QUADUINO - AN ARDUINO-BASED EDUCATIONAL QUADRUPLEX-COMPUTER-SYSTEM

Bastian Luettig, Constantin Frey, Johannes Reinhart, Michel Bunza, Jan Freyhardt, Jiajun Li, Benedict Roth, Johannes Wuebbeling, Bjoern Annighoefer University of Stuttgart, Institute of Aircraft Systems, Pfaffenwaldring 27, Stuttgart, GERMANY

Abstract

Redundant avionics computers are essential for aircraft safety, but practical education on their behavior is limited. We introduce Quaduino, an educational tool developed at the Institute of Aircraft Systems of the University of Stuttgart. Quaduino comprises four Arduino ATmega2560 microcontrollers, forming a redundant computer, and additional instrumentation for in-depth assessment. Our software ensures separation of platform management and control function, leveraging Arduino's ecosystem. A web interface enables real-time data observation, enhancing students' understanding of redundant systems. Quaduino demonstrates challenges such as asynchrony, faults, and communication. This tool empowers students to grasp redundant avionics principles and was successfully introduced into a laboratory course.

Keywords

Redundant avionics; safety-critical systems; Quaduino; Arduino ATmega2560; educational tool; redundant computer; instrumentation; prototyping; synchronization; teaching device

1. MOTIVATION

Avionics safety is key to ensure public trust in commercial air transportation. During the previous decades, aircraft became more electric and more digital than ever before [1]. Current research projects advocate for a further increase in these trends: intelligent wings [2], adaptive cabins [3], machine learning and AI for image recognition, as well as more digital operations. Proven tools, books, and lectures exist for the development of the non-redundant function. For the redundant implementation, those books and lectures are rare and tools are basically non-existent.

The Institute of Aircraft Systems / University of Stuttgart dedicates the research and courses towards the challenges of redundant implementation. Two common pure redundancy strategies are featured within separate lectures and laboratory courses: all-active and active-standby. For this research, we focus on an all-active strategy with its most prominent implementations: duplex, triplex and quadruplex architectures, as shown in Fig 1.

In the lecture, we intend to show all theoretical effects within such a redundant system to the students. During lab courses, students should work on the systems themselves, develop different redundancy mechanisms and observe how the actual hardware reacts to the implementation. This includes the correct redundant operation, consistent failures in the computer system and even byzantine failures. Hardware similar to avionics is hard to buy and actual commercial redundant avionics hardware is basically impossible to obtain. Hence we developed our own equipment. We chose to build a new laboratory based on commercial-off-the-shelf components and open source all materials.

Within this paper, you will find the fundamentals for operation in all-active systems, the Quaduino concept in terms of teaching, hardware and software. Then we will introduce the actual implementation, our current verification state and conclude the paper with summary and outlook.

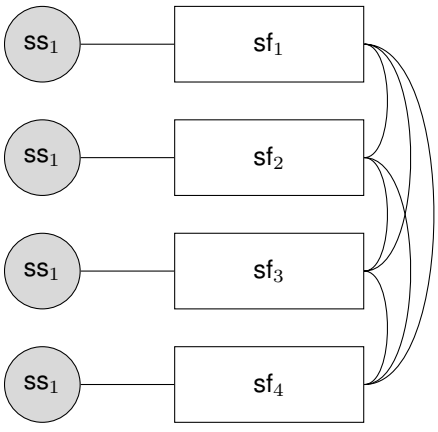

FIG 1. Diagram of the four computing lanes (sf) with each one single sensor (ss) and cross-lane-connections. The four lanes exhibit output consensus.

2. FUNDAMENTALS

In an all-active strategy, multiple single computer lanes operate together and form a redundant computer

doi: 10.25967/610350

system. Each computing lane computes the same functions and the lanes compare each others outputs. All computer lanes contribute to both, integrity and reliability of the redundant computer, i.e., if the computers detect a discrepancy from one computing lane, they will passivate the faulty lane (integrity). If a lane fails, the others still perform the function (reliability). To ensure the integrity comparison works correctly, i.e., not too early and not too late, the system designer must implement a mechanism to detect erroneous outputs reliably during the mission. There are two options: (a) use wide monitoring limits or (b) enforce computer-replica-determinism.
Most commercial applications implement option (a), as this allows to leave the redundant lanes asynchronous and improve independence - at the cost of false-positives [4].
For option (b), we need to ensure computer-replica-determinism, which strictly speaking means, that if you input the same values ($\underline{x}_{\text{in}}$) in the same order to redundant units, all redundant units will produce the same commands ($\underline{x}_{\text{out}}$). We define: Given vectors with signals in all redundant units, they show exact agreement ($A_=$), if they are identical:

$$\forall i, j : \underline{x}_{\text{i}} = \underline{x}_{\text{j}}$$

We shorten this to:

$$\underline{x}\big|_{A_=}$$

(the lanes show exact agreement for this vector).
Strict computer-replica-determinism will be defined as:

$$\underline{x}_{\text{in}}\big|_{A_=} \Rightarrow \underline{x}_{\text{out}}\big|_{A_=} \tag{1}$$

For two computers, this is an option, but for more redundant computer lanes, we cannot guarantee this anymore with reasonable effort and modify in a way that similar values (A_Δ) lead to similar outputs. First, we need to separate analog and discrete values. Analog values are signals represented by larger data types, discrete values are smaller data types, e.g. boolean or short integers. With this change, equation 1 becomes:

$$\begin{pmatrix} \underline{x}_{\text{in,analog}} \\ \underline{x}_{\text{in,discrete}} \end{pmatrix}\Bigg|_{A_=} \Rightarrow \begin{pmatrix} \underline{x}_{\text{out,analog}} \\ \underline{x}_{\text{out,discrete}} \end{pmatrix}\Bigg|_{A_=} \tag{2}$$

We define delta-agreement A_Δ as the analogue values may differ slightly:

$$\forall i, j : \underline{x}_{\text{analog,i}} \approx \underline{x}_{\text{analog,j}}$$

We shorten this to:

$$\underline{x}_{\text{analog}}\big|_{A_\Delta}$$

The less strict computer-replica-determinism shows as follows

$$\begin{pmatrix} \underline{x}_{\text{in,analog}}\big|_{A_\Delta} \\ \underline{x}_{\text{in,discrete}}\big|_{A_=} \end{pmatrix} \Rightarrow \begin{pmatrix} \underline{x}_{\text{out,analog}}\big|_{A_\Delta} \\ \underline{x}_{\text{out,discrete}}\big|_{A_=} \end{pmatrix} \tag{3}$$

We separated this into analog and discrete values, because we can implement communication protocols that ensure reliable broadcast [5] properties for discrete values. For analog values this could overload the underlying bus system. For those (in terms of bits) larger values, we implement a simple broadcast. This will ensure A_Δ for any single failure within a quadruplex system. For each scenario which does not exhibit an asymmetric communication fault, those values fulfill $A_=$.
To fulfill the above equation, the system needs to ensure consensus. Which means, it needs to fulfill consensus conditions: (1) synchrony, (2) agreement, and (3) integrity [6, 7]. Which translates to: the single computers have to synchronize as close as possible, acquire and produce similar values, and produce *correct* values.
Despite not being favored in its pure form anymore by aircraft manufacturers, the basic concepts are still being used as part of active-standby configurations [8] [9].

3. CONCEPT

The Quaduino is designed with a distinct pedagogical purpose in mind, leading to a set of requirements that diverge significantly from those associated with conventional avionics hardware.

3.1. Teaching Goals

When implementing consensus in such a redundant system, the engineer faces typical challenges. The Quaduino concept aims at increasing the visibility of those challenges and possible solutions. The corresponding lecture teaches the general requirements and concepts to implement consensus, failure handling, scheduling, and operation.
With the Quaduino, the students will:
1) observe typical challenges in redundant operation;
2) implement concepts to solve the challenges;
3) observe the redundant system handling these challenges;
4) gain a deeper knowledge and experience with redundant systems;
5) experience the effect of failures.

Furthermore, we will utilize the concept to teach single-computer operation.
Therefore, the students also will:
1) learn how to program an embedded systems;
2) learn to work with hardware debugging tools to observe memory and registers;
3) learn to operate embedded systems;
4) see how computers store program and data in memory.
5) understand basic machine instructions and learn how function calls are realized

3.2. Hardware Considerations

We want to decrease the distance between students and avionics hardware, therefore we selected widely available commercial-off-the-shelf components. A student should have the opportunity of building an own Quaduino. Hence we want to achieve a price of 200€ per build.
We considered multiple ecosystems including Raspberry Pi, STM32, Arduino, ESP8266/32. In order to show and implement a redundant computer, the ecosystem should fulfill these requirements:
The computers and ecosystems for the single lanes
1) can communicate on dedicated channels;
2) have Ethernet support;
3) each supports at least four avionics-typical sensors, e.g., distance, acceleration, stick-input;
4) each supports at least two servo motors;
5) support low-level languages like C, C++, or Rust;
6) are as simple as possible;
7) are readily available for less than 50€.

Next to hardware, basic software is essential.

3.3. Software Considerations

The Arduino ecosystem primarily relies on the Arduino IDE and utilizes C/C++ programming languages.
To bridge the gap between students and software, our aim is to provide users with maximum access to every facet of the system. Unlike a typical avionics system, which operates on a cyclic schedule with each cycle lasting approximately $10\,\mathrm{ms}$, our approach prioritizes the need for students to observe changes within a redundant computer system. Consequently, we deliberately introduce measures to slow down the system and allow to opt for slower microcontrollers.
Key software requirements for our system encompass:
1) Implementation in a low-level programming language.
2) Adherence to a cyclic scheduling approach.
3) Adoption of a software architecture suitable for educational purposes.
4) Provision of the capability to analyze inner variables in real-time during system operation.

4. DEMONSTRATOR IMPLEMENTATION

In the process of implementing our demonstrators, we followed a systematic approach. Initially, we developed the single-lane architecture, subsequently integrating these components into a redundant computer system, and ultimately, we implemented fundamental software functionalities.

4.1. Single Computer Architecture

The choice to opt for the single computer architecture stemmed from several factors. Firstly, the Raspberry Pi, while initially considered, was ruled out due to a prevailing shortage and its inherent complexity and cost.
On the other hand, the STM32, although relatively new to the market, presented itself as a viable option.
Meanwhile, the ESP8266 and ESP32 exhibited intriguing features, including WiFi connectivity and competitive pricing. However, they lacked the required dedicated cross-lane communication capabilities, which were pivotal for our project.
Within the Arduino ecosystem, the Arduino Mega2560 emerged as the most suitable choice, offering an ATmega2560 microcontroller with a clock speed of 16 MHz [10], $256\,\mathrm{kB}$ of flash memory, and $8\,\mathrm{kB}$ of RAM. To enhance its capabilities, we incorporated a W5500 Ethernet module [11] for Ethernet support, and four RS485 modules [12] to facilitate dedicated cross-lane communication. Fig. 2 shows the single computer architecture with the designated pin assignment.

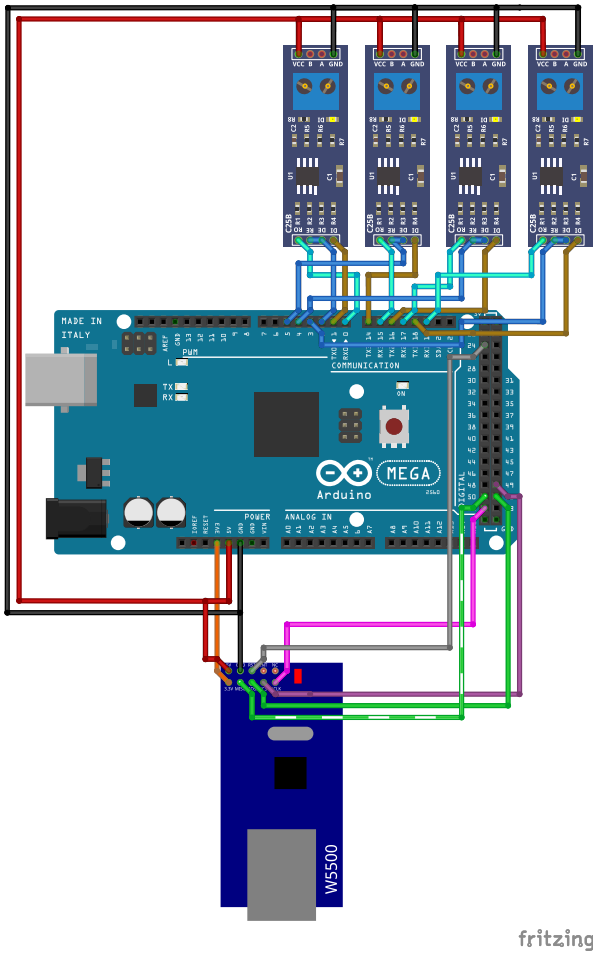

FIG 2. Single computer lane, consists of one Arduino Mega 2560, four RS485 MAX modules, and one W5500 Ethernet module

4.2. Redundant Computer Architecture

Matching the educational concept, we combined four single computers to one redundant computer. To achieve consensus across the computers, they need communication channels. The Arduino Mega 2560R3 features four serial interfaces. In order to limit the direct connection across computers, we implemented the cross communication using RS485 serial interfaces. The RS485 busses that run from the sending lane to

each other lane for receiving. The modules must be configured to either send or receive. This is done via the RE/DE pins, if both are set HIGH, the module sends. If both are LOW, the module can receive data. Each RS485 module connects to one of the Serial interfaces 1-3, the sender module to Serial0.
To ensure correct failure indication, each computer lane must know which neighbor sends on which interface. Hence we implemented pin programming that tells the program what their individual ID is.
Given all these components, we came to around 250€ per redundant computer. With clone boards, the target price can easily be achieved.

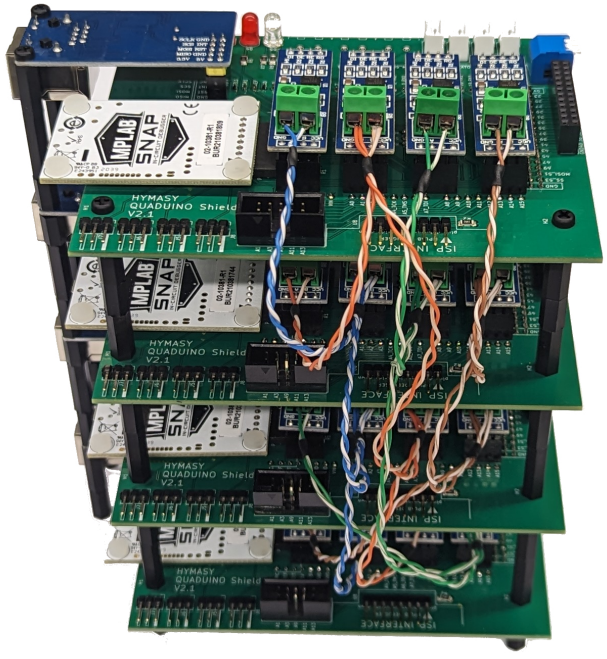

FIG 3. Redundant computer, consists 4 single computers stacked on-top each other, includes MPLAB SNAP debugger

We prototyped the hardware and then needed the basic software.

4.3. Software Architecture

The initial proof of concept showcases the successful implementation of cross-lane synchronization, a critical feature enabling synchronous execution within the redundant computer system.
At its core, the onboard software comprises several components, each playing a pivotal role in ensuring the reliability and functionality of the system. Fig. 4 depicts the Quaduino software architecture, borrows [13] and implements the following services:
Pin Programming: To ensure, that each lane knows its own ID, we implemented a service that reads 2 discrete pins to compute an ID. Additionally, we implemented a function that maps 0-2 to the actual neighbor lane ID.
Synchronization Service: The synchronization service implements an event-synchronization mechanism that allows precise coordination among the four redundant lanes. It ensures that all lanes execute their tasks synchronously. This service achieves synchronization by transmitting synchronization messages and resetting the internal clock when a common event is reached. It is part of the platform management (plama) for the redundant computer (core).
Timing Service: To guarantee timely execution and provide support for synchronization, we introduced a layer that translates the hardware clock within each Arduino into the logical time of the redundant computer. This service ensures that all actions occur in accordance with the synchronized clock.
Scheduler/Dispatcher: The scheduler or dispatcher constitutes the backbone of the software. It orchestrates the execution, guaranteeing that the computer performs services in the correct order and at the precise time. This coordination is essential for the system's correct operation.
Database: An array of database entries facilitates seamless communication between services and permits the exchange of critical debugging data with the laboratory computer. This database acts as a central hub for data management.
RS485 Communication Service: To set the modules to be sender or receiver, our software driver needs to set the RE/DE pins accordingly. Furthermore, the service needs to implement a message protocol that distinguishes among the different types, adds a message buffer and assigns the correct lane IDs to the received messages.
Ethernet Communication Service: To streamline communication with the laboratory computer, the system integrates a specialized Ethernet communication service. This service ensures efficient data transfer between the Quaduino and the laboratory computer, enabling real-time monitoring and control, which are essential for educational purposes. The eventual concept is similar to the SPY function [14].
Collectively, these software components collaborate to construct a user-friendly and highly functional redundant computer system. Not only do they demonstrate the successful implementation of cross-lane synchronization, but they also provide a solid foundation for the Quaduino project's future extensions and enhancements. This software architecture marks a significant milestone in the development of our educational system.

4.4. Development Environment

In our search for the ideal development environment for the Quaduino project, we evaluated the tools available within the Arduino ecosystem. While the Arduino Integrated Development Environment (IDE) had proven its accessibility and suitability in initial prototyping and smaller-scale ventures, we encountered constraints as our project's complexity expanded, particularly when handling multiple files and modules within a project.
In response to these limitations and to facilitate the project's scalability, we transitioned to PlatformIO and Cmake. This shift proved instrumental, as PlatformIO offered us a robust and adaptable framework that seamlessly integrates Arduino-compatible libraries

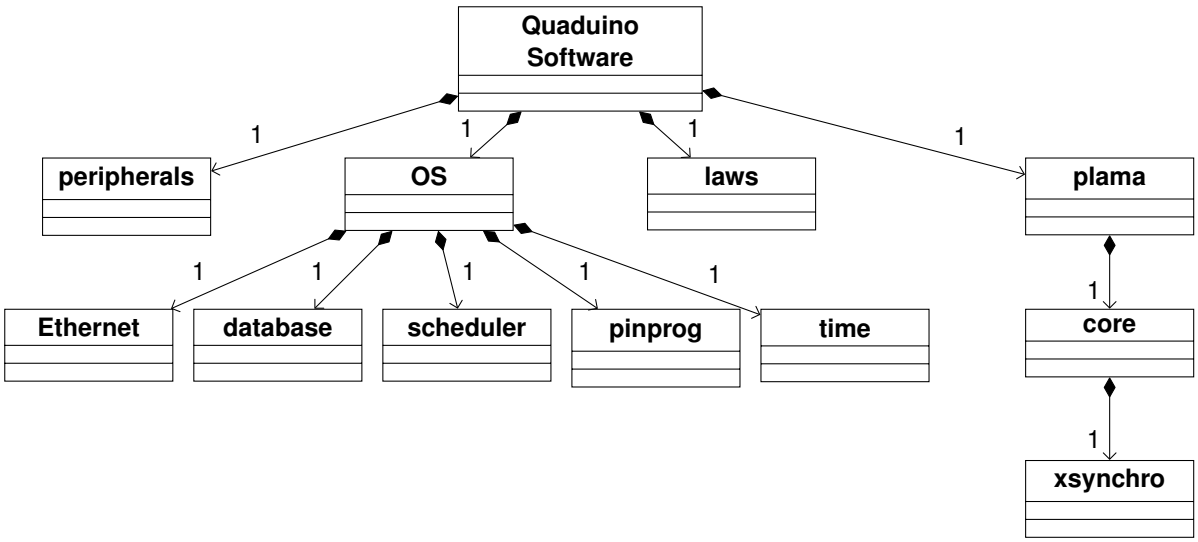

FIG 4. Quaduino Software Architecture with its main domains OS (fundamental services), peripherals (drivers for sensors/actuators), laws (actual application), plama (platform management, handles redundancy management)

while accommodating intricate projects encompassing multiple modules.

One of PlatformIO's standout advantages lies in its ability to streamline the development process, providing a unified and efficient platform for managing dependencies, libraries, and build configurations. This feature became particularly beneficial within the Quaduino project, where diverse sensors, communication modules, and peripheral devices demanded seamless integration.

To cater to the preferences and needs of our students, we've adopted a flexible approach. Recognizing that the choice of an Integrated Development Environment (IDE) can be a matter of personal preference, we encourage students to select any IDE that supports the C++ programming language. This flexibility empowers students to customize their development environment according to their individual preferences and also equips them with skills applicable to a broader spectrum of programming challenges beyond the Quaduino project.

4.5. Software Loading

The choice of development tools for programming and debugging is important for a smooth user experience. PlatformIO, our selected development environment, emerges as a versatile and practical choice for the Quaduino project. It creates one common binary file, which is loaded onto all four lanes of the system. This approach ensures that each lane executes identical software. The loading of these binaries onto the Arduino boards is facilitated through the use of the Atmel AVR programmer AVRdude via USB connections.

However, our teaching goals and plans for future laboratory courses needed a method for inspecting each individual lane's inner workings, including registers, data and program memory.

To meet this requirement, we integrated the MPLAB SNAP JTAG debugger (see Fig. 5) into our system architecture. This specialized hardware debugger not only provides debugging capabilities but also offers an interface for connecting with individual Arduinos, with each board identifiable by its unique serial number.

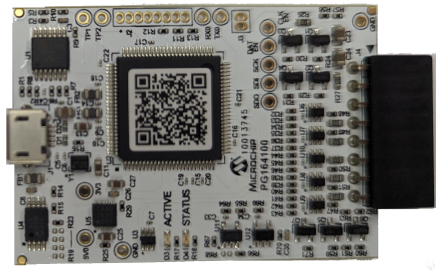

FIG 5. MPLAB SNAP debugger, can connect to ICSP and JTAG interface or Arduino Mega 2560

This integration effectively solves a significant challenge: the need for individual flashing of each lane. This challenge arose as a consequence of the laboratory's PC boot-up process, which assigned ports to connected Arduinos in a random sequence, making conventional USB connections impractical. As a solution, we transitioned to using the MPLAB debugger for flashing new software.

To enable the JTAG interface, each Arduino must undergo configuration, i.e., setting the fuses accordingly. This temporarily renders the previous method of software loading via USB unusable, until the fuses are reset. To program the Arduinos, we initially connect the MPLAB debugger to the ICSP interface, set the new fuse values and can then connect the debugger to the JTAG interface.

In summary, we have made the deliberate choice to rely solely on the new JTAG debugger for both debugging and software loading tasks, aligning with our project's teaching goals and ensuring efficient deployment in a laboratory setting.

4.6. Debugging via JTAG

The JTAG debugger allows programming the microcontroller's flash and EEPROM storage for uploading com-

piled programs, setting fuse and lock bits, halting and resuming its core as well as reading and overwriting registers and RAM. It is supported by the MPLAB-X IDE, which allows setting breakpoints at specific lines in code, and provides a graphical user interface for interacting with the debugging system, such as halting the microcontroller or manipulating memory. The Atmega2560's JTAG interface and on-chip debugging capabilities must be specifically enabled by setting the JTAGEN and OCDEN fuses.
Besides programming the microcontroller, the debugging capability of the Quaduino setup is used for interactive hands-on exercises for students: They first learn basic machine instructions by reading the microcontroller's instruction set manual and by translating a minimalistic C-code snippet manually. After that, they are tasked to let a C-compiler translate the code and compare their solution with the contents of the microcontroller's program memory. For learning how data is stored in memory, students compile and upload a program with static data. After the variables have been initialized, they download a snapshot of the RAM using the debugger. Students also learn the principles of the call stack, by writing programs with function calls and by inspecting the stack. For this, they read the stack-pointer register at different times during program execution and look at the corresponding RAM content. These exercises are supposed to convey the operating principles of a digital computer and make students aware of the challenges and pitfalls when developing safety critical digital systems: Many details, such as endianness, execution timing and memory organization, which are usually abstracted away by a programming language or operating system, become visible.

5. LABORATORY SETUP

The Quaduino itself is part of the laboratory setup that accompanies a PC running Debian, a relay-board, an oscilloscope, a networking switch, and another Arduino. The PCs hosts all necessary tools, the IDE and connection to git repositories. The users work directly on these PCs - either locally or via VNC.
The entire setup rests in and on an acrylic glass case. The Quaduino tower is screwed to the top plate and the other components on the lower plate. This way, the students can see and observe each component, have easy access to all wires and peripherals, can perform custom experiments and use the oscilloscope to view specific signals.

5.1. Web Interface

The students need to access the internal values for select database entries, e.g., sensor values, failure indications, synchronization information, and debugging data. We can use the Ethernet service to transmit this data to a server on the laboratory computer.
For visualization, we developed a simple interface (see Fig. 6) that displays the four computing lanes, shows if they are on-line and displays the internal values.

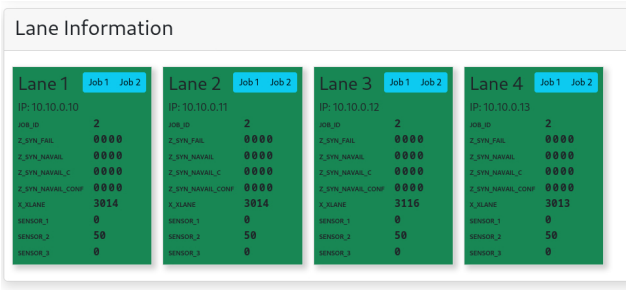

FIG 6. Web-Interface for the Qaduino that displays the online status for each computing lane and internal values

5.2. Peripherals

With the vast Arduino ecosystem, we can directly incorporate different sensors and actuators in our system. We can use the respective libraries, which facilitates the driver development immensely and use PlatformIO to keep track of package updates.
Initially, we tested mainly potentiometers as **sensors**. We needed to write a short wrapper that allows correct addressing, initialization and receiving values from the sensor.
Initially, we included LEDs as **actuators** - as they are easy to control and allow to show different information.
We further included RGB LEDs and servo motors SG90. Eventually, we will implement a model of the space shuttle orbital vehicle as demonstration scenario, hence we will use servos for the elevon, add time-of-flight sensors to measure the distance between shuttle and base and control the elevon using a pilot stick - or by students at home using the keyboard.

5.3. Control Arduino

For instrumentation purpose, we added another Arduino Mega including an Ethernet Shield to the laboratory setup. It connects to each computing lane via five analogue outputs and five analogue inputs. This allows transmission and reception of signals from and to the computer lanes. Additionally, it can receive data via Ethernet from the laboratory computer. It will additionally control the motor for the shuttle z-axis.

5.3.1. Virtual Sensors

For select use cases, adding multiple sensors for the same value becomes infeasible. We want to include a pilot's stick into the system, which connects to USB. It is impossible to connect an USB device to the Arduino itself and furthermore we would still have only one sensor value for each axis. To solve this, we introduce virtual sensors, that the additional Arduino (Control Arduino) distributes via pulse-width modulated signal onto each single lane. The Control Arduino receives the signals via Ethernet from the laboratory computer. This in turn is fed using the web interface, that can read data from the USB joystick. The web interface offers a tool to alter the signal individually for each computing lane and thus to

simulate multiple sensors for the same physical value. The signals can have fixed or relative offsets.
From the Quaduino point of view, a virtual sensor acts identical to an actual sensor. The data is read as pulse and hence needs a driver.

5.3.2. Measuring Asynchrony

In order to detect the remaining asynchrony after successfully executing the synchronization service, we needed a global time. The Control Arduino receives signals from each computing lane upon successful synchronization and stores the time, the signals occurred. It then computes the median reception time and computes the deviations from each lane to the median value.

6. VERIFICATION

During this year's lab course "Systementwurf II", we incorporated the Quaduino platform. The students had to implement the synchronization within the redundant computer, the initial code included all parts except for the actual synchronization service.

6.1. Tasks

The primary task was the implementation of the synchronization service. This service encompasses the setting of states, broadcasting of messages, and the establishment of a common time base.
The students received the algorithm in pseudo code form. It consists of a pre-sync and a sync-phase, which end upon time-out or receiving messages.
The validation and verification of the implemented function were carried out using six test scenarios. These tests were conducted on a laboratory setup. Each scenario had a clearly defined expected system response. During the tests, the actual system response was observed and compared to the expected response. The results obtained were then analyzed and classified.
The six test scenarios include:

1) No Synchronization: This scenario analyzes the behavior of the system without any synchronization.
2) Lane is off-line: Here, the system behavior is checked when a lane is inactive.
3) Lane exhibits consistent errors in Quadruplex configuration: In this scenario, the behavior is analyzed when a lane consistently delivers erroneous results.
4) Lane exhibits inconsistent errors in Quadruplex configuration: Here, the behavior is examined when a lane produces inconsistent errors towards its neighbors.
5) Lane exhibits inconsistent errors in Triplex configuration: This scenario examines the behavior of the system when one lane is already off-line and another lane exhibits inconsistent errors.
6) Pre-sync phase is too short: This test investigates the impact of an overly short pre-synchronization phase on the overall system.

Through the application of these test scenarios, a comprehensive assessment and verification of the student's implementation of the synchronization service were achieved. Fig. 7 shows the remaining asynchrony in each lane after the successful synchronization service.

6.2. Lessons Learned and Bugs

During the course, three student's groups used a total of four laboratory setups and came across some bugs that we investigated together.
The **Ethernet Module** proved to have spurious faults during operation whenever a single lane went off-line and was to be restarted. Whenever a computing lane is off, it does not receive 9V power from the relayboard, but it still connects via the RS485 modules on the wires A and B. Turns out, the wires have a pull-up and pull-down resistor towards the 5V and GND pins of the Arduino. This way, the off-line single lane still receives some 5V from the other lanes. Fig. 8 shows the result from a voltage measurement experiment: all four lanes are on-line, then lane 1 is shut off via the relay board and turned back on again after 40 seconds. We expected lane 1 to return to $0\,\mathrm{V}$ and the other lanes remaining at $5\,\mathrm{V}$. However, lane 1 does not return to $0\,\mathrm{V}$, it remains at $1.5\,\mathrm{V}$. Furthermore, Lane 3 shows spurious voltage drops to this level.
This in turn leads to the Ethernet Module not restarting correctly without the reset pin being triggered correctly. This posed an issue, as we left the reset pin open - which works fine in a single setup. To solve this issue, we added a dedicated reset wire that we trigger during boot-up of the single lane.
The **Fuses** can render the Arduino unusable and even change the internal clock speed. Initially we set the fuses using the MPLAB IDE, which sometimes lead to the wrong values. To solve this, we use avrdude to change the fuses to a fixed value.
The **synchrony** of the redundant computer was destroyed whenever a single-lane went off-line. This happened due to the RS485 timeout setting, which was initially at $1000\,\mathrm{ms}$. This lead to individual lanes running into the timeout whenever they received an incomplete message from the off-line lane and thus leading to an unsuccessful synchronization. This was easily solved by adjusting the timeout setting.
Software loading is broken whenever an incomplete software load runs on the Arduino. This happens whenever two Arduinos claim the sending property for the RS485 module. This error can currently only be fixed by removing the receiving RS485 modules and loading the software again.

6.3. Aftermath

Each group succeeded in their task within the expected time frame.
One Quaduino tower was destroyed, because a wrong power adapter was connected. This lead to random behavior of the single lanes.
Generally, students perceived the new laboratory well and especially mentioned positively the numerous challenges with embedded hardware.

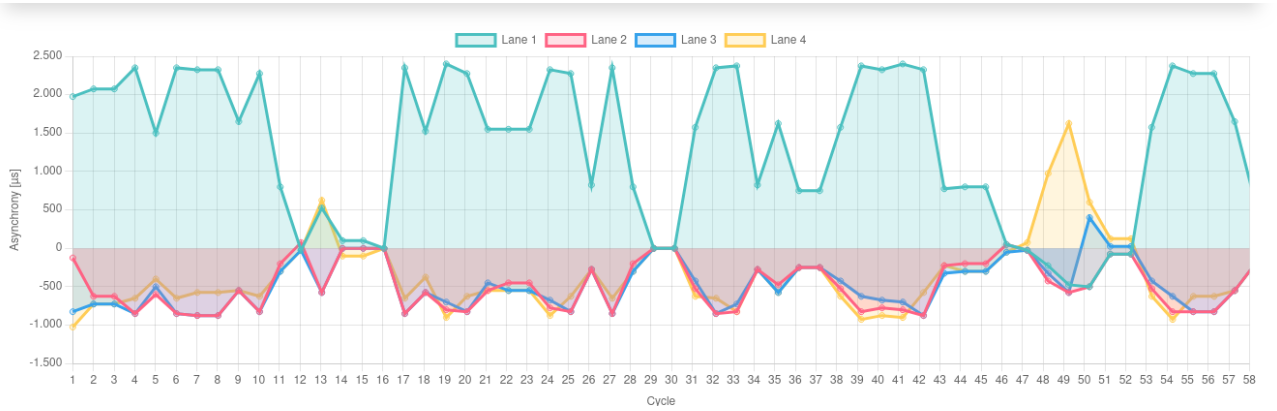

FIG 7. Plot of remaining asynchrony in micro-seconds over cycles in for lanes 1-4 after successful synchronization over cycles, cycle time is 1 s

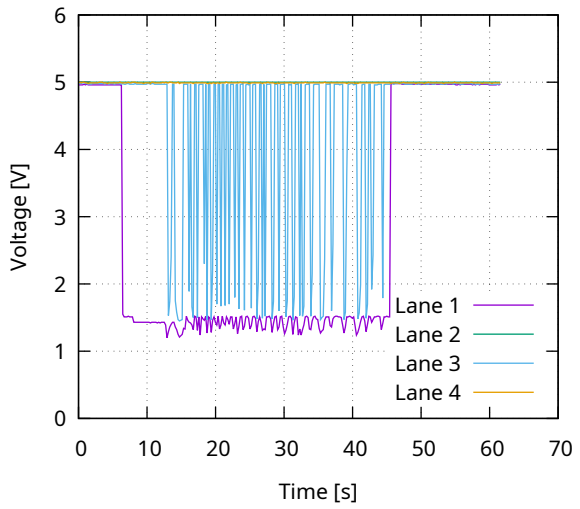

FIG 8. Plot that shows the voltage level of neighboring lanes upon lane 1 being off-line, i.e., without 9V from the relay board

7. SUMMARY AND OUTLOOK

We implemented an Arduino-based quadruplex-computer-system used for teaching computer basics and redundancy mechanisms deployed in safety-critical aircraft. The use of COTS hardware components together with established software tools resulted in an easy to use and affordable setup. We have so far implemented parts of the embedded software including drivers for several peripherals, a scheduler/dispatcher and a synchronization mechanism as well as a graphical user interface for interacting with the setup. A first use of the setup in a students lecture was successful

Currently, we extend the software with the platform management parts for sensors, laws and actuators. The laboratory is intended as hybrid workplace, which means students can either work remotely or in the avionics lab. Therefore we currently implement the infrastructure for VNC connections and data handling using gitlab. Each laboratory setup will then feature two webcams which allow for live observation of the servos and LEDs.

Eventually, we will utilize this concept in the three fundamental lectures on avionics - either in the lectures and exercises as demonstration, as actual training ground, and in the lab course.

For the near future, we will improve the laboratory by extending with a full one degree-of-freedom space shuttle orbital vehicle including all necessary sensors, actuators, external controls, and software. It will move its elevons and moves up and down - depending on the elevon angle. Furthermore, we aim at implementing the industry standard ARINC653 API [15].

For a distant future, we can utilize the four Arduinos in different architectures, e.g., using verifiable computing [16], the setups track all internal information in a centralized database and thus produce data for use in machine learning applications. This could then be used for health determination [17] or during initial deployment to find faulty solder points.

Acknowledgement

Results were enabled by the project HYMASY funded by the „Stiftung Innovation in der Hochschullehre“ in the scope of the program "Freiraum 2022".

Contact address:

bastian.luettig@ils.uni-stuttgart.de
quaduino.org

References

[1] Ian Moir, Allan Seabridge, and Malcolm Jukes. *Civil Avionics Systems*. John Wiley & Sons, 2013. Google-Books-ID: 8XFwAAAAQBAJ. ISBN: 978-1-118-53672-8.

[2] Mario Werthwein, Darbaz Darwesh, and Bjoern Annighoefer. Using electronic data sheets for an automatic detection of peripheral devices in a smart wing's digital infrastructure. In *2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC)*, pages 1–10, 2022. DOI: 10.1109/DASC55683.2022.9925884.

[3] Bjoern Annighoefer, Marc Riedlinger, Oliver Marquardt, Reza Ahmadi, Bernd Schulz, Matthias Brunner, and Reinhard Reichel. The adap-

tive avionics platform. *IEEE Aerospace and Electronic Systems Magazine*, 34(3):6–17, 2019. DOI: 10.1109/MAES.2019.2900903.

[4] Vivi Yang. Releases final report of china airlines flight CI202 occurrence investigation. Publisher: Taiwan Transportation Safety Board.

[5] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, USA, 1994.

[6] Stefan Poledna. *Enforcing replica determinism*, pages 61–88. Springer US, Boston, MA. 1996.

[7] Simon Goerke, Rolf Riebeling, Florian Kraus, and Reinhard Reichel. Flexible platform approach for fly-by-wire systems. In *2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC)*, pages 2C5–1–2C5–16, 2013. ISSN: 2155-7209. DOI: 10.1109/DASC.2013.6712542.

[8] Y. C. Yeh. Triple-triple redundant 777 primary flight computer. In *1996 IEEE Aerospace Applications Conference. Proceedings*, volume 1, pages 293–307 vol.1, 1996. DOI: 10.1109/AERO.1996.495891.

[9] Marc Fervel, Arnaud Lecanu, Antoine Maussion, and Jean-Jacques Aubert. Aircraft control system with integrated modular architecture. patentus 8600584B2, Airbus Operations SAS, 2013. Library Catalog: Google Patents.

[10] Mega 2560 rev3 | arduino documentation.

[11] WIZnet Co., Ltd. W5500 datasheet.

[12] V Maxim Integrated Products, Inc. Datasheet for MAX481/MAX483/MAX485/MAX487–MAX491/MAX1487 low-power, slew-rate-limited RS-485/RS-422 transceivers.

[13] Bastian Luettig, Bjoern Annighoefer, and Reinhard Reichel. A service provisioning layer enabling simplex-minded function development on integrated modular avionics hardware. In *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*, pages 1–9, 2018. ISSN: 2155-7209. DOI: 10.1109/DASC.2018.8569364.

[14] Matthias Lehmann. Testumgebung für komplexe, verteilte Avionikplattforminstanzen, 2012.

[15] Aeronautical Radio, Incorporated. Arinc 653: Avionics application standard software interface PART 1 – REQUIRED SERVICES. Technical report, Aeronautical Radio, Incorporated, 2019.

[16] Johannes Reinhart, Bastian Luettig, Nicolas Huber, Julian Liedtke, and Bjoern Annighoefer. Verifiable computing in avionics for assuring computer-integrity without replication. In *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, pages 1–10, 2023.

[17] Bastian Luettig and Bjoern Annighoefer. Using autoencoders to identify aged, faulty and unknown peripherals in the adaptive ima system. In *2023 IEEE/AIAA 42nd Digital Avionics Systems Conference (DASC)*, pages 1–9, 2023.