network desynchronization problem

slot, which may correspond to the case of transmission big packets over a slow network.

Scenario of a schedule slot smaller then RTOS major time frame is not supported for two reasons:

- First, because of **module schedule desynchronization** different modules work with different schedules.
- Second, even with synchronized schedules, there are **dead schedule slots**, during which transmission is impossible with existent hardware.

Both of the problems may be resolved with support on protocol and hardware level, as it is done in MIL-STD-1553 protocol [14].

Both problems could be avoided entirely if we give up the model of guaranteed maximum delay of packet delivery in favour of model with guaranteed bandwidth. This will require a corresponding redesign of the network scheduling algorithm [15].

C. Coordinated packet buffering

In order to satisfy teal-time requirements data transmission must be guaranteed (no packet loss) and fast (low latency). Both of these parameters are affected by the number of buffers in the packet transmission path — each buffer adds **excess latency for packet copying** between the buffers. Besides, **buffer coordination** must be done: packet must be retrieved from a buffer only in case there is buffer space in the next one — otherwise packet is dropped.

In developed network stack we use buffering scheme, remotely similar to sk_buff in OS Linux [16]. It's main feature is the lack of buffering in network scheduler and switch: scheduler requests buffer space from a NIC driver through the switch, and only in case of success uses it as a destination buffer in 'RECEIVE_QUEUING_MESSAGE' system call. This way excess latency for packet copying is excluded, kernel and NIC driver buffers are coordinated with each other.

Coordination of NIC driver buffer and SpaceWire controller buffers is done by the hardware developer on used platforms. Coordination of controller buffers of sender and receiver is provided by the SpaceWire protocol by means of the Flow Control Token mechanism.

It is important to note that total coordination of all buffers is possible for SpaceWire networks, though it is unwanted. It may lead to slow data flow blocking outgoing port of a switch and making it unusable for a faster and more critical data flow — this effectively breaks data flow isolation.

It is technically feasible to get rid of userspace buffer, as it is done in io_uring in Linux OS [17] — using circular buffer

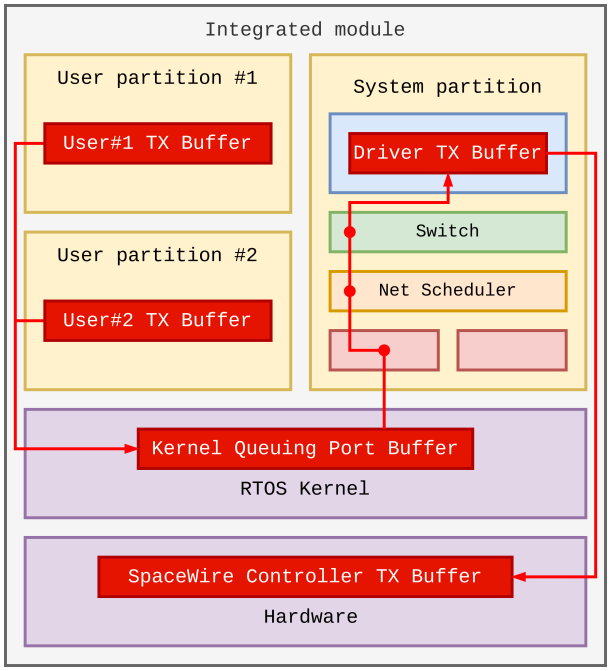

Figure 7: Coordinated packet buffering

in memory block shared among user and system partitions. However, this will require serious development effort from system and user partition developers, will make verification harder and will make it impossible to transmit data from multiple partitions.

D. Driver Finite State Machine

Final step towards a deterministic network stack is to develop a harware interaction algorithm that has a bounded WCET.

Data exchange with SpaceWire controller is done via four seperate DMA-channels: data and descriptor channels for transmission and for reception. Each of the channels requires driver to start DMA tarnsfer list, monitor the end of transfer and handle it — reclaim driver buffers, pass received packet to user partition.

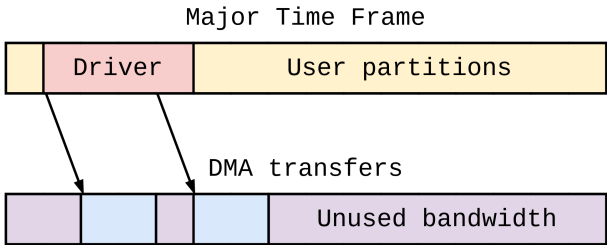

Figure 8: Single DMA-transfers

An important technique used to increase network stack throughput is replace simple DMA-transfers with DMA list transfers. It allows to use network bandwidth more effeciently, but complicates control of precise packet transmission time.

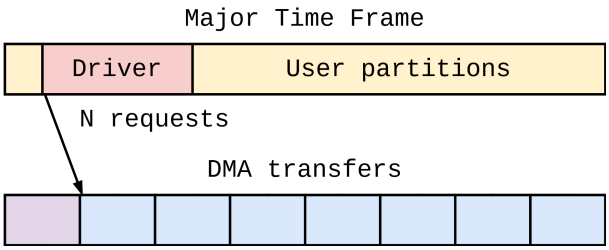

Figure 9: DMA list transfer

SpaceWire driver main activity may be viewed as handler of transitions in some Finite State Machine (FSM). FSMs for transmissions and reception are independent and equivalent, thus we need to describe only one of them. It's state is defined by values of three flags: tx_dma_on (at least one DMA channel is running), $tx\_desc\_dma\_on$ (DMA descriptor channel is running), $tx\_data\_dma\_on$ (DMA data channel is running).

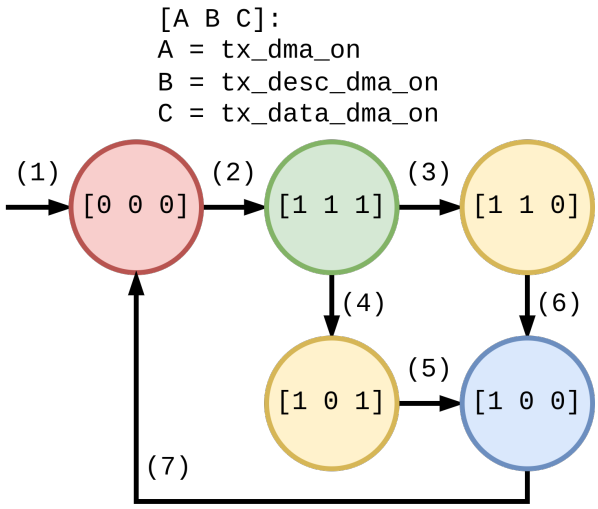

Figure 10: FSM for DMA TX transfers

Transitions in an automaton are performed either automatically or based on current DMA controller state:

1) Transition №1 — FSM initialization.
2) Transition №2 — start of DMA transfers (both for data and descriptor channels) for all currently buffered packets.
3) Transition №3 — end of a DMA list transfer on data channel.
4) Transition №4 — end of a DMA list transfer on descriptor channel.
5) Transition №5 — equivalent to transition №3.
6) Transition №6 — equivalent to transition №4.
7) Transition №7 — reclamation of buffers used in transmission.

In order to increase network stack throughput transmission buffers are freed not only after finishing all DMA transfers (transition №7) but also after every transmitted packet, provided that controller allows to monitor this.

Tranisions №2 and №7 have the most impact on WCET. However, as none of the transitions have busy-wait infinite cycles, their WCET is a linear function of transmission queue size and maximum packet size.

V. Network stack verification

A. Testbench

In order to verify the network stack we manufactured a testbench and connected different development boards with a cable. Low quality SpaceWire cable was manufactured without struct conformance to the specification [18] and without modern cable production technology [19]. Providing a common ground with a seperate cable turned out ot be an essential condition for robust communication.

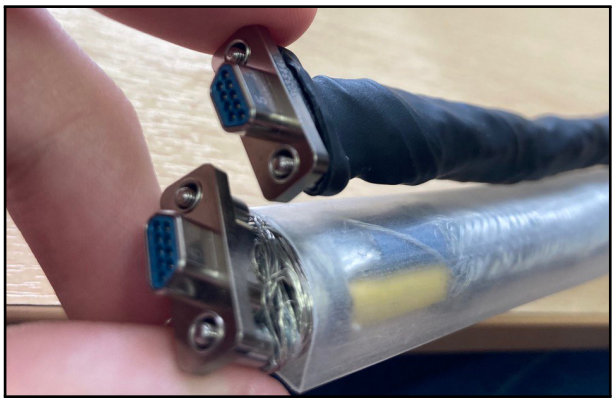

Figure 11: Manufactured SpaceWire cable

Testbench composed of two development boards showed the network stack to be valid for data transmission and reception in a simplest network.

B. Test suite

For verification purposes we developed an automated test suite. It operates in loopback mode, when two ports of single device are connected with a cable. Tests in a test suite:

1) **Test for correct addressing**: device sends packets to different correct addresses. Each receiver must get all packets addressed to him and must not receive any excess messages.
2) **Test for incorrect addressing**: device sends packets to several non-existend addresses. All packets must be dropped.
3) **Estimation of packet delivery time**: device transmits N packets to each of several destinations and receives them. For each packet, end-to-end delay is measured. Sampling averages, sampling deviations and maximum values are computed. The test is considered successful if it's result does not exceed the theoretical limit.
4) **Test of network schedule enforcement**. Device sends packets to three addresses in three distinct time slots. In any given schedule slot scheduler must send only the allowed packets.

VI. Conclusion

In this paper we presented an architecture of deterministic SpaceWire network stack for an ARINC 653 RTOS. In order to implement it several tasks were solved:

1) **Unification of network stack architecture**. We managed to unify interfaces of two distinct implementations of SpaceWire controller and provided system integrator with platform-independent way of configuring the network.
2) **Development of network scheduler**. We used the mechanism of SpaceWire time markers to enforce network schedule for the case of big time slot (slow exchange of big packets between packet modules).
3) **Coordination of buffering in network stack**. To minimize packet loss in case of buffer overflows inside network stack we coordinate all our RTOS buffers on SpaceWire packet transmission.
4) **Development of FSM to handle DMA transfers**. We use list of DMA transfers to transfer packets over the network and the DMA handling code has a bounded Worst Case Execution Time (WCET).

The developed architecture is implemented in RTOS CLOS, developed at ISP RAS, and verified on availible hardware used in space industry.

VII. Acknowledgements

To all the members of the hard real-time operating systems development and verification group at ISP RAS for their invaluable assistance. Specifically, Ilya Rusetskiy for network stack verification.

References

[1] *Avionics Application Software Standard Interface Part 1 – Required Services*, Aeronautical Radio, Inc. ARINC Specification 653P1-5, Dec. 2019.

[2] *Aircraft Data Network Part 7 – Avionics Full Duplex Switched Ethernet (AFDX) Network*, Aeronautical Radio, Inc. ARINC Specification 664P7, Jun. 2005.

[3] *SpaceWire - Links, nodes, routers, and networks*, Std. ECSS-E-ST-50-12C Rev.1, May 2019.

[4] N. Finn, P. Thubert, B. Varga, and J. Farkas, "Deterministic Networking Architecture," RFC 8655, Oct. 2019. [Online]. Available: https://datatracker.ietf.org/doc/rfc8655/

[5] K. Mallachiev, N. Pakulin, and A. Khoroshilov, "Design and architecture of real-time operating system," *ISP RAS Preceedings*, vol. 28, no. 2, pp. 181–192, 2016. [Online]. Available: https://doi.org/10.15514/ISPRAS-2016-28(2)-12

[6] Deredempt, Marie-Hélène and Kollias, Vangelis and Sun, Zhili and Canamares, Ernest and Ricco, Philippe, "Spacecraft Data Handling Architecture based on AFDX network," in *Embedded Real Time Software and Systems (ERTS2014)*, Toulouse, France, Aug. [Online]. Available: https://hal.science/hal-02271363/file/ERTS_2014_submission_19.pdf

[7] *Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*, IEEE Std. 802.3, 2002.

[8] A. F. Steve Parkes, *SpaceWire-D: Deterministic Control and Data Delivery Over SpaceWire Networks*, Std., Apr. 2010. [Online]. Available: http://spacewire.esa.int/WG/SpaceWire/SpW-WG-Mtg15-Proceedings/Documents/SpW-D%20Protocol%20Draft%20B.pdf

[9] V. Olenev, E. Podgornova, I. Lavrovskaya, I. Korobkov, and N. Matveeva, "Development of the transport layer scheduling mechanism for the onboard spacewire networks," 2016. [Online]. Available: https://fruct.org/publications/volume-16/acm16/files/Ole.pdf

[10] I. L. Korobkov, "Method for estimating the maximum packet transmission time in networks with end-to-end routing and time multiplexing," 2022. [Online]. Available: https://openbooks.itmo.ru/ru/article/21026/

[11] K. Mallachiev, N. Pakulin, A. Khoroshilov, and D. Buzdalov, "Using modularization in embedded OS," 2017. [Online]. Available: https://doi.org/10.15514/ISPRAS-2017-29(4)-19

[12] J. Krodel and G. Romanski, "Handbook for Real-Time Operating Systems Integration and Component Integration Considerations in Integrated Modular Avionics Systems," Tech. Rep. DOT/FAA/AR-07/48, Jan. 2008. [Online]. Available: https://www.tc.faa.gov/its/worldpac/techrpt/ar0748.pdf

[13] V. Cheptsov and A. Khoroshilov, "Robust Resource Partitioning Approach for ARINC 653 RTOS," 2023.

[14] "Review and Rationale of MIL-STD-1553A and MIL-STD-1553B," 2012. [Online]. Available: https://www.milstd1553.com/wp-content/uploads/2012/12/MIL-STD-1553B.pdf

[15] B. Hubert, *Linux Advanced Routing and Traffic Control HOWTO*, 2000. [Online]. Available: https://lartc.org/lartc.pdf

[16] A. Cox, "Network Buffers And Memory Management," 1996. [Online]. Available: https://tldp.org/LDP/khg/HyperNews/get/net/net-intro.html

[17] J. Axboe, "Efficient IO with io-uring," 2019. [Online]. Available: https://kernel.dk/io_uring.pdf

[18] "ESCC Detail Specification No. 3902/003," Jun. 2008. [Online]. Available: https://spacecomponents.org/specification/view?id=3125

[19] R. Kuznetsov, A. Lobanov, and N. Molchanov, "Specific Development and Production of cables for the SpaceWire standard," 2021. [Online]. Available: https://spetskabel.ru/files/articles/SpaceWire/spacewire.pdf