DOI: 10.15514/ISPRAS-2025-37(4)-15

Architecture for Time Synchronization in an onboard SpaceWire Network of ARINC 653 Nodes

I.V. Rusetskiy <rusetskiy@ispras.ru>
V.V. Aleinik <valeinik@ispras.ru>
V.Y. Cheptsov <cheptsov@ispras.ru>

Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. In this paper we present the architecture for time synchronization in an onboard network. The architecture is specific to SpaceWire protocol and is based on mechanism of broadcast codes introduced in ECSS-E-ST-50-12C standard. We examine synchronization of real time clocks as well as synchronization of ARINC 653 node schedulers. We discuss the modification to ARINC 653 Interrupt Services required for time synchronization to operate. Achieved precision of synchronization is no worse than 5 ms.

Keywords: real-time operating system; network stack; distributed systems; ARINC 653; SpaceWire; time synchronization**.**

For citation: Rusetskiy I.V., Aleinik V.V., Cheptsov V.Y. Architecture for clock synchronization in an onboard SpaceWire network of ARINC 653 nodes. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 2, 2025. pp. 7-16. DOI: 10.15514/ISPRAS-2025-37(4)-15.

Acknowledgments. All the members of the hard real-time operating systems development and verification group at ISP RAS for their support and ideas. Savva Mitrofanov and Rustam Subkhankulov for their hands-on contributions to the result of this paper. RC Module for their invaluable technical support regarding the internals of SpaceWire hardware.

Архитектура системы синхронизации времени в бортовой сети SpaceWire из ОСРВ с поддержкой стандарта ARINC 653

И.*В. Русецкий* <*rusetskiy@ispras.ru*>
В.*В. Алейник* <*valeinik@ispras.ru*>
В.Ю. Чепцов <*cheptsov@ispras.ru*>

Институт системного программирования РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В данной работе предлагается архитектура системы синхронизации времени вычислителей сети. Предложенная архитектура специфична для сети SpaceWire и основывается на механизме широковещательных кодов из стандарта ECSS-E-ST-50-12C. Рассматривается не только задача синхронизации часов реального времени вычислителей, но и задача синхронизации планировщиков операционных систем стандарта ARINC 653. Обсуждаются дополнение к интерфейсу ARINC 653 Interrupt Services, необходимые для функционирования системы синхронизации времени. Достигается точность синхронизации не хуже 5 мс.

Ключевые слова: операционные системы реального времени; сетевые стеки; распределённые системы; стандартный интерфейс ARINC 653; бортовая сеть SpaceWire; синхронизация часов.

Для цитирования: И.В. Русецкий, Алейник В.В., Чепцов В.Ю. Архитектура системы синхронизации времени в бортовой сети SpaceWire из ОСРВ с поддержкой стандарта ARINC 653. Труды ИСП РАН, том 37, вып. 4, часть 2, 2025 г., стр. 7–16 (на английском языке). DOI: 10.15514/ISPRAS–2025–37(4) -15.

Благодарности. Авторы благодарны всем членам группы ИСП РАН по разработке и верификации операционных систем жёсткого реального времени за их поддержку и идеи. Авторы благодарят Савву Митрофанова и Рустама Субханкулова за их практический вклад в результат этой работы, а также сотрудников НТЦ Модуль за высококвалифицированную техническую поддержку по работе с оборудованием SpaceWire.

1. Introduction

The problem of time synchronization [1] is a classical problem of distributed system theory and it has serious practical implications. Synchronicity of nodes in a network and accuracy of their time sources may affect the correctness and robustness of network interactions. Unsynchronized nodes are unable to perform data transfers to a predefined static schedule – this poses a difficulty for construction of a synchronous deterministic network and complicates organization of hot spare redundancy. At the same time, there are numerous desynchronization mechanisms: frequency drift and frequency instability of crystal oscillators, timer reset on node failure. As a result, the exact definition of «time synchronization» depends on the environment and the guarantees required for network operation.

In this work we present a scheme for time synchronization in an onboard network of nodes controlled by ARINC 653 compliant real-time operating system [2]. Orientation towards avionics and space industry has a couple of important features. First, avionic network and all of its nodes (modules) are hard real-time systems – thus, our scheme must have a relatively low overhead. Second, network use cases (synchronous timed transfers, hot spare redundancy) imply module schedule synchronization as well as astronomic time distribution. Last, spacecraft networks base on specialized physical layer protocols (SpaceWire [3], MIL-STD-1553 [4]) that have unique facilities for time distribution.

We aim for a time distribution architecture that could be adapted to any ARINC 653 compliant RTOS. Even though our scheme is based on broadcast codes special to SpaceWire, we hope it is adaptable to MIL-STD-1553 and other protocols. We propose a simplistic solution for time and schedule synchronization with precision of 5 ms.

Proposed architecture is implemented in CLOS, a real-time operating system developed at ISP RAS, and tested on hardware platforms used in space industry.

2. State of the art

2.1. Time synchronization in ARINC 653

Synchronization of time in a network of modules running an ARINC 653 compatible RTOS has a couple of distinctive features. In contrast to POSIX, ARINC 653 performs static allocation of execution time, memory and other resources to entities called «partitions». Thus, all partitions of a module execute according to a fixed schedule. Module schedule is periodic (its period is called «Major Time Frame») and consists of windows assigned to partitions. After module initialization its major time frame executes periodically until system restart or shutdown, or until module schedule is switched. Module schedule is enforced by partition scheduler, which switches partition windows based on system time. In case RTOS supports ARINC 653 extended services [5] there is a system call to switch module schedules.

If we want to build a deterministic network with timed data exchanges, we need to synchronize module schedules of different nodes. Thus, synchronization subsystem must perform time distribution and shift module schedule for one or both modules backward or forward based on accumulated time difference.

According to ARINC 653, module contains user and system partitions. System partition has privilege to access a wider selection of system calls and often contains file system, network stack and other peripheral drivers (ARINC 653 implies microkernel OS architecture).

ARINC 653 has several system calls related to system time – TIMED_WAIT, PERIODIC_WAIT, GET_TIME – and a subsystem for working with real time – Calendar Time API. Calendar Time API allows to read or write real time using GET_CALENDAR_TIME and SET_CALENDAR_TIME system calls. Setting of time is supported in two modes: one can set calendar time directly or temporarily slew calendar time to prevent backward time shifts or forward jumps.

ARINC 653 does not specify how one can set or adjust system time. A good mechanism to adjust time and frequency for the Unix kernel is described in [6] for the ticking timer: clock frequency is changed by adding or subtracting a fixed amount at each system timer interrupt for a calculated number of ticks. This scheme to not easily adaptable to tickless timer architecture, though.

2.2. The problem of time synchronization

There is a variety of clock synchronization problem formulations, and it is important to concentrate on one. First, it is important to distinguish logical clock synchronization [7], which aims to build a total order of events in the distributed system, from physical clock synchronization. Physical clock synchronization [8] may aim at agreement of clocks in the system (internal synchronization, «any two clocks deviate no more than Δ»), or at correspondence to some external precision time source (external synchronization, «any clock deviates from real time source no more than Δ»).

Synchronization problems can also be classified depending on symmetry of roles in a network (symmetric or asymmetric) or on presumed message propagation delay [9] («synchronous» for bound delay, «asynchronous» for unbound delay).

Approaches toward time synchronization may also be classified by their tolerance to faults [10].

Following [8], we divide the time synchronization scheme into three parts:

1. Clock source selection and resynchronization event detection.
2. Time distribution protocol.
3. Time correction mechanism.

According to all the mentioned criteria in this paper we analyze physical clock synchronization and aim for clock agreement. Our scheme is asymmetric (also called «central master synchronization» [10]) and the network has bound propagation delay. We do not perform clock source selection and simply fix time source node. Our resynchronization event and time correction mechanism are trivial as well – we resynchronize periodically with fixed period and correct time by shifting time forward by a computed delta. This work is about time distribution over SpaceWire network and about OS support required for its operation.

2.3 Time distribution infrastructure

Physical clock synchronization may be achieved either with software or with hardware.

Network Time Protocol [11] is a time synchronization protocol widely used since its development in 1985. NTP builds on top of UDP/IP transport and distributes time from atomic clocks according to a hierarchical client-server model. On each synchronization step using a pair of transfers (from client to server and to back) NTP estimates current round-trip delay θ_i and clock offset δ_i between nodes. Pairs (θ_i, δ_i) are filtered over time and used to update current real-time clock frequency. It has a precision of around 100 μs in fast LANs. Adaptation of NTP into a hard real-time operating system is problematic as NTP expects network stack to handle arrivals of UDP/IP packets instantaneously (in interrupt handler) while RTOS tend to prohibit device interrupts in favor of code execution determinism.

Another protocol is PTP [12]. Like NTP, PTP has a hierarchical time distribution. Best master clock algorithm (BMCA) determines optimal clock source for a given node and synchronizes with it. However, to reach precision of 1-10 μs PTP requires hardware integration into the network layer protocol. Its integration into Ethernet is called Time-Sensitive Ethernet [13] and reaches precision of around 500 ns.

Synchronous Ethernet [14] is a hardware approach that achieves synchronization with precision of 100 ns by transmission of reference signal over a dedicated line and subsequent adjustment of receiver clock frequency with a PLL.

SpaceWire and MIL-STD-1553 have lower layer mechanisms for synchronization pulse distribution (broadcast codes for SpaceWire, «synchronize» mode code for MIL-STD-1553), however they require a separate software solution to transfer timestamps. It is important that SpaceWire broadcast codes (time markers and distributed interrupts) have an increased priority compared to data packets and do not need to wait in queues during switching – these properties allow to build network-wide distribution of time instead of hop-by-hop time distribution. SpaceWire broadcast code skew and jitter can be estimated as 14 (ESC + data character) and 10 (one preempted data character) bit-periods which is equivalent to 140 ns and 100 ns for one 100 Mbit/s link.

SpaceFibre network is analogous to SpaceWire network in terms of time-distribution mechanism, however its increased bandwidth allows implementations of heavy-weight custom hardware protocols analogous to PTP [15] with accuracy of 1 μs.

3. Time synchronization architecture

3.1 Approach to time distribution

Our time distribution scheme is asymmetric. Network has a single fixed «master» node, that periodically broadcasts a SpaceWire distributed interrupt. All other nodes in a network are «slave» nodes. After the distributed interrupt a separate message containing a timestamp is sent.

One iteration of time distribution is as follows:

1. Master node broadcasts distributed interrupt to the network and saves current time T_1.
2. Slave node handles the interrupt and saves moment of its reception T_2.
3. Master node transmits value of T_1 in a separate data packet over the SpaceWire network.
4. Slave node estimates the clock offset between nodes: $\theta = T_2 - T_1$.

In order to critically analyze the approach let's focus on each step in details. What we seek is the delay and jitter introduced on each step of time distribution (Fig.1):

1. Let's focus on the time difference between broadcast code distribution and saving of current time T_1. First of all, in case the system timer is implemented as ticking timer (in contrast to tickless timer) a timer interrupt may arrive. Timer interrupt time can be roughly estimated as 1000 instructions which is equivalent to 5 μs on a 200 MHz board ($T_{tickint}$ = 5 μs). Secondly, there may be a wraparound jitter in the implementation of a system timer. Hardware platforms with 32-bit system counter have to account for wrapping to produce a 64-bit timestamp – they have a cycle in the timestamp generation code. Yet it is guaranteed to add only a single spare iteration in the worst case (20 instructions), which is bound by 100 ns on a 200 MHz microprocessor ($T_{wraparound}$ = 100 ns). Yet another possible source of time slew from the timer is the possible context switch: on some architectures system timer is read-protected for unprivileged code, so to read a timestamp it is required to switch to the kernel and back ($T_{syscall}$ = 5 μs).
2. Time distribution over SpaceWire network has three major sources of delay or jitter: broadcast code transmission through data link layer, link failure, interrupt handling. As we've discussed earlier, broadcast code skew and jitter can be estimated as 140 ns and 100 ns for one 100 Mbit/s link (T_{bcslew} = 140 ns, $T_{bcjitter}$ = 100 ns). This number is multiplied by a number of links broadcast code traverses from master to slave (diameter of the network graph). Time to restart the link on error detection is $T_{linkfail}$ = 19.2 μs. Precise synchronization is impossible in presence of link failures, and it is possible to drop time samples if link failure is detected. Interrupt state preservation and handler dispatch may introduce jitter related to caching: in case kernel stack got evicted from cache processor may require to load it from memory. Rough estimate for interrupt handling is T_{int} = 2 μs on a 200 MHz board.
3. A probable source of error on this step is packet loss. In case one timestamp or one interrupt is lost in the network, time distribution mechanism will set a wrong association between the interrupt and the timestamp, which will result in totally incorrect clock offset. This may be handled by proper filtering of collected values of θ.

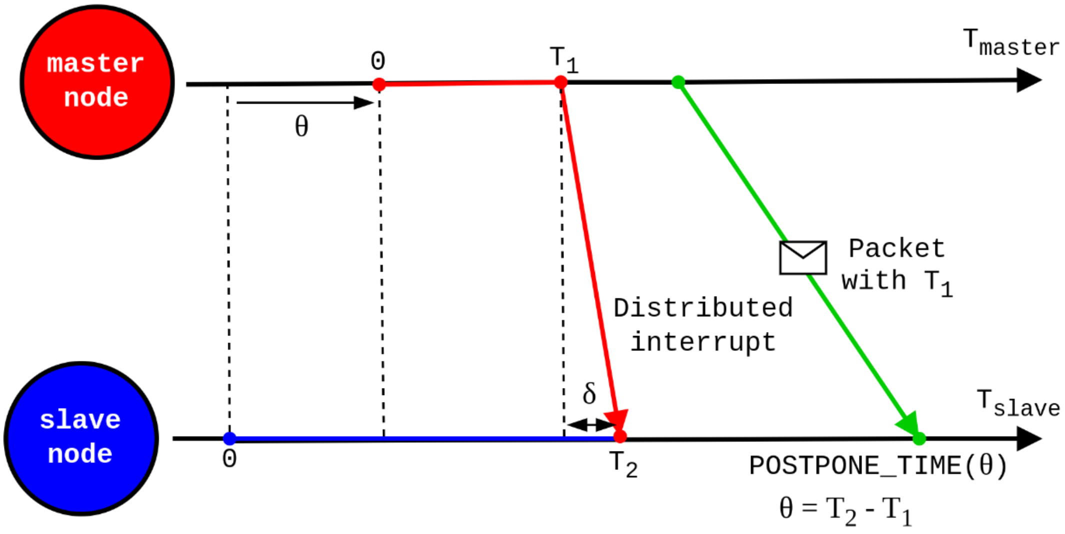


Fig. 1. One iteration of time distribution.

Our approach is simplistic and works with an ideal network that satisfies properties:

- No data packets or broadcast codes are missed or corrupted.

- Links are always functional.

A big disadvantage of our scheme is that we do not account for time propagation delay δ. At the time of writing, we do not have a methodology for precise time measurement in a distributed system, and fine-tuning is not possible at that stage.

3.2 Time synchronization subsystem In ARINC 653 RTOS

Our architecture for time synchronization subsystem in ARINC 653 real-time operating system has a set of functional software elements (Fig.2):

1. **Synchronization software**. It consists of interrupt and timestamp generation, timestamp transfer, time offset estimation and clock correction.
2. **Network stack**. It provides a simple interface for broadcast code and data packet transmission and reception, stores all hardware configurations and network routing tables.

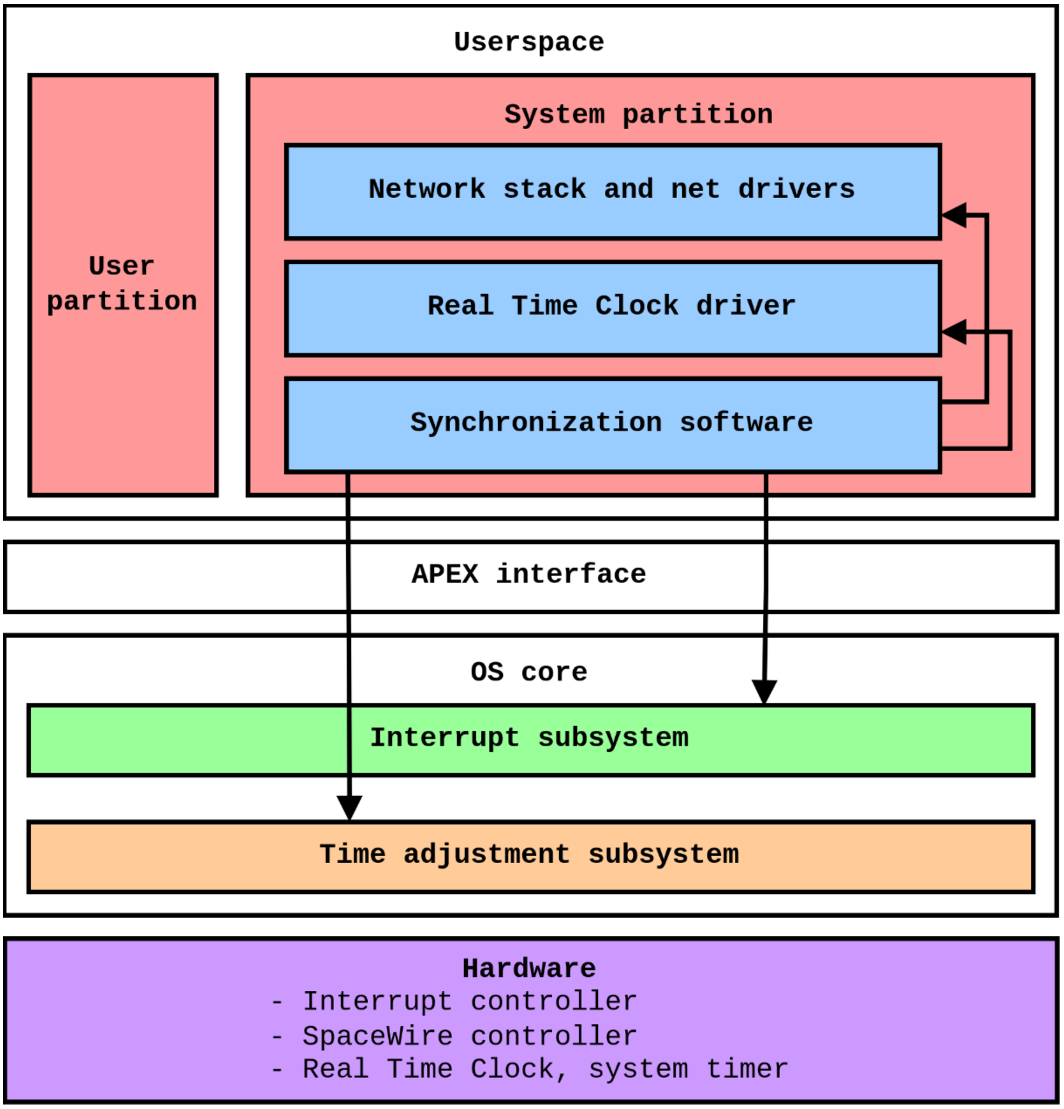


Fig. 2. Architecture for time synchronization subsystem.

3. **Interrupt handling subsystem**. Its main purpose is to perform robust partitioning of external interrupts. Only the partition that owns the interrupt source must spend time to handle the interrupt. This is achieved by unmasking the interrupt only for the duration of the interrupt source owner partition. Interrupt subsystem must provide a system call for partition to know whether a given interrupt is received.
4. **Time adjustment API**. It is important for the operating system to provide a proper set of system calls to adjust system time. The implementation of this API poses a challenge as it interferes with RTOS scheduler. Another non-standard API that turned out to be useful is the system call to get start of the current Major Time Frame and offset of the current module window.

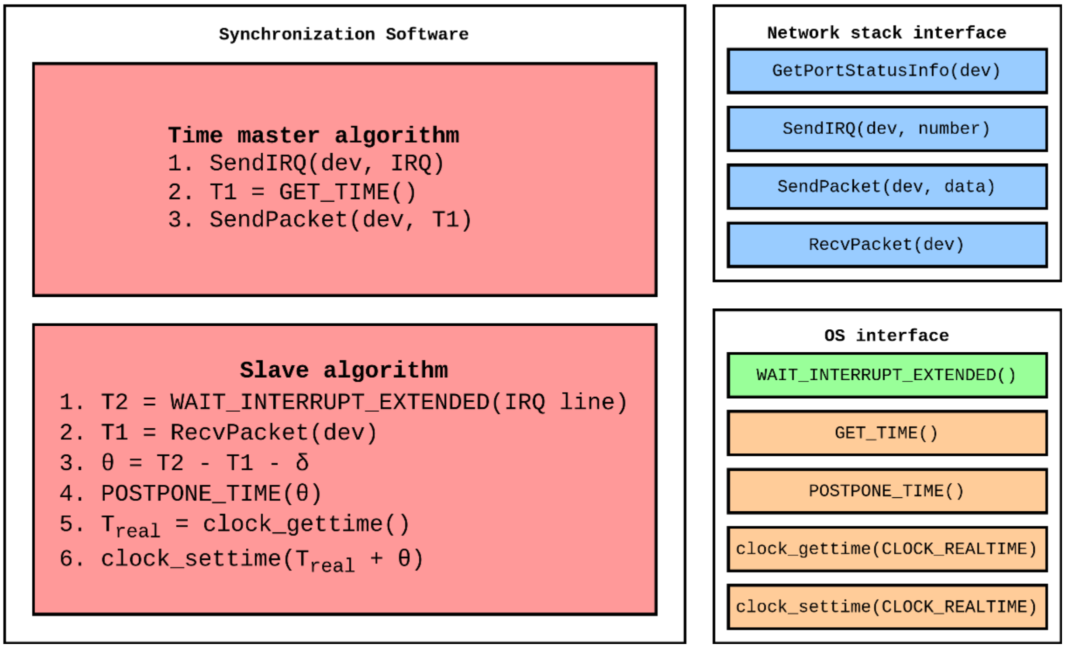
Synchronization software, SpaceWire network stack and a real-time clock driver have to be a part of a dedicated system partition.

3.3. Synchronization software

Synchronization software is a central control unit of synchronization. Synchronization software executes periodically during every Nth Major Time Frame and contains: either master or slave time distribution algorithm, a possible role selection scheme (for now, we fix roles in the network for the sake of simplicity), a proper data filtering algorithm for received data.

For synchronization software it is important to have up-to-date status information of the corresponding SpaceWire device. For the sake of simplicity, we fix that each node in a network has only one port connected to the network: master node broadcasts distributed interrupts through a single port, slave node receives distributed interrupts from a single port.

At startup, synchronization software executes either master node algorithm or slave node algorithm (Fig. 3). Master algorithm broadcasts a distributed interrupt, saves a timestamp and transmits it to the broadcast SpaceWire logical address. Slave node handles the interrupt received from OS kernel via a system call, handles the timestamp received from SpaceWire network stack and performs a system call to adjust time.



```
Time master algorithm
1. SendIRQ(dev, IRQ)
2. T1 = GET_TIME()
3. SendPacket(dev, T1)
```

```
Slave algorithm
1. T2 = WAIT_INTERRUPT_EXTENDED(IRQ line)
2. T1 = RecvPacket(dev)
3. $\theta$ = T2 - T1 - $\delta$
4. POSTPONE_TIME($\theta$)
5. $T_{real}$ = clock_gettime()
6. clock_settime($T_{real}$ + $\theta$)
```

Fig. 3. Algorithms for synchronization software.

It is important to handle error scenarios:

- The clock offset value is too big (timestamp or sync pulse missed).
- The link was down since last sync pulse (timestamp or sync pulse missed, sync pulse delayed).
- The clock offset value is negative (our time adjustment only postpones time).

Described approach lacks redundancy and will not outlive permanent failure of a master node or a link to it. A possible fix to this problem: one can assign master role to three nodes in a network (each with unique interrupt id). In this scheme slave node will receive up to three distinct timestamps T_1, T_2, T_3 and will try to adjust its time towards median of T_1, T_2, T_3.

3.4 Integration with SpaceWire Network

During the integration with SpaceWire network we faced several challenges. First challenge is the dissimilarity of time distribution interfaces provided by SpaceWire hardware. One of the IP cores we work with aimed at automatic time marker generation and the documentation did not specify how to generate a single time marker.

Second, distributed interrupts and interrupt acknowledgments are an extension to SpaceWire standard. It turned out one of the three hardware platforms we work with supports them in non-straightforward fashion.

Another problem we faced is that during link shutdown some hardware platforms do not disable interrupt generation. Thus, SpaceWire codec may recognize broadcast code symbol in the noise and generate a spurious interrupt. This could be partially mitigated by checking error bits on interrupt reception in the interrupt handling code.

3.5 Interrupt handling considerations

ARINC 653 Extended Services specifies a set of system calls that are required for a conforming OS implementation:

- CREATE_INTERRUPT – creates an object to be used to wait for interrupt.
- GET_INTERRUPT_ID – obtains ID of an interrupt configured for use by the partition.
- WAIT_INTERRUPT – provides a means for a process to wait upon the occurrence of a partition-specific interrupt.

None of the specified system calls transfer timestamp of synchronization pulse from the kernel, so we used WAIT_INTERRUPT_EXTENDED that does. Another important feature of this system call is the ability to control whether interrupt is masked during windows of other partitions. Its use enforces strict partitioning of external interrupts (which is good for system reliability and robustness) but requires nodes in a network to be already «almost synchronized» (in case an interrupt is masked and set pending during other partition window, the sync pulse will be late).

3.6 Time adjustment subsystem

We perform adjustment of both real time and system time.

Real time clock (RTC) driver is implemented in userspace and resides in system partition. However, system partition with an RTC can be a distinct partition from time synchronization partition. In this scenario client performs a remote procedure call to set time and reads a value from ARINC sampling port to get time. These two actions are wrapped in POSIX clock_gettime/clock_settime API familiar to most developers.

System time is a more fragile substance to work with. ARINC 653 time management services do provide a way (GET_TIME() APEX call) to get current time, but we need to synchronize Major

Time Frame (MTF) start times. Thus, we implemented a seqlock-based system call to observe start of the previous MTF.

In order to adjust time we used a non-standard POSTPONE_TIME() system call. Its major drawback is that it only shifts time forward (it suspends execution of current partition window and starts the next one earlier). It means that even for a little negative time offset between master node and slave node slave node will have to wait for a whole Major Time Frame.

4. Results

In order to measure accuracy of our time synchronization scheme we built a testbench of two different development boards. We marked one BSP as slave node, another one as master node, started synchronization process and observed printouts at system partition start times.

We provided output of each BSP with automatically generated timestamps and compared them. As a result of synchronization process, arbitrarily desynchronized system synchronizes to a precision of around 5 ms.

This is 3 orders of magnitude worse than what we’ve been expecting from estimation of distributed interrupt transfer time. However, for the moment of writing, we have a serious weakness in our measurement methodology: timestamps are generated on instrumental machine that receives output from remote synchronizing boards. Data follows a serial line at 115200 baud, OS network stack, Python multiplexing software, a series of network appliances, another OS network stack and another multiplexing software. This is hardly a suitable methodology for time measurement. A better solution would be to implement a couple of GPIO drivers, to set output pins to one on system partition start and to perform measurement with an oscilloscope.

5. Conclusion

In this paper we presented an architecture for time synchronization in an onboard SpaceWire network of ARINC 653 nodes. The developed architecture is implemented in CLOS real-time operating system, developed at ISP RAS, and tested on available hardware used in the space industry.

References

B. Simons, J. L. Welch, and N. Lynch. 1990. An overview of clock synchronization. Fault-tolerant distributed computing. Springer-Verlag, Berlin, Heidelberg, 84–96.

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

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

Review and Rationale of MIL-STD-1553A and MIL-STD-1553B, 2012.

https://www.milstd1553.com/wp-content/uploads/2012/12/MIL-STD-1553B.pdf (дата обращения: 14.07.2025).

Avionics Application Software Standard Interface Part 2 – Extended Services, Aeronautical Radio, Inc. ARINC Specification 653P2-5, Dec. 2024.

D. Mills. 1994. RFC1589: A Kernel Model for Precision Timekeeping. RFC Editor, USA.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 7 (July 1978), 558–565. DOI: 10.1145/359545.359563.

Puaut, Isabelle. A Taxonomy of Clock Synchronization Algorithms. 1997.

B. Simons, J. L. Welch, and N. Lynch. 1990. An overview of clock synchronization. Fault-tolerant distributed computing. Springer-Verlag, Berlin, Heidelberg, 84–96.

Kopetz, H., Steiner, W. (2022). Global Time. In: Real-Time Systems. Springer, Cham. DOI: 10.1007/978-3-031-11992-7_3.

D. L. Mills, Internet time synchronization: the network time protocol, in IEEE Transactions on Communications, vol. 39, no. 10, pp. 1482-1493, Oct. 1991, DOI: 10.1109/26.103043.

IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems", in IEEE Std 1588-2019 (Revision ofIEEE Std 1588-2008), vol., no., pp.1-499, 16 June 2020, DOI: 10.1109/IEEESTD.2020.9120376.

IEEE Standard for Local and Metropolitan Area Networks – Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks", in IEEE Std 802.1AS-2011, vol., no., pp.1-292, 30 March 2011, DOI: 10.1109/IEEESTD.2011.5741898.

ITU-T Recommendation G.8262 (2007), Timing Characteristics of Synchronous Ethernet Equipment Slave Clock (EEC).

E. Suvorova, Time Synchronization in SpaceFibre Networks, 2021 28th Conference of Open Innovations Association (FRUCT), Moscow, Russia, 2021, pp. 439-450,
DOI: 10.23919/FRUCT50888.2021.9347624.

Информация об авторах / Information about authors

Илья Владиславович РУСЕЦКИЙ – сотрудник отдела технологий программирования ИСП РАН; магистр ВМК МГУ. Сфера научных интересов: операционные системы реального времени, распределённые системы, тестирование сетевых стеков.

Ilya Vladislavovich RUSETSKIY – employee of the department of Programming Technologies of the Ivannikov Institute for System Programming of the Russian Academy of Sciences; graduate master student at Moscow State University. Research interests: real-time operating systems, distributed systems, network stack verification.

Владислав Владимирович АЛЕЙНИК – сотрудник отдела технологий программирования ИСП РАН; аспирант МФТИ. Сфера научных интересов: операционные системы реального времени, архитектура сетевого стека, компьютерные сети SpacWire.

Vladislav Vladimirovich ALEINIK – employee of the department of Programming Technologies of the Ivannikov Institute for System Programming of the Russian Academy of Sciences; PhD student at Moscow Institute of Physics and Technology. Research interests: real-time operating systems, network stack architecture, SpaceWire networks.

Виталий Юрьевич ЧЕПЦОВ – архитектор операционных систем в ИСП РАН; автор проекта OpenCore и регулярный контрибьютер в Tianocore EDK II и UEFITool. Занимается бортовыми операционными системами с жёстким реальным временем и исследованиями безопасности ОС общего назначения.

Vitaliy Yurievich CHEPTSOV – OS architect at ISP RAS; author of OpenCore project; contributor of Tianocore EDK II and UEFITool. Research interests: onboard hard-real time operating systems, security of general-purpose operating systems.