NTNU
Norwegian University of Science and Technology

TTK4550 - SPECIALIZATION PROJECT

INTEGRATION OF A NETWORK STACK ON A NANO-SATELLITE PAYLOAD

Author:
Magne Hov

December 18, 2018

Supervisor 1: Tor Arne Johansen
Supervisor 2: Milica Orlandic
Supervisor 3: Roger Birkeland

NTNU
Norwegian University of
Science and Technology

Abstract

The SmallSat Laboratory at the Norwegian University of Science and Technology is developing a Hyperspectral Imager Payload for the HYPSO nano-satellite mission. This work presents the payload network stack consisting of a Controller Area Network bus link and the CubeSat Space Protocol.

The details of CAN-bus and CSP are explained, before multiple internal communication architectures are evaluated for integration into an Embedded Linux system. Architectures are implemented and tested before one is suggested as a proposed solution.

Acknowledgement

I would like to give my thanks to the HYPSO project for the opportunity to work with the communication architecture of the Hyperspectral Imager Payload. The opportunity to use the working areas of the SmallSat Laboratory at the Norwegian University of Science and Technology is highly appreciated.

I extend my gratitude to my supervisor Tor Arne Johansen, and my co-supervisors Milica Orlandic and Roger Birkeland for their general supervision, helpful comments, and help in reviewing this document. Their contributions have been important the completion of this work.

Contents

1 Introduction

1.1 Problem description

Through the project work, the student shall integrate the network protocol Cubesat Space Protocol (CSP) and a Controller Area Network (CAN) link into the software stack of the Hyperspectral Imager (HSI) payload system for the HYPSO nano-satellite mission.

1.2 Small Satellites

The names *minisatellites*, *microsatellites* and *nanosatellites* are given to categories of human made satellites that fit within certain mass ranges. Collectively, these categories can be referred to as small satellites, emphasising their size relative to traditional satellites that perform heavy tasks such as radio communication, global navigation systems and earth observation. The range specifications range 100 – 500kg for minisatellites, 10 – 100kg for microsatellites and 1 – 10kg for nanosatellites [1].

Small satellites are much cheaper to produce and operate than traditional satellites for several reasons. The smaller mass accounts for less material to produce, procure and validate. However, not being able to put as many subsystems into the satellite is also one of the main disadvantages. Smaller satellites do often not have the necessary space or mass to include propulsion systems.

The lower weight means that a smaller launch vehicle with less fuel can be used to place the small satellite in orbit. However, the largest savings come from being able to piggy-back as secondary or tertiary payloads on launch vehicles that carry larger primary payloads.

The build time of a satellite tends to be proportional to its size. Some small satellite projects use the low cost and low build time to justify lighter requirements for verification and validation, and the use of Commercial off-the-Shelf (COTS) components. Fire-and-forget seems to be a common approach, where a project develops a small satellite following a short and strict timeline. In the case that the mission fails, it is considered more valuable being able to reflect back on the previous design and iterate with a new small satellite. This is only possible with the low cost and build time associated with small satellites.

Low cost makes small satellites an attractive option for low budget projects, and the increasing availability of launch options for small satellites has aided the small satellite community in growing considerably the last few years. Industry actors who previously did not have the funds or skills to develop satellites may to a greater extent rely on small satellites to reach their goals. Small satellites are a driving factor for the democratisation of space [2].

Planet Labs is an example of a company reaching new goals with small satellites. They have been able to create a constellation of over 150 nanosatellites that together produce an image of the entire earth at least once a day [3].

1.3 The HYPSO mission

The Centre for Autonomous Marine Operations and Systems (AMOS) at The Norwegian University of Science and Technology (NTNU) develops technology for autonomous maritime operations and control systems, in order to meet the challenges related to the coasts and the oceans. The centre places an emphasis on autonomous vehicles, ocean structures and advanced navigation and guidance systems to fit the needs of scientists, fisheries and industries.

Various robotic agents for underwater, in-air and on-surface operations have already been developed through AMOS-related activities to collect climate data from maritime environments.

The SmallSat Lab at the NTNU is a project group working to build and launch nano-satellites. The current mission, Hyperspectral surveillance of the oceans (HYPSO), aims to supplement these robotic agents with satellites. The mission carries a Hyperspectral Imaging (HSI) camera to enhance the surveillance of coastlines, and a Software Defined Radio (SDR) to aid communication in the arctic.

By placing the satellite in a near-polar orbit, one achieves a coverage and revisit time that is expensive and impractical to match by using surface and in-air robots. The hyperspectral camera onboard the satellite will be able to capture images of remote locations without having to deploy robots on lengthy missions. Additionally, the satellite will have a low revisit-period, meaning that changes in a specific area can be surveyed over time, without having to deploy multiple surface or in-air missions to the same area.

1.3.1 Hyperspectral Imaging

The HSI camera captures individual light intensities throughout the colour spectrum, as opposed to only a few select bands as in traditional cameras. Hyperspectral images contain two spatial dimensions and one spectral dimension, and are often depicted as cubes. This is illustrated in Figure 1.

The data that is captured by the HSI camera is stored and processed by

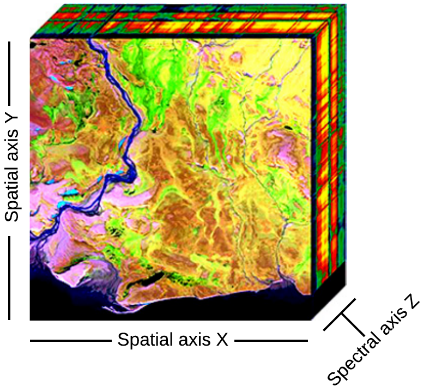


Figure 1: Graphic representation of hyperspectral data [4].

an image processing system. After processing, the data needs to be transferred to a ground station over a radio link. The radio link is typically very restricted in bandwidth for small satellites due to limitations in available transmission power. The hyperspectral imager produces large data sets that are of considerable size even after compression. As an example, a raw hyperspectral image can easily occupy several gigabytes of storage. Given a S-band radio link with a data rate of 512 Kbps [5] it would take 35 minutes to transfer a single image of one gigabyte.

1.3.2 Satellite platform

The HYPSO mission uses a commercially available satellite bus from *NanoAvionics*. In the remainder of this paper, the term *satellite bus* is taken to mean the collection of subsystems that support the payload systems in performing their functions. This includes mechanical structures, electrical power systems, communication systems and more.

The M6P satellite bus from *NanoAvionics* provides a mechanical frame that is compliant with the 6U CubeSat Design Standard [6]. This means that the volume and shape of the satellite must follow these specifications, and that the maximum weight of the entire satellite must be kept within 12 kg, including payloads and supporting subsystems. By following the CubeSat Design Specification [6], the satellite is compatible to be released with standardised deployers, which increases the availability of launchers, and decreases the required efforts to successfully integrate the satellite into the launch vehicle.

The subsystems present on the M6P satellite bus, including payloads, are outlined below. The physical architecture is illustrated in Figure 2.

- Flight Computer (FC), which is tasked with generating and sending telemetry data, configuring an UHF radio, as well as housing the ADCS (see next entry).
- Attitude Determination and Control System (ADCS), tasked with gathering navigation data from various sensors systems, including sun sensors, magnetometers, a startracker and a Global Navigation Satellite System (GNSS) receiver.
- Electrical Power System (EPS), which receives power from solar panels to charge batteries, and provides both regulated and unregulated voltages to the other subsystems.
- Payload controller (PC), which acts as a Cubesat Space Protocol (CSP) communication bridge between the payloads, primary radio and the rest of the satellite. It is able to buffer payload data before forwarding it to the radio when transmitting it to the ground [7].
- S-Band Radio, primary radio, intended to provide relatively high bandwidth uplink and downlink.
- Ultra-High Frequency (UHF) Radio, a secondary radio, which can be configured to send a beacon signal with telemetry data, and to receive from, and forward CSP packets to ground stations.
- HSI Payload, consisting of a Breakout Board (BB) with a processing module which connects to a HSI camera and a Red-Green-Blue (RGB) camera.

When an image is requested to be downloaded from the HSI payload, the payload sends the image over a communication network that connects all communicating subsystems, including the ground station and other supporting systems on the ground. In the M6P satellite bus, this network is implemented using the open source network library CSP [9]. In order to comply with the M6P satellite bus, the HSI and SDR payloads are required to communicate with the satellite bus using the CSP library.

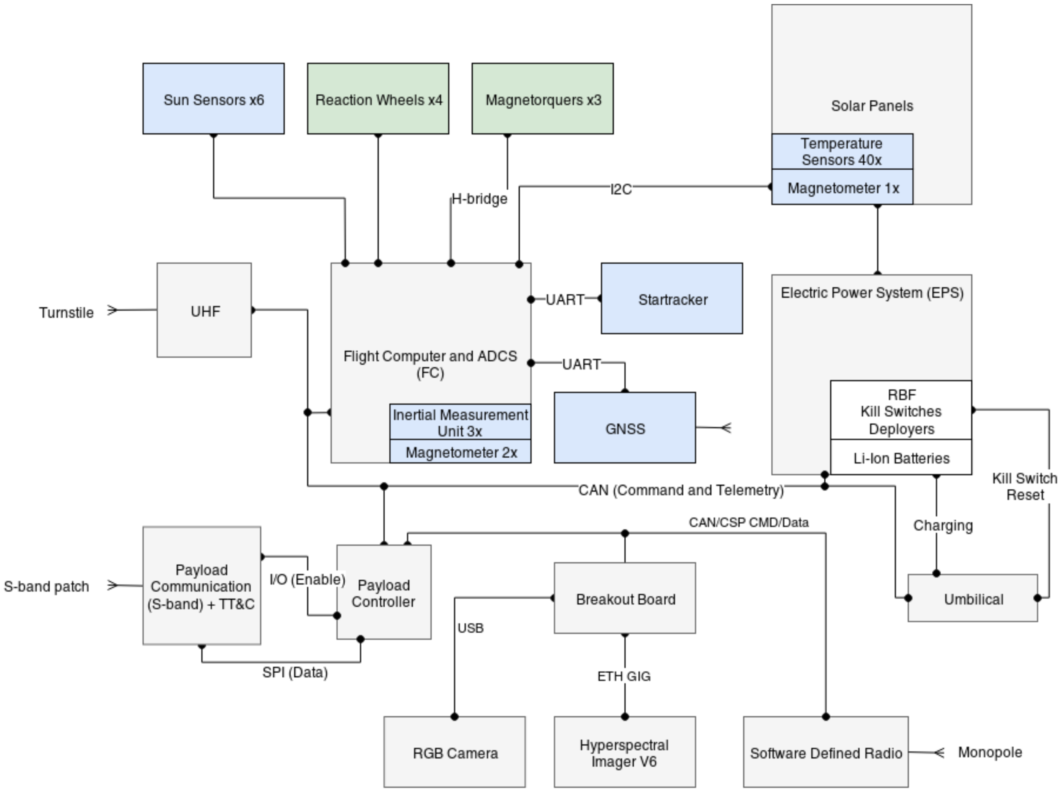


Figure 2: M6P Physical Architecture. Adapted from NanoAvionics M6P Platform ICD [8].

1.4 Report Outline

The remainder of this report details the work of investigating, integrating and testing CSP for the HYPSO HSI payload. The following section 2 explains constraints, concepts and modules that are used in the work, providing a background of knowledge on which the work can be reviewed and understood. Thereafter, section 3 presents the external network architecture, before internal architecture designs are evaluated. The term external architecture is taken to mean the interfaces that connect the payload with neighbouring subsystems, while internal architecture is taken to mean how the software services access these interfaces. Multiple architectures are tested and profiled in section 4, where the report also discusses issues, and suggested improvements. A conclusion is finally given in section 5.

2 Background and Context

This chapter discusses concepts central to the HSI payload. First, the constraints and requirements of the HYPSO project are explained. Then the network components of the Payload platform are presented.

2.1 Project Constraints and Requirements

The HYPSO project employs a System Engineering methodology based on systems engineering standards from European Cooperation for Space Standardization (ECSS). Mission objectives, mission requirements, system requirements, and project constraints are organised in a hierarchical fashion. High level objectives and requirements are refined into increasingly detailed requirements at the derived system and subsystem levels.

The HYPSO project requirements are kept in centralised documents that always are kept up to date with the newest requirements. The requirements that are relevant to this work have been included here to form a basis of evaluation for proposed solutions. See Table 1.

Requirement IF-001 is derived from the fact that the HYPSO project has procured a M6P satellite bus [8] from *NanoAvionics*. This limits which protocols can be used to communicate with the remainder of the M6P satellite bus. As illustrated in Figure 2, the HSI Payload shall be connected to the PC, and must communicate through this subsystem whenever it needs to communicate with the rest of the satellite bus, or with the ground station. Therefore, the HSI payload system is required to use a Controller Area Network (CAN)-bus to communicate. Furthermore, the PC employs CSP on

Table 1: HSI Payload requirements that are relevant for communication.

Req. ID	Definition
IF-001	The payload shall comply with NanoAvionics mechanical and software ICD
M-1-015	L1A data product shall be downlinked in less than 24 hrs and be ground truthed
M-1-016	Operational data shall be downlinked and ground truthed in less than 3 hrs
M-2-026	S/C shall communicate to ground and downlink housekeeping telemetry data of up to 200 kb for at least 1 pass per day
SYS-3-007	Data rate from payload computer (FPGA) to radio should be at least 0.8 Mb/s (Serial or CAN protocol)
SYS-3-016	S/C software should be sufficiently open (e.g. SDK, API, protocols, open source) to enable the user to configure the S/C and mission, develop own code, and integrate payload.
SW-4-006	Shall be able to output level 1a and level 4 data
SW-4-007	Shall provide a file transfer service that can send and receive file reliably. Transfers must be able to continue over multiple sessions or passes.

top of the CAN-bus link to perform packet routing, meaning that the HSI payload also is required to use CSP for communication.

The mission requirements M-1-015, M-1-016 and M-2-026 regard the speed at which HSI data must be downlinked from the satellite. To fulfil the requirements, the communication from the HSI Payload to the downlink radio must be able support a minimum effective data rate. The most strict of theses three requirements is to downlink a L1A image in 24 hours. The HYPSO Mission Scenario budget [10] estimates a total downlink duration of 5394 s over the span of one day. The L1 data sets are estimated to a size of 420 MB [10]. Using these estimates, the minimum needed data rate is roughly 81 Kbps.

System requirement SYS-3-007 states that the data rate between PC and the radio subsystem should be at least 0.8 Mbps. This is a requirement of the M6P satellite bus, but implies that the data rate between payload and PC also must be at least 0.8 Mbps, or that the internal buffering capabilities of the PC must be utilised. The CAN-bus has a maximum bitrate of 1 Mbps. Furthermore, subsubsection 3.1.2 will illustrate how the effective data rate of the CAN bus will be considerably lower than 0.8 Mpbs. This already implies that to in order to fulfil requirement SYS-3-007, the buffering capabilities of the PC must be used.

Requirement SYS-3-016 is a requirement for the M6P satellite bus, however, it is also taken into consideration when designing the internal communication architecture in the HSI Payload. Multiple developers will be concerned with the communication interface when developing services and processing applications for the HSI payload, therefore, the CSP communication interface should be integrated such that it is as easy as possible to use for service developers.

Requirement SW-4-006 requires the HSI payload to be able to communicate at least two types of image data. Level 1A data is raw data, with attached metadata and no non-reversible processing applied to it [11]. One image of level 1a data is estimated to occupy roughly 450 MB, according to the HYPSO data budget [10]. The level 4 data is heavily processed and compressed, estimated to occupy only 6MB. For transmitting both of these, a dependable communication path is required. Therefore, together with requirement SYS-3-007, there is a need to verify that the communication link can operate with a high bus utilization over an extended period of time.

Requirement SW-4-007 demands that a reliable file transfer service shall be provided. For this to be possible, the CAN and CSP link from HSI payload to PC and forward to the radio must be working. Furthermore, some mechanism must be implemented to guarantee that packets are reliably delivered, for example through retransmission of lost and corrupted packets. This requirement is considered to be the responsibility of the upper application layers, and is therefore out of scope for this report.

2.2 HSI Payload Hardware

The HSI Payload requires hardware capable of processing large amounts of image data, while simultaneously being able to communicate with other satellite subsystems. System on Chips (SoCs) combine processing units with communication and control peripherals in a single chip, providing highly flexible and capable processing systems. System on Modules (SoMs) combines even more features, like external transceiver circuits, memory banks and data storage units, on a small Printed Circuit Board (PCB), to provide even more functionality out of the box.

When building systems with ready-built SoMs, a connector board must necessarily be built to break out pins, routing them from the SoM to more manageable connectors. However, most of the electrical design work is already finished on the SoM, and the core development can be focused on making software applications instead of building hardware. Therefore, building systems with SoMs is beneficial for projects with a tight schedule.

The HYPSO Payload is equipped with a PicoZed SoM from Avnet, which itself is equipped with a Zynq 7000-series SoC from Xilinx. This SoC integrates a Processing System (PS) based on ARM Cortex-A9 cores together with a section of Programmable Logic (PL) [12]. The PS has two levels of caching, 256 KB On Chip Memory (OCM), support for external DRAM, and several interfaces for external static memory. The SoC offers a selection of external peripherals such as a SD-card (Secure digital) controller, gigabit Ethernet, CAN-bus, Universal Serial Bus (USB), General-Purpose IO (GPIO), Universal asynchronous Receiver-Transmitter (UART), Inter-Integrated Circuit (I2C) and Serial Peripheral Interface (SPI). The PS and PL can interface each other through a shared memory bus, and can exchange data using one of the chip's 8 Direct Memory Access (DMA) controllers through Advanced Extensible Interface (AXI) interfaces. The PicoZed SoM provides 1 GB of Double Data Rate Type Three Low Voltage (DDR3L) memory, 8 GB of Embedded MultiMediaCard (eMMC) storage, a USB 2.0 transceiver and a gigabit Ethernet transceiver [13].

During development of the HSI Payload, a ZedBoard development kit from Avnet is used instead of the PicoZed SoM. The ZedBoard development kit is fully integrated on a single PCB and has transceivers and connectors for almost all peripherals that are required by the HSI Payload. This board allows development to begin before a breakout board has been created for the PicoZed SoM.

2.3 Operating system

When designing an embedded system in general, the choice of whether to use an Operating System (OS) and which one to use is an important decision. The choice of OS is reflected in the amount of programming tools and documentation available to the software designer. There exist various categories of OS, depending on the type of tools they provide, which abstraction level and platforms they target, and whether they deal with real time requirements.

An OS takes care of tasks such as scheduling, memory management, low level IO and security [14]. They provide a uniform interface on which applications can be built, without having to worry about specific implementations of resources, drivers and interfaces.

Many embedded systems perform tasks with hard or firm deadlines. In order to cater for these requirements, an OS must provide real time capabilities. Real time scheduling algorithms can be used to prove the timely execution of critical tasks.

A well established general-purpose OS such as those based on the Linux kernel, have available a vast collection of software. This is a great help for the developer who is able to use existing libraries and drivers.

Embedded systems usually require a smaller set of the available software resources than the general-purpose computer. In this cases, the plethora of software modules that come shipped with many distributions becomes bloatware that slows down the system, uses more memory than necessary and can increases the total complexity of the system. Additionally, the extra software means that more code needs to be verified. However, many of the available resources are open-source and the OS can be adapted to fit a specific application [14].

From a development point of view, the large amount of software available for an OS such as Linux makes it an attractive platform to work on. Although one runs the risk of ending up with a bloated system if using too many general purpose libraries, it is often possible to strip the system down to its bare essentials right before deploying it. In this way, a larger set of software tools may aid the development process, while not interfering with the performance or requirements of the final product.

Embedded systems often favour a light weight OS because of limited resources and ease of verification due to their simplicity. These are often not as established as the general purpose OS, and as a result they do not offer the same amount of libraries and ready made software modules. An example is FreeRTOS which provides several real time scheduling algorithms [15]. It provides a simple tasking interface, as well as a few OS primitives such as mutexes, semaphores, queues and timers. FreeRTOS is a popular OS for CubeSats, with several CubeSat-specific systems offering it as a kernel. An example is *GomSpace's* NanoMind A712D On-board Computer (OBC) [16].

2.3.1 Board Support Packages

SoCs and SoMs generally have a large amount of peripherals that need to be initialised before they can be used. To mitigate the setup cost related to writing all of these drivers, SoM producers usually provide board support packages for their modules. The Board Support Package (BSP) aids in initialising hardware components such that they are visible and accessible to an OS. These packages can save hours of work in just getting a minimal system up and running and lets the developer focus on applications.

2.3.2 PetaLinux

Xilinx maintains an embedded Linux build system and reference distribution called PetaLinux. This framework employs project configuration files that allow components to be included or excluded from the Linux kernel, including custom built modules. The build system also provides a generous amount of utility programs, libraries and drivers that can be compiled into a root file system that works together with the compiled Linux kernel. The ZedBoard and PicoZed both have BSPs for PetaLinux, released by Avnet. Additionally, the PetaLinux build system makes available tools for modifying and creating BSPs for custom boards.

The PetaLinux build system is based on tools developed by the Yocto project [17]. This open-source project aims to provide better support in creating Linux distributions for embedded platforms. It is made up of build tools that configure and build Linux kernels and packages to deploy on top of those system. It provides a reference configuration of an embedded Linux system, and allows for heavy customisation. It can cross build for a large selection of targets.

The PetaLinux OS was chosen for the HSI Payload because it is natively supported on the ZedBoard and PicoZed platforms. Additionally, the provided BSP and the large amount of available drivers and libraries save development time. The PL section on the Zynq 7000 Series SoC is capable of running real-time specific modules, if such a requirement should appear.

2.4 Previous work with CSP at NTNU

There has been previous efforts to use CSP for the NTNU Test Satellite (NUTS) project [18] at NTNU.

In 2012 Muench, Marius developed a Hash-Based Message Authentication Code (HMAC) scheme called the NUTS Authentication Protocol to provide authentication of telecommands for the NUTS project [19]. The protocol was formally verified, implemented and tested, but never integrated back into the NUTS project.

In 2015 Jahren, Erlend Riis developed a reliable transport protocol (NUTS Reliable Protocol) after concluding that the reliable transport protocol (Reliable atagram Protocol (RDP)) which is included in CSP was non-functional. The NUTS Reliable Transport Protocol was concluded to be mostly working with only a few unaddressed issues. However, no further efforts were made to fully integrate the protocol into the NUTS software architecture.

In 2016 Normann, Magne Alvar made a Software Design for an Onboard-Computer [20], for a ARM based platform running FreeRTOS. The proposed software architecture is properly justified, and parts may be adapted for the HSI Payload software architecture.

2.5 Alternative to CSP

Although the payload system is programmatically required to use CSP as its network and transport protocol, it is worthwhile investigating alternative protocols. Comparison of these might reveal areas where CSP falls short, and can indicate what protocols and features might be implemented on top of CSP.

The Jet Propulsion Lab (JPL), California, has implemented a Delay Tolerant Network (DTN) called *Interplanetary Overlay Network*. Rather than being a single network protocol, it is a collection of protocols that enable a network to span the entire solar system [21]. This implementation makes heavy use of zero-copy techniques to avoid rewriting the contents of packets. This is achieved by using memory sharing between processes and by passing references to object buffers instead of copying the buffer contents.

2.6 Coding Standards

In order to provide some amount of consistency throughout the code produced in this project, the C coding style from the Linux Kernel project was adopted. By installing a style definition file into the project, the `clang-format` tool from the Clang Project [22] can be used to automatically format the source code.

2.7 Testing

Verification and validation are two development steps that are very important for the success of a project, but are often skimped on or skipped in small satellite projects. Many small satellite projects don't have resources to support a full verification and validation program. A recent survey by Jacklin [23] details methods that are often employed in small satellite software development. Simulation and testing is said to be the technique that is most often applied, with model-based design being a technique which is gaining popularity.

3 Design

In this section, the lower layers of the the network stack are presented in detail in subsubsection 3.1.1, subsubsection 3.1.2 and subsubsection 3.1.3. Different communication architectures using CSP are presented and discussed in subsection 3.2.

3.1 Network Architecture

When discussing networks it is helpful to have a common classification of modules and protocols. There exist protocols that perform the same conceptual function, while the way in which they work and how they are implemented may be completely different. These should be categorised together to illustrate that they perform related tasks.

The Open Systems Interconnect (OSI) network model is an often referenced model on how the functionality of a communication network can be classified [24]. It delegates the various tasks and responsibilities of a communication network into layers, where higher layers have a higher level of abstraction from the physical world. The names and functions of the layers in this model are used when discussing network protocols in this report. See Figure 3 for an overview of layers, names and respective functions.

Layers		Protocol data unit	Function Examples
Host Layers	Application	Data	Services, file access, database access
	Presentation		Data encoding, compression
	Session		Handling of connections
	Transport	Datagram	Retransmission, fragmentation, acking
Media Layers	Network	Packet	Addressing, routing, traffic control
	Link	Frame	Transmission of data frames
	Physical	Symbol	Transmission of single symbols

Figure 3: Illustration of the different layers in the OSI reference model for networks, and their functions summarised.

3.1.1 Physical Layer - Controller Area Network

The physical layer is related to mechanical, electrical and wireless interfaces. Some protocols define specific mechanical connectors and put requirements on cabling and signal propagation mediums. Signal timing and electrical properties are often defined, as well as working conditions that the medium might have to withstand.

CAN is a physical and link layer protocol originally developed for the automotive industry. It is defined in a series of standards released by the International Organization for Standardization (ISO): ISO11898-1 [25], ISO11898-2 [26] and ISO11898-3 [27]. The physical layer protocol connects two or more nodes via a physical serial bus, allowing them to exchange data frames using a protocol-specific addressing scheme.

The physical bus is made up of a pair of twisted wires, with the two wires being named *CAN High* and *CAN Low*. The wires are electrically terminated with resistors. Active signalling on the bus is performed by a node pulling the *CAN High* wire to a reference voltage of 5.0 V, and the *CAN Low* wire to a reference voltage of 0 V. This represents the symbol '0', which is termed a dominant symbol. To represent the recessive symbol '1', the node stops driving the wires, allowing the wires to equalise in voltage with the help from the terminating resistors. Listening nodes will measure the voltage difference between the *CAN High* and *CAN Low* wires, and if the difference is lower than some threshold the bus state will be interpreted as a recessive symbol '1', otherwise it is interpreted as a dominant symbol '0'.

By using twisted wires and differential signalling, the bus gains protection to electromagnetic noise [28]. The twisting of the wires eliminates unbalanced induction caused by non-uniform fields that originate from components that are in close proximity to the bus. In an untwisted pair of wires, the wire closest to the source of a magnetic field will experience a stronger field, resulting in a higher induced Electromotive Force (EMF) in that wire. By twisting the wires, they alternate on being closest to the noise source, averaging out the induced electromotive forces. Now the induced voltage will be the same in both wires, only causing an elevated common mode voltage. The common mode voltage is removed from the signal when measuring the difference between the wires. In order to provide protection against large common mode voltages, CAN-bus transceiver circuits are dimensioned to be able to operate on higher voltages than the nominal 5.0 V, and to be able to survive much higher transient voltages.

There exists several variations and extensions on the physical implementation of the CAN-bus protocol, and they are targeted at different speed classes.

High Speed CAN is defined in ISO 11898-2 [26] and its extensions, and allows bandwidths of up to 1 Mbps. This variant has a linear topology, and is terminated with 120 Ω resistors at each end of the bus. Nodes can connect anywhere on the bus by electrically connecting to the *CAN High* and *CAN Low* leads. This configuration is illustrated in Figure 4.

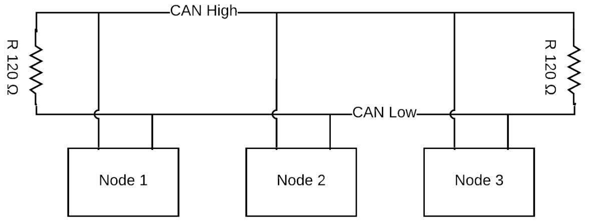


Figure 4: The physical CAN bus is terminated at each end. Nodes can connect anywhere between the termiantors.

Low Speed Fault Tolerant CAN is defined in ISO 11898-3 [27], and is limited to a bandwidth of 128 Kbps. This variant allows for a richer topology, in which linear buses, star shaped buses, and hybrids are allowed. The termination resistors are divided between the nodes of system, and must together total to the desired impedance value of 100 Ω. The variant also changes the thresholds for dominant and recessive differential voltages.

3.1.2 Link Layer - Controller Area Network

A link layer is responsible for encapsulating and transmitting units of data (often called frames) [24]. The structure of the data within the frame is specified, and additional header data is added to provide for various functions. A length header field is often used to encode the length of a frame. When multiple devices are accessing the same medium, techniques must be employed to avoid jamming the bus. There exists collision detection and avoidance techniques, as well as a wide selection of bus arbitration schemes. Error detection, signalling and recovery can also be implemented as link layer features. Some link layer protocols also include Error-Correcting Code (ECC) to improve the integrity or reliability of the link.

The CAN link layer protocol is described in ISO11898-1 [25], and defines frame format, Medium Access Control (MAC) addressing, bus arbitration, error handling and more. The frame format includes fields for identification, payload length, payload data, checksum and a few other fields used for flow control. There are two different frame formats depending on whether a 11 bit or 29 bit frame identifier is used. The frame layouts are illustrated in Figure 5 and Figure 6.

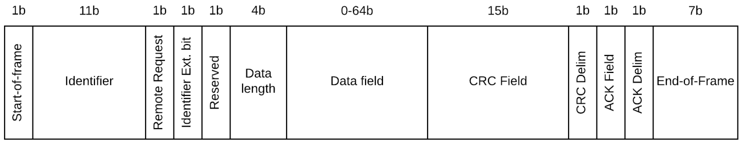


Figure 5: Illustration of the *base* CAN bus frame format, with 11 bits for identifiers. Frame format is derived from ISO11898-1 [25].

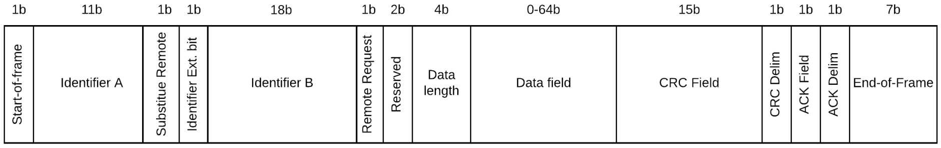


Figure 6: Illustration of the *extended* CAN bus frame format, with a 29 bits for identifiers. The identifier is split up between two fields, A and B. Frame format is derived from ISO11898-1 [25].

Addressing is performed using 11 bit or 29 bit identifiers , that simultaneously represent the priority of the frame. A lower frame identifier has higher priority, and is used to provide decentralised bus arbitration. All nodes that are physically connected to the bus (see Figure 4) must refrain from driving it as long as they measure traffic on it, meaning as long as they measure the bus being pulled to a dominant state. When the bus is perceived as free, any node may start transmitting a frame. With the exception of a start-of-frame bit which is always dominant ('0'), the identifier is the first field to be transmitted on the bus. While transmitting, the sender simultaneously samples the bus state to verify that it was successful in transmitting the symbol. Another node might initiate a transmission at the same time. Then, either of the transmitting nodes might measure the bus being in a dominant state ('0') when itself is transmitting a recessive symbol ('1'). The dominant signal overrides the recessive state of the bus. The node that is transmitting the recessive symbol will then interpret this as another node transmitting a frame with a higher priority than its own. It will therefore stop transmitting its own frame and wait until the bus is available before trying again. In this way, CAN-bus implements Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA). In the rare case where two frames with the same identifier are broadcast simultaneously, the arbitration boils down to observing which frame contains a dominant symbol earliest in the remainder of their frames.

All versions of the physical CAN protocol use a Non-Return-To-Zero (NRZ) representation for physical bit states [25]. With this technique, the state of the physical bus is not brought back to a default state between symbols. The physical bus state will stay constant through the transmission of multiple identical symbols.

The CAN-bus protocol does not employ a shared clock to dictate when nodes should sample the state of the bus. Therefore, all nodes synchronise on transitions from recessive to dominant bus states, aligning their phase to the clock of the current sender. A prerequisite for continued synchronisation is that all nodes in the network are configured to operate on the same frequency.

Because of NRZ and the lack of a sampling clock, a bit stuffing rule is used as a run-length-limiting technique to avoid bit-slips [25]. After five consecutive symbols of the same state, a symbol of the opposite state must be sent, and it is not considered part of the frame. This ensures frequent transitions on which nodes can synchronise their local clocks. If six consecutive identical symbols are sent, the bit stuffing rule is violated, and an error is signalled.

All CAN frames are tailed by a time period in which one or several receivers of the frame put the bus into a dominant state to indicate an

Acknowledgement (ACK) on receiving the frame. During this period, the sender leaves the bus in a recessive state in order to measure the transition from a recessive to a dominant state.

Cyclic Redundant Check (CRC) codes are included in the frame to allow bit errors to be detected [25]. A CRC field encodes information about the bit pattern of the remainder of the frame. Its value is computed as the remainder of a polynomial division. Most of the frame fields are included in the calculation.

When violations of ACK, CRC or bit stuffing are detected, by any node, they signal an active error on the bus by pulling it to a dominant state for at least six consecutive symbols. This should again be detected as a bit stuffing error by other nodes. In this way, an error detected by a single node can be signalled to all nodes.

Error handling is an integral part of the CAN specification [25]. Sources of errors include invalid CRCs, bit stuffing violations and corrupt frames with missing ACK. After a node has detected and signalled (six or more successive dominant symbols) a certain number of errors, it will enter a passive-error state where it may only signal errors with passive frames (six or more passive symbols). If a node then continues to detect sources of errors it enters a state where it logically disconnects, and stops participating on the bus. Most CAN controller circuits include some mechanism to recover from passive and hard error states by observing some amount of successful bus traffic.

3.1.2.1 SocketCAN Driver As part of the Linux kernel modules, SocketCAN provides a link layer interface to CAN devices. The programming interface is identical to that of the Transmission Control Protocol (TCP)/Internet Protocol (IP) network interface. The driver code initialises a Linux socket object from the networking protocol family `PF_CAN`, after which the Linux system calls `read` and `write` can be used to communicate data with the lower level CAN controller.

3.1.2.2 Transfer Rates Although the ISO11898-2 CAN-bus is able to operate at 1 Mbps, the effective data transfer rate is significantly lower than that.

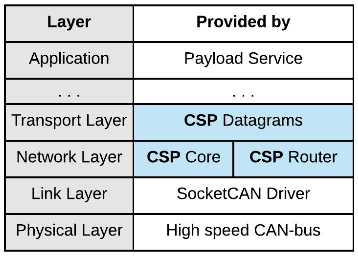
In the best case, the extended CAN frame (shown in Figure 6) is filled with a maximum number of data bytes, equal to 64 bits of data. The remaining 64 bits are used for addressing, CRC and control, meaning only 50 % of the transferred symbols are used for data. Between each frame, an interframe space consisting of at least three recessive bits must be inserted. Additionally, depending on the content of the frame, extra bits may be inserted by the bit stuffing rule, further lowering the effective transfer rate.

When only considering the 50 % frame overhead, the theoretical upper limit on the data transfer rate is 500 Kbps.

3.1.3 Network Layer - Cubesat Space Protocol

A network layer is normally responsible for tasks related to host addressing, packet forwarding and routing, and traffic control (see Figure 3).

CSP is a network library developed by *GomSpace* for use in CubeSats [9]. It performs the functions of the network layer (layer 3) and transport layer (layer 4). The role of CSP in the network stack is illustrated in Figure 7.



Layer	**Provided by**	
Application	Payload Service	
. . .	. . .	
Transport Layer	**CSP** Datagrams	
Network Layer	**CSP** Core	**CSP** Router
Link Layer	SocketCAN Driver	
Physical Layer	High speed CAN-bus	

Figure 7: Illustration of the role of CSP in a network stack using the SocketCAN Linux driver for CAN-bus.

The protocol has been adopted by several small satellite designers and has flight heritage. CubeSats such as GOMX-3 and AAUSAT3 have flown successful missions with CSP. Several small satellite vendors, for example *NanoAvionics* [29] and *GomSpace* [16] include CSP as part of their commercial platform, showing the readiness of the technology.

In many ways, CSP performs the same functions as the Internet Protocol (IP), but offers a lighter implementation which is suitable for resource limited systems like those found in nano-satellites. The similarity of the application programming interface (API) makes it easy to adopt for people who have previous knowledge of network socket programming.

The CSP network facilitates communication by allowing data packets to be sent between any two nodes in the CSP network. Each node in the network has an unique 5 bit ID number. When sending a packet, the packet is addressed with the ID number of the destination node, and a port number which indicates which service inside the destination node should receive the packet. The packet header also includes the source ID number and a source port number to let the receiver know from which node and service the packet was sent from.

Each packet is fronted by a 16 bit data length field, encoding the number of bytes in the data field. Eight flag bits encode CSP options that are enabled for that packet. Options are detailed in Section 3.1.3.5. The complete CSP frame layout, with a CRC option enabled, is shown in Figure 8.

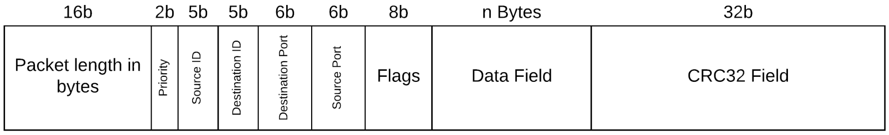


Figure 8: A CSP frame with the CRC32 option enabled.

3.1.3.1 Buffer management When creating a CSP packet, a buffer must first obtained from the CSP core. A call to `csp_buffer_get()` returns a pointer to an unused packet buffer. The pointer is used to modify the packet data field and to set the data length field. This must be done manually. All packet buffers are kept in a central bank of buffers which is allocated with `malloc()` when CSP is initialised, normally at the start of a program. By passing buffer pointers around, one circumvents the need to copy the contents of the packet between each user of the packet. This zero-copy technique reduces unnecessary duplication of packet data in memory.

3.1.3.2 Router The driver code that receive packets from the OS or lower level devices copies the packet data into a free packet buffer, and appends the packet pointer to an input queue. Packets that are inserted into the input queue must be routed to their final destination. The CSP library provides a `csp_route_work()` function, which is usually run in its own thread (`csp_route_start_task()`). This function is the single reader from the queue of packets which all receiver tasks write to. It picks out the first packet in the queue and examines the packet identifier. Packets that are addressed to other nodes are redirected to the appropriate outgoing interface. Packets that are addressed to the receiving node are appended to the CSP socket which is bound with the appropriate port number. If no such port has been bound, the packet is rightfully dropped. The router refers to a routing table to resolve which interface it must forward a packet on. This routing table is static and must be manually configured.

3.1.3.3 Required functionality and resources

CSP was developed to initially run on FreeRTOS, but has also been ported to run on Portable Operating System Interface (POSIX), Windows and MacOS. There is a set of functional resources that CSP requires to run. These must be provided by the OS, and will differ between different architectures. A list of the most important resources is shown below.

Threading Threads are required to run routing work and to receive packets from interfaces in the background.

Memory allocation Malloc is required to allocate the packet buffer bank upon initialisation of CSP.

Queues Manipulation of incoming connections and packets requires an implementation of queues to keep them in an organised structure.

Synchronisation Primitives Mutexes and semaphores are required to ensure safe access to the shared queues that hold the connections and packets.

System Services Ports 1 to 6 are reserved for system management and CSP remote configuration. Some of these services require the system to be able to supply information about the uptime of the system, the amount of free memory in the system, and to provide commands for reboot and shutdown.

Functions that are specific to the OS are declared in a compatibility layer in `libcsp/src/arch/`. Functions that are supported by the OS are wrapped and renamed to CSP-specific identifiers in order to provide a consistent interface that the library can use. When the OS lacks certain functionality, it is implemented directly in the compatibility layer files, such as for example the implementation of queues for the POSIX architecture.

3.1.3.4 CAN interface

The CAN protocol frame length is limited to a maximum of eight bytes, implying that some kind of packet fragmenting must be implemented on top of the CAN protocol. The CSP code base includes a CAN interface implementation with automatic fragmentation of CSP packets into CAN-bus frames. The Maximum Transfer Unit (MTU) over the interface is limited to 256 bytes, which should be enough to allow the majority of telecommands and telemetry messages to fit within one CSP packet. When transferring large amounts of data, such as a file, it will be necessary to fragment the data object at a higher level, probably in the application code.

The MTU of this interface can be decreased, but it should be noted that the MTU represents a trade-off between low overhead when using large MTUs, and a lower chance of packet loss as an effect of data corruption when using small MTUs.

3.1.3.5 CSP Options Confidentiality, integrity, and authenticity (CIA-triad) forms the foundation of a secure service [28]. The CSP library offers mechanisms to support each of these concepts.

Cyclic Redundancy Check If compiling the library with the CRC option enabled, packets can be flagged to use 32-bit CRC to protect their integrity. The packet feature is enabled by creating a socket or connection while passing a $\{\text{CSP}\backslash_{\text{O}}\backslash_{\text{CRC32}}\}$ flag as a function parameter.

Checksum creation and verification is done by the interface modules, and the checksum is not visible to the user of CRC32-enabled packets. When sending packets on a CRC32-enabled connection, the CRC32 flag of the header will be set, and every interface that supports CRC32 will then add a checksum at the end of the data field before transmitting the packet. Note that there must be enough remaining space in the packet buffer to append the checksum on the end of the data field.

Hashed Message Authentication Code When compiling with the HMAC option enabled, packets can be protected with a message authentication code field. Packets can then be verified to have been signed with the correct key, proving the authenticity of the packet. The key which is used to verify the packets must be distributed to the satellite in a safe manner, for example by installing the key before launch.

Extended Tiny Encryption Algorithm When compiling with the Extended Tiny Encryption Algorithm (XTEA) block cipher option enabled, packets may be encrypted using a symmetric key. The satellite must have the shared key installed in order to to decipher the encrypted packets. Used together with HMAC, it allows new security keys to be safely uploaded.

3.1.4 Transport layer - CSP Unreliable Datagram Protocol

There are two different transport protocols offered by CSP. The *Unreliable Datagram Protocol*, which is similar to, but more simple than the User Datagram Protocol used with IP, and provides the simplest form of transport without any guarantee of successful delivery. The *Reliable Datagram Protocol* (RDP) is another compiled option, and adds support for acknowledgement of received packets and automatic retransmission of lost packets. It is similar to the TCP protocol used with IP, but is a smaller and more specialised implementation.

3.1.5 Higher Layers - CSP Applications

Applications can easily be built as clients and servers, using a traditional request-response pattern. A correspondence between a client and server is shown in Figure 9. Packets are sent between CSP nodes by creating and writing to CSP connection objects. A connection object can be obtained by calling `csp_connect()` with the appropriate destination address and port, after which packets can be sent to the node by calling `csp_send()` with the connection and a packet as argument. To receive packets, a CSP socket object must be created, bound and be listening to the correct port. A socket pointer is first obtained with `csp_socket()`, then it must be associated with a specific port using `csp_bind()`. A connection queue must be allocated via `csp_listen()`. A connection handle for an incoming packet may be obtained with `csp_accept()`, and packets may be read from that connection using `csp_read()`. The same connection can be used to send replies with `csp_send()`.

3.2 Internal Communication Architecture

In addition to implementing simple test programs that verify the basic functionality of CSP across nodes, like sending and receiving ping packets, a general communication architecture is sought for integrating and combining the various services of the HSI payload. Although no fixed software architecture for the HSI payload exists yet, it is natural to expect that there might be multiple concurrent services that need to communicate over CSP. The way in which CSP is integrated into the processing system directly affects the software architecture, and will restrict the ways in which the services may communicate.

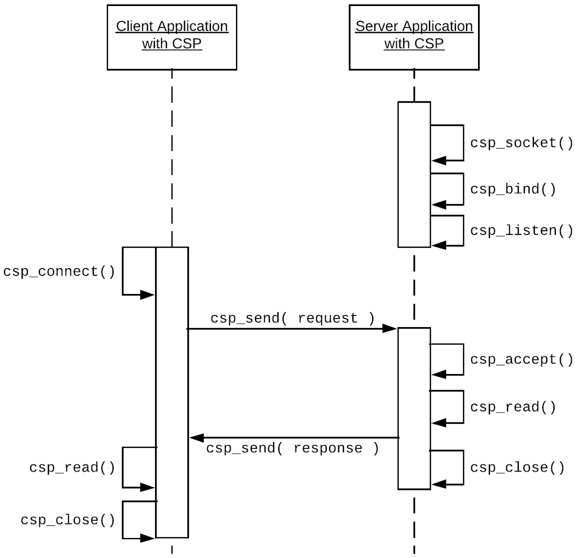


Figure 9: Basic CSP communication. A client application sends a request packet to and receives a response packet from a server application.

3.2.1 Processes vs Threading

In the Linux OS, processes and threads are both concepts for concurrent execution of program contexts. However, there are significant differences between them that must be taken into account when thinking about software architecture. Linux processes are created when a new program is loaded into memory and executed. Processes have their own memory segments for instruction and data memory. Threads are different in that they share these memory and instruction segments with the process that they were spawned from. Threads keep their own Central Processing Unit (CPU) register values and stack memory in order to facilitate execution of different contexts [14].

3.2.2 Inter process communication

While memory sharing is default behaviour between Linux threads, Linux processes can not directly access the memory of each other. Therefore, in order to integrate CSP into a Linux environment, where multiple processes might work and communicate between each other and with a CSP interface, it is necessary to consider Inter-Process Communication (IPC). A small inquiry into the available methods of IPC under the Linux OS is performed.

3.2.2.1 Signals Signals in POSIX (and the equivalent implementation in Linux) is a way to send simple commands between processes (`SIGNAL(2)` [30]). A signal is made up of a single integer, taking on one of the values from a predefined set. Signals are usually not used to pass data, but are perfect for passing small commands and notifications, as long as the signalled values fit within the predefined set of signals. Some of the standard signal values are intended to invoke a specific functionality, like terminating or restarting a process. However, most of these (there are exceptions like `SIGKILL` and `SIGSTOP`) can be repurposed to perform any function by registering custom signal handling routines. For the purpose of passing network data between processes, signals score a low score.

3.2.2.2 Shared memory The Linux OS provides methods to allocate chunks of working memory (`mmap`) that may be shared between processes (`MMAP(2)` [30]). This provides a flexible way to communicate between processes, because all processes read and write directly to the working memory that has been mapped to their virtual memory space. It is also a very fast method of communication because access to the shared data does not require the process to requests expensive system calls from the kernel. This is a clear advantage when compared to other mechanisms offered by the OS, which often depend on system calls to pass data.

If used across multiple cores, the method may create challenges with regard to synchronisation, because the coherence between the data caches of different cores needs to be considered. When used together with synchronisation primitives, or an explicit protocol to safely govern concurrent access to the shared memory, the method offers one of, if not *the* fastest way to communicate any type of data between processes.

This method is used in the *Interplanetary Overlay Network* developed by the Jet Propulsion Laboratory in California (see [21] for an overview of the project), which implements a whole suite of network protocols intended for the use in embedded systems in space. The project employs a module that manages a chunk of shared memory. The various communication protocols attach to the shared memory, and request buffers of dynamic length from this pool. The buffers may be passed to any other processes that also have attached to the shared memory. In addition to manipulation of the shared memory being fast compared to system calls, the buffers are also being passed around using zero-copy mechanism.

3.2.2.3 Files All processes can write and read to files, as long as they have the correct access rights. Naturally then, a process can communicate data with another process by opening and writing to a file, while the other process opens and reads from the same file. This is a very simple approach, but lacks intrinsic mechanisms to encapsulate the concept of a *message* or *packet*. Readers and writers must agree on some protocol, for example to always write new data at the end of the file, and to always read from the start of the file, while making sure to remove data when it has been read. In the discussion about pipes and First in First outs (FIFOs) below, it will become clear that those methods are preferred because of the way they are interfaced.

3.2.2.4 Pipes and FIFOs Pipes are channels on which unidirectional communication can take place. It consists of two ends, one for reading and one for writing (`PIPE(7)` [30]). A pipe is first opened by a process, which must then pass the file descriptor pointing to either the write or the read end of the pipe, to a different process. Child processes can easily be passed the file descriptor of either end of the pipe upon creation with `fork()`. Passing a file descriptor to an unrelated process is slightly more complicated, requiring the use of a separate method of IPC called UNIX domain sockets. Internally, buffers are allocated and data is moved, by system calls to the kernel.

Named pipes, or FIFOs as they are also called, provide an interface to using pipes that is interleaved with a virtual file system (`FIFO(7)` [30]). Instead of being returned two file descriptors like ones does when creating a pipe, an `open` system call is called on a special file in the file system. Standard file access semantics govern whether a process may read or write to a specific FIFO file. Options passed along when opening the FIFO file determines whether a file descriptor for the writing, the reading, or for both ends are returned. FIFOs may be read and written to by multiple processes, although for the sake of simplicity and coherence, it is often clever to limit the users at each endpoint. The way to address a FIFO is to specify its full path name when opening it, such as `"/tmp/proc_fifo_rx"`.

3.2.2.5 UNIX Domain Sockets Sockets in Linux provide a general interface which can be used to communicate internally between processes (then referred to as UNIX domain sockets (`SOCKET(7)` [30]), and to communicate over an IP network (then referred to as INET domain sockets), to name a few underlying channels. Sockets are appropriately named to appear as something you can put objects into, and then expect the objects to appear at the intended opposite end of the communication channel. When using Linux sockets, data is buffered in kernel space.

When using UNIX domain sockets, as is the case for inter-process communication, addressing is done by specifying a file path to a special socket file. Communication is slow compared to a shared memory method, and should offer speeds similar to that of pipes and FIFOs, as those methods also do buffering in kernel space.

3.2.2.6 IPC Conclusion The shared memory (`mmap`) method offers the highest performance for IPC, identical to that of shared memory between POSIX threads. However, the method requires explicit synchronisation to provide safe concurrent access to the shared memory, demanding more development work than the alternatives.

Named pipes (FIFOs) offers a small but safe interface for sending data between processes. It is at least as fast as the other alternatives that also use kernel space for data buffering. Additionally, there exists examples of how to implement a simple prototype of a CSP interface based on FIFOs. Therefore, it is chosen as the method to be used for IPC in the internal communication architecture.

3.2.3 Monolithic Process

This and the following sections explore different communication architectures within the OS, and how the different ways CSP can be integrated.

The simplest approach to an internal communication architecture is the one where all communicating services reside in the same monolithic process as threads.

The important distinction between processes and threads for the use of CSP is that threads are spawned from the same process and may access the same CSP instance. Since threads execute from the same chunk of linked CSP code, they use the same instruction code which points to the same bank of allocated packet buffers. This is advantageous when implementing payload services with threads, because all services can communicate as the same CSP node, like illustrated in Figure 10.

The Monolithic Process approach is most similar to how CSP is used on the lower level OS such as FreeRTOS, where all tasks run in the same access space and can interact with a central CSP instance directly (illustrated in Figure 11). This is the case in the work of Normann [20], where CSP is proposed as an interface for inter process communication. CSP can also be used for inter-thread communication on Linux, and provides the same

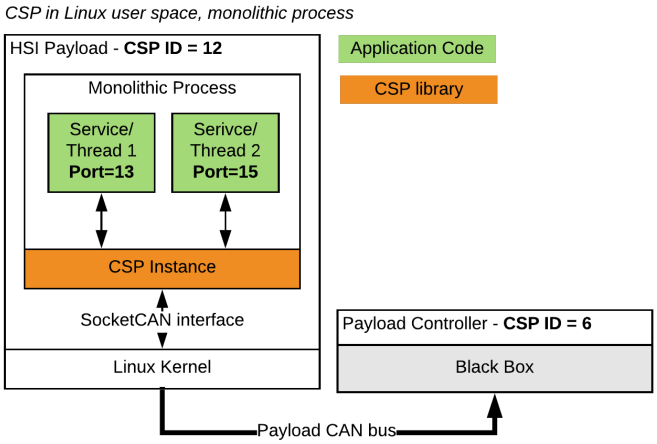

CSP in Linux user space, monolithic process
HSI Payload - CSP ID = 12
Monolithic Process
Service/ Thread 1 Port=13
Serivce/ Thread 2 Port=15
CSP Instance
SocketCAN interface
Linux Kernel
Application Code
CSP library
Payload Controller - CSP ID = 6
Black Box
Payload CAN bus

Figure 10: Under Linux with threading, services are run in separate threads that all connect to the same CSP instance.

benefit of fast communication thanks to shared memory, as explained in Section 3.2.2.2.

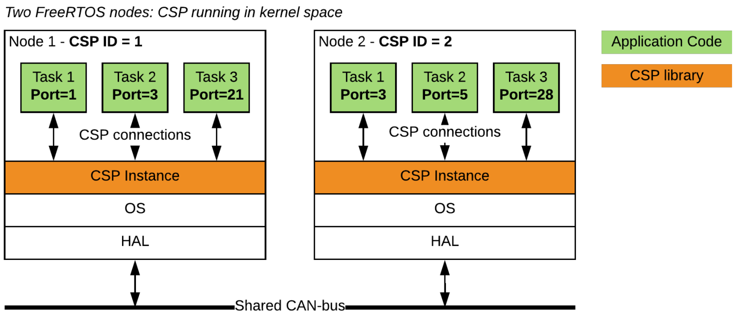


Figure 11: An example of CSP in an OS without user space. In FreeRTOS, CSP is linked together with the whole system, allowing all tasks to access the CSP interface.

There is a disadvantage with the monolithic process approach. All communicating services will have to be linked into a single executable. If the code for one services is to be changed, then the program file containing the code of all services must be changed. This might pose a challenge when upgrading single services.

3.2.4 Separate Processes with unique CSP IDs

As soon as services are implemented as separate processes instead of threads, they cannot access the same CSP instance, which is a major disadvantage. A work-around for this is to expand the CSP network by instantiating multiple CSP nodes in different processes internally in the Payload Processing System. In an e-mail correspondence with one of the CSP developers [31] this was suggested as a solution that had been used for Linux systems. This would then require an inter-process CSP interface to be implemented.

In order to facilitate routing of packets between the processes, a routing process is created. The separate communicating processes connect to the Router Process via a FIFO based inter-process CSP interface. The design is illustrated in Figure 12.

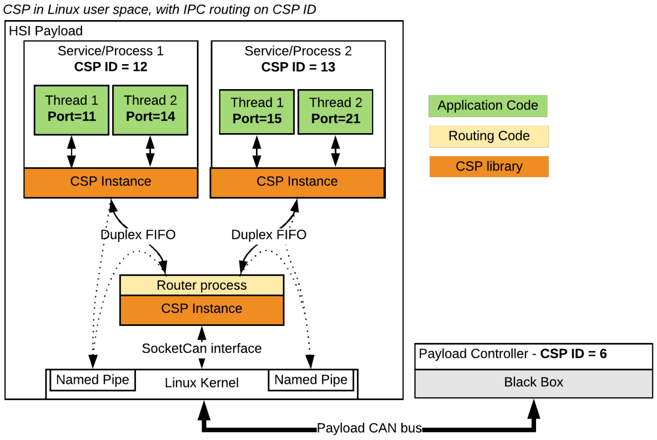


Figure 12: Each service is run as its own process, with its own instance of CSP. A Router Process makes sure that internally addressed packets are routed internally, while externally addressed packets are sent to the CAN bus interface.

There exists advantages to implementing services as distinct processes instead of threads. Since processes have their own instruction segments, a service's code may be changed without halting the other services. Another advantage is that there exists a larger set of management and monitoring tools for processes than threads. For these reasons, architectures using both processes and threads as a basis for services have been investigated.

Each communicating process would require its own CSP ID. One must consider that there are only 32 unique IDs in CSP's entire address space, and furthermore that there are only 4-8 unique CSP IDs allocated for the payloads [29]. It becomes clear that address exhaustion could be a problem if using separate processes for services. Additionally, having multiple CSP IDs for a single Payload is not consistent with the CSP ID assignment scheme that is used with the remainder of the satellite, where a single CSP ID is given to each subsystem.

Buffering Issue The fact that the CSP packet data is routed through FIFOs means that the data must be copied to and from kernel space more times than in the Monolithic Process approach. This is expected to cause overhead that can be observed as delay on round trip times. Additionally, when a node that is connected to the Router Process sends packets quickly, the FIFO interfaces may fill up faster than the Router Process is able to dispatch packets on the CAN-bus. The result is that the receive thread of the FIFO interface uses up all free packet buffers while reading from the FIFO.

3.2.5 Separate Processes with identical CSP IDs

In an attempt to solve the shortcomings of the approach using separate processes with unique CSP IDs, another design is proposed which lets separate processes use the same CSP ID.

As illustrated in Figure 13, this approach has a similar architecture to that of the approach with unique CSP IDs, but now the processes all have the same CSP IDs. The Router Process must now inspect the destination port of incoming packets to determine which process it is addressed to.

In order to perform the routing correctly, the Routing Process must know which process owns the destination port of the incoming packet. For incoming connections, a predefined range of ports can be reserved for each communicating process. The Router Process can then bind to all incoming ports, and check who owns a specific destination port by looking it up in a new static port routing table.

3.2.5.1 Source Port handling Each CSP instance has a pool of ephemeral ports. Ephemeral ports are temporarily reserved as source ports for outgoing connections, such that replies to that specific connection can be correctly routed back to it. Specifically, when a reply to an incoming connection is created, the source port will be used as a destination port.

Since ephemeral ports are chosen at random (initialised with `srand()`), the router will have trouble knowing which process a specific reply is intended

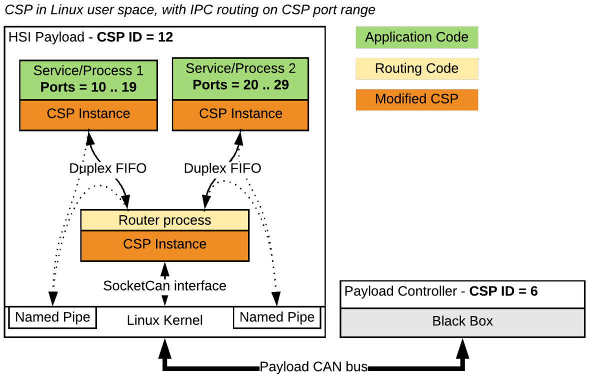


Figure 13: Earch service is run with its own instance of CSP, but with the same CSP ID. A Routing Process inspects the destination port of incoming packets to determine which process it is addressed to.

for. A solution to this is to limit each CSP instance to a range of ephemeral ports, such that their source ports may only be drawn from that range. In this way, the destination port of replies will also reside in that same range, and the router will have all the necessary information to route it back to the sender. The downside of this solution is that the CSP library does not support setting a range for ephemeral ports. In order to implement this solution, the library must be patched. It is undesirable to change the library because that can cause compatibility issues. However, the necessary patch is relatively small, only a few lines of code, and provides a solution the routing problem.

3.2.5.2 CRC field handling The way in which CSP implements CRC handling creates problems for the implementation of the port-based Routing Process. When CSP routes a packet, it checks whether the packet was sent from its own CSP address to decide whether to append a CRC checksum before passing it to the link layer. When routing on ports, the Router Process necessarily has the same CSP address as the leaf CSP node that it serves. This means that a CRC field will be added when sending the packet to the Routing Process, as well as when the Routing Process sends it further on.

When a packet has arrived at the correct node with a Port based Routing Process, it is first accepted by a socket listening on all ports. The action of reading from the socket destroys the CRC field. When the Routing Process forwards the packet to the destination process, the CRC field is only potentially restored. If the packet is sent from the current node, another CRC field is automatically added, such that the CRC field is present when the packet is passed to a FIFO interface. If the packet was not sent by the current node, a CRC field will not be automatically added before passing the packet to a FIFO interface. Therefore, a field is manually added in the implementation of the FIFO interface when this scenario is detected.

When a packet from the current node has arrived at the port based Routing Process, but is addressed to some other node, it will not be accepted by the socket listening to all ports. Therefore, the existing CRC field will not be removed. However, the CSP send-function still appends a new field because it thinks it is the original sender of the packet. To mitigate this, the FIFO interface is implemented to remove the CRC field if it detects this scenario. A packet addressed from the current node to some other node will then arrive at the Routing Process without a CRC field, but with the CRC-flag set. The CSP send-function will then correctly add the CRC field before sending it on the external interface.

The fact that a workaround like this is necessary to make port-based routing work, makes the solution less attractive as it increases complexity. The problem was first discovered after already having tested the implementation through multiple iterations. The reason why a certain action is required in each of the described scenarios is not obvious and required close examination of the implementation of CSP. This also means that this approach is more difficult. The ID-based Routing Process does not have any of the problems associated with CRC field handling, and is therefore simpler to understand and maintain.

3.2.6 CSP as a Kernel Module

When a module is to be interfaced by lots of different processes, it might seem a natural choice to place the module in kernel space, as illustrated in Figure 14.

A CSP kernel module would allow multiple processes to access the same CSP interface, without needing to employ a Routing Process. A single CSP ID could be used for the kernel instance of CSP.

However, the disadvantages of developing a kernel module greatly outweighs the advantages. The implementation of the CSP library as a kernel module would require significant development efforts. This contradicts the idea of saving development effort by using a ready developed network library.

Kernel space is a dangerous place to make code for. When faults are allowed to exist and propagate it can cause the entire kernel to crash. In

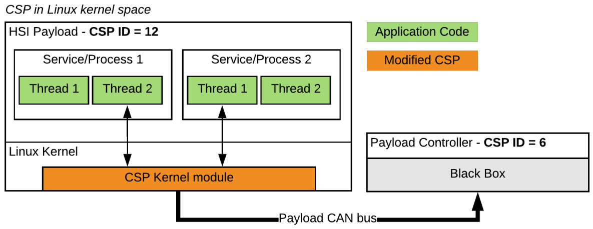


Figure 14: A CSP kernel module would allow multiple proceses to acces the same CSP interface.

comparison, running faulty code in a user space program will only cause the single process to crash.

The Linux kernel has a high readiness level as a result of being heavily reviewed by the whole Linux kernel community. Introducing kernel modules with lower levels of review and testing brings down the readiness of the entire OS.

3.2.7 Summary

The advantages and disadvantages of the proposed architectures are summarised in Table 2.

The Monolithic Process approach offers the best performance, but the worst flexibility. The ID-based Routing Process approach sacrifices performance for greater flexibility, while the port-based Routing Process approach offers even greater flexibility with a similar sacrifice. Depending on the test results, either the Monolithic Process approach or the port-based Routing Process approach is thought to be the most suitable solution.

The development efforts and risks associated with the kernel module approach are too large to consider it a suitable solution.

3.3 Implementation

This section describes the physical setup of the system.

3.3.1 Physical setup

Although the final version of the Payload Processing System is meant to include a PicoZed SoM with a Zynq 7030 SoC, the breakout-board which makes the physical pins accessible to standard connectors is not yet ready.

Table 2: Summary of the proposed internal architectures.

Architecture	Advantages	Disadvantages
Monolithic Process	Minimal overhead with regards to inter-thread communication. Only uses a single CSP ID. CSP can be used for inter-thread communication with the performance of shared memory.	Monolithic binary means that it is impossible to replace the code of a service without having to stop the others.
Separate Processes routing on CSP ID	Services can exists without depending on the binary or state of other services. CSP can be used for IPC.	Overhead from IPC. Having to buffer packets multiple times. Exhaustion of the CSP address space.
Separate Processes routing on CSP ports	The CSP interface is identical for all connecting processes. The added routing is invisible to the services. CSP can be used for IPC.	Overhead from IPC. Having to buffer packets multiple times. Requires code changes to the CSP library.
CSP as a kernel module	No limitations on implementing services as processes or threads. Shared CSP ID for all processes and threads.	IPC via CSP not as fast as the shared memory in Monolithic Process. Significant development cost and risk.

Therefore, the CSP communication stack is implemented to run on a ZedBoard development board from Avnet. This board is equipped with a Zynq 7020 SoC, which is close enough to the Zynq 7030 SoC in terms of processing power and identical in terms of available peripherals. An on-board UART to USB converter circuit allows terminal access to the Zynq 7020 on-board the ZedBoard, and the ARM cores can be booted with a Linux image loaded from a SD-card.

The ZedBoard development kit lacks a CAN bus transceiver circuit. Therefore, an external MCP2551 CAN bus transceiver was acquired and connected to a pair of GPIO pins. The MCP2551 transceiver operates on 5.0 V logic, while the Zedboard is configures to operate on 3.3 V logic. Therefore, a BSS138 based logic level converter circuit was added to the external circuit to step the voltages between the CAN-bus transceiver and the Zynq 7000 SoC. The external circuit was realised on a breadboard, and the complete setup is illustrated in Figure 15 and Figure 16.

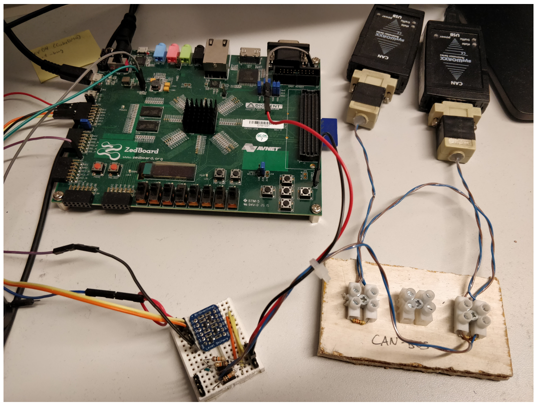


Figure 15: Setup of the ZedBoard development board, and an external CAN transceiver circuit which connects to a CAN bus.

3.3.2 Setting up Operating System

The procedure of building a custom kernel and a root filesystem in PetaLinux proved to be more challenging than anticipated. This procedure is not required to be repeated many times, and is primarily a cost when setting up the system. The setup and build procedures for boot image and root file systems are described in detail in the Xilinx document UG1144: *PetaLinux Tools Documentation - Reference Guide* [32].

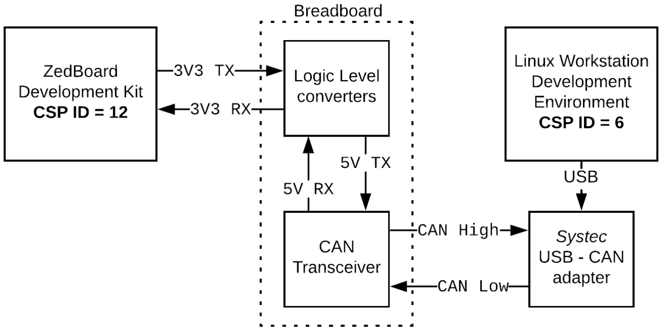


Figure 16: Schematic of test setup, with Zedboard, CAN-bus and Linux workstation.

The first steps include importing the BSP for the ZedBoard SoM, provided by Avnet. This BSP lacks a CAN interface. To add the CAN interface, the BSP is imported into Xilinx's Integrated Development Environment (IDE) Vivado. In addition to being a development environment for designing, simulating and implementing Field-Programmable Gate Array (FPGA) modules and Application-Specific Integrated Circuits (ASICs), it also allows the user to configure the Zynq peripherals to be connected on specific Multiplexed IO (MIO) pins. After configuration, an updated BSP is exported.

At this stage, the modified BSP is imported into a PetaLinux project, which is then configured to include the necessary libraries, and utilities. A boot image and root files system is created and loaded to a SD card. The development board can now be booted from the SD card, and run stored applications that are built with CSP.

3.3.3 Proposed Architectures

With the exception of a kernel module, all the proposed internal communication architectures were implemented. This allowed them to be compared in terms of performance and usability. Some guidelines on how to compile and use the CSP library is gives in Appendix A. All produced code is available via a GitHub link in Appendix B.

The Routing Process that was implemented for the architecture with separate processes was used for both the ID based architecture and the port based architecture.

A FIFO-based inter-process interface for CSP was created. The code was adapted from one of the application code examples provided by the

CSP library. The interface uses Linux FIFOs to send and receive packets. The CSP packet length field is sent first. This fixed-length field is first read to see how many bytes should be read for the remainder of the packet.

To test bandwidth utilization, Trivial File Transfer Protocol (TFTP) was adapted [1] and implemented on top of CSP. Files are fragmented into CSP packets by one node, and recombined by the other. The fragment size is set to the MTU of a CSP packet. However, the MTU was also lowered in certain scenarios to observe what effect the MTU has on the link layer.

4 Results and Discussion

To verify that the implemented communication architectures fulfil the listed requirements in subsection 2.1, multiple tests were carried out. The test results provide grounds for evaluating the architectures.

4.1 Basic Communication

A Command Line Interface (CLI) capable of sending CSP service packets was created to test the basic functionality of CSP. The functionality was tested between the ZedBoard and a Linux Workstation. The results are summarised in Table 3.

Table 3: Summary of tests of basic CSP functionality.

Function under Test	Test Result
Ping	Successful
Request amount of free memory	Successful
Request number of free buffers	Successful
Request uptime	Successful
Reboot system	Successful
Shutdown system	Successful

4.2 Round trip time

When sending a packet, it must pass through multiple execution contexts before reaching the physical interface. After the sending thread has queued a packet, the router thread must be given a chance to execute before the packet is passed on to the appropriate interface. In the case of SocketCAN

---

[1] Adapted from RFC-1350 https://tools.ietf.org/html/rfc1350

interface, the packets are queued in kernel buffers, and is dependent on yet another kernel thread to execute before it is actually dispatched to the CAN controller. All of these delays add up when transmitting, but also when receiving.

The architectures with a Routing Process will have each packet pass through another three execution contexts, as it is picked up on a FIFO interface, routed internally again, and possibly handled by a port-based routing thread.

To measure the impact of this extra overhead, a round trip time test is created. The round trip time was calculated by sending ping like packets and measuring the time to receive a response packet. The measurements were averaged over 10000 packets.

The different architecture were tested on the development workstation, while the ZedBoard responded to the packets.

Table 4: Averaged results of round trip test.

Architecture	Round trip time, averaged [us]
Monolithic Process	1475
Separate Processes, routing on CSP ID	1643
Separate Processes, routing on CSP Ports	1700

From the results presented in Table 4 it is evident that the Routing Process is adding overhead into the network stack, and that the port-based Routing Process adds more than the ID-based Routing Process.

Even when averaged, the measured round trip time varied a lot, and was heavily dependent on instantaneous CPU utilization.

Although the architectures with a Routing Process increase the round trip delay, the increase is insignificant when compared to the delays that can be expected from the rest of the communication chain. The delay should also not affect the data bandwidth as much if using delay tolerant protocols.

4.3 Bus Utilization

Files were transferred between CSP nodes to put strain on the CAN link. The `canbusload` utility from the Linux package `can-utils` was used to monitor the bus utilization.

The transfer rate for theoretical maximum bus utilization is also calculated. In a single packet, the CSP header occupies 4 bytes, the CSP length field occupies 2 bytes, and the CRC field occupies 4 bytes. Given that the CAN interface has a MTU of 256 bytes, there are only 246 bytes left for data. Additionally, the CAN link layer adds a substantial amount of header bits to the frames before they are sent. Given the 50 % data utilization of the CAN link, the overall data utilization for one CSP packet of maximal length is approximately 48 % as calculated in Equation 1. This equals a theoretical maximum transfer rate of 480 Kbps. This is still unobtainable because of the lesser effects explained in Section 3.1.2.2.

$$\eta = \frac{246\text{B}}{512\text{B}} = 0.480 \tag{1}$$

4.3.0.1 SocketCAN MTU issues When transmitting large packets (more than 50 % of the maximum 256 bytes in a packet) from the Linux Workstation, the communication link became unresponsive. Sometimes, after a large CSP packet had been written to the CAN socket, the last few bytes of the CSP packet would not be transmitted on the CAN-bus. Only when a new packet was written to the CAN socket did the last few bytes of the previous packet get transmitted on the bus, before the new packet was transmitted. The effect was more frequent for larger packets, and the effect was rarely observed for packets of size 64 and below. The author was not able to find the root cause of this issue. However, the issue was only observed on the development Linux Workstation, and not on the ZedBoard or PicoZed.

In Table 5, the results of transferring a 1.51 MB file are shown. The file was transferred from the ZedBoard to the Linux Workstation. When using Stop-and-Wait flow control, the transfer was able to reach an average data rate between 350 Kbps and 380 Kbps.

Table 5: Transfer rates of TFTP using Stop-and-Wait flow control, and 64B fragment size.

Architecture	`canbusload`, estimate [%]	Average transfer rate [Kbps]	Percentage of Theoretical Maximum [%]
Monolithic Process	90	380.68	79.3
Separate Processes, routing on CSP ID	85	356.52	74.3
Separate Processes, routing on CSP Ports	83	354.45	73.8

The decrease in transfer rates for the two architectures that use a Routing Process is accounted to their increased round trip time. This effect is prominent when waiting for ACKs, such as with Stop-and-Wait flow control.

Buffering Issues As discussed in subsubsection 3.2.4, sending CSP packets too quickly over a FIFO interface to a Routing Process will exhaust its packet buffers. The effects are observed when transferring files with TFTP without any flow control. When the Routing Process's packet buffer bank has been exhausted, the `csp_get_buffer()` function starts running expensive system calls to print error messages. The observed effect is that the data rate drops to a few percent of what the link normally can handle.

The issue stems from the fact that the CSP library has no method to have a thread block or suspend until there are available packet buffers. This could be implemented using POSIX condition variables, but would require a larger modification to the CSP library.

The Monolithic Process architecture does not suffer from this exact problem, because all buffering is performed by the SocketCAN networking module inside the Linux kernel. The buffer space offered by the networking module can also be observed to become exhausted, but the CSP SocketCAN driver implements a short `usleep()` to allow the kernel to process the data before attempting again.

The same work-around using `usleep` when no buffers are available was implemented in the FIFO interface, resulting in the wanted effect. Most, but not all warning and error prints were eliminated. The achieved data rates was restored to more reasonable speeds. The now infrequent, but still present warning and error prints, caused this transfer mode to not perform as well as the one with Stop-and-Wait flow control.

Table 6: Transfer rates of TFTP using *no* flow control and 64B fragment size.

Architecture	`canbusload`, estimate [%]	Average transfer rate [Kbps]	Percentage of theoretical maximum [%]
Monolithic Process	82	367.74	76.61
Separate processes, routing on CSP ID	82	367.67	76.60
Separate processes, routing on CSP ports	82	367.24	76.51

In Table 6, the results from transferring a 1.51 MB file *without* any flow control is shown. The file was transferred from the ZedBoard to the Linux Workstation. The transfer rates are comparable, with only a slight decrease in throughput for the architectures with a Routing Process, compared to the Monolithic Process approach.

The highest obtained data transfer rate is not high enough to fulfil requirement SYS-3-007, which demands a transfer rate of 800 Kbps (which is actually higher than the theoretical maximum). However, all architectures achieve a rate that is high enough to fulfil requirements M-1-015, M-1-016 and M-2-026, which at most requires a data rate of 81 Kbps.

Given that the buffering capabilities of the PC are used, the requirement SYS-3-007 may still be fulfilled. As the satellite orbits around the earth, most of the time spent in one orbit will be time without free sight to a ground station. A solution could then be to use this period of time to pre-buffer data on the PC. When the ground station comes into sight, the PC is able to downlink the data at full speed, thus fulfilling the requirement.

4.4 Stress test

A 1005 MB file was transferred over the link using the architecture based on port routing. The transfer lasted approximately six hours, and completed without any faults. This demonstrates how the network is able to withstand a high bus utilization over a longer period of time, as was required by requirements SW-3-007 and SW-4-006.

4.5 Validating the M6P platform

The M6P satellite bus was tested by connecting to a so called remote *FlatSat*. The payload's CAN-bus link was connected to a workstation in the SmallSat Lab running a piece of software that acts as a TCP/IP bridge. The bridge connects to a server in *NanoAvionic's* facilities, which forwards all communication to and from a complete M6P satellite bus over the Internet. This is similar to connecting to the a real physical satellite bus at the SmallSat Lab, with only a few limitations. One thing is the extra delay that is experienced when communicating between Payload and PC. Another is the fact that the satellite bus subsystems can not be physically observed or manipulated, due to the system being connected over the Internet.

Basic CSP functionality was verified to be working. Ping was received from all subsystems present in the FlatSat.

Telemetry data was received from telemetry services on the EPS and FC nodes. This demonstrated that the Flatsat and CSP network was able to transfer packets of considerable size (roughly 250 bytes).

5 Conclusion

The CSP network library and a CAN-bus link have been integrated into the HSI Payload.

Several internal communication architectures have been evaluated. One architecture that use a port-based Routing Process has achieved greater flexibility than the other architectures, in the sense that it offers multiple processes to communicate as the same CSP node. However, the added complexity and the additional issues that were discovered in the architectures that utilise a Routing Process is not justified by the increased flexibility. Therefore, the thread-based Monolithic Process approach is preferred and offered as a proposed solution.

Although the SYS-3-007 requirement was not fulfilled by the proposed architecture, it is believed that the buffering capabilities of the PC will make it possible to fulfil that requirement.

The findings of this report are thought to be valuable to the process of defining a overall software architecture for the HSI Processing System.

Future Work The services that implement the functionality needed to operate the HSI camera are currently under development. In order to integrate these, a software architecture needs to be defined. The software architecture should manage services by starting, stopping and restarting them as necessary, and should integrate the communication architecture that is proposed in this report.

During the work, all testing has been carried out with a ZedBoard an Linux Workstation, as well as a remote FlatSat. It still remains to test the CSP network with physically connected PC hardware.

The SocketCAN MTU issue described in subsubsection 3.2.4 should be further investigated, despite the issue not manifesting itself on the ZedBoard.

A solution for the Routing Process buffering issue could be sought.

References

[1] D. D. E. Buchen, "2014 nano/microsatellite market assessment," SpaceWorks Enterprises, Inc. (SEI), Tech. Rep., 2014.

[2] "How New Technology Is Democratizing Access To Space," 2018. [Online]. Available: https://www.forbes.com/sites/mitsubishiheavyindustries/2018/08/24/how-new-technology-is-democratizing-access-to-space/#

[3] A. Escher, ""inside planet labs' new satellite manufacturing site"," *TechCrunch*, 2018. [Online]. Available: https://techcrunch.com/2018/09/14/inside-planet-labs-new-satellite-manufacturing-site/

[4] S. Dr. Nicholas M. Short, "Graphic representation of hyperspectral data," 6 2007. [Online]. Available: https://en.wikipedia.org/wiki/Hyperspectral_imaging

[5] *Full-duplex Low-power S-band Transceiver*, Satlab ApS, 2018.

[6] *6U CubeSat Design Specification*, California Polytechnic State University, 7 2018.

[7] NanoAvionics, *Payload Controller, NA-PC-G0-R0, Data Sheet*, 2018.

[8] *M6P Platform/Payload Interface Control Document*, NanoAvionics, 2018.

[9] H. D. Ledet-Pedersen J., Christiansen J.D.C, "Cubesat Space Protocol," https://github.com/GomSpace/libcsp, 2018.

[10] "HYPSO Mission Budgets - Project Document," 2018.

[11] S. Martin, *An Introduction to Ocean Remote Sensing*, 2014.

[12] *Zynq-7000 SoC Technical Reference Manual, UG585*, 1st ed., Xilinx, Inc., July 2018.

[13] *PicoZed 7Z015 / 7Z030 SOM - Hardware User guide*, Avnet, 2018.

[14] W. Stallings, *Operating Systems, Internals and Design Principles.* Pearson Prentice Hall, 2009.

[15] Real Time Engineers Ltd., "The FreeRTOS™ Kernel," www.freertos.org, 2017.

[16] *NanoMind A712D Datasheet*, 1st ed., GomSpace ApS, March 2016.

[17] *PetaLinux Tools Documentation Reference Guide, UG1144*, 2018th ed., Xilinx, Inc., June 2018.

[18] "NTNU Test Satellite (NUTS), Student Satellite Project," 2011-2017.

[19] M. Muench, "Integration and verification of a keyed-hash message authentication scheme based on broadcast timestamps," Master's thesis, NTNU, 2014.

[20] M. A. Normann, "Software design of an onboard computer for a nanosatellite," Master's thesis, NTNU, 2016.

[21] S. Burleigh, "Interplanetary overlay network: An implementation of the dtn bundle protocol," in *2007 4th IEEE Consumer Communications and Networking Conference*, 1 2007, p. nil. [Online]. Available: https://doi.org/10.1109/ccnc.2007.51

[22] "Clang: a C language family frontend for LLVM [Software Project]," 2014.

[23] S. A. Jacklin, "Survey of Verification and Validation Techniques for Small," 2015.

[24] *ISO/IEC 7498-1:1994 - Information technology - Open Systems Interconnection - Basic reference model: The basic model*, 1994.

[25] *ISO11898-1 Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*, International Organization for Standardization (ISO), 2015.

[26] *ISO11898-2 Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit*, International Organization for Standardization (ISO), 2016.

[27] *ISO11898-3 Road vehicles – Controller area network (CAN) – Part 3: Low-speed, fault-tolerant, medium-dependent interface*, International Organization for Standardization (ISO), 2006.

[28] C. Doerr, *Network Security in Theory and Practice.* Christian Doerr, 2018.

[29] *M6P Platform Software Interface Control Document*, NanoAvionics, 2018.

[30] *Linux Programmer's Manual*, 4th ed., Linux Foundation, 2018.

[31] "Personal Communication with CSP developer Johan De Claville Christiansen," 2018.

[32] *Payload Controller, NA-PC-G0-R0, Data Sheet*, NanoAvionics, 2018.

Acronyms

ACK Acknowledgement. 22, 44

ADCS Attitude Determination and Control System. 8

AMOS Centre for Autonomous Marine Operations and Systems. 6

ASIC Application-Specific Integrated Circuit. 40

AXI Advanced Extensible Interface. 13

BB Breakout Board. 8

BSP Board Support Package. 15, 40

CAN Controller Area Network. 10, 12, 13, 18–22, 25, 34, 39, 40, 42, 43, 45, 46

CLI Command Line Interface. 41

COTS Commercial off-the-Shelf. 5

CPU Central Processing Unit. 28, 42

CRC Cyclic Redundant Check. 22, 24, 26, 35, 36, 43

CSMA/CA Carrier Sense Multiple Access with Collision Avoidance. 21

CSP Cubesat Space Protocol. 8, 10, 12, 16, 23–28, 31–46

DDR3L Double Data Rate Type Three Low Voltage. 13

DMA Direct Memory Access. 13

ECC Error-Correcting Code. 20

ECSS European Cooperation for Space Standardization. 10

EMF Electromotive Force. 19

eMMC Embedded MultiMediaCard. 13

EPS Electrical Power System. 8, 45

FC Flight Computer. 8, 45

FIFO First in First out. 30, 31, 33, 34, 36, 40–42, 44

FPGA Field-Programmable Gate Array. 40

GNSS Global Navigation Satellite System. 8

GPIO General-Purpose IO. 13, 39

HMAC Hash-Based Message Authentication Code. 16, 26

HSI Hyperspectral Imaging. 6, 8, 10–13, 15, 16, 27, 46

HYPSO Hyperspectral surveillance of the oceans. 6, 7, 10, 12, 13

I2C Inter-Integrated Circuit. 13

IDE Integrated Development Environment. 40

IP Internet Protocol. 22, 27, 45

IPC Inter-Process Communication. 28, 30, 31, 38

ISO International Organization for Standardization. 18

MAC Medium Access Control. 20

MIO Multiplexed IO. 40

MTU Maximum Transfer Unit. 25, 26, 41, 43, 46

NRZ Non-Return-To-Zero. 21

NTNU Norwegian University of Science and Technology. 6, 16

NUTS NTNU Test Satellite. 16

OBC On-board Computer. 15

OCM On Chip Memory. 13

OS Operating System. 14, 15, 24, 25, 28, 29, 31, 32, 37

OSI Open Systems Interconnect. 17

PC Payload controller. 8, 10, 12, 45, 46

PCB Printed Circuit Board. 13
PL Programmable Logic. 13, 15
POSIX Portable Operating System Interface. 25, 29, 31, 44
PS Processing System. 13

RDP Reliable atagram Protocol. 16
RGB Red-Green-Blue. 8

SDR Software Defined Radio. 6
SoC System on Chip. 13, 15, 37, 39
SoM System on Module. 13–15, 37, 40
SPI Serial Peripheral Interface. 13

TCP Transmission Control Protocol. 22, 27, 45
TFTP Trivial File Transfer Protocol. 41, 44

UART Universal asynchronous Receiver-Transmitter. 13, 39
UHF Ultra-High Frequency. 8
USB Universal Serial Bus. 13, 39

XTEA Extended Tiny Encryption Algorithm. 26

Appendix A Cubesat Space Protocol

The CSP source code is downloaded from the public GitHub repository: `https://github.com/libcsp/libcsp`.

A.1 Compiling CSP Library

The library is configured with the python based build tool `waf`. The build program is included in the repository. Python version 2.7.15 was used to build the library, after failing to run the `waf` scripts with Python version 3.7.0. Other version of `python3` were found to work. The `virtual environment` python tool can be used to enforce a specific python version.

The CSP library can be configured with a wide selection of options. The options can be applied with

- `>> ./waf configure <option1> <option2> . . .`

The following options were used when configuring

- `--enable-crc32`

  allows CRC32 field to be added to packets.

- `--enable-if-can`

  adds the CAN interface, which includes an automatic fragmentation protocol etc.

- `--enable-can-socketcan=socketcan`

  adds a simple SocketCAN CSP driver.

- `--with-os=posix`

  adds the POSIX compatibility layer. Necessary to run on Linux.

- `--with-max-bind-port=PORT`

  was used when implementing the Routing Process, in order to be able to bind source and destination ports.

- `--with-router-queue-length=COUNT`

  was used to increase the number of packets the Router Process could buffer.

- `--with-max-connections=COUNT`

  used to increase the number of connections the CSP instance can keep active at one time.

The library is compiled with:

```
>> ./waf build
```

The build folder can be changed by supplying the option:

```
--out=<build_dir>
```

The compiled library and library header files can be installed with

```
>> ./waf install
```

The install directory can be changed by supplying the option:

```
--prefix=<install_dir>
```

Problems were encountered when trying to configure, compile and install the library in separate **`waf`** commands. When supplying all the commands simultaneously, no problems were encountered:

- `>> ./waf configure <option1> <option2> ... build install`

The following Makefile wrapper was used to configure and build the library:

```
# If installing outside libcsp directory, please define
↪  LIBCSP_BUILD_DIR

# Default architecture
ARCH ?= x86
BUILD=build_$(ARCH)

PYTHON2 = python2

ROOT = $(abspath ..)

LIBCSP_DIR = $(ROOT)/libcsp

LIBCSP_CONFIG_COMMON =  --enable-crc32  --enable-if-can
↪  --enable-can-socketcan=socketcan --with-os=posix
↪  --with-padding=0 --install-csp --with-connection-so=64
```

```
ifeq ($(ARCH),arm) # ARM toolchain
        LIBCSP_CONFIG_PLATFORM =
         ↪  --toolchain=arm-linux-gnueabihf-
endif

# Allow a custom build dir to be specified,
# along with extra configuration options LIBCSP_CONFIG_EXTRA
ifeq ($(LIBCSP_BUILD_DIR),)
        LIBCSP_CONFIG_BUILD =
         ↪  --out=$(LIBCSP_DIR)/build_$(ARCH)
         ↪  --prefix=$(LIBCSP_DIR)/build_$(ARCH)
        LIBCSP_BUILD_DIR = $(BUILD)
else
        LIBCSP_CONFIG_BUILD =  --out=$(LIBCSP_BUILD_DIR)
         ↪  --prefix=$(LIBCSP_BUILD_DIR)
endif

LIBCSP_CONFIG_FULL = $(LIBCSP_CONFIG_COMMON)
 ↪  $(LIBCSP_CONFIG_PLATFORM) $(LIBCSP_CONFIG_BUILD)
 ↪  $(LIBCSP_CONFIG_EXTRA)

all:$(LIBCSP_BUILD_DIR)/libcsp.a

%/libcsp.a: $(shell find src -type f) $(shell find include
 ↪  -type f)
        cd $(LIBCSP_DIR) && \
        $(PYTHON2) waf configure --jobs=$(shell nproc)
         ↪  $(LIBCSP_CONFIG_FULL) build install

clean:
        cd $(LIBCSP_DIR) && \
        $(PYTHON2) waf configure $(LIBCSP_CONFIG) clean
```

A.2 Using CSP Library

To use the library it must be statically linked with the application code. Here it is assumed that the GNU Compiler Collection (`gcc`) is used for building programs.

One way to link the library is to provide a library search path `-L<CSP_-`

`install/lib` and a library name `-lcsp`:

- `>> gcc -L<CSP_install/lib> main.c <other sources> -lcsp`

Or one can also just include the static library like any other object file:

- `>> gcc main.o <other objects> <CSP_install/lib>/libcsp.a`

Appendix B Project Code

The project code is released on and can be downloaded from GitHub: `https://github.com/magne-hov/hypso/releases/tag/TTK4550`