Journal of the Brazilian Computer Society, 2026, 32:1, doi: 10.5753/jbcs.2026.5575

Integrating ZBus into Zephyr RTOS: An Experience Report of Knowledge Transfer from Academia to Industry Through Open-Source Contributions

Rodrigo Peixoto [**Federal University of Pernambuco** | ***rjsp@cin.ufpe.br***]
Leopoldo Teixeira [**Federal University of Pernambuco** | ***lmt@cin.ufpe.br***]
Baldoino Fonseca [**Federal University of Alagoas** | ***baldoino@ic.ufal.br***]

Informatics Center, Federal University of Pernambuco. Av. Journalist Anibal Fernandes, s/n, Cidade Universitária (Campus Recife), CEP: 50.740-560. Recife - PE, Brazil.

Received: 15 March 2025 • **Accepted:** 08 May 2026 • **Published:** 23 June 2026

Abstract Embedded systems are present in a wide range of products, and advances in the Internet of Things, wireless sensor networks, and cyber-physical systems have increased their use. This growth has made embedded software more complex, requiring new real-time operating systems, development kits, libraries, and platforms. Despite these tools, the need for better software engineering practices remains a significant issue in embedded software, to prevent lower software quality and increased maintenance costs. To make matters worse, the knowledge transfer of research advances from academia to industry is still challenging. This paper presents ZBus, an advanced software bus that enables many-to-many communication between threads, enhancing modularity and decoupling in embedded software development. ZBus is a result of a collaborative effort between industry and academia. We also detail the development process, the challenges encountered, the lessons learned, and the collaborative efforts that culminated in integrating ZBus into the widely adopted Zephyr real-time operating system. This integration serves as an experience report on technological knowledge transfer from academia to industry through open-source contributions, as well as a practical guide for researchers and practitioners seeking to engage with industrial open-source initiatives in the embedded software domain.

Keywords: Embedded Systems, Software Engineering, Software Bus, Open-Source, Real-time Operating Systems, Technical Debt

1 Introduction

Embedded Systems (ES) play an essential role in our daily lives. They are present in various devices, from washing machines to airplanes. The advancements in the Internet of Things (IoT) Miorandi *et al*. [2012], Wireless Sensor Networks (WSNs) Akyildiz *et al*. [2002], and Cyber-Physical Systems (CPS) Wan *et al*. [2011] have enabled the widespread adoption of these devices globally. As the use of embedded systems advances, the size and complexity of embedded software also increase. To address this, developers and researchers have introduced various solutions, including new real-time operating systems (RTOS), Software Development Kits (SDKs), libraries, and platforms Fariha *et al*. [2024].

Despite the availability of software development resources, the need for improved software engineering practices remains a significant issue in embedded software development. Antonino *et al*. [2016] states that computer scientists typically have limited knowledge about embedded systems concepts. Consequently, engineers from other fields end up acting as software architects in embedded software development. However, these engineers usually have limited knowledge of software architecture and other software engineering practices. This lack of understanding may lead to the reduction of embedded software quality. For example, Ampatzoglou *et al*. [2016] and Ciolkowski *et al*. [2021] indicate that embedded software projects of different maturity levels present technical debt in their architecture, code, and tests, resulting in high maintenance costs and other issues.

Techniques and tools to introduce better software engineering practices to embedded software have been proposed in previous works AUTOSAR [2025]; ROS [2025]; Marzi *et al*. [2009]; Love [2005]; Rajkumar *et al*. [1995]; Schoettler [2017]. However, these works are tailored to specific application areas or architectures (such as automotive or robotics) AUTOSAR [2025]; ROS [2025]; Marzi *et al*. [2009], high-level operating system classes (Linux, Unix, etc.) Love [2005]; Rajkumar *et al*. [1995], or are simply unknown or unused by the community Schoettler [2017]. Consequently, these works have technical limitations that hinder their widespread use in embedded systems. Thus, applying adequate software engineering practices is still challenging for embedded software developers Antonino *et al*. [2016].

To make matters worse, transferring knowledge from academia to industry is challenging Daniel and Alves [2020], making it difficult for embedded software developers to benefit from research advances quickly. In this context, involvement in the open-source community can be a promising approach to enhance knowledge transfer between academia and industry. Open-source embedded software allows developers worldwide to contribute to various projects, fostering rapid innovation and diverse input that can lead to more robust, flexible, and secure embedded solutions. However, the highly specialized nature of embedded software can limit potential

contributors, making it hard to build a large and active community. To address this, open-source contributions processes and documentation must be clear, comprehensive, and accessible to newcomers, lowering entry barriers and encouraging more developers to contribute. Previous studies have investigated barriers to open-source contributions Rossoni *et al*. [2023]; Fronchetti *et al*. [2023], but none focus on a case study related to embedded software. Such a case study could provide insights into newcomers' contributions to open-source projects in the embedded software domain.

This paper introduces ZBus, a software bus enabling many-to-many communication between threads, enhancing modularity and code decoupling in embedded software development. ZBus was developed at the Edge Innovation Center EDGE [2025], where a team of embedded software developers tackled various software engineering challenges for clients in the automotive, medical, and consumer electronics industries.

ZBus emerged from applying established software engineering principles—specifically publish-subscribe patterns Eugster *et al*. [2003], active databases Paton and Díaz [1999], and event-driven architectures—to the constraints of resource-limited embedded systems. Drawing conceptually from software buses such as D-Bus Love [2005], but optimized for RTOS environments, ZBus provides deterministic many-to-many communication between threads while maintaining a minimal memory footprint and predictable real-time behavior. Interestingly, the resulting architecture exhibits structural similarities to hardware communication buses such as CAN International Organization for Standardization (ISO) [2016], FlexRay FlexRay Consortium [2010], and LIN LIN Consortium [2010], reflecting convergent evolution driven by similar design constraints rather than direct emulation.

We also discuss the process, challenges, collaborations, and lessons learned during the submission and integration of ZBus into the Zephyr real-time operating system (RTOS) Zephyr-RTOS [2026], which is widely used by the embedded systems community. Since version 3.3, Zephyr has included ZBus as its official software bus, making it a valuable tool for creating decoupled embedded software. In the Zephyr community, one of the authors became the ZBus maintainer, evolving the bus based on community feedback, assisting developers through the project Discord channel,[1], and presenting sessions at the Zephyr Developer Summit in 2023, 2024, and 2025. Various organizations and individuals are integrating ZBus into their software projects. Notably, open-source initiatives such as CogniPilot[2] and ZSWatch[3], as well as companies like Nordic Semiconductor[4], Freedom[5] and Plentify[6], are incorporating it into their libraries, application examples, and commercial products.

The next section gives an overview of ZBus origin, detailing its technical aspects and the critical decisions made during its development. Section 3 describes the process of contributing ZBus to become the official software bus for the Zephyr RTOS. Section 4 discusses the lessons learned, along with the implications and limitations of our work, setting the stage for future improvements. Section 5 reviews relevant literature in the field, providing a foundation for further exploration. Section 6 provides the final remarks and outline the potential directions for future research.

[1] https://discord.com/channels/720317445772017664/1042530682011926631

[2] https://www.cognipilot.com/

[3] https://github.com/jakkra/ZSWatch

[4] https://www.nordicsemi.com/

[5] https://freedom.ind.br/

[6] https://plentify.io/

2 ZBus

In recent years, our team in the Embedded Systems R&D division of the Edge Innovation Center in Brazil has gained valuable experience by addressing challenges in developing 45 embedded systems projects. These projects vary in size, with codebases ranging from 10,000 to 50,000 lines of code. We have collaborated with 12 national and international companies with a presence in Brazil across various application domains, including automotive, medical, and consumer electronics.

Throughout this process, we have developed numerous applications, including electronic shelf labels and customized Android tablets. Additionally, we have been actively exploring ways to enhance software modularity for embedded systems, testing various approaches and tools along the way. We identified well-suited design patterns, architectural patterns, communication paradigms, and techniques, but few widely adopted tools, primarily for enhancing modularity in embedded software for constrained devices. As a result, we created ZBus, a software bus that helps developers build modular, event-driven architectures, improving maintainability, testability, and flexibility.

The current version of ZBus is the outcome of several iterations driven by functional and non-functional requirements. We began with a request to implement a system to monitor streetlights in Recife, Brazil. For such cases, we developed the VSX system, a complex solution that involves multiple threads and a hybrid network combining GPRS and Bluetooth Mesh. This complexity presented significant challenges in both development and maintenance. These various issues led us to rethink our approach to embedded software development. Figure 1 shows the technical progress that led to the present version of ZBus.

2.1 Design Lineage and Convergent Evolution

The development of ZBus followed a pragmatic evolution driven by software engineering principles rather than hardware bus emulation. Our design emerged from applying established software architectural patterns to resource-constrained RTOS environments.

Software Engineering Foundations. The ZBus architecture draws from several software engineering approaches. The publish-subscribe paradigm Eugster *et al*. [2003] provides the foundational communication model, enabling decoupled many-to-many interactions. Active databases Paton and Díaz [1999], which trigger autonomous actions when data changes, directly inspired the event-driven notification mechanism. The property system pattern Fowler [1997] influenced our centralized state management approach.

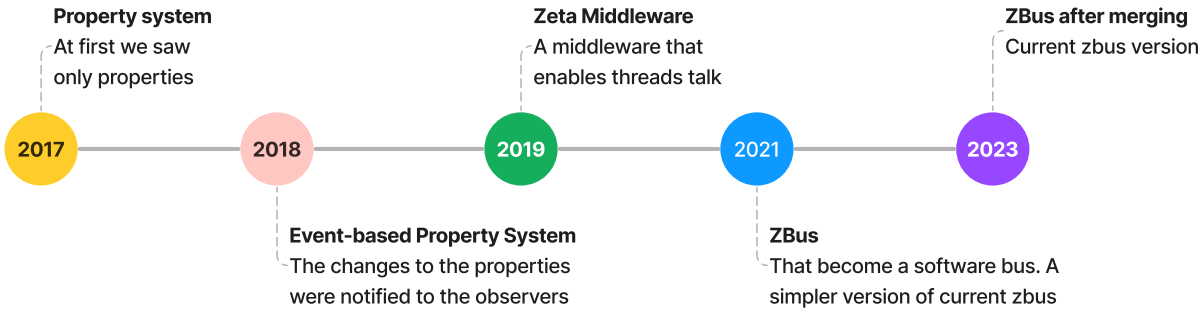


Figure 1. ZBus evolution.

Most directly, ZBus draws conceptual inspiration from D-Bus Love [2005], the widely-adopted software bus in Linux environments. D-Bus demonstrated the viability of software bus abstractions for inter-process communication. However, its complexity, dynamic memory allocation, and dependencies on high-level operating system features made it unsuitable for resource-constrained RTOS environments. This gap motivated the development of ZBus specifically optimized for embedded constraints: limited RAM, deterministic real-time behavior, static memory allocation, and priority-based preemptive scheduling.

Convergent Evolution with Hardware Communication Buses. The resulting ZBus architecture exhibits structural similarities to classical hardware communication buses, particularly CAN International Organization for Standardization (ISO) [2016], FlexRay FlexRay Consortium [2010], and LIN LIN Consortium [2010]. These similarities arose through convergent evolution rather than intentional emulation. Both hardware and software buses in embedded environments must address analogous challenges: priority-based resource arbitration, deterministic message delivery, efficient broadcast communication, and operation under severe constraints.

This convergence manifests in specific patterns. Priority-based notification delivery in ZBus—where notifications execute in the publishing thread's context—parallels CAN's priority-based arbitration, though it is implemented via RTOS scheduling rather than hardware signaling. The deterministic Virtual Distributed Event Dispatcher (VDED) provides predictable ordering similar to FlexRay's time-triggered model, implemented via synchronous callbacks rather than hardware time slots. The channel-based broadcast model mirrors CAN's identifier-based messaging, implemented through shared memory and mutex protection.

These parallels emerged because embedded systems constraints—whether hardware or software—naturally drive similar solutions. Priority-based access prevents unbounded blocking in real-time systems. Deterministic delivery ensures predictable behavior. Broadcast communication enables efficient one-to-many distribution. Convergence to analogous patterns arises from addressing the same fundamental constraints.

This convergent evolution validates the ZBus design: independent arrival at patterns proven effective in hardware domains suggests fundamental correctness. However, ZBus remains distinctly software-optimized, providing RTOS-specific features like compile-time configuration, flexible synchronous/asynchronous modes, and high-level abstractions unavailable in hardware implementations.

2.2 Property system

Initially, we adopted a Property System (PS) Fowler [1997], in which we centralized software information into global properties accessible to all software modules. The PS functions as a streamlined, lightweight database, providing a reliable source for all threads to access software information and communicate changes to other components.

```
u8_t manual_timed_load_control_data[3] = {0};
nb_property_get(
    NB_MANUAL_TIMED_LOAD_CONTROL_PROPERTY,
                    NB_REF(
    manual_timed_load_control_data));
```

Listing 1: Property System get.

```
u8_t dimmer = manual_timed_load_control_data[2] &
     0x7F;
nb_property_set(NB_DIMMER_PROPERTY, NB_REF(dimmer
    ));
```

Listing 2: Property System set.

Listings 1 and 2 illustrate `get` and `set` operations. The former reads three-byte data from the manual timed load control property through the `nb_property_get` function. The latter changes the one-byte dimmer level property using the `nb_property_set` function based on the value of the former property.

After using the PS to develop the VSX software, we observed that the system required enhanced capabilities to meet our needs effectively. For instance, it became common for various software modules to rely on information from the PS to alter their behavior. This dependency led to the emergence of a polling data pattern in our code, which adversely impacted both latency and overall performance, making the PS inadequate for our needs. We needed to improve that, taking into account the software behavior after changing a property.

2.2.1 Event-driven Property System

To enhance the monitoring of behavior changes, we evolved the property system to an event-driven architecture. Specifi-

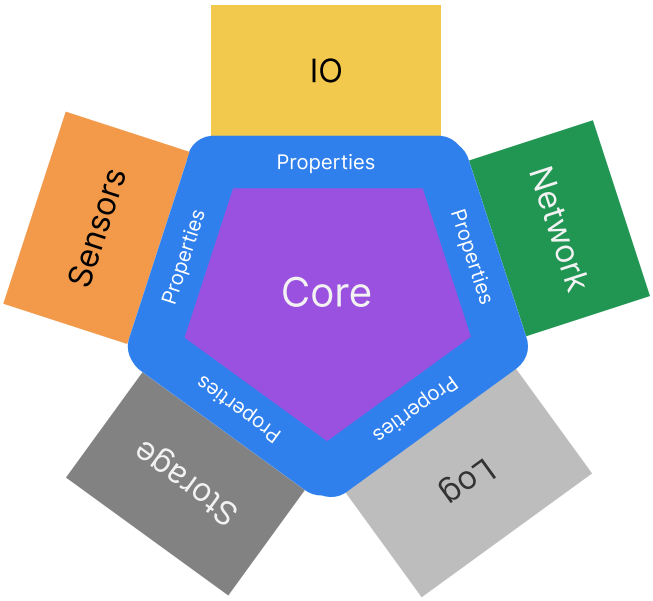


Figure 2. The overview of the VSX project architecture after the refactoring and the advent of the event-driven property system.

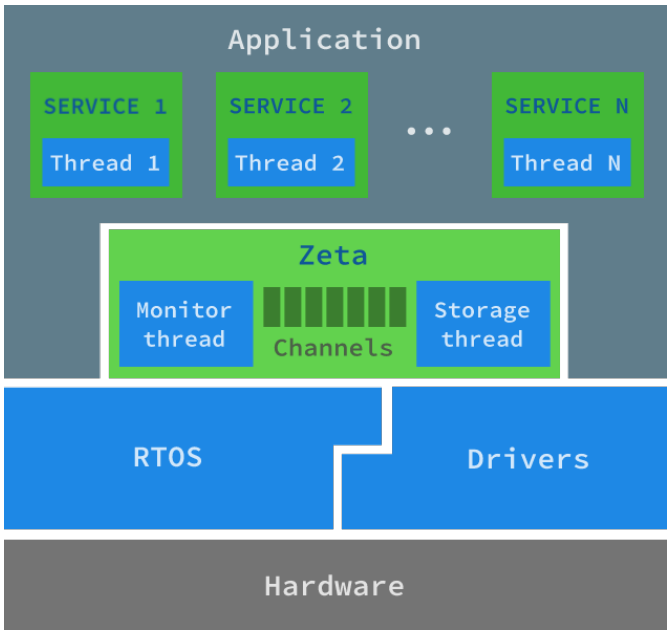


Figure 3. Zeta Middleware overview

cally, the event-driver property system operated similarly to an active database Paton and Díaz [1999]. Developers could store data, and the system actively notified observers about property modifications.

The modules that compose the event-driven property system illustrated in Figure 2 are as follows: **Log.** This module abstracts the system logs by logging application state and events in different ways, such as UART (Universal Asynchronous Receiver/Transmitter) and Bluetooth; **Storage.** This module abstracts non-volatile memory storage and enables the storage of data using different approaches, such as file systems or simple NVS (Non-Volatile-Storage) RTOS implementation; **IO.** This module abstracts user interactions with the device. It enables users to interact with the device through buttons and LEDs; **Sensor.** This module abstracts all sensors' configuration and reading. It enables setting up sensors and fetching data from them; **Network.** This module abstracts all network configuration and access. It enables the network to send and receive data to and from other nodes or the Internet via mesh or GPRS. Additionally, this module automatically detects the presence of the GPRS module and configures the device as an ordinary node or a sync node, thereby changing its behavior; **Core.** This module abstracts all the application logic by coordinating the other modules to make the system operate properly.

We observed that the event-driven property system met our requirements for developing the VSX software. In particular, we could test the **Core** module using mocks to interact with it via its properties, which proved particularly useful for this task. Any change made to a module did not affect other modules. We could even completely replace modules with no system code modifications. The development speed increased significantly compared with the previous version of the property system.

After discussions with the VSX project team, we concluded that the event-driven property system is suitable not only for developing the VSX software but also for other IoT projects. Thus, we decided to adopt this solution company-wide. However, the event-driven property system was developed for the VSX software. To use this solution company-wide, we needed to make it more generic and less dependent on the VSX project.

2.2.2 Zeta

Encouraged by the advantages of the event-driven property system, we decided to further develop a more robust and comprehensive tool. As a result, we developed a middleware, *Zeta*, specifically designed to assist developers in creating modular embedded software. Figure 3 depicts the Zeta middleware overview. This middleware adopted a communication model that enabled modularity via a publish-subscribe mechanism Eugster *et al.* [2003]. In particular, *Zeta* could generate code from channel definition files, where developers could outline the interplay between channels and subscribers. Thus, *Zeta* empowered developers to make diverse architectural choices for solutions on resource-limited devices running real-time operating systems (RTOS). Additionally, we introduced the Zeta CLI (command-line interface), which provides tools for code generation functionalities.

From the bottom up, in Figure 3, the hardware block represents the solution hardware (MCU, peripherals, sensors, etc.), the RTOS, and the drivers that illustrate the system software available to the developer. The Zeta middleware corresponds to channels, as well as the monitor and storage threads. The latest is the application that gathers all the services.

After finishing the first Zeta version, we used it to develop an embedded software product. We observed that Zeta significantly benefited developers by improving code maintainability. However, we observed that the development team had difficulty using the Zeta CLI and interpreting the generated code. To mitigate this issue, we improved the *Zeta* as follows.

First, we intended to replace the Zeta CLI. Although the CLI offers several technical advantages, it introduces complexity for developers. To improve Zeta's usability and maintainability, we required an artifact that enables us to implement code directly rather than generate it from a model using

a tool. Several established projects, such as Quantum Leaps[7] and the IAR state machine tool[8], which rely on software tools to generate code for the solution. Some clients and partners complained about costs and reported problems with model-based code-generation tools related to version compatibility in long-term projects. Thus, it must be open-source, enabling the community to maintain it independently of our efforts. The goal is to spread out the technology.

Second, we considered using only Zeta's thread communication mechanism, rather than the entire Zeta middleware, to generate all architectural code. In that direction, we developed a mechanism that enables threads to communicate in an event-based, many-to-many topology, without making the threads code-coupled in time, space, or synchronization. As a benefit, Zeta became a software bus that enabled developers to create modular and decoupled software architectures.

Third, we decided to contribute to a consolidated open-source project rather than maintaining an independent, complex tool. We started searching for an open-source RTOS community to contribute to the evolution of the Zeta. We observed that FreeRTOS[9] was the most used RTOS at that date. However, Amazon assumed a stewardship role[10], which discouraged other companies from investing in it and could affect its community adoption.

As an alternative to FreeRTOS, we examined other open-source RTOS repositories that use C as the programming language and were the most starred on GitHub Explore query[11]: RT-Thread[12], Zephyr RTOS[13], RIOT[14], AliOS[15], ThreadX[16], and Nuttx[17]. Table 1 lists the inter-thread communication capabilities of the considered RTOS. In addition to feature capabilities, we considered governance, neutrality, and community growth. Following a systematic evaluation of available platforms, we directed our contributions to Zephyr RTOS, a Linux Foundation project that has demonstrated increasing visibility and adoption within the embedded systems community.

2.2.3 Zeta Becomes ZBus

Pursuing our earlier decisions and our choice to engage with the Zephyr RTOS project, we renamed Zeta to ZBus (meaning Zephyr-Bus) and submitted it to this community. Integrating ZBus into the Zephyr codebase was challenging. It involved numerous change requests from collaborators and maintainers. However, this effort produced a stable, complete version of ZBus and ensured its maintenance through Zephyr releases. In the subsequent sections, we will delve into the technical details of ZBus.

[7] https://www.state-machine.com/
[8] https://www.iar.com/products/iar-visual-state/
[9] https://www.freertos.org/index.html
[10] https://www.freertos.org/FAQ_Amazon.html#has_amazon_forked_freertos
[11] https://github.com/topics/rtos?l=c&o=desc&s=stars
[12] https://www.rt-thread.io
[13] https://www.zephyrproject.org/
[14] https://www.riot-os.org/
[15] https://alios.cn/
[16] https://https://azure.microsoft.com/pt-br/products/rtos/
[17] https://nuttx.apache.org/

2.3 Early Versions of ZBus

Figure 4 illustrates the ZBus overview. The bus comprises channels that contain control metadata and the message itself, as well as subscribers and listeners. The ZBus event dispatcher uses shared memory to enable fast message exchange and implements two communication paradigms: publish/subscribe and message passing.

Communication participating entity coupling is a key aspect of a software bus, defining how entities interact. It can be decomposed into space coupling, where producers and consumers must know each other; time coupling, where producers and consumers must participate in communication simultaneously; and, last, synchronization coupling, where publishers are blocked while generating events Eugster *et al*. [2003].

Message passing is a low-level form of distributed communication, usually asynchronous for producers and synchronous for consumers, inducing temporal coupling for the latter. On the other hand, Publish/Subscribe is a more abstract communication paradigm that provides decoupling in time, space, and synchronization among publishers and subscribers Eugster *et al*. [2003]. We have chosen to use the channel-based publish-subscribe variation Jacobsen [2009] instead of the topic-based variation, which is typically string-based and incurs significant filtering overhead on the broker side. This variation provides a simpler approach in which entities focus on channels rather than on messages or topic names.

We opted to provide both communication paradigms to make ZBus sufficiently generic and powerful to address the community's needs. Message passing addresses speed and low-latency scenarios, and publish/subscribe provides a high level of abstraction with strong decoupling. ZBus uses shared memory as the communication mechanism to make messages available to consumers, since it outperforms sockets and pipes Venkataraman and Jagadeesha [2015]. It also leverages copy semantics to make implementation and maintenance more straightforward Freeh *et al*. [2003].

2.3.1 Channels

A channel stores a message and metadata in a shared memory managed by the bus. The message is a predefined structure that describes the data exchanged over the channel. The Listing 3 illustrates a ZBus message using a plain C `struct`.

```
1 struct acc_msg {
2     uint32_t x;
3     uint32_t y;
4     uint32_t z;
5 };
```

Listing 3: ZBus message structure.

The metadata describes the following channel elements: **Mutex.** Enables the RTOS to control access to the channel by using priority inheritance and thread mutual exclusion to avoid race conditions; **User-data.** Enables developers to increase ZBus channel capabilities by adding some metadata to it; **Message validator.** This function checks the message's validity before publishing it. If the message is invalid, the publishing action cannot occur; **Observers.** It stores the list

Table 1. RTOS Comparison

RTOS	Shared Memory	Message Queue	Stream/ Pipe	Mailbox	Event Flag/Signal
FreeRTOS	✓	✓	✓		✓
RT-Thread	✓	✓		✓	✓
Zephyr	✓	✓	✓	✓	✓
RIOT	✓	✓			
ThreadX	✓	✓			✓
NuttX	✓	✓	✓		✓

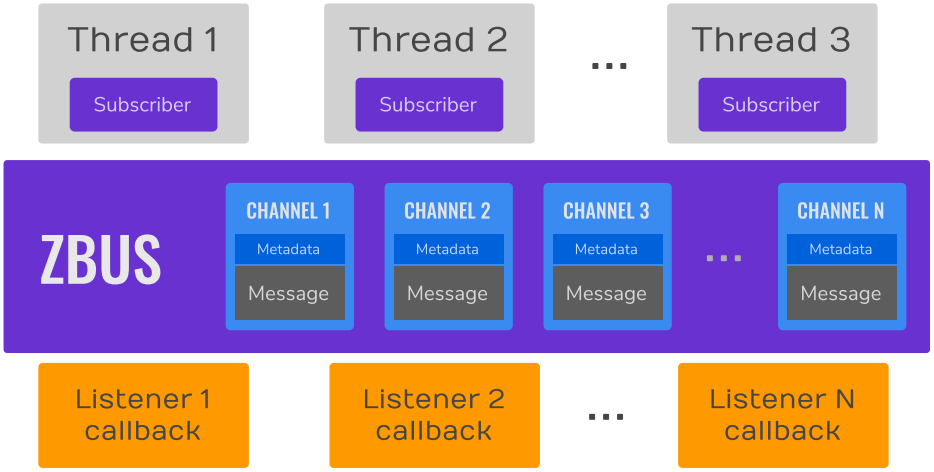


Figure 4. ZBus architecture

of the channels' observers. Observers can be listeners or subscribers for synchronous and asynchronous notifications, respectively.

2.3.2 Messages

Messages are structured data that channels use to exchange information. In ZBus, a message is a C `struct` defined at compile time and assigned to a channel. All messages are statically allocated in the shared memory. Nevertheless, the developer can attach complementary data to a channel by referencing it in the message. Hence, the channel message is in shared memory, and the user's complementary data is in a separate location. The user can allocate the external data dynamically or statically.

2.3.3 Virtual distributed event dispatcher

The Virtual Distributed Event Dispatcher (VDED) constitutes the bus logic responsible for disseminating notifications regarding message publications to the channel's observers. In ZBus, no central entity functions as an event dispatcher. We therefore describe it as virtual because it behaves as an event dispatcher without actually being one, and as distributed because its execution occurs in the context of each publishing thread.

During the notification process, the VDED transmits the message to the listeners, whereas for subscribers, it forwards only the channel identifier. To complete message transmission, subscribers must subsequently perform an explicit read operation to retrieve the message, since the interaction is asynchronous by design.

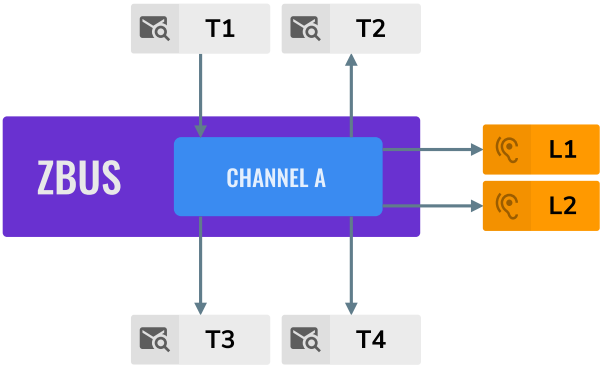


Figure 5. Publication scenario where priorities of threads are $T1 < T2 < T3 < T4$.

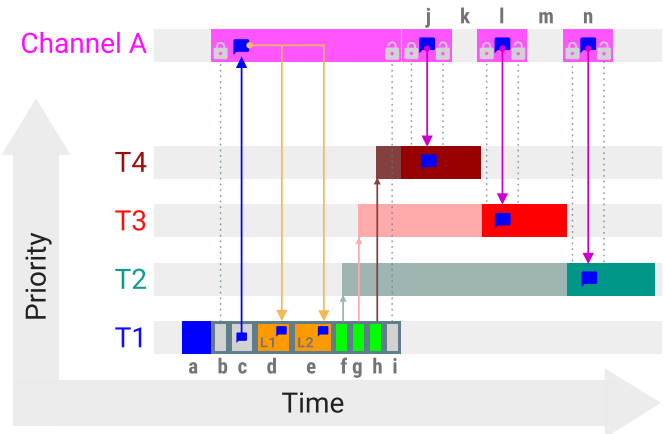


Figure 6. VDED delivery message process for scenario illustrated in Figure 5.

Because the thread that executes the publishing action also handles notification distribution (VDED logic), communi-

cation can occur in different priority contexts. To illustrate the ZBus message delivery process, we suppose a scenario depicted in Figure 5, while Figure 6 illustrates the execution step-by-step. T1, the lowest priority thread, publishes a message to Channel A. The VDED delivery process comes from Figure 6(b) up to Figure 6(n). It starts with a channel mutex lock (b); it executes a message copy (c); it executes the listeners (d, e), it notifies the subscribers (f, g, and h); it unlocks the channel (i); T4 retrieves the message and performs some action (j, k); T3 retrieves the message and performs some action (l, m); at last, the T2 retrieves the message (n).

2.3.4 Basic operations

Threads can publish to, read from, and observe (listen and subscribe) channels. Publishing and reading are supported in all RTOS contexts, except when executing within an Interrupt Service Routine (ISR). The publishing and reading operations are designed to be simple and efficient: each consists of acquiring a mutex and then performing a memory copy of a shared memory region. Channel observation is supported both at compile time (static observers) and at run time (dynamic observers). At runtime, however, it is possible to disable a static observer and remove a dynamic observer if necessary.

Suppose a usual sensor-based solution, as depicted in Figure 7. The application functioning based on the bus configuration is as follows: 1) When triggered, the Timer publishes to the `Start trigger` channel using a work queue; 2) As the sensor thread subscribed to the `Start trigger` channel, it receives the notification and starts to fetch the sensor data. Notice that the VDED executes the blink callback because it also listens to the `Start trigger` channel; 3) When the sensor data is ready, the sensor thread publishes it to the `Sensor data` channel; 4) The core thread as a `Sensor data` channel subscriber processes the sensor data and stores it in an internal sample buffer. It repeats until the sample buffer is full; when it happens, the core thread aggregates the sample buffer information, prepares a package, and publishes it to the `Payload` channel; 5) The Lora thread receives the message because it is a `Payload` channel subscriber and sends the payload to the cloud; and, 6) When the Lora thread completes the transmission, it publishes to the `Transmission done` channel. The VDED executes the blink callback since it listens to the `Transmission done` channel.

The depicted implementation approach offers flexibility, allowing us to make changes independently. It supports independent modifications across threads without affecting other system components. If we want to switch the communication interface from LoRa to Bluetooth, we only need to adjust the LoRa thread. No other changes are necessary to enable this switch. As a result, developers can apply this change to each block in the image.

2.4 ZBus Quality Characteristics

In this section, we discuss the quality characteristics of ZBus as outlined in ISO [2023]. The principal advantages of using ZBus include enhanced solution maintainability, stemming from its modularity and reusability, and software flexibility, which facilitates adaptability and scalability. Conversely, the drawbacks of ZBus include relatively low performance and a notable learning curve, which limit its interaction capabilities.

Maintainability. Software modularity is crucial in modern software development. ZBus is an effective tool for promoting modular software design in embedded systems. By using ZBus, developers can create decoupled code that can be reused with minimal or no changes in other software. For instance, a well-defined software module with clearly defined behaviors can use ZBus to interact with other application modules. The developer can reuse the module across different solutions or even across various hardware architectures if the module contains no architecture-specific, hardware-dependent code. The reused solution must implement only the interfaces (sets of channels) the module requires to function.

Flexibility. The flexibility of ZBus is a vital aspect to consider. As a fundamental subsystem of the Zephyr RTOS, it was designed to be adaptable to various hardware and solutions. Whether it's a small sensor or a processing-intensive cluster, ZBus is useful thanks to its features. For example, messages can be allocated dynamically or statically, and notifications can be synchronous or asynchronous. Moreover, developers can control the channel by claiming it and adding metadata information using the user-data field. Developers can use validators to ensure that messages are created correctly. These characteristics make ZBus a versatile and valuable open-source community tool.

Performance Efficiency. Efficient and flexible functioning of ZBus required the implementation of various techniques. As with any abstraction, ZBus will consume memory, primarily in Flash, to store metadata for channels, and the VDED will require processing time to distribute notifications across threads. To address the issue of communication latency, we have introduced "listeners" - an observer type designed for applications with speed and reliability requirements.

Interaction Capability. In opposition to flexibility, the number of possibilities and details of ZBus can steepen the learning curve. Additionally, the shared-memory mechanism used in ZBus message exchange over channels can be harder to understand than message passing. Implementing embedded systems software using a software bus and relying on its architecture differs from current practices among embedded developers, which may take time for the community to understand and adopt. To help overcome these challenges, we have created detailed documentation, numerous ZBus samples, and implemented a rich set of assertion mechanisms that flag common compile-time errors to developers.

The following section outlines how we integrate ZBus into the Zephyr RTOS project. We selected the Zephyr Project, a widely adopted in the industry and well-governed OSS project, as a spreading venue for the ZBus benefits to the worldwide embedded systems community.

3 ZBus as the official Zephyr bus

The Zephyr RTOS is currently one of the most popular topics in the evolution of embedded systems software. Zephyr originated in 2015 as a Wind River Real-Time OS (RTOS)

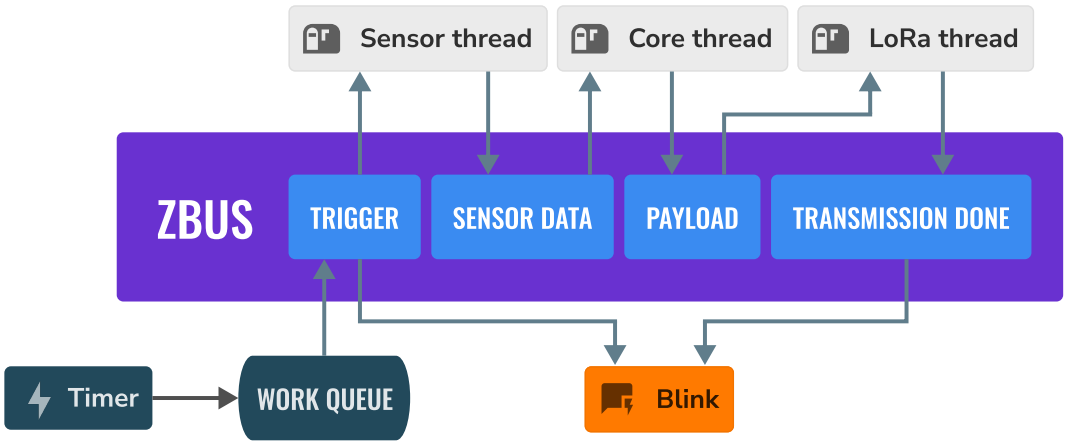


Figure 7. Three threads/subscribers, one callback/listener, and bus interaction.

designed for resource-constrained systems. Its former name was Rocket OS. In 2016, Rocket OS became the Zephyr project, and Wind River transferred it to the Linux Foundation for management as an open-source project THORNTON [2018]. Since then, renowned companies (such as Intel, Nordic, NXP, Google, and Meta) have collaborated to maintain the RTOS Zephyr-RTOS [2026]. Project users, contributors, maintainers, stars, and forks have steadily increased as more companies adopt the RTOS.

There is a growing trend among companies to provide Zephyr-based software development kits (SDKs). For instance, Nordic Semiconductor offers an official SDK based on vanilla or upstream Zephyr (the terms the community uses for the official Zephyr code base). As a result, many companies have shifted from using FreeRTOS (and other real-time operating systems such as Micrium and ThreadX) to adopting Zephyr as the foundation software for their products.

Zephyr's consistency and continuous improvement result from the community's collective effort, which is closely tied to the Linux Foundation's project governance. With its extensive experience managing Linux projects, the Linux Foundation brought many successful practices and advancements to Zephyr. As a result, Zephyr has benefited from several aspects and evolutions pioneered by its "big brother."

Zephyr is currently on version 4.3. To put its community effort into perspective, it has over 128,000 commits, more than 8,000 Discord users, 3,005 technical contributors, and over 3 million lines of code across 20k files. This tremendous effort by the community has enabled Zephyr to reach a high level of maturity, leading to various products that use it as an RTOS entering the market.

Despite significant community efforts and contributions, Zephyr RTOS primarily offers standard thread communication and interaction mechanisms, such as FIFOs, LIFOs, message queues, events, mailboxes, semaphores, and mutexes, which are similar to those of other RTOSs. However, to make the bus available to the growing community, we decided to integrate ZBus into the Zephyr RTOS.

In the next sections, we discuss the Zephyr contribution process, ZBus submission process, a comparison with existing solutions, post-submission community effects, and a report from the maintainer experience.

Zephyr has a well-defined process for developers to submit code changes to the codebase. These changes can range from minor fixes, like correcting a typo in a documentation section, to developing an entirely new subsystem. Depending on the type of change, the developer must follow specific guidelines to work with the community and get the code merged. The process for submitting code to Zephyr's code base is illustrated in Figure 8.

3.1 Is a substantial new contribution?

The contributor should open an issue or start a discussion on the Discord channel dedicated to architectural discussions.[18] The community analyzes the demand and decides if it would be better to contribute to an existing implementation that works similarly or if it would be necessary to write an *Request For Comments* (RFC) in which other community members could contribute to building a non-biased and generic specification to something completely new.

For the ZBus, the first step was to determine how to contribute. During the first interactions with the community, @carlescufi pointed out a Pull Request (PR) about the event manager[19] from Nordic Semiconductors. That PR was an attempt to submit the Nordic Event Manager, which shared similarities with ZBus mainly in its purpose of creating an event bus, but the feature sets between the Nordic Event Manager and ZBus were quite different. Also, the Nordic Event Manager's PR stalled and was abruptly closed due to conflicts during the review stage.

Consequently, the community considered ZBus a new subsystem, since Zephyr lacked a subsystem providing a software bus that enabled threads to communicate in a decoupled way. Thus, we started the ZBus's RFC by detailing all features and contributions to the community.

3.2 Write an RFC

When a substantially new feature is proposed, developers must create an RFC[20] document to discuss the idea before

[18] Discord channel: `https://discordapp.com/channels/720317445772017664/883445484923011172`

[19] Event Manager.`https://github.com/zephyrproject-rtos/zephyr/pull/38611`

[20] Request for Comments. `https://docs.zephyrproject.org/latest/contribute/proposals_and_rfcs.html`

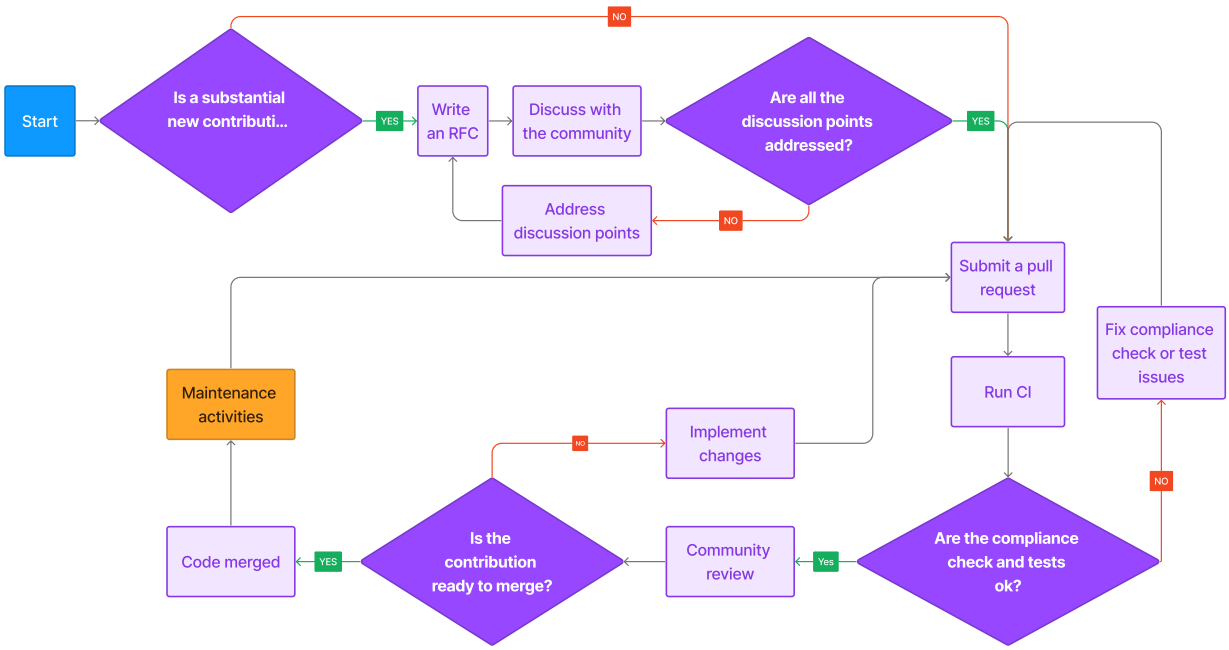


Figure 8. Zephyr contribution process.

submitting the code. The RFC template contains all the necessary information, and the contributing author describes the submission's characteristics in the document. The RFC must follow the template structure[21] defined by the community, which contains information about the problem description, proposed change, dependencies, concerns, unresolved questions, and alternatives.

As ZBus was considered a new subsystem to the Zephyr, we needed to write ZBus's RFC. This RFC provided the community with a clear vision of what ZBus was and what we aimed to contribute. The ZBus's RFC[22] contains the following sections.

Introduction. We describe the targets and users that the RFC will directly affect. It must also include a brief explanation of the scenario in which the new feature will operate.

Problem Description. We describe the technical problem addressed by the ZBus in the context of the Zephyr, as described below:

> Zephyr does not offer any IPC mechanism that supports many-to-many thread communication. There is no straightforward way to implement decoupled many-to-many thread communication in Zephyr. The closest IPC available for that is the mailbox, but it does not deliver the message to many, in fact. It delivers to the first reader. Developers must reinvent the wheel whenever they need many-to-many thread communication.

Proposed Change. We summarize the changes added to Zephyr in a language comprehensible to a broader audience, as described below:

> Add a fast, decoupled inter-process communication mechanism (a message bus, hereafter referred to as ZBus, from "Zephyr-bus") to Zephyr to enable a many-to-many communication model. Using this IPC, developers can easily implement thread communication, even in a many-to-many model. Besides its communication capabilities, as proposed, the bus will offer asynchronous, structured communication with time, space, and synchronization decoupling between the communication entities. Even ISRs (Interrupt Service Routines) can send messages over the bus. The bus, in the current implementation, is made by a static shared-memory portion, semaphores, and message queues working together.

In addition to this summary, we provided a detailed description of the proposal in Section 2.

Dependencies. We describe how the ZBus features may affect the Zephyr system as well as which teams and subsystems of the Zephyr can be affected as well;

Concerns and Unresolved questions. We provide a list of concerns, unknowns, and generally unresolved questions regarding the proposition, as follows: i) The heavy use of the preprocessor may generate unreadable compilation errors when the developer makes a mistake; ii) There is no pub/sub loop checking. It is not an easy task, as it can occur across a chain of calls. If a thread is subscribed to a channel and published to it, maybe a high-priority loop will occur (the event dispatcher must have a high priority), causing a kind of starvation; iii) The bus is not made for streaming purposes. It would be used only for control messages. It will be necessary

[21] Zephyr RFC template. `https://github.com/zephyrproject-rtos/zephyr/blob/main/.github/ISSUE_TEMPLATE/003_rfc-proposal.md`

[22] ZBus's RFC. `https://github.com/zephyrproject-rtos/zephyr/issues/45910`

to measure performance; iv) It will increase the footprint of the solution. Each channel allocates the message structure and a metadata portion of 32 bytes. Each subscriber will allocate a message queue to receive change notifications; and, v) We previously implemented systems using an earlier version of zbus, and the results were promising. Nonetheless, the code needed to be more mature.

Alternatives. We list any considerable alternatives for the ZBus, and, most importantly, discuss the reasons for choosing the ZBus over them, as described in Table 2. Besides discussing the differences between the ZBus and existing alternatives, we perform an experiment to compare the ZBus with the Nordic Event Manager[23].

After finishing the experiments and writing the RFC, we submitted it to the community. From this point, the community reviews the RFC and may request additional information or modifications. Sometimes, the community may ask the author to clarify their proposal during the *Architecture Work Group Meeting*, which is held every Tuesday. We describe these discussions in the following section.

3.3 Discussion with the community

After submitting the RFC, the contributor should discuss the idea openly with the community. This discussion has two stages: trial and contribution. The trial starts with extensive community feedback and questions. Most participants test the concept by deconstructing it and pointing out possible issues to see how the proposer responds. At the start of the trial, much attention is given to the alternatives section. Although the alternatives are often more palatable to maintainers—given their existing implementation and minimal impact on the broader codebase—the contributor's proposal must demonstrate sufficient merit to justify the additional complexity and code changes required.

When the idea passes the trial, the community is satisfied with the results, and there are no more contests; the contributions start. The contribution phase involves collaboratively developing an idea that addresses community problems and sets aside the contributor's bias by creating something generic that solves the community problem, not a particular case.

Regarding the ZBus, the community discussion began with some contributors pointing out other issues and RFCs related to the ZBus. For instance, the users @ck-telecom and @hongshui3000 pointed out the event manager PR[24], which proposed a similar feature but did not successfully merge to the code base. In this case, the authors and the community did not resolve some code-related conflicts, and the authors gave up trying to add it to Zephyr.

Also, the user @henrikbrixandersen asked that KBUS be considered as an alternative to ZBus.[25] We examined the documentation and identified that KBUS had a Linux kernel dependency that prevented us from considering it as a viable alternative. The user @stephanosio indicated the Android CHRE port to Zephyr[26] as an alternative. However, that seemed to be a parallel implementation that does not directly contest ZBus.

@nordicjm commented about one implementation done by another manufacturer inside a framework[27]. We added the alternative and started comparing that with ZBus. In parallel, @carlescufi asked about the main differences between ZBus and the already mentioned alternative, the event manager. We listed the differences. Requests from the community regarding ZBus features, compared with the alternative, prompted us to elaborate on the comparison in Table 2, highlighting the relevant aspects of the solutions. The table presents concepts such as *metaphor* and *made for*, as well as usability characteristics, such as *message allocation style*, and maintainability points, such as *implementation approach* and *maturity level*. All the information together helps reviewers get a broader idea of what ZBus is compared to other alternatives.

With the table stated, we requested @zycz (one of the event manager authors) and @lairdjm (one of the Laird Zephyr Framework authors) to analyze the table information. After reviewing it, we finalized the table. After the table, the community members requested a comparison of the alternative performance with the current version of the ZBus code. @hongshui3000 provided an executable module that allows us to test the event manager against ZBus in the same conditions. The Laird alternative could not participate in the comparison due to insufficient information and test code.

To compare ZBus against the event manager, we designed an experiment called 256KB benchmark described in detail in Section 3.2. We asked the event manager author to review the proposed code. ZBus demonstrated potential by achieving better performance and lower memory usage across all benchmark scenarios compared to its contender.

After the comparison's success, the contributions' tone began pointing to a better feature set rather than trying to beat the idea. With that mindset shift, participants offered good suggestions. @cfriedt highlighted the importance of providing a mechanism for dynamically allocating channels, given that companies are increasingly using Zephyr in complex projects with high-performance processors, and the once-valid solution for constrained devices would be inadequate for this new scenario. Thus, we added a mechanism for creating dynamic channels in ZBus.

The latest interaction before the community accepted the RFC was with @gregshue. We discussed several concepts, from terms to use to the event dispatcher approach. His contributions were significant to ZBus. After several discussions, we replaced the centralized event dispatcher with a virtual distributed one, which would profoundly change its functioning. The change improved the consistency, performance, and composability of both the ZBus and the systems based on it.

3.4 Address discussion topics

Addressing the discussion topics increases the likelihood that the idea will be accepted and continue. After addressing all

[23] https://github.com/zephyrproject-rtos/zephyr/issues/45910

[24] https://github.com/zephyrproject-rtos/zephyr/pull/38611

[25] http://kbus.readthedocs.io/en/latest/specification.html

[26] https://github.com/zephyrproject-rtos/zephyr/issues/37223

[27] https://github.com/LairdCP/zephyr_framework

Table 2. Features comparison.

Comparison item	ZBus	Event Manager NCS	Laird Messaging Bus
Made for	Increase code quality by enabling reuse and increasing testability	Reduce the number of threads using events and callbacks	Multiple receivers broadcast messaging framework system
Metaphor	Messaging bus	Event manager	Event manager
Message definition time	Compile-time	Compile-time	Compile-time, though a custom message could be created dynamically, but the receivers would need to know how to parse the struct
Message definition approach	Centralized, single file to describe channels and subscriptions	Descentralized, one file per each message and subscription would occur in every place	Descentralized, multiple files, usually one per module, all combined into an auto-generated output file by CMake
Message Allocation Style	Static (compile-time)	Dynamic or static (execution-time, it uses a weak alloc function	Dynamic (execution-time)
Message Persistency	Persistent still exists after processed	Transient, deleted after processed	Both, cleaned up if DISPATCH_OK is returned by a handler, otherwise remains
Message Distribution pattern	Publish/subscribe	Message passing (Event/listener)	Message passing (Event/listener)
Subscription style	Compile-time, but it can be disabled in execution-time by masking the subiscriber	Compile-time, but run-time could be added using extension hooks	Compile-time but execution-time filtering was planned and partially added
Message transmission style	Direct transmission when using callbacks style.	Direct transmission	Direct transmission.
Subscriber execution style	Synchronous (by callback), Asynchronous (by queue)	Synchronous (by callback)	Asynchronous (by queue)
Implementation approach	Static memory, semaphores, and queues	Dynamic memory, kernel spinlock, LD files, sys APIs	Dynamic memory (buffer pool), message queues
Code Generation	No, only macros	No, only macros	CMake used for autogenerating: message IDs, message code, and message types
Extension mechanism	Yes (uart|_bridge, remote_mock)	Yes (app_event_manager_profiler, event_manage_proxy)	No built-in
Maturity	Feature-complete	Production-ready	Produciton-ready

the points, a new round of discussions starts, and the process continues until the community has no more points to address. Many contributors give up in this phase, usually because of a lack of time or interest in addressing all the points. This is a patience exercise that the contributor must pass to be accepted.

Once all issues related to the RFC have been resolved and the proposal is accepted, the author can proceed with the second phase of contribution, which involves submitting code.

In summary, based on our proposal (RFC45910[28]), the community could discuss what the bus should take into account to address most of its use case scenarios, leading to changes, additions, and removals to the original proposal. We presented the RFC twice during the Architecture Working Group weekly meeting. After all the modifications and feature adjustments, the bus passed the first trial and was requested to be compared against another existing solution, as described in Section 3.2. The approval rate increased after the positive results of the comparison, and after several discussions and modifications over 69 days, the community finally approved the RFC.

It is important to mention and thank the RFC phase contributions: @carlescufi, @nordicjm, @zycz (Nordic Semiconductors); @cfriedt (Meta); @henrikbrixandersen (Vestas); @gregshue (Legrand); @ck-telecom, @hongshui3000, @stephanosio, and @andrea-cowboy (Independent or company not declared).

3.5 Submit a pull request

The contribution starts with a Pull Request (PR) that includes all the code and related artifacts, such as documentation, samples, and tests. Submitting the code in small increments through the PR is recommended to simplify the revision process. However, this may not always be feasible, and sometimes, the PR may become quite large and complex to review. The time required to merge complex submissions is a mapped issue discussed in the community Process Group. There is an open issue[29] where community members are discussing ways to improve merge time for complex changes.

After the RFC approval, we moved on to the next step of opening a pull request to submit the ZBus code to the repository. So, we started to receive feedback through reviews. The first review came from @gmarull, who provided valuable insights on how to contribute appropriately to the codebase. We made several changes based on the feedback we received, including improving how we organized the code and structured our commits. One critical area of feedback was on the way we defined channels. Our approach to defining channels in a central file was

[28] https://github.com/zephyrproject-rtos/zephyr/issues/45910

[29] https://github.com/zephyrproject-rtos/zephyr/issues/56304

problematic. @gmarull suggested using iterable sections and improving the codebase's composability, allowing developers to create channels more easily throughout the codebase. Summary of @gmarull review: i) The commits must be atomic. They need to work with no dependencies to a future commit; ii) The guard in `ifdef` must follow the path of the file; iii) We should use `zephyr/` prefix on all the includes related to Zephyr; iv) We should use `__DOXYGEN__` flag to enable conditional features during the documentation phase; v) We should avoid log function aliases; vi) We should use `cond INTERNAL_HIDDEN` to hide internal helper functions from the public functions documentation; vii) He suggested a proper name for the function called `ZBUS_SUBSCRIBER_DECLARE`. The name should be `ZBUS_SUBSCRIBER_DEFINE` since the function semantic was about defining a subscriber and not declaring it; viii) He pointed out that API variety (`zbus_chan_pub`, for example) could confuse users; ix) We should not use hardcoded sizes in the code. It must use configuration flags from Kconfig instead; x) He also suggested not reinventing things and using the already implemented things like `CODE_UNREACHABLE`; xi) He suggested the `zbus_info_dump` should be replaced by a function that gives the user the power to decide what to do with the Zbus info instead of printing it. Based on that, we created the channels and observers iterators; xii) He indicated that some verifications of the code should be done in an assert instead of by checking it directly, since it was not the standard way of doing things in Zephyr; xiii) We should avoid log tags like `[ZBUS]`; and, xiv) At last, he commented on some other minor improvements.

Following @gmarull, @carlocaione provided several tips for improving the code and identified a potential issue with missing mutex unlocking when a failure occurs. However, upon investigation, the code was correct.

@pdgendt started his comments by trying to make the ZBus APIs available on ISR contexts, but unfortunately, it was not possible based on the mutex's restrictions. He also points out an improvement to the code regarding the timeout calculation for ZBus functions. In the worst-case scenario, the function would take twice as long as the chosen timeout. Accordingly, we needed to fix it by calculating the timeout correctly. After the first fixes, @pdgendt conducted another review and suggested additional improvements, such as extracting the complex code that calculates the remaining timeout into a helper function. Lastly, he indicated ways to improve the code by using the macro `MAX` correctly. He suggested a new way of implementing ZBus' functions, which would enhance the readability of the code. Interacting with him in the community chat, he gave us great advice regarding vertical indentation: separating actions vertically (by adding blank lines) to make clear which commands are working in the same activity and which are not.

With @yperess, the review sparked a discussion on the best way to test assertions in Zephyr. We created a `_ZBUS_ASSERT` that overwrites the system assertions to enable us to test them during automated tests. No other subsystem or Zephyr code did that before. As all of ZBus functions use asserts to check the parameters, we needed a way of testing it automatically. After some discussion, @yperess concluded that the way we were doing it was acceptable and allowed us to keep our approach. In addition, he suggested that @alevkoy use the same approach for the Vbus code.

@fabiobaltieri started his review, discussing the documentation and suggesting several corrections and improvements. After that, he began indicating infrastructure improvements for the samples on the CMake, overlay, Python scripts, etc. He offered several structural code improvements, such as avoiding mixing `printk` with logs, adding blank lines where needed to separate code, not initializing variables that were later changed, removing unnecessary casts, and so on. He was strict about the usage of static for all the local variables and functions in an implementation file. He finished his review by suggesting improvements and corrections to the tests.

@anangl questioned the read-only channels feature. Another suggestion from him was to adjust the inline usage. The way we were using that was not effectively improving the code performance. The correct method was to create an inline function in the header file rather than the implementation file. @anangl also pointed out the unnecessary RAM by keeping mutable variables that could be constant. Constant variables usually go to Flash. He also suggested fixing ZBus' functions signature by adding the `const` on the channel and the message since they must not change. He instructed us to add `static` to the mutex by preventing other code from accessing it externally. He later suggested hiding every variable and function that should be used only locally in a specific file as `static`. At last, he suggested some changes to the conditional code compilation to make that more efficient in case some fields and functions were not necessary, so we added some configurations to the Kconfig file to enable names like `CONFIG_ZBUS_CHANNEL_NAME` and `CONFIG_ZBUS_OBSERVER_NAME`. We added a configuration to allow the runtime observers as well. In this way, the code became more versatile.

@desowin pointed out several improvements and problems in code indentation and documentation. He also discussed read-only channels, which convinced us to remove them. We also discuss Zephyr coding style, how ZBus sorts the observers and channels, and other minor clarification points. After @desowin, @anangl gave the latest suggestions about the publication process, which is the more sensitive part of ZBus. The points were: i) Timeout calculation and usage. ii) The return point of a failure. iii) Protection of the runtime observer's pool. It was susceptible to concurrency problems. All the comments were addressed or clarified. After more requests from @anangl and @fabiobaltieri to fix minor issues and improve code performance, ZBus was finally accepted.

The modifications ranged from simple documentation typo corrections to significant architectural changes. The process took 163 days (over 5 months), from May 23rd to November 1st, 2022. The following Zephyr community members have made an essential contribution to ZBus evolution and acceptance to the PR merge: @gmarull (TESLABS); @anangl (Nordic Semiconductors); @fabiobaltieri, @yperess (Google); @pdgendt (basalte); @cfriedt (Meta); @desowin (Independent or company not

declared); @carlocaione (BayLibre); @MaureenHelm (Analog Devices); and, @nashif (Intel).

3.6 Run CI

The Zephyr RTOS has a comprehensive continuous integration (CI) process for verifying pull requests (PRs). The CI includes 36 activities that ensure code style, documentation consistency, licenses, commit text and format, running and implementing tests, regression tests, and more. The review process will begin only after the CI passes all checks.

The *Check Patch* step is essential in making Continuous Integration (CI) routines more robust and complete. This step ensures the code follows the Linux kernel coding style as a reference. If the code does not follow the coding style rules, such as a line being in the wrong place or a simple space not being correctly placed, the checker will reject the code and prevent a merge. The *Gitlint* is also a part of this process. The CI will reject the submission if the commit does not follow the rules.

The contributor is responsible for implementing test routines for the new feature. The Zephyr project usually uses tools to check test coverage and verify that the addition is well-tested using code structural coverage metrics. Historically, the system used the Codecov[30] tool, but it now uses the Coverity[31] tool. The latter additionally performs static analysis on the code.

We based the ZBus tests on the techniques provided in the book *Effective Software Testing: A developer's guide* Aniche [2022], as it provides practical ways to implement tests. To ensure good coverage, we mainly implemented the unit tests using specification-based, boundary, and structural tests. We also implemented integration tests to verify that the bus behaved properly with a couple of threads using it, similar to an actual application. Ultimately, the test coverage of ZBus reached 93% of line coverage and 81% on branch coverage metrics. We could improve this result further because the tool did not account for coverage on macro lines, which we used frequently in the code.

The documentation is another essential part of the submission. The compliance check also verifies that all functions in the header files have documentation and that all arguments are correct and documented. Documentation must describe how to use the new contribution for a broad audience, using clear, accessible writing that accommodates varying levels of expertise.

Writing the ZBus documentation was challenging because the audience ranged from novice to experienced developers. Consequently, we created our sections based on the documentation of other Zephyr subsystems. We decided to describe that in more detail and let it grow and be complete. The community members and the author carefully reviewed the documentation.

The last important part is the samples. They are used to show users how to use something through examples. On Zephyr, the CI uses samples as part of the testing routines by checking some output results to verify if the sample is still running. In addition to the additional tests and samples, the CI will run the regression test to ensure the new addition does not affect unrelated project code. The CI process takes some minutes to complete.

For ZBus, we have samples to illustrate the usage of each main feature. The samples also serve as tests, as we can set up a sample to run as a test and verify that the output text is correct, increasing the test count and scope.

3.7 Fix compliance or test issues

To avoid compliance issues during the check, the developer can run the same procedure in a local environment. It would save a lot of time waiting for the CI to run and make the submission cleaner. The developer must fix all issues before resubmitting, and only after resolving all of them will the CI phase succeed, and the PR will be ready to merge for a compliance check. However, the community will start the review rounds.

The *Check Patch* captured several non-compliant codes based on the Zephyr coding style[32] derived from the Linux Kernel coding style[33] during the ZBus submission process. We needed to read all that and make several changes to the code. After some iterations, we learned the coding style and used tools to ensure it was correct before pushing the code to the repository, which triggered the CI process. The *Check Patch* can also be performed in a local environment. Thus, we did that as part of the submission process.

Incomplete documentation also makes the CI fail. The CI process also indicates a failure if the submitted code lacks documentation for a function or if the documentation is incomplete. Hence, the function must be documented or properly flagged as not part of the module documentation by using Doxygen flags. The proper way is to add `@cond INTERNAL_HIDDEN` and `@endcond` around the hidden code lines. We have that on ZBus code for the helper functions and macros.

The CI runs tests and samples across all available platforms to ensure the code is available on all project targets. In that phase, some architecture-specific details can cause tests to fail. The contributor should be aware that the contribution must be runnable on all Zephyr's supported architectures and available boards.

The CI executes all tests against all the virtually available platforms. That can generate several issues. In ZBus, some tests, mainly the ones derived from samples, rely on output sequences. Several platforms presented different output sequences, specifically when the Synchronous Multi-Processor (SMP) was enabled. That happened several times within ZBus submission. In some situations, changes may cause unexpected behavior when running ZBus on certain platforms, potentially affecting test results. After investigation, we found the issue was due to a recent change that introduced defects into the platform's execution. In the last instance, we can exclude a platform from the possible platforms to test the code. The most complicated platforms for ZBus

[30] https://about.codecov.io/

[31] https://scan.coverity.com/

[32] https://docs.zephyrproject.org/latest/contribute/guidelines.html#coding-style

[33] https://kernel.org/doc/html/latest/process/coding-style.html

tests execution were `fvp_base_revc_2xaemv8a_smp_ns` and `qemu_leon3`. The tests usually gave us false positives for both.

3.8 Community review

The community starts the review when the CI routine concludes with no issues. Sometimes reviewers can start before that, but only in rare circumstances. The community contribution review is a critical part of the process. During this phase, Zephyr's current contributors and maintainers examine the code to identify issues and provide potential solutions.

It is common for reviewers to request changes in this phase. The author must convince them that the modification is unnecessary, discuss the proposed solution, implement it, and resubmit the Pull Request (PR) with the changes. This step is essential to ensure the code is high-quality and meets the project's standards.

The community review has several nuances. It starts with a functionality inspection where the reviewers focus on the feature aspects, architecture, and implementation approach. With ZBus, the first phase began with discussions about architectural choices: how to define channels, discourage logging within the ZBus core, and so on. After the first phase, the coding details came into the spotlight. A more detailed review of the code began with an eye on basic coding style and naming choices, and even discussed the complexity of functions. Once the code passed review, the documentation revision began. Usually, the documentation counts on different people reviewing it. The documentation revision includes corrections of typos, suggestions on the section disposition, and changes to figures. When the documentation is ready, the code is prepared to be merged upstream. As the ZBus has extensive and detailed documentation, that phase took much time.

3.9 Implement changes

The contributor must address all change requests by either implementing them or convincing the requester that the change is unnecessary. Sometimes, there is no agreement about the change requests, and the escalation process begins. The PR must pass that process to resolve all technical disagreements about some aspects of the contribution.

The first stage of the resolution is to discuss that in the PR. If the contributor and the reviewer cannot converge at some point, the contributor can ask to discuss that in the Zephyr Dev Meeting, the second stage. If the disagreement remains unresolved after the Dev Meeting, the contributor can dismiss some irrelevant requests by adequately documenting the dismissal. However, suppose the reviewer disagrees with the dismissal. In that case, they can escalate the situation to the Architecture Working Group, which can help with offline moderation or even an online meeting.

If all avenues of resolution and escalation have been exhausted, the involved parties can escalate the matter to the Technical Steering Committee (TSC). The TSC is a neutral group of representatives from companies and the community that drives the project. They will decide a verdict about the situation, and the contributor or reviewer must attend the resolution. That is quite a rare situation, but it is expected, and the project governance provides ways of resolving it.

Once all the requested changes have been resolved, the PR must be approved by at least two reviewers before a highly privileged community member can merge it. During ZBus submission, no higher escalation stage was necessary other than the usual discussion with the reviewer. No conflicts were generated, and all requests were addressed.

The most critical implemented changes the community requested, along with their implications during the code review process, are as follows.

Decentralize the event dispacher. The first implementations of ZBus used a centralized event dispatcher that would block all operations while propagating notifications for a changed channel. Thus, that could increase latency and interfere in the wrong context. Sometimes, it was possible to see a low-priority thread publication interfering with a high-priority thread operation because the event dispatcher should be the higher-priority thread among the preemptive ones. The community members pointed this as an open issue to be improved.

We removed the event dispatcher thread and implemented the notification process on the publishing function. This way, the publication process began to run directly from the publishing thread context, which would not interfere with higher-priority thread contexts and did not require a dedicated thread for the notification process. All the involved entities now contribute to that process.

The changes ended up making the event dispatcher distributed and virtual, because it does not exist, and several entities act like that to fulfill its attributions. We now call it the Virtual Distributed Event Dispatcher (VDED), which decreases memory usage and increases communications speed, flexibility, and consistency through ZBus.

Listeners. At first, ZBus only implemented asynchronous communication based on threads and queues (called subscribers). The Nordic Event Manager (the ZBus competitor) implemented synchronous communication only. The community considered the additional features proper to fit more needs and solve different problems and scenarios. We added the listeners as a variation on observers and changed the notification process to consider calling callbacks for listeners before starting to notify subscribers. ZBus provides a faster and guaranteed delivery communication mechanism with listeners to fit better in constrained devices and high-speed applications.

Mutex instead of semaphores. ZBus employed semaphores to lock the channels, to avoid concurrency problems. However, the community advised using a mutex instead because it could benefit the priority inheritance feature, since semaphores do not implement that. By using mutexes, the VDED can operate even more consistently. Given the mutex's priority inheritance, the communication occurs on the higher-priority pending thread. It reduced latency and improved communication consistency. The disadvantage of the solution is that the ZBus features cannot be used inside interrupt service routines.

Decentralize channel creation and subscription. At first, the developer must create the channels in a single file. The community referred to that as a too-limiting approach, giving

the composability perspective. Wherefore, we changed the way in which channels (and observers) are created, and now it is possible to define and declare them in different files. In addition, the operating system (or services) can create channels and observers independently from the developers. Therefore, this approach increases composability, code reuse, and abstraction.

Dynamic observation. In ZBus, a channel could only add an observer to its observers list during compile time. The only available flexibility was disabling the observer at runtime with a boolean flag. Community members suggested improving this by implementing a way to add and remove observers from channels at runtime. We added a dedicated memory pool to allocate observers' links to the channels' observer lists. Adding and removing observers to and from a channel at runtime is possible. That increases flexibility and allows changing the system behavior by changing observations at runtime.

Unlock mutex before propagating the notifications. In ZBus, the VDED blocks the channel while propagating the notification to all observers. One community member asked to remove the block and leave the scheduler to handle the sequence and notification procedures, even if that allows nested notifications. He suggested that it would avoid communication limitations. We disagreed with this request. The current blocking approach ensures consistency and avoids chained notifications, which could lead to unpredictable notification ordering and potential priority inversion in nested scenarios. A different thread (from the publishing one) can access a notified channel only after the notification procedure ends.

With the cited changes, ZBus was improved before being accepted into Zephyr. The submission process made it a better solution that fits more use cases and is more flexible, enabling developers to build composable, reusable applications.

3.10 Code merged

The contribution activities end with the code merging. If it is their first contribution, a GitHub robot will send a message congratulating the author. The community may assign them to future issues on the contributed code. However, depending on the nature of the contribution, the contributor may also become a maintainer and have their name added to the MAINTAINERS file. With that, the community will assign them all the issues or RFCs related to the maintenance code.

On November 1st, 2022, ZBus was merged into the Zephyr code repository by the community. As usual, one of us was established as the maintainer of ZBus[34]. From that point on, the community assigned him all issues or related movements regarding ZBus on the repository. The first official release of Zephyr that included ZBus was 3.3.0, which was released on February 22nd, 2023.

The first moment of community engagement occurred when the Golioth[35] company requested a call to discuss a new feature. That resulted in a ZBus post on their blog[36]. The Zephyr Discord server opened a ZBus dedicated channel at that moment. The Golioth blog drove some movement to the channel with questions and suggestions. The first discord channel interaction with the community was related to the CogniPilot[37] project, where some contributors began considering using ZBus to exchange events in their software. Another notable project is the ZSWatch[38], an open smartwatch that relies on ZBus to orchestrate the events and messages of the system. On 2023 June 19th, the ZBus Discord channel had 238 messages.

It's worth noting that a session and tutorial called "ZBus - the lightweight and flexible Zephyr message bus" were accepted at the Embedded Open Source Summit held in Prague in 2023. The first author was able to attend the event in person thanks to travel funding provided by the Linux Foundation. The session[39] was held on June 27th, and the tutorial[40] on June 29th, 2023. Similarly, this happened in the North American edition of the same event, the Embedded Open Source Summit NA 2024, held in Seattle, Washington. The first author gave the session "ZBus - New Features and Roadmap"[41].

Beyond the main events, the following items describe other notable happenings from the ZBus addition: i) Discussions about adding ZBus as a backend of the Input subsystem. The subsystem sends events from keys pressed. But the first version does not consider using ZBus as the event dispatcher. The conclusion was to submit ZBus as a backend alternative in a pull request after the Input acceptance; ii) ZBus addition solved an open issue[42] from 2018. ZBus performs precisely what the developer requested on the issue; iii) The ZBus pull request added 100 files of ZBus code, documentation, samples, and tests. The review process took a lot of work to complete. ZBus was cited as, based on the size, a hard-to-review example during the contribution process improvement discussion[43]. On the other hand, the organization and clarity of the addition were praised with the following phrase[44]: "*Usually PRs of this size are painful to review, but this one reads very easily. Thank you!*"; iv) ZBus helped to find mangling logging bugs on other parts of the code during the test procedures on two occasions[45,46]; v) An issue[47] raised indicates ZBus cannot be used in architectures with cache incoherance; vi) An issue[48] opened regarding improvements on how ZBus propagate messages. The case became a discussion that ended with the issue creator understanding the way of work of ZBus and proposing an addition to the

[34] https://github.com/zephyrproject-rtos/zephyr/blob/327eb119b6bc3166b2e623792567e9831f4385d7/MAINTAINERS.yml#L3398

[35] https://golioth.io/

[36] https://blog.golioth.io/

[37] https://github.com/CogniPilot

[38] https://github.com/jakkra/ZSWatch

[39] https://youtu.be/EBnFqARqyeQ?si=3soeVcGI3k1fEf0V

[40] https://youtu.be/4TRrSmPTei8?si=F_3-WaawdRqPAAqT

[41] https://youtu.be/f8saK6fcf00

[42] https://github.com/zephyrproject-rtos/zephyr/issues/9045

[43] https://github.com/zephyrproject-rtos/zephyr/issues/50836

[44] https://github.com/zephyrproject-rtos/zephyr/pull/48509#pullrequestreview-1127281924

[45] https://github.com/zephyrproject-rtos/zephyr/issues/55071

[46] https://github.com/zephyrproject-rtos/zephyr/issues/55679

[47] https://github.com/zephyrproject-rtos/zephyr/issues/54121

[48] https://github.com/zephyrproject-rtos/zephyr/issues/58004

documentation. The repository representatives merged the documentation improvement into the code via PR[49].

3.11 Maintenance activities

The author automatically becomes the maintainer of a feature if it is a subsystem or module. However, maintainers must adhere to the community's well-defined code of conduct. The maintenance phase involves supporting the community in fixing bugs and working on new features and improvements. As the documentation[50] states, A Maintainer is a Collaborator who is also responsible for knowing, directing, and anticipating the needs of a given zephyr source code area.

Being a maintainer is challenging and requires a lot of work, especially if you are independent (not supported by a company). We receive a lot of feedback from the community, from encouraging words to severe, maybe even offensive comments. Many developers are giving feedback about the subsystem. As far as the work goes, there's more pressure. Being part of the release process is an exciting activity. There is the right time window to contribute, the feature freeze, and other stages of the release. We need to stay focused and adequately plan the features to be sure they will be available in the next release.

Another challenge of being a maintainer is dealing with the community work overload. Everyone has a lot of work to do, and reviewing PRs is demanding. There is an instant moment in which you have to decide to make your life easier by creating an extensive and complete PR, or you can make the reviewer's lives better by splitting up your contribution into small steps. However, if you take too many steps, you will take months to have a completely new (maybe complex) feature in the system and skip one official release. Thus, even the size of the contribution is a decision moment.

As ZBus is a secondary feature and not an essential part of the system, it can be challenging to draw the attention of other maintainers. ZBus is a tool that assists developers in creating high-quality embedded software. However, the system software does not utilize it. Therefore, the benefits of ZBus are more noticeable to external developers than to internal ones. We are seeking other developers to assist in maintaining ZBus because we have numerous ideas for enhancing its user experience.

The bias is a vital aspect to avoid during the maintenance phase. Nobody plans the subsystem feature with you when you are the sole maintainer. The community gives us some tips, but the maintainer is usually responsible for reviewing the feature roadmap. Therefore, maintainers must often try to see a new feature from another perspective because a feature that seems beneficial may not make sense to others. Companies and developers are using Zephyr on a wide variety of devices. Thus, it is complex to design such a fundamental part of the system to fulfill the needs of someone who uses an 800MHz processor with 1GB of RAM and another who needs that on a battery-operated device running a 16MHz MCU with 32KB of RAM. So, bias and a lack of vision are among the most challenging aspects of the subsystem maintenance phase.

Maintaining a constructive mindset when dealing with others' feedback is crucial. Engaging in discussions with experienced developers during the ZBus submission and maintenance process proved to be a valuable opportunity for learning and gaining insights. The feedback from reviewers provided new perspectives and knowledge, which helped the ZBus maintainer to improve their understanding. By considering the reviewers' suggestions, the maintainer learned several new techniques and applied good practices to the subsystem code, resulting in a more robust and efficient solution.

Based on the feedback from the community, we have categorized some comments into a list of the most favorable and unfavorable words:

"I am a junior embedded software engineer. I've been developing embedded software with Zephyr RTOS for almost a year. Discovering the ZBUS and trying to understand it thoroughly has really changed my perspective on developing with a RTOS".

"Now zbus is not a proper data transfer mechanism. Calling it a bus is a misnomer and creates almost unlimited confusion in people."

Once words are spoken, they cannot be unsaid. Therefore, we must keep moving forward, believing in our efforts to improve the project for the entire community. It is essential to have the maturity to process user feedback and determine its value for the group, enabling us to consistently enhance our contributions. Remember, we have chosen to undertake this project, so we should continue working to improve it.

We would like to thank everyone from the Zephyr community, the Federal Universities of Alagoas and Pernambuco, Citrinio members, other partner companies, and the Edge Innovation Center for their contributions to ZBus and making it a reality.

4 Lessons learned

Information is moving faster with the globalization of the Internet. In the media, once spoken, the words can quickly become a new truth for many people. It is not that fast with science. Promising research results can take years to become utilized by the industry. Salatino *et al*. [2020] mentioned that academic publications precede industrial ones by 5.6 years. However, the approach applied in this work demonstrates that recent academic knowledge can quickly become part of the daily work of several companies. Applying scientific discoveries and study results to open-source projects can be an effective venue for sharing recent academic knowledge with the industry. In this case, throughout Zephyr RTOS, companies have the opportunity to use ZBus.

During the submission process of our ZBus, which involved an RFC and pull request phases, we encountered several communication challenges. We found that the community was more interested in the results of our contribution rather than its academic origins. As a result, we focused on discussing the features and benefits of our work rather than using academic terms or references. On the other hand, inter-process communication, software bus, and software engineer academic

[49] `https://github.com/zephyrproject-rtos/zephyr/pull/58101`

[50] `https://docs.zephyrproject.org/latest/project/project_roles.html#maintainer`

knowledge helped us to convince reviewers during discussions and demonstrate the benefits of the selected implementation approaches. Therefore, when researchers publish their work to the open-source community, adapting their communication strategy to be more pragmatic and practical than usual is essential.

The acceptance of a submission is just the beginning. After submission, maintaining the contribution requires a high effort to keep it attractive. We must engage with the community to understand their needs, update our contribution features, and address scenarios that have been hidden before. Doing that in parallel with the research is challenging. We must continue to do the academic work simultaneously with the project contribution. When noticed, we are more community members than researchers, and we need to return to the academy to research the new feature additions.

It is essential to be clear about the contribution's purpose and ensure that all scientific knowledge is available through comprehensive documentation. That is typically accomplished through an RFC or similar process. It is common for contributions to require better documentation, and we have ensured that the documentation for ZBus is as complete and detailed as possible. That included documentation artifacts, such as schemes and results in figures and tables, demonstrating how to apply academic knowledge in an industrial context. Shmerlin *et al*. [2015] have noted that developers often view documentation as tedious, complicated, and time-consuming.

One thing to avoid is bias. We must keep our contributions generic when contributing to a general-purpose open-source project, such as an RTOS. It is usual to see people trying to add features to the project that solve only their own problems. The community must be the more significant beneficiary in the game. People are good at pointing out scenarios. Listen to them and create a compilation, striving to remain impartial and objective.

Ultimately, being prepared is crucial. One must thoroughly understand the theory surrounding the idea. The people involved in the process have extensive experience and have seen many unconventional ideas. They will, at first, try to destroy the concept before contributing to it. If the proponent is firm and consistent, they will listen and consider the possibility. One of the most challenging aspects of the contribution was the Architecture Meeting Group, which took place twice, where we needed to support ZBus concepts against a skeptical and experienced audience.

4.1 Developer survey

To better understand the demographic distribution, popularity, and other related aspects of Zbus, we conducted a survey within the community. The survey[51] ran from July 7 to August 22, spanning a total of 47 days, and we received responses from 56 developers worldwide. One of the authors presented the findings of the survey at the Embedded Open Source Summit 2025[52], held in Amsterdam. The complete dataset collected from the survey, along with the script used to process the data, is available online in a GitHub repository[53].

Figure 9. What are the primary benefits you've experienced using Zbus?

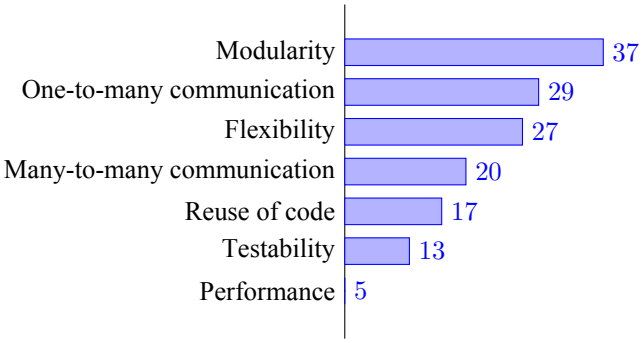
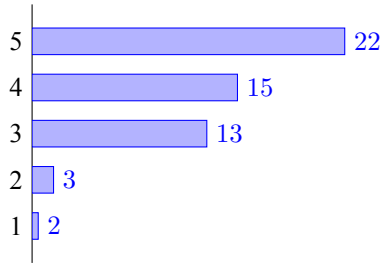


Figure 10. Considering its capabilities, how would you rate Zbus's overall contribution to your embedded software development process?

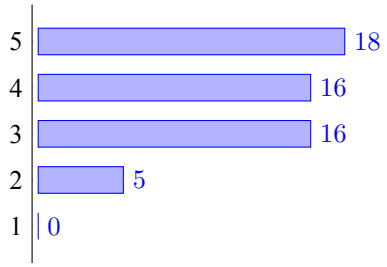


The majority of respondents (see Figure 9) consider ZBus as modular and flexible. They also highlighted that ZBus effectively supports both one-to-many and many-to-many communication topologies. In our survey, we categorized ratings of 4 and 5 as positive, a rating of 3 as neutral, and ratings of 1 and 2 as negative. According to Figure 10, 67.27% of respondents believe that ZBus contributes positively to their embedded system development process. Additionally, 61.82% of respondents view the ZBus documentation as positive (refer to Figure 10). Furthermore, 76.36% of respondents consider ZBus easy to use (see Figure 12). Ultimately, 75.86% of responses indicated that ZBus inspired them or their teams to have a positive transformation in their perspective on embedded system software architecture.

4.2 Limitations

We develop ZBus to solve open issues in the embedded system software field. No other RTOS we assess or are aware of has a bus as ZBus. That remains a unique piece of software that helps developers create better embedded software. However, as we focused on constrained devices during the design phase, the technology presents some limitations.

Figure 11. How would you rate the documentation for Zbus?



[51] https://docs.google.com/forms/d/e/1FAIpQLSe0epPjzBKbnWFX5OMhO1cM5I3zpCGPqiVpslb34hJsvi4ZzQ/viewform

[52] https://www.youtube.com/watch?v=WAosr1sRZgk

[53] https://github.com/rodrigopex/zbus_survey_2025

Figure 12. How would you rate the overall ease of use of Zbus?

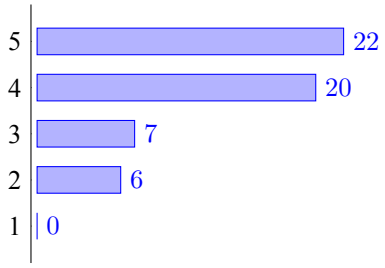
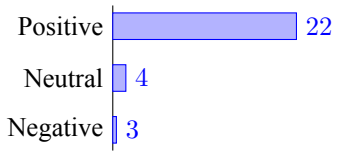


Figure 13. Did Zbus inspire you or your team to transform your perspective on embedded systems software architecture?



We created a listener to ensure the notification is sent as quickly as possible. This consists of a callback executed when someone publishes to a channel that the listener is observing. To make it possible, we followed persistent communication, which stores the message on the channel once it gets published. Persistent communication helped resolve the scenario, but it also increased memory usage, making the notification process less familiar and more challenging to understand.

The subscribers are asynchronous observers who exclusively receive the notification that the observed channel has changed, not the message itself. It caused considerable confusion among community members. Only some developers can use that properly. Another issue that subscribers raised was the loss or repetition of packages. The developers publish a message several times to a channel that a subscriber observes, and the subscriber receives just the latest message several times. That is not wrong. It is correct, but as they need to understand the concept deeply, they imply that the implementation is incorrect.

We utilize many macros in the ZBus implementation to simplify its operation, which can make it difficult to understand. Another point in the implementation is that the community has several needs, so we had to adapt the bus, which comes in various configurations and variations. Therefore, mastering it takes a long time. The documentation is lengthy and dense, with many details and aspects. Hence, one who wants to start using ZBus will find almost everything they need, but it will require attention and practice to work correctly.

The only maintainer is a limitation that can prevent the bus from growing faster. Nowadays, one of the authors is the sole ZBus maintainer, using his spare time to develop new features and fix bugs. Some developers started to help with simple things by doing small PR, but it was usually only superficial aspects of the code or the documentation.

4.3 Implications

The ZBus development, submission, improvement, and addition to the Zephyr codebase show us how fascinating open-source contributions associated with research can be. ZBus results from a mixing of practice and research. Its initial stage of development takes place in the academy. It receives several inputs from the industry through a complex and demanding submission process for the extraordinary, worldwide, and well-adopted RTOS project, Zephyr. At the end of the process, ZBus is available to many people and companies as a powerful development tool. The history of ZBus indicates a shortened path from the academy to the industry, allowing someone to contribute using science directly to the industry.

There are some open gaps in ZBus functionality regarding topics we consider when ZBus runs in the real world. In this manner, next, we will discuss some available directions where the academy can help the industry again.

Bus synchronization is a problematic factor of ZBus when considering multi-core (Asymmetric Multi-processing or Symmetric Multi-processing) and multi-target (embedded and Desktop) scenarios. When you have two or more cores/-targets, they all have a ZBus clone. The following questions emerge: i) How do I synchronize more than one ZBus instance, keeping consistency and performance? ii) What are the steps to take in the synchronization process? iii) Do we let the event dispatcher deliver all the notifications on the originating core before synchronizing or after? iv) What are the possible consequences of each choice? v) What is the most suitable format/protocol to exchange binary data in these scenarios? vi) Can the techniques applied to multi-core scenarios solve multi-target, even remote ones, where the targets are on different machines using the network to exchange messages? vii) Can the same techniques be used on low-speed wireless networks like Bluetooth Mesh?

Those questions can serve as a roadmap for advancing research on ZBus, allowing it to operate effectively in increasingly complex scenarios.

5 Related work

This work contributes twofold: First, we create a software bus that facilitates a many-to-many communication topology between threads, promoting modularity and code decoupling in the embedded systems RTOS scenario. Second, we transfer knowledge from academia to the industry through an OSS contribution.

5.1 Software Bus Architectures and IPC Mechanisms

The concept of software buses for inter-process communication has been established in desktop and server environments, with D-Bus Love [2005] representing the most prominent example. Love [2005] described D-Bus as middleware providing IPC to meet the needs of the Linux desktop environment, successfully replacing CORBA as GNOME's remote object system. D-Bus demonstrates the viability of software bus abstractions for decoupling communicating components through message-based interactions. Its architecture provides mechanisms for message routing, service discovery, and event notification that enable loosely coupled system designs.

However, D-Bus's design reflects the constraints and capabilities of desktop Linux systems rather than resource-constrained embedded environments. Its reliance on dynamic memory allocation, dependency on high-level operating system features (such as Unix domain sockets), and relatively

large memory footprint make it unsuitable for deployment in typical RTOS environments where memory is measured in kilobytes rather than megabytes. ZBus adopts the conceptual model of software buses from D-Bus—centralized message routing, publish-subscribe communication, and observer patterns—while fundamentally redesigning the implementation for embedded constraints: static memory allocation, minimal dependencies, integration with priority-based scheduling, and deterministic real-time behavior.

While alternative IPC approaches exist in the literature, they address distinct problem domains or impose architectural constraints that render them unsuitable for general-purpose embedded systems. The IPC mechanism described by Marzi *et al*. [2009] uses message-passing designed for specific chip architectures, creating strong hardware dependencies that limit portability. Rajkumar *et al*. [1995] presented an IPC model for distributed real-time systems using C++ and dynamic registration, targeting systems with 32MB RAM running LynxOS with UDP/IP networking—resource levels an order of magnitude beyond typical embedded constraints. Schoettler [2017] proposed a publish-subscribe framework for embedded systems implemented on FreeRTOS; however, specific architectural decisions—including a centralized high-priority broker that introduces priority inversion risks and a constraint of thirty-two observers per channel—limit its applicability to more complex system configurations. Despite being publicly available for almost a decade, our review of the literature revealed no documented evidence of substantial industrial adoption of this framework.

These limitations across existing solutions motivated the development of ZBus as a software bus specifically optimized for resource-constrained RTOS environments, balancing the architectural benefits of software buses with the practical constraints of embedded systems.

5.2 Publish-Subscribe and Event-Driven Patterns

The publish-subscribe interaction paradigm provides the foundational communication model for ZBus. Eugster *et al*. [2003] comprehensively investigated publish-subscribe variants, identifying their commonalities and divergences. Key characteristics include the decoupling of communicating entities along three dimensions: time (publishers and subscribers need not be active simultaneously), space (entities need not know each other's identities or locations), and synchronization (operations need not block while waiting for coordination). These properties directly address modularity challenges in embedded software by enabling components to interact without tight coupling.

ZBus implements publish-subscribe specifically for RTOS thread communication, adapting the paradigm to the constraints of embedded systems. Unlike enterprise publish-subscribe systems that often rely on dynamic subscription management and complex routing algorithms, ZBus emphasizes static configuration for predictability and minimal overhead. The channel abstraction in ZBus serves as a publish-subscribe topic, with a compile-time declaration that enables static memory allocation and eliminates runtime subscription overhead. However, ZBus retains flexibility through runtime observer masking and dynamic observer registration when needed.

The integration of event-driven architecture principles complements the publish-subscribe model. Active databases Paton and Díaz [1999], which trigger autonomous actions rather than requiring explicit polling, directly inspired ZBus's event-driven notification mechanism. Property systems Fowler [1997], which provide observable state with automatic change notification, influenced the centralized state management approach. The evolution of ZBus from a simple property system through event-driven enhancements to a full software bus demonstrates progressive application of these patterns to embedded system constraints.

5.3 Embedded System IPC Solutions

Within the Zephyr RTOS ecosystem and the broader embedded systems domain, various IPC mechanisms exist, though none specifically address the need for lightweight many-to-many thread communication via a software bus abstraction. Standard RTOS primitives—message queues, mailboxes, semaphores, and mutexes—provide point-to-point or one-to-one synchronization but require custom implementation patterns for many-to-many scenarios. This gap led to repeated reinvention of communication mechanisms across projects.

Alternative solutions targeting similar problems exist, but with different trade-offs. The Nordic Event Manager, proposed for integration into Zephyr, shares the goal of creating an event-based communication mechanism but differs significantly in design philosophy and feature set. As detailed in Section 3.2, our experimental comparison demonstrated ZBus advantages in latency, memory efficiency, and architectural flexibility. The Laird messaging framework provides another alternative with dynamic memory allocation and message-passing semantics, but targets different use cases than ZBus's focus on static configuration and shared-memory efficiency.

The absence of a standardized, lightweight software bus for general-purpose RTOS environments motivated ZBus development. While domain-specific frameworks exist—such as AUTOSAR in automotive or DDS in distributed systems—these typically impose significant overhead or architectural constraints unsuitable for resource-constrained devices. ZBus fills this gap as a general-purpose solution optimized for the constraints typical across embedded systems domains.

5.4 Convergent Evolution with Hardware Communication Buses

An interesting observation during the development and community review of ZBus is its structural similarity to classical hardware communication buses widely used in embedded systems, particularly in automotive and industrial domains. CAN (Controller Area Network) International Organization for Standardization (ISO) [2016], FlexRay FlexRay Consortium [2010], and LIN (Local Interconnect Network) LIN Consortium [2010] represent mature, proven technologies for reliable communication in resource-constrained, real-time environments.

These similarities emerged through convergent evolution rather than intentional emulation. Both hardware buses and

software buses operating in embedded environments face analogous fundamental challenges: priority-based arbitration of shared resources under real-time constraints, deterministic and reliable message delivery, efficient broadcast communication to multiple recipients, and operation within severe memory and processing limitations. The ZBus priority-based notification delivery, which executes notifications in the publishing thread's context, parallels CAN's priority-based arbitration. The deterministic behavior of the Virtual Distributed Event Dispatcher provides predictable ordering similar to FlexRay's time-triggered model. The channel-based broadcast resembles CAN's identifier-based messaging.

This convergence validates the ZBus design: independent arrival at architectural patterns proven effective in hardware domains over decades suggests fundamental soundness for the problem space. However, the convergence reflects common constraints rather than direct inspiration—ZBus remains distinctly a software solution optimized for RTOS threading models, compile-time configuration, and software abstraction levels unavailable in hardware implementations. As detailed in Section 2.1, ZBus development followed from software engineering principles rather than hardware bus emulation.

5.5 Software Quality and Modularity

The quality characteristics of ZBus align with established software quality models. The ISO/IEC 25010:2023 standard ISO [2023] defines a comprehensive quality model for software products, identifying nine product quality characteristics: functional suitability, performance efficiency, compatibility, interaction capability, reliability, security, maintainability, flexibility, and safety. The ZBus design particularly emphasizes maintainability (through modularity and reusability) and performance efficiency (through deterministic behavior and resource optimization). These quality characteristics are detailed in Section 2 in the context of resource-constrained embedded systems.

Earlier work by McCall *et al*. [1977] identified three software product activities: operation, revision, and transition. The operation activity is related to factors such as correctness, reliability, efficiency, integrity, and usability. On the other hand, the revision activity is related to maintainability, flexibility, and testability, while the transition activity is related to portability, reusability, and interoperability. Our focus is on the last two activities, which can be improved by implementing disciplined teamwork to reduce a project's technical debt.

Lee [2014] mentioned modularity is one of the most affecting criteria for software quality. Reinforced by Parnas [1972], which stated developers can use modularization to enhance the flexibility and comprehensibility of a system while shortening its development time. This directly affects maintainability, flexibility, testability, portability, reusability, and interoperability. Therefore, these principles drove our efforts to develop a tool that enables embedded software modularization.

Ampatzoglou *et al*. [2016] described an industrial case study related to the perception of Technical Debt (TD) in the embedded systems domain. To achieve the results, they conducted multiple case studies across projects from five countries, spanning short- to long-term lifecycles and small- to large-scale industries. The findings indicate that one of the most recurring TD is architectural, which ZBus aims to address. Ampatzoglou *et al*. [2016] also stated that the main reason for the TDs is the short time-to-market required for the business. ZBus is a suitable architectural choice for reducing development time and decreasing maintenance costs.

5.6 University-Industry Knowledge Transfer and Open-Source Contribution

The second contribution of this work addresses knowledge transfer from academia to industry through open-source contributions. Daniel and Alves [2020] highlighted university-industry collaboration as a fruitful arrangement, bringing universities' knowledge about practical problems while providing industry access to research advances and helping students prepare for industrial work. ZBus, created within the university context and refined through practical projects, demonstrates this bidirectional knowledge flow. Its integration into Zephyr RTOS makes academic innovations directly accessible to the embedded systems industry.

Rossoni *et al*. [2023] identified cultural differences between university and industry as critical barriers to technology transfer, with secrecy being a common challenge. Open-source contributions circumvent this barrier—everyone contributes to and uses the same codebase. During ZBus submission, cultural differences manifested through community requests for broader problem-solving approaches and code standardization, ultimately improving the contribution's quality and generality.

The contribution process itself presented typical open-source challenges. Fronchetti *et al*. [2023] noted that CONTRIBUTING documentation often neglects barriers for newcomers, with communication gaps in 75% of projects. Zephyr proved exceptional in this regard, providing comprehensive documentation and accessible community communication through Discord. However, as a mature project, the submission of an entirely new subsystem faced high standards and numerous change requests. We experienced all Modern Code Review (MCR) stages identified by Davila and Nunes [2021]: preparation, reviewer selection, notification, code checking, reviewer interaction, and review decisions. The rigorous process, including strict CI requirements described by Wessel *et al*. [2023], resulted in a more generic ZBus better aligned with community standards.

6 Conclusion

This paper introduced ZBus, an open-source software bus designed to facilitate many-to-many inter-thread communication, thereby significantly enhancing modularity and architectural flexibility in embedded systems. Developed at the Edge Innovation Center of the Federal University of Alagoas (UFAL) in Brazil, ZBus addresses critical challenges in embedded software engineering, such as tight code coupling and resource-constrained synchronization.

We detailed the process and obstacles encountered while integrating ZBus into the widely adopted Zephyr real-time

operating system project. The case of ZBus demonstrated that open-source software can benefit from diverse contributors, including academia, providing robust solutions for the embedded systems community.

The successful deployment and acceptance of ZBus among organizations and individual developers highlight its potential impact in the field. We emphasize the significance of knowledge transfer between academia and industry, as well as the importance of open-source communities in advancing the field of embedded software engineering.

ZBus is a valuable software artifact for collaborative development and open-source contributions, offering a helpful tool to the embedded software community.

ZBus is an academically-grounded open-source software bus designed to facilitate modularity and collaborative development within the embedded systems domain. By bridging the gap between theoretical pub-sub models and practical resource-constrained implementations, this work provides a scalable foundation for both open-source evolution and standardized industrial applications. Our experiences and lessons can inspire future research and development efforts in this vital domain.

Given the open opportunities related to ZBus, identifying a suitable synchronization solution for ZBus would represent a significant advancement for the bus system. Exploring embedded software architecture approaches, such as event-driven architectures, creating communication abstractions, and testing techniques that rely on channels to assess module functionality, can offer valuable new solutions to the community.

6.1 Future work

There are several open challenges in the embedded systems field related to ZBus. One of the most pressing issues is the asynchronous multiprocessor scenario, which can be divided into two configurations: (i) a homogeneous environment, where multiple real-time operating system (RTOS) applications run on the same microcontroller (MCU) but on different cores; and (ii) a heterogeneous environment, where two distinct operating systems run on separate cores (possibly with different architectures) and interact with each other, such as Linux on a CPU and Zephyr on an MCU within the same system-on-chip (SoC).

In scenario (i), although the environment uses the same real-time operating system (RTOS), achieving efficiency, consistency, and security remains complex. Scenario (ii), which involves integrating different technologies, introduces even more challenges. We plan to first address the issues in scenario (i) before moving on to scenario (ii). As ZBus achieves broader international adoption, we aim to foster strategic partnerships with academic institutions and industry leaders to catalyze further innovation. A notable current development is the ongoing collaboration with Nordic Semiconductor, which is actively addressing a PR to implement scenarios (i) and (ii) via an open pull request[54] to the Zephyr RTOS repository.

Declarations

Authors' Contributions

Rodrigo is the main contributor and writer of this manuscript, responsible for the conceptualization, methodology, software implementation, and original draft. Leopoldo and Baldoino contributed to the conceptualization and methodology, supervised the research, validated the results, and were actively involved in reviewing and editing the manuscript. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Funding

This work was partially supported by CNPq and INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) `www.ines.org.br`, CNPq grant 408817/2024-0. Leopoldo is partially supported by CNPq grant 310405/2025-4. Baldoino is partially supported by CNPq grant 309227/2025-9.

Availability of data and materials

The datasets (and/or software) generated and/or analyzed during the current study are available in the Zephyr RTOS GitHub repository[55].

References

(2023). ISO/IEC 25010:2023. Available at:`www.iso.org`.

Akyildiz, I., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002). Wireless sensor networks: a survey. *Computer Networks*, 38(4):393 – 422. DOI: 10.1016/S1389-1286(01)00302-4.

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Abrahamsson, P., Martini, A., Zdun, U., and Systa, K. (2016). The perception of technical debt in the embedded systems domain: An industrial case study. In *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, pages 9–16. DOI: 10.1109/MTD.2016.8.

Aniche, M. (2022). *Effective Software Testing: A developer's guide*. Simon and Schuster. Book.

Antonino, P. O., Morgenstern, A., and Kuhn, T. (2016). Embedded-software architects: It's not only about the software. *IEEE Software*, 33(6):56–62. DOI: 10.1109/MS.2016.142.

AUTOSAR (2025). Automotive open system architecture. Available at:`https://www.autosar.org/`. Accessed: February 17, 2025.

Ciolkowski, M., Lenarduzzi, V., and Martini, A. (2021). 10 years of technical debt research and practice: Past, present, and future. *IEEE Software*, 38(6):24–29. DOI: 10.1109/MS.2021.3105625.

[54] https://github.com/zephyrproject-rtos/zephyr/pull/96086

[55] `https://github.com/zephyrproject-rtos/zephyr`

Daniel, A. D. and Alves, L. (2020). University-industry technology transfer: the commercialization of university's patents. *Knowledge Management Research & Practice*, 18(3):276–296. DOI: 10.1080/14778238.2019.1638741.

Davila, N. and Nunes, I. (2021). A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software*, 177:110951. DOI: 10.1016/j.jss.2021.110951.

EDGE (2025). Centro de inovação edge. Available at:`https://www.edge.ufal.br/`. Accessed: February 17, 2025.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/-subscribe. *ACM Comput. Surv.*, 35(2):114–131. DOI: 10.1145/857076.857078.

Fariha, A., Alwidian, S., and Azim, A. (2024). A systematic literature review on requirements engineering and maintenance for embedded software. *IEEE Access*, 12:114263–114279. DOI: 10.1109/ACCESS.2024.3443271.

FlexRay Consortium (2010). Protocol Specification, Version 3.0.1. October 2010. Available at:`http://www.flexray.com`.

Fowler, M. (1997). Dealing with properties. Available at:`https://martinfowler.com/apsupp/properties.pdf`.

Freeh, V. W., Xu, J., and Lowenthal, D. K. (2003). Hybrid messaging passing in shared-memory clusters. Technical report, Technical report, Department of Computer Science, University of Georgia. Available at:`https://www2.cs.arizona.edu/~dkl/Publications/Papers/hybrid.pdf`.

Fronchetti, F., Shepherd, D. C., Wiese, I., Treude, C., Gerosa, M. A., and Steinmacher, I. (2023). Do contributing files provide information about oss newcomers' onboarding barriers? In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 16–28, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3611643.3616288.

International Organization for Standardization (ISO) (2016). Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling in general applications. Standard ISO 11898-1:2016, ISO. Available at:`www.iso.org`.

Jacobsen, H.-A. (2009). *Channel-Based Publish/Subscribe*, pages 321–323. Springer US, Boston, MA. DOI: 10.1007/978-0-387-39940-$9_1$207.

Lee, M.-C. (2014). Software quality factors and software quality metrics to enhance software quality assurance. *British Journal of Applied Science & Technology*, 4(21):3069–3095. DOI: 10.9734/BJAST/2014/10548.

LIN Consortium (2010). LIN Specification Package Revision 2.2A. Technical Report Revision 2.2A, LIN Consortium. Available at:`www.linconsortium.org`.

Love, R. (2005). Get on the d-bus. *Linux J.*, 2005(130):3. Available at:`https://www.linuxjournal.com/article/7744`.

Marzi, H., Hughes, L., and Yanting Lin (2009). Embedded systems with improved interprocess communication design. In *2009 7th IEEE International Conference on Industrial Informatics*, pages 200–203. DOI: 10.1109/INDIN.2009.5195803.

McCall, J., Richards, P., and Walters, G. (1977). *Factors in Software Quality. Volume I: Concepts and Definitions of Software Quality*. AD A049. General Electric. Available at:`https://books.google.com.br/books?id=vtrBSgAACAAJ`.

Miorandi, D., Sicari, S., Pellegrini], F. D., and Chlamtac, I. (2012). Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497 – 1516. DOI: 10.1016/j.adhoc.2012.02.016.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058. DOI: 10.1145/361598.361623.

Paton, N. W. and Díaz, O. (1999). Active database systems. *ACM Comput. Surv.*, 31(1):63–103. DOI: 10.1145/311531.311623.

Rajkumar, R., Gagliardi, M., and Sha, L. (1995). The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation. In *Proceedings Real-Time Technology and Applications Symposium*, pages 66–75. DOI: 10.1109/RTTAS.1995.516203.

ROS (2025). Robot operating system. Available at:`https://www.ros.org/`. Accessed: February 17, 2025.

Rossoni, A. L., de Vasconcellos, E. P. G., and de Castilho Rossoni, R. L. (2023). Barriers and facilitators of university-industry collaboration for research, development and innovation: a systematic review. *Management Review Quarterly*, pages 1 – 37. DOI: 10.1007/s11301-023-00349-1.

Salatino, A., Osborne, F., and Motta, E. (2020). Researchflow: Understanding the knowledge flow between academia and industry. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 219–236. Springer. DOI: 10.1007/978-3-030-61244-3_16.

Schoettler, M. R. (2017). A publish-subscribe framework for embedded systems: simplifying the development process. Master's thesis, University of California, Irvine. Available at:`https://www.proquest.com/openview/162fc9c230544ca80c9c90103c05a841/1?pq-origsite=gscholar&cbl=18750`.

Shmerlin, Y., Hadar, I., Kliger, D., and Makabee, H. (2015). To document or not to document? an exploratory study on developers' motivation to document code. In *Advanced Information Systems Engineering Workshops: CAiSE 2015 International Workshops, Stockholm, Sweden, June 8-9, 2015, Proceedings 27*, pages 100–106. Springer. DOI: 10.1007/978-3-319-19243-7_10.

THORNTON, S. (2018). Ee world: An open source rtos for iot. Available at: `https://www.zephyrproject.org/ee-world-an-open-source-rtos-for-iot/`. Accessed: February 17, 2025.

Venkataraman, A. and Jagadeesha, K. K. (2015). Evaluation of inter-process communication mechanisms. Available at:`https://api.semanticscholar.org/CorpusID:6899525`.

Wan, J., Yan, H., Suo, H., and Li, F. (2011). Advances in

cyber-physical systems research. *TIIS*, 5:1891–1908. DOI: 10.3837/tiis.2011.11.001.

Wessel, M., Vargovich, J., Gerosa, M. A., and Treude, C. (2023). Github actions: the impact on the pull request process. *Empirical Software Engineering*, 28(6):131. DOI: 10.1007/s10664-023-10369-w.

Zephyr-RTOS (2026). Zephyr project members page. Available at: `https://www.zephyrproject.org/project-members/`. Accessed: January 14, 2026.