SpaceCANvas: A low-cost open-source CAN-based hardware and software framework for space systems networks

Duarte Galvão[(1)], **Mário Vástias**[(2)], **José Teixeira de Sousa**[(1)], **Horácio Neto**[(1)], **Rui Policarpo Duarte**[(3)]
[(1)] INESC-ID; Instituto Superior Técnico, Universidade de Lisboa, duarte.galvao@tecnico.ulisboa.pt, jts@inesc-id.pt, hcn@inesc-id.pt;
[(2)] Instituto Superior de Engenharia de Lisboa, Instituto Politecnico de Lisboa, mvestias@deetc.isel.ipl.pt;
[(3)] INESC-ID; Celestia Portugal, rpd@inesc-id.pt

Abstract

Novel space missions require spacecraft with faster, versatile, reliable, and non-proprietary communication network for interconnection of its components. CAN buses have been adopted into space systems because of their resilience and fault-tolerance properties. Moreover, there is a new trend in the space industry to adopt off-the-shelf devices which are more computationally powerful and cheaper, instead of the typical radiation-hardened FPGAs, which are more expensive and less powerful. This work presents SpaceCANvas, a turn-key solution focused on the ECSS-E-ST-50-15C specification, a space-tailored CAN-based standard. This solution includes an open-source hardware and software framework targeting COTS MCUs and SoC devices. So far, there are no open-source solutions which provide this.

Keywords: System on a Chip (SoC), Microcontroller Unit (MCU), ECSS-E-ST-50-15C, Space Engineering, Fault Tolerance

1. HARDWARE IN CONTEXT

Recent space missions using SmallSats [21] and CubeSats [23] rely on Commercial Off-The-Shelf (COTS) with modern System on Chip (SoC) devices, which are energy efficient, computationally powerful, and exhibit enough radiation-tolerance to withstand space conditions [15]. One of the challenges of designing a satellite or spacecraft computer system is the communication between its components. Furthermore, the communications must be reliable, compact and energy-efficient. In this paper we present an open-source solution for a reliable interconnect system for space systems using low-cost COTS components.

The Controller Area Network (CAN) Specification is a technology employed in the automotive industry since 1991 [19]. CAN has a simple and easy-to-discriminate physical medium – two possible values at any point in time, transmitted via two wires with opposing polarities. Additionally, redundant buses further increase the reliability of the network.

The CANopen protocol is defined in the CAN in Automation (CiA) 301 2011 standard [5] and was designed for unifying the application layer of the CAN stack. In CANopen, all communications are executed by transmitting and receiving objects out of a pre-defined set. The set is defined in the Object Dictionary (OD), which is a structured collection of properties to be used on the configuration of the communication and of the nodes themselves. A space-tailored CANopen-based standard was released in 2015 by the European Cooperation for Space Standardization (ECSS), the ECSS-E-ST-50-15C [8]. This specification is intended for usage in environments with low-power requirements, high radiation incidence, and stronger mechanical stresses. It is an alternative to other expensive, obsolete, or closed-source protocols, such as SpaceWire [7]. The network stack defined in the ECSS-E-ST-50-15C contains fault-tolerance mechanisms, both original and inherited from the aforementioned CAN specifications.

Space missions, such as ExoMars [4, 20] and ESAIL [2, 24], have already deployed this standard. It is mostly used for command and control. Yet, there are no open-source and configurable ECSSE-ST-50-15C modular transceiver circuit designs and source-code in C which can target small microcontrollers or SoCs. Thus, for each ECSS-E-ST-50-15C system implementation it is necessary to design and manufacture customized Printed Circuit Boards (PCBs), and develop software from scratch. This constraint causes higher costs and longer development and prototyping times. In this work we present a scalable, lightweight, and low-cost framework which facilitates the implementation of CAN systems with different target devices and requirements such as the ones on space missions. SpaceCANvas is a hardware and software turn-key solution of CAN communication for embedded and SoC systems. The following subsections present the detailed context for each component, finishing with a summary of the contributions made in this work.

1.1 SpaceCANvas Shield

Most recent COTS Microcontroller Unit (MCU) and SoC Field Programmable Gate Array (FPGA) development kits have built-in CAN controllers but do not embed CAN transceivers. Currently, and to the best of the author's knowledge, there are no available boards or modular solutions for interfacing with the embedded processing systems' CAN controllers. There are CAN Shields for Arduino which include a CAN controller [1]. In addition, the communication between the development kit and the controller is performed via Serial Peripheral Interface (SPI). Thus, we lose the benefit of the proximity of the controller to the Central Processing Unit (CPU), adding latency and increased design complexity. Moreover, the embedded system offers the device-driver for its CAN controller, while an Arduino CAN Shield requires the development of a device-driver for each different hardware platform. Finally, most of the Arduino CAN Shields include only one CAN interface.

The SpaceCANvas Shield is an Arduino shield developed for a compact and portable mounting of the transceivers and their related circuitry. The Arduino interface was selected because of its availability in development boards, ensuring a future-proof design. The shield targets development kits with CAN controllers – these controllers can either be integrated on the MCU/SoC device or programmed on the Programmable Logic (PL) of an FPGAs.

1.2 ECSSopenNode Library

There are several implementations of the CANopen protocol: CanFestival [25], CANopenNode [3], openCANopen [18], and Lely CANopen [17]. An examination was performed to determine the most appropriate library to serve as a base for SpaceCANvas. The chosen criteria for differentiating the libraries are based on the ISO/IEC 25010:2011 [16] system and software quality standard. The ISO/IEC standard defines several characteristics. Each characteristic was evaluated with a set of criteria:

- *Functional suitability* – some libraries do not fully implement the CiA 301 standard or the ECSS-E-ST-50-15C standard. Thus, to avoid repeating work previously implemented by others, a more complete library is desired.
- *Performance efficiency* – since the target devices for this framework are MCUs and SoC FPGAs for use in space systems, the memory availability is limited. Thus, this assessment focuses on memory efficiency.
- *Compatibility* – proper separation practices in C are observed, such as the use of prefixes for names to avoid collisions with other libraries, as well as use of synchronous or asynchronous calls.
- *Usability* – the user is the programmer who will build their application on the framework. As such, usability is the easiness of configuration for a desired application. The provided documentation and examples are also taken into consideration.

- *Reliability* – the focus of the assessment for this characteristic is the libraries' maturity. Other aspects of reliability, such as availability, fault-tolerance and recoverability, are highly platform-specific and not trivial to evaluate for this selection analysis.
- *Security* – the framework is to be used in controlled environments, so security is not a relevant characteristic.
- *Maintainability* – the framework must be scalable, expandable, and testable. Therefore, it is important for the source code to follow modern guidelines for code quality, such as parameter checking, abstraction and proper design pattern implementations. Since the ECSS-E-ST-5015C standard defines several optional features [8], modularity is particularly important to adapt the library to each use case. Testability is also important to fulfill the ECSS software product assurance standard [9], and is evaluated by checking for the presence of unit tests, given that code developed without having unit tests defined before or simultaneously tends to not be fully testable.
- *Portability* – the platform support provided by the library. Libraries conceived for use in embedded systems should not use resources that are often unavailable, such as POSIX threads. Number of dependencies is also considered.

CANopenNode was selected as the base CANopen library for the SpaceCANvas framework because of its open-source, modern, modular and configurable source-code.

1.3 Contributions

The following hardware and software contributions were made as part of the framework development:

- Evaluation and assessment of the existing CANopen implementations based on software quality standards – necessary for the selection of the base library for the ECSS-E-ST-50-15C implementation.
- Implementation of CANopen device-drivers for two platforms common in space systems: Xilinx's Zynq-7000 SoC FPGA and Atmel's SAM V71 MCU.
- Design and build from scratch of a two-channel CAN transceiver add-on board for testing the framework with development kits which do not include CAN transceivers.
- Implementation of parallel bus redundancy in the selected CANopen library for ECSS-E-ST50-15C compliance.
- Implementation of a foundation for a self-test suite, with example tests to ensure the software quality.

2. HARDWARE DESCRIPTION

SpaceCANvas was based on an existing open-source implementation [3] and improved to achieve compliance with the ECSS-E-ST-50-15C standard. Moreover, SpaceCANvas also includes the transceiver hardware and the respective software components to control it. The framework's structure is summarized in Figure 1. The nodes comprise an MCU/SoC development kit with the custom-built SpaceCANvas Shield. The nodes interconnect with a redundant CAN, as per ECSSE-ST-50-15C requirement. The nodes also connect via USB to a host computer for performing automatic testing.

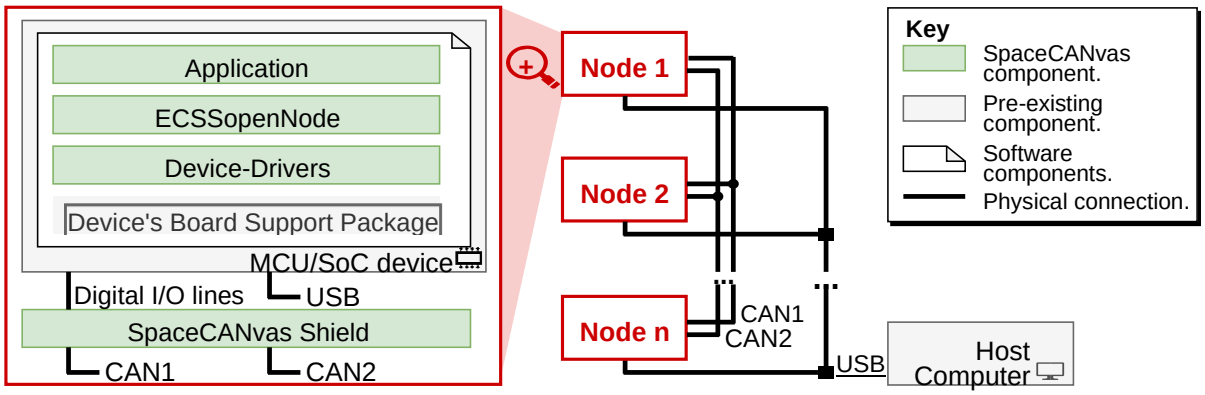

Figure 1: SpaceCANvas hardware and software overview.

The most advantageous features of this framework are:

- The hardware was designed with low-cost COTS components and is available as open-source;
- The CAN Transceiver board enables the use of the on-chip CAN controller, improving performance and reducing the number of necessary Input/Outputs (I/Os);
- The software has a small memory footprint and no OS or external dependencies / standalone, thus being adaptable for use in varied environments;
- Includes a test platform for validating the system;
- Ready to use in a laboratory environment or on real space-systems such as CubeSats or SmallSats.

The following subsections detail on the hardware and software components of the framework.

2.1 Physical Network

At least two nodes are necessary to create a network. Each node runs on an embedded device, such as a MCU/SoC development kit. These nodes must be capable of:

(a) running the test applications;
(b) connecting to a redundant CAN bus.

For the requirement (a), the limitation is the available memory: it is necessary to have at least 20 KB of Flash/Electrically-Erasable Programmable Read-Only Memory (EEPROM) (program) and 5 KB of Random Access Memory (RAM) (data). For the requirement (b), each device must have two CAN interfaces, or the ability to connect two CAN interfaces. The components of the interface are:

- CAN transceiver – transforms the electrical signals from the CAN High and CAN Low lines into a two-line, Receive (Rx) and Transmit (Tx), digital bitstream, with dominant translated to low signal and recessive translated to high.
- CAN controller – accessed from the CPU via registers. Receives a bitstream from the transceiver and stores incoming frames onto designated registers, and transmits frames from determined registers as a bitstream to the transceiver, when the channel is available. It also performs frame acknowledgement and error counting and signalling.

Embedded systems which only integrate the CAN controllers and include Arduino headers may make use of the SpaceCANvas Shield.

2.2 SpaceCANvasShield

SpaceCANvasShield contains the SpaceCAN Transceiver Shield PCB's Fritzing project (project.fzz) and its exported Extended Gerber (RS-274X) files (in gerber/).

3. SOFTWARE LIBRARY

The ECSS-E-ST-50-15C library, denominated *ECSSopenNode*, is designed around CANopenNode's structure and code, with the ECSS-E-ST-50-15C-specific features implemented as new modules within the library. Additionally, two new modules were created: the *Library Self-Tests* module, which performs unit tests at start-up and reports the results in the Test Anything Protocol (TAP) format; and the *UART-CAN Bridge* module, which allows the transmission of CAN frames with UART, necessary for executing the network tests.

ECSSopenNode uses the same modular structure as CANopenNode. To comply with the ECSSE-ST-50-15C, the library implements a new module for ECSS-E-ST-50-15C compliant time distribution, and a module for performing automated testing to comply with code quality requirements. These features are implemented in a manner as to maintain CANopenNode's code conventions, as well as to be fully customisable.

The ECSS-E-ST-50-15C standard defines a parallel bus access architecture for the implementation of a two-bus redundancy [8]. In this scenario, all nodes transmit and receive all messages on both buses simultaneously. The duplicate frame handling must be implemented in the application by the user. In case of a temporary or permanent fault on a node leading to a communication failure, the communication will proceed normally on the other active bus, without any delay.

CANopen includes a time distribution service, in which the master node sends a message with 2 bytes corresponding to the number of days since 1 January 1984, and 4 bytes corresponding to the number of milliseconds after midnight [5]. The ECSS-E-ST-50-15C standard considers CANopen's time distribution service to be inadequate for space requirements [8]. Thus, the ECSS proposes a solution which supports a 16-bit 'submilliseconds of ms' field and distributes time information with two different modes: the *time distribution protocol*, which consists of broadcasting the time object using a Process Data Object (PDO); and the *high-resolution time distribution protocol*, which uses CANopen's SYNC protocol in conjunction with the PDO. Nodes can keep and synchronize Spacecraft Elapsed Time (SCET), Spacecraft Universal Time Coordinated (UTC), or both. In the framework, a new module was created in the ECSSopenNode library for managing the time distribution. The selection of time resolution (normal or high) and type (UTC or SCET) are defined in the library configuration at compile time, while the necessary PDO are configured in the Object Dictionary.

To fulfil the ECSS software quality assurance standard [9] and the ISO/IEC's software quality [16] standards, it is essential that the framework is properly tested with sufficient coverage. An existing open-source unit test library was used to avoid implementing unnecessary code. An assessment of fourteen open-source test libraries available was performed in [14]. CMocka was selected for the execution of the tests since it is compatible with the target embedded systems.

3.1 ECSSopenNode

ECSSopenNode is a fork of the CANopenNode repository [3] with additional functionality for ECSS-E-ST-50-15C compliance.

3.2 ECSSopenNodeDemo

The repository contains a demo SpaceCANvas network scenario to run on up to five devices. The objective of this demo is to simulate a real-world application of ECSSopenNode for a small space system. The example scenario consists of a satellite equipped with an On-Board Computer (OBC), an altimeter and accelerometer module, a thermometer, a Synthetic Aperture Radar (SAR) sensor and an image

processor. The sensors generate synthetic data, and the receiving nodes confirm the correctness of the transmissions.

3.3 CANopenSAMV71

CANopenSAMV71 holds the ECSSopenNode device-drivers for devices of the Atmel SAM V71 family. The repository contains an Atmel Studio 7 project for the ATSAMV71Q21B, which can be added to a solution. The structure of the src/ directory is as follows:

- ASF/ – Atmel Studio / ASF generated libraries. Equivalent to a board support package. The following modules are included:
  - Generic board support (driver)
  - Delay routines (service)
  - GPIO – General purpose Input/Output (service)
  - IOPORT – General purpose I/O service (service) – Standard serial I/O (stdio) (driver)
  - CAN – Control Area Network Controller (driver) – MPU – Memory Protect Unit (driver)
  - PIO – Parallel Input/Output Controller (driver)
- CANopenNode/ (git module) – CANopenNode or ECSSopenNode library.
- config/ – Board configuration files.
- SAMV71/ – CANopenNode device-drivers for the Xilinx Zynq-7000 platform.
- example SAMV71/ – Example CANopenNode application.

3.4 CANopenArduZynq

CANopenArduZynq holds the ECSSopenNode device-drivers for devices of the Xilinx Zynq-7000 family. The repository follows the structure of a Vivado project for the Trenz TE0721-03M "ArduZynq" development kit.

- can0-2018.sdk/ – Xilinx SDK projects.
  - can example0/ – A CAN example application that tests communications on both CAN controllers at 1Mbps. Adapted from Xilinx's xcanps_polled_example.c. – canopennode/ – CANopenNode port for the Xilinx Zynq-7000 platform.
    - src/CANopenNode/ (git module) – CANopenNode or ECSSopenNode library.
    - src/ZYNQ7/ – CANopenNode device-drivers for the Xilinx Zynq-7000 platform.
    - src/example_ZYNQ7/ – Example CANopenNode application.
  - design_1_wrapper_hw_platform_0/ – Trenz TE0723 automatically generated hardware platform wrapper.
  - hello_world/ – Contains the default Xilinx "Hello World" application, to be used as a sanity check for correct FPGA configuration and UART set-up.
  - Standalone_bsp_0/ – Zynq board support package. Automatically generated by the Xilinx SDK.
- can0-2018.srcs/ – Hardware design source files. Includes imported files and automatically generated files.
- doc/ – Directory with documentation.
- README.md – Instruction file.

4. BUILD INSTRUCTIONS

4.1 SpaceCANvas Shield

Due to the lack of a medium to attach the CAN transceivers Integrated Circuit (IC) to the boards, a daughter PCB was developed, in the form of an Arduino shield. Fritzing [10], an open-source Computer Aided Design (CAD) software, was used to design the circuit schematic and the fabrication output1 (Gerber files). The shield's circuit schematic is presented in Figure 2.

The shield is intended to be compatible with various development boards. The CAN transceiver has variable I/O voltage, which is assigned to the IOREF pin on the embedded system, which usually ranges from 1.8V to 5V. The shield can set the CAN transceivers to standby mode. This mode can be activated via a physical toggle switch (SW1 and SW2), so it can be set on-the-fly for testing, or an I/O signal trough the Arduino interface (D2, D3), so the software can control the standby status. The selection between these two is made through jumpers J1 and J2.

The Tx and Rx lines for each CAN controller are connected to the digital I/O pins D4 trough D7. Digital I/O pins D0 and D1 are left unconnected due to often being reserved for Universal Asynchronous Receiver/Transmitter (UART) Rx and Tx lines respectively. Three LEDs are used as status indicators for power (red) and standby status of each CAN transceiver (orange).

The shield can be used on any MCU or SoC with one or two CAN controllers via the Arduino headers. The developed shield can be used for testing ECSS-E-ST-50-15C compliant CAN space systems in laboratory environments.

The following steps must be performed to build the SpaceCANvas Shield:

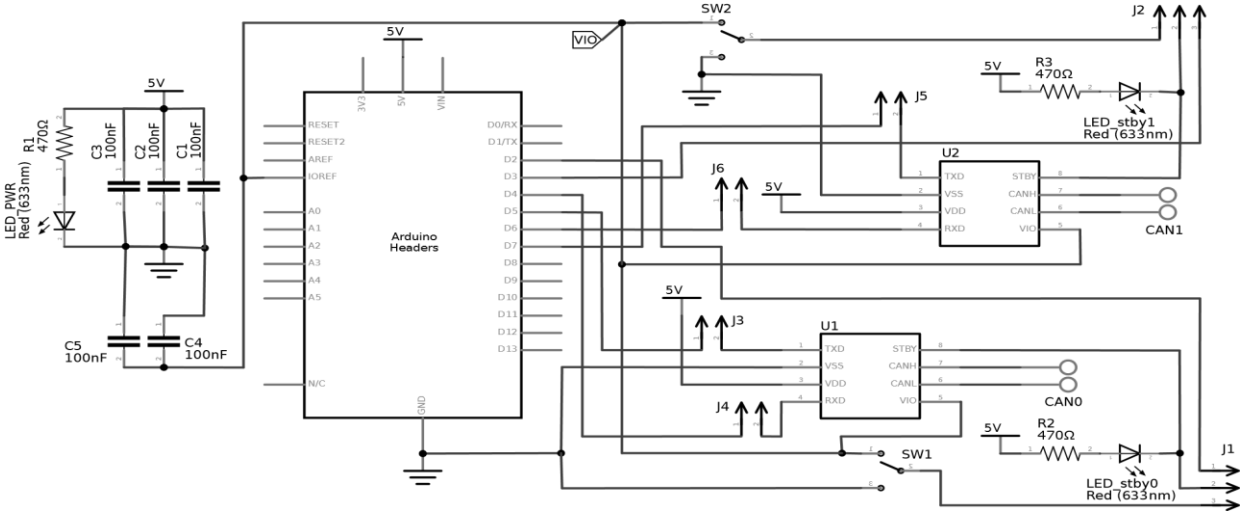

Figure 2: SpaceCANvas Shield circuit schematic.

1. Fabricate the two-sided PCB from the Gerber files available in the SpaceCANvasTShield.zip archive, and solder the components according to Figure 3.
2. Attach the shield to the development kit via the Arduino headers.
3. Set the shield's jumpers and switches according to the desired configuration:
   (a) J1: CAN0 transceiver's standby, controlled by the digital I/O pin 2 (position 1–2) or by the output of the SW1 slide switch (position 2–3).
   (b) J2: CAN1 transceiver's standby, controlled by the digital I/O pin 3 (position 2–3) or by the output of the SW2 slide switch (position 1–2).
   (c) J3 selects I/O pin 5 or an external input for CAN0 Tx.

[1] https://oshpark.com/shared projects/8SErD3bW

(d) J4 selects I/O pin 4 or an external input for CAN0 Rx. (e) J5 selects I/O pin 7 or an external input for CAN1 Tx.

(f) J6 selects I/O pin 6 or an external input for CAN1 Rx.

(g) SW1: CAN0 transceiver's standby. Position 1–2 on, position 2–3 off. (h) SW2: CAN1 transceiver's standby. Position 2–3 on, position 2–3 off.

4.2 Multi-node Test Network

To assemble the multi-node test network it is necessary to connect each of the CAN buses lines (CAN High and CAN Low) to each other with a pair of twisted wires, as shown in Figure 4. Each

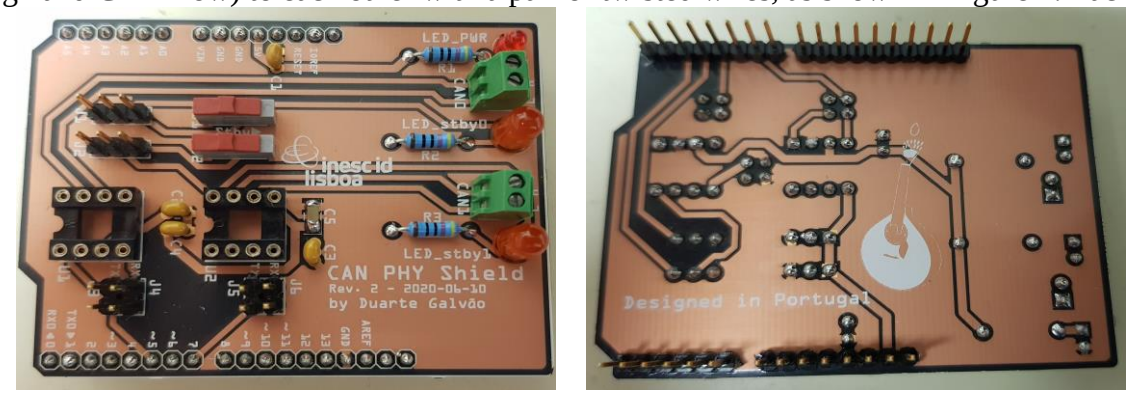

Figure 3: front and back of the assembled SpaceCANvas Shield PCB.

CAN bus requires a network termination circuit on each end. The termination consists of a 120Ω resistor. Resistors R1 and R2 are the CAN 0 network terminations, and resistors R3 and R4 are the CAN 1 network terminations. A CAN bus analyzer [27] may be attached to the network to aid the debugging process.

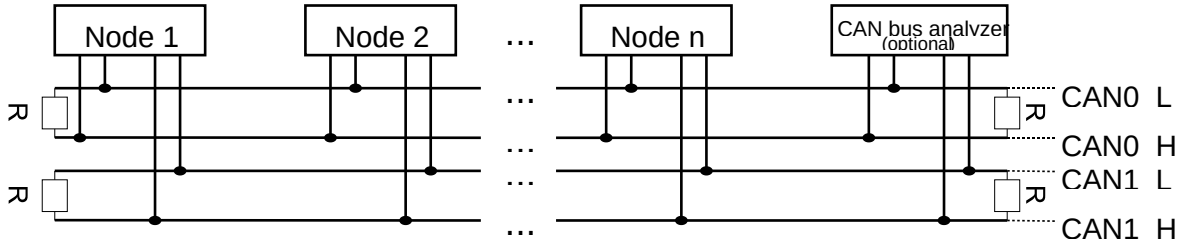

Figure 4: Multi-node test network diagram.

4.3 Software

It is necessary to connect a host computer to the nodes configure them. The following procedure must be executed on the host computer:

1. Install Git.
2. Install the SDKs for the target platforms:
   (a) If the target platform is a Xilinx Zynq SoC FPGA, install Vivado 2018.3 or later.
   (b) If the target platform is an Atmel SAM V71 MCU, install Atmel Studio 7 or later. (c) Otherwise, follow the recommendation on the target platform's manual.
3. Download and configure the project on the SDK.
4. Configure the Object Dictionary and CANopen settings.

Step 3 is highly platform-dependent. Thus, the detailed instructions for configuring the project on each platform are presented in Sections 5.3.1 and 5.3.2. Finally, the demo application configuration is described in Section 5.3.3.

4.4 Atmel SAM V71 Project Configuration

Download the project by cloning the GitHub repository [12]. To clone the repository, use Git Bash on the desired target directory, and then run the following commands:

1. `git clone https://github.com/duartegalvao/CANopenSAMV71.git` to clone the repository;
2. `git checkout ecss` to switch to the ECSS-E-ST-50-15C compliant drivers;
3. `cd CANopenSAMV71/` to enter the newly created clone repository;
4. `git submodule init` to initialise the Git submodules;
5. `git submodule update` to clone the contents of the Git submodules.

Then, to configure the project on the SDK, the project must be added to an Atmel Studio Solution. Create a new Blank Solution using the project creation wizard. Then, on the Solution Explorer pane, right-click on the solution name and add the provided project (.cproj file). The project should be able to be built and run successfully.

4.5 Xilinx Zynq-7000 Project Configuration

Before opening the provided project, *CANopenArduZynq*, the board files and constraints must be loaded onto Vivado. The provided project is specific to the Trenz TE0723-03M "ArduZynq" development kit, and set-up instructions for this board are available in the Step 1.1 of Galvão's "Getting Started with an ArduZynq Board" guide [13]. If using the same board, open the project on Vivado and launch the SDK. To use the project with a different board, follow these steps:

1. Import the manufacturer-provided board files onto Vivado.
2. Open the project in Vivado (.xpr file).
3. Change the board in the project settings.
4. Import the board's manufacturer-provided constraint files.
5. Delete the existing Zynq-7000 Processing System IP block from the block design and recreating it with the appropriate settings for the target board.
6. If the digital I/O lines for pins D2 through D7 must be routed trough the Emulated Multiplexed Input/Output (EMIO), configure the necessary ports and constraints.
7. After the block design is validated, generate block design, run the synthesis and implementation processes, generate the bitstream and export the hardware to the SDK. 8. Launch the SDK.

The SDK projects should all be ready to build and run on the connected device.

4.6 ECSSopenNodeDemo Configuration

The ECSSopenNodeDemo repository's contents should be placed in the source-code directory of a CANopenNode project with the appropriate device drivers pre-loaded. The repository's root folder must be added to the Include directories in the build configuration. No other CO_OD.h header should exist within any included directory.

The demo network's global parameters are set in the demo parameters.h header. This header should be identical for all nodes running in the same demo network. To run the demonstration, first set the participating slave nodes in the ECSSDEMO PARTICIPATING SLAVES bitmask. The demo must be run with the master node (Node 1) and any combination of slave nodes (Nodes 2 through 5).

For each node, the conf_demo.h configuration header must be created and define CO DEMO NODE to correspond with the desired node for the program execution. For example, on the device that runs Node 2, the conf_demo.h's source code should be as follows:

```
#ifndef CONF_DEMO_H
#define CONF_DEMO_H
#define CO_DEMO_NODE 1
#endif
```

5. OPERATION INSTRUCTIONS

In this section we show how to use the SpaceCANvas Shield and the software to control it and the system. The first subsection details on how to build an application for a CAN system. The following subsection describes how the Object Dictionary and CANopen Settings are defined. Finally, the device driver development is described, for use with devices which have no device-drivers available.

5.1 SpaceCANvas Shield Preparation

1. Insert the shield in the development board with the correct orientation.
2. Power-up the development kit and confirm that the LED PWR turns on.

5.2 Implementing a User Application

New applications using the ECSSopenNode library must implement five methods, similarly to abstract functions in C++: *programStart*, *communicationReset*, *programAsync*, *programEnd* and *programSync*. These methods are called by the device-driver in accordance with the flowchart in Figure 5. The main function execution flow is shown on the left side, while the synchronous task flow is shown on the right side. The yellow boxes represent application-side functionality and the grey boxes detail the library's functionality.

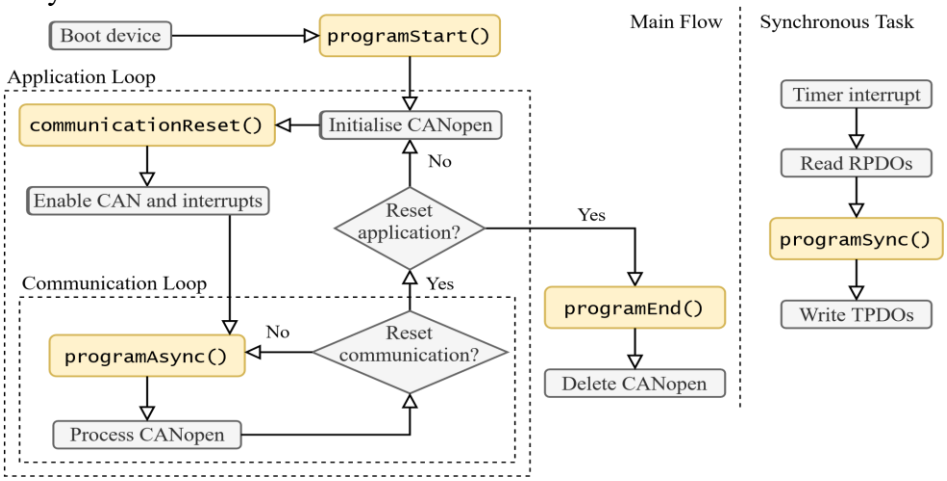

Figure 5: Application flowchart.

The *programStart* method is called only once in the program execution and can be used for device set-up and variable initialization. There are two loops in the *Main Flow*: the communication loop and the *Application Loop*. The *Communication Loop* executes continuously unless an error condition causes a communication reset to occur, in which case a new iteration of the Application Loop is executed. The *communicationReset* method is called at the start of each iteration of the Application Loop, and initializes the variables related to the communications and timing. If there are no communication errors, *communicationReset* will be invoked only once. The *programAsync* method, which should process the application's blocking tasks, is called every communication cycle and receives the time difference (in microseconds) from the previous call. When the application is reset the *programEnd* method is called to deallocate all the application's objects from the memory, followed by the deallocation of the CANopen objects and the program's exit.

The synchronous task is executed cyclically with a 1ms interval by default. The interval is set in the device-drivers and may be increased or reduced depending on each device's capabilities and

communication needs. The synchronous task processes the incoming Receive PDOs (RPDOs), calls the *programSync* method, and processes the outgoing Transmit PDOs (TPDOs). If the device's platform supports threads, the synchronous task may be executed as a thread. Otherwise, the task may be called by interrupt if proper care is taken to ensure that the *programSync* method is non-blocking, or be called in the main flow's communication loop at fixed time intervals if *programAsync*'s processing time is considerably less than the synchronous task interval.

5.3 Object Dictionary

The OD is a set of objects. Each object has a unique index between 1000*h* and FFFF*h*. An object can be either:

- A *variable* with a user-defined datatype.
- An *array* of sub-objects of the same datatype with sequential sub-indexes between 01*h* and FF*h*.
- A *record* of sub-objects of different datatypes with sequential sub-indexes between 01*h* and FF*h*.

The OD is split into three sections:

1. *Communication Specific Parameters* (Indexes 1000*h* trough 1F80*h*) – communication-related definitions, such as bus status and PDO mapping.
2. *Manufacturer Specific Parameters* (Indexes 2100*h* trough 2402*h*) – node identifier and other device-dependent parameters, such as bitrate, temperature, and voltage.
3. *Device Profile Specific Parameters* (Indexes 6000*h* trough FFFF*h*) – application-specific parameters, such as telemetries and control instructions. The expected entries on the Communication Specific Parameters and the Manufacturer Specific Parameters sections are specified in the CiA 301 standard [5].

The OD is defined in a C source file and a header, CO_OD.c and CO_OD.h respectively. These files are automatically generated with the EDSSharp v0.8 tool by Robin Cornelius [6]. The tool outputs an OD for CANopenNode. The EDSSharp ECSSopenNode Patch tool is used to patch the output for ECSSopenNode compatibility.

4.4 Stack Configuration

The library requires the definition of a set of macros for the configuration of each feature: Network Management (NMT), Service Data Object (SDO) client and server, Heartbeat producer and consumer, PDO, SYNC, Time Distribution, and automated test execution. These definitions are either placed in the CO_driver_target.h header or a dedicated header included in CO_driver_target.h. The documentation for the stack configuration is available under the CO_driver.h Doxygen documentation.

4.5 Device-Drivers

The ECSS-E-ST-50-15C library was designed to be backwards-compatible with standard CANopenNode device-drivers. Nevertheless, modifications to the device-drivers are required to make use of the parallel bus access redundancy. The CANopenNode developers provide a template for developing device-drivers, in the example/ directory, a header declaring the necessary data-types and methods in the library stack itself, 301/CO_driver.h, as well as several implementation examples.

The OD may have persistent objects defined by the user, which are to be stored in non-volatile memory (EEPROM).

The device-drivers comprise the following files:

- *CO_driver_target.h* – declares the device-specific data-types and macros.
- *CO_driver.c* – has the implementation of most of the methods declared in the *CO_driver.h* header.

- *eeprom.h* and *eeprom.c* – Application Programming Interface (API) for storing persistent data in the device's EEPROM. The provided example is composed of stubs which store the data in the heap. Since persistent storage is outside the scope of this work, the example EEPROM API is sufficient.
- *main.c* – the application's main function performs device-specific initialisations, and calls all of the necessary library methods and application methods.

The required device-specific data-types are:

- *CAN module* (CO_CANmodule_t) – a single object is instantiated at start-up. The object stores all data associated with the CAN controller, including Tx and Rx buffers, and error status flags. Depending on the device, the structure may store the base pointer to the CAN controller's registers.
- *Received message buffer* (CO _CANrx t) – an array of these objects is instantiated by each module at start-up, and each object stores the identifier filter for a type of message and a pointer to a callback function. When a received message's identifier matches one of the buffers' filter, the object's callback function is called.
- *Transmit message buffer* (CO_CAN_tx_t) – akin to the received message buffer, an array of these objects is instantiated at start-up, each associated with a different functionality and a pre-defined message identifier. Each module stores pointers to its corresponding buffers. When a module intends to send a frame, it writes the data to the appropriate buffer's data field and requests the transmission to the device-driver.

The implementation details of the above data-types are device-specific, thus there may be additional structures declared in the *CO_driver_target.h* header, as well as additional fields in the listed structures, as needed. The required methods to be implemented in are presented in Table 3, and are similarly device-specific, often requiring additional methods, such as interrupt handlers.

The structure of the device-drivers reflects the structure of the framework's modules. Each module has a header in which data-types, macros and function prototypes are declared, and an object in which the methods are implemented. Each module has a structure that stores its instances' state, an initialisation method, and a cyclically-called processing method. In the device-drivers, these elements correspond to the CAN module, *CO_CANmodule_init* and *CO_CANmodule* process respectively.

The device-driver template implements a mechanism for sending frames asynchronously when the outgoing buffers are full, which should be replicated for all implementations if possible. When *CO_CANsend* is called, the method checks whether the CAN controller's Tx buffer is full. If not, the method writes the frame to the buffer. Otherwise, the method keeps the frame in the module's buffer. Then, the interrupt handling routine sends the frame when an end-of-transmission interrupt is received.

For incoming frames, the device-driver filters the identifier and calls the appropriate callback function. However, if the CAN controller supports identifier filters and its Tx buffer is the same size or larger than the received message buffer array, it is possible to use the controller's filters and associate each controller's buffer position to each of the received message buffers, optimising the message receiving process. This feature was not implemented for the device-drivers developed in this work.

6. VALIDATION AND CHARACTERIZATION

This section presents an assessment on the performance, effectiveness, and testing methodology of SpaceCANvas. It starts with basic communication tests to guarantee all configurations and connections are correct, and later, stress tests to determine the limits of the demonstration apparatus.

6.1 Processing Platforms

The test of the platforms' built-in CAN controllers and their interfaces starts with a loopback connection inside the controller, and later on a loopback connection with the SpaceCANvas Shield. Xilinx and Atmel provide BSPs and example applications for their respective platforms [26]. These applications use a single CAN controller in loopback mode, in which the controller automatically transfers frames from the Tx buffer to the Rx buffer. An application that sends a message via the first controller and receives it via the second controller was developed for each of the platforms by calling the transmission and reception methods on different controllers. The controllers were connected externally with the SpaceCANvas Shield, and tested with Figure 6 and Figure 7's scenarios, thus simultaneously testing the software, controllers and shields. The Zynq-7000 application, can example0, is available on the CANopenArduZynq repository. The Atmel SAM V71 application is available on Github [11].

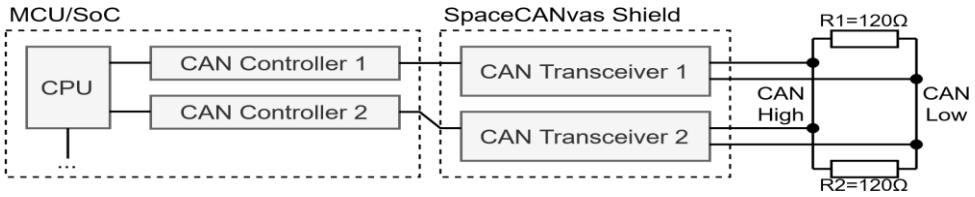

Figure 6: Loopback testing network schematic.

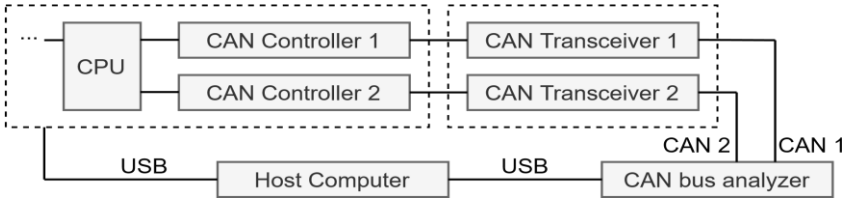

Figure 7: Single-node CAN bus analyser testing network schematic.

The maximum baudrate can be assessed after assuring the communication is established between all components on the CAN. The baudrate is selected by defining the bit duration in reference to the CPU clock by adjusting a set of parameters: the *first time segment*, the *second time segment*, and the *baudrate prescaler*. A bit is split into two *Time Segments*, in which the length of the segments is defined in *Time Quantas*. The Time Quantas' length depends on the value of the *baudrate prescaler*, which is defined as the number of clock cycles per Time Quanta. Bus state switches occur during the first Time Segment, and the sampling of the bus state occurs between the two Time Segments [22]. The Time Segments must be defined to ensure that the controller samples the bus state after the state switch has finalized. There can be several combinations of parameter values for each desired baudrate, and the optimal values depend on the latency caused by the bus cables and the controllers in use.

6.2 SpaceCANvas Shield

The CAN Transceiver Shield should be electrically and functionally tested independently. Stress tests were performed by sending a constant stream of CAN frames from the development board to the CAN bus analyser for thirty minutes, followed by constant stream of CAN frames from the CAN bus analyser to the development board. An oscilloscope can be used to monitor the CAN signals.

6.3 ECSS-E-ST-50-15C Library

The library was tested for its communication capabilities with the aid of the CAN bus analyser [27] and the Integrated Development Environment (IDE)' debuggers. At each step of the implementation, the

message transmission is verified by configuring an application to send messages in regular intervals and using the CAN bus analyser to verify the messages' contents. Message reception is verified by configuring an application to accept certain CANopen objects, transmitting the expected object with the CAN bus analyser, and using a breakpoint in the debugger to verify that the proper callback function is called and the received frames' contents are correct.

The parallel bus access redundancy is tested by performing the aforementioned experiments, and manually disconnecting one of the buses during a transmission, observing the resulting behaviour of the CAN. The bus disconnection was performed by: a) turning on a transceiver's standby; b) disconnecting one of the lines of the bus; c) disconnecting both lines of the bus. This method simulates a permanent circuit failure, which can be caused by exposure to space radiation or mechanical stresses. The application behaved as expected in all these fault scenarios. It is worth noting that short-duration disconnections of the bus result in the CAN controllers re-transmitting the dropped frames, which may result in out-of-order messages. Applications must be developed with this concern in mind.

7. SOURCE FILE REPOSITORY

All source files are shared an Apache-2.0 Open-Source License, and can be found here:
Software library: https://doi.org/10.5281/zenodo.5196584
Arduino shield: https://oshpark.com/shared_projects/8SErD3bW/

8. ACKNOWLEDGEMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under grants with references UIDB/50021/2020 (INESC-ID multi-annual funding), SARRROCA (PTDC/EEI-HAC/31819/2017), and BI| 2021/128 (research grant).

9. REFERENCES

[1] Arduino. CAN-BUS Shield v2. [Online]. Available: https://store.arduino.cc/can-bus-shield-v2. Accessed: 2021-05-10.
[2] M. Brada, Y. Isasi, and C. Molon-Noblot. The ESAIL mission: a CANopen high-performance platform. In *CAN in Space Workshop 2019*, June 2019.
[3] CANopenNode. CANopen stack base. [Online]. Available: https://github.com/ CANopenNode/CANopenNode. Accessed: 2019-11-30.
[4] M. Caramia. CAN @ Thales – Beyond Exomars. In *CAN in Space Workshop 2016*, Mar. 2016.
[5] CiA. CANopen application layer and communication profile. Standard CiA 301 version 4.2.0, CAN in Automation (CiA) e. V., Feb. 2011.
[6] R. Cornelius. EDSSharp. [Online]. Available: https://github.com/robincornelius/libedssharp. Accessed: 2021-02-15.
[7] ECSS Secretariat. Space Engineering – SpaceWire - links, nodes, routers and networks. Standard ECSS-E-ST-50-12C, ECSS – European Cooperation for Space Standardization, July 2008.
[8] ECSS Secretariat. Space Engineering – CANbus extension protocol. Standard ECSS-E-ST50-15C, ECSS – European Cooperation for Space Standardization, May 2015.
[9] ECSS Secretariat. Space product assurance – Software product assurance. Standard ECSSQ-ST-80C Rev. 1, ECSS – European Cooperation for Space Standardization, Feb. 2017.
[10] Fritzing. Fritzing desktop application. [Online]. Available: https://github.com/fritzing/ fritzing-app. Accessed: 2020-03-18.

[11] D. Galvão. Atmel SAMV71 MCAN@1Mbps Example Project. [Online]. Available: https: //github.com/duartegalvao/SAMV71-MCAN-1Mbps-Example. Accessed: 2021-01-25.
[12] D. Galvão. CANopenNode on ATSAMV71. [Online]. Available: https://github.com/ duartegalvao/CANopenSAMV71/tree/ecss. Accessed: 2021-07-03.
[13] D. Galvão. Getting Started with an ArduZynq Board. [Online]. Available: https://github. com/duartegalvao/ArduZynq-Tutorials. Accessed: 2019-09-19.
[14] D. Galvão. ECSS-E-ST-50-15C Compliant CAN for Space Systems – Open-Source HW and SW Framework for Microcontrollers and Systems-On-a-Chip. Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, 2021.
[15] M. Glorieux, A. Evans, T. Lange, A. In, D. Alexandrescu, C. Boatella-Polo, R. G. Alía, M. Tali, C. U. Ortega, M. Kastriotou, P. Fernández-Martínez, and V. Ferlet-Cavrois. Single-Event Characterization of Xilinx UltraScale+® MPSOC under Standard and Ultra-High Energy Heavy-Ion Irradiation. In *2018 IEEE Radiation Effects Data Workshop (REDW)*, pages 1–5, July 2018.
[16] ISO/IEC JTC 1/SC 7. Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Standard ISO/IEC 25010:2011, ISO - International Organization for Standardization, Mar. 2011.
[17] Lely Industries. Lely core libraries. [Online]. Available: https://gitlab.com/lely industries/ lely-core. Accessed: 2019-11-30.
[18] Marel Technology Centre. openCANopen – Open Source CANopen Stack. [Online]. Available: https://github.com/marel-keytech/openCANopen. Accessed: 2019-11-30.
[19] Mercedes-Benz USA. Controller Area Network – CAN. Technical Report 507 HO CAN B (ICC) 10-28-02, Mercedes-Benz USA, LLC, Oct. 2002.
[20] P. Poulakis, J. Vago, D. Loizeu, C. Vicente-Arevalo, R. McCoubrey, J. Arnedo-Rodriguez, B. Boyes, S. Jessen, A. Otero-Rubio, S. Durrant, G. Gould, L. Joudrier, Y. Yushtein, C. Alary, E. Zekri, P. Baglioni, A. Cernusco, F. Maggioni, R. Yague, and F. Ravera. Overview and development status of the exomars rover mobility subsystem. In *ASTRA 2015 – 13th ESA Workshop on Advanced Space Technologies for Robotics and Automation*, May 2015.
[21] E. Rapuano, G. Meoni, T. Pacini, G. Dinelli, G. Furano, G. Giuffrida, and L. Fanucci. An fpga-based hardware accelerator for cnns inference on board satellites: Benchmarking with myriad 2-based solution for the cloudscout case study. *Remote Sensing*, 13(8), 2021.
[22] S. Robb. CAN Bit Timing Requirements. Technical Report AN1798, Freescale Semiconductor, Inc., East Kilbride, Scotland, 1999.
[23] A. Scholz, T.-H. Hsiao, J.-N. Juang, and C. Cherciu. Open source implementation of ECSS CAN bus protocol for CubeSats. *Advances in Space Research*, 62(12):3438–3448, Dec. 2018.
[24] K. Schwarzenbarth. LuxSpace CAN activities. In *CAN in Space Workshop 2016*, Mar. 2016.
[25] E. Tisserant, F. Dupin, and L. Bessard. *The CanFestival CANopen stack manual.*, v3.0 edition, Jan. 2019.
[26] Xilinx. Xilinx SDK Drivers API Documentation – canps. [Online]. Available: https://xilinx. github.io/embeddedsw.github.io/canps/doc/html/api/index.html. Accessed: 2019-09-15.
[27] Zhuhai Chuangxin Technology Co., Ltd. *USB-CAN Bus Interface Adapter & CANalyst-II Analyzer Manual*, 2017.