sid.inpe.br/mtc-m21c/2021/01.19.10.57-TDI

SPECIFICATION OF A NEW DATA LINK LAYER PROTOCOL AND ASSOCIATED "CHANNEL" API FOR EMBEDDED NETWORKS ONBOARD OF AEROSPACE VEHICLES

Sérgio Duarte Penna

Doctorate Thesis of the Graduate Course in Space Engineering and Technology/Space Systems of Management and Engineering, guided by Dr. Marcelo Lopes de Oliveira e Souza, approved in February 1st, 2021.

URL of the original document:
<http://urlib.net/8JMKD3MGP3W34R/442G7R2>

INPE
São José dos Campos
2021

PUBLISHED BY:

Instituto Nacional de Pesquisas Espaciais - INPE
Coordenação de Ensino, Pesquisa e Extensão (COEPE)
Divisão de Biblioteca (DIBIB)
CEP 12.227-010
São José dos Campos - SP - Brasil
Tel.:(012) 3208-6923/7348
E-mail: pubtc@inpe.br

sid.inpe.br/mtc-m21c/2021/01.19.10.57-TDI

SPECIFICATION OF A NEW DATA LINK LAYER PROTOCOL AND ASSOCIATED "CHANNEL" API FOR EMBEDDED NETWORKS ONBOARD OF AEROSPACE VEHICLES

Sérgio Duarte Penna

Doctorate Thesis of the Graduate Course in Space Engineering and Technology/Space Systems of Management and Engineering, guided by Dr. Marcelo Lopes de Oliveira e Souza, approved in February 1st, 2021.

URL of the original document:
<http://urlib.net/8JMKD3MGP3W34R/442G7R2>

INPE
São José dos Campos
2021

Cataloging in Publication Data

Penna, Sérgio Duarte.

P381s Specification of a new data link layer protocol and associated "channel" API for embedded networks onboard of aerospace vehicles / Sérgio Duarte Penna. – São José dos Campos : INPE, 2021.

xxviii + 257 p. ; (sid.inpe.br/mtc-m21c/2021/01.19.10.57-TDI)

Thesis (Doctorate in Space Engineering and Technology/Space Systems of Management and Engineering) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos, 2021.

Guiding : Dr.Marcelo Lopes de Oliveira e Souza.

1. Communication protocol. 2. Data Link Layer. 3. Application Layer. 4. Ethernet. 5. Embedded systems. I.Title.

CDU 629.7.054

MINISTÉRIO DA CIÊNCIA, TECNOLOGIA E INOVAÇÕES

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS

Serviço de Pós-Graduação-SEPGR

Pós-Graduação em ETE/Engenharia e Gerenciamento de Sistemas Espaciais.

DEFESA FINAL DE TESE DE SÉRGIO DUARTE PENNA

No dia 01 de fevereiro de 2021, as 09h, por videoconferência, o(a) aluno(a) mencionado(a) acima defendeu seu trabalho final (apresentação oral seguida de arguição) perante uma Banca Examinadora, cujos membros estão listados abaixo. O(A) aluno(a) foi APROVADO pela Banca Examinadora, por UNANIMIDADE , em cumprimento ao requisito exigido para obtenção do Título de Doutor em Engenharia e Tecnologia Espaciais/Eng. Gerenc.de Sistemas Espaciais. O trabalho precisa da incorporação das correções sugeridas pela Banca Examinadora e revisão final pelo(s) próprio Orientador.

Título: "SPECIFICATION OF A NEW DATA LINK LAYER PROTOCOL AND ASSOCIATED "CHANNEL" API FOR EMBEDDED NETWORKS ONBOARD OF AEROSPACE VEHICLES"

Eu, Maurício Gonçalves Vieira Ferreira , como Presidente da Banca Examinadora, assino esta ATA em nome de todos os membros.

Membros da Banca

Dr. Maurício Gonçalves Vieira Ferreira , Presidente INPE.
Dr. Marcelo Lopes de Oliveira e Souza, Orientador(a) INPE.
Dr. Paulo Giacomo Milani, Membro da Banca INPE.
Dr. Rômulo Silva de Oliveira , Convidado(a) UFSC.
Dr. Dr. Fernando José de Oliveira Moreira, Convidado(a), Embraer.

Documento assinado eletronicamente por **Mauricio Goncalves Vieira Ferreira**, **Coordenador de Rastreio, Controle e Recepção de Satélites**, em 01/02/2021, às 14:55 (horário oficial de Brasília), com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.mctic.gov.br/verifica.html, informando o código verificador **6451242** e o código CRC **7F8BB0D6**.

Referência: Processo nº 01340.000742/2021-80 SEI nº 6451242

“It is our choices [...] that show what we truly are, far more than our abilities”.

Albus Dumbledore in “The Chamber of Secrets”, 1999

To my dear Professor Dr. Marcelo Lopes de Oliveira e Souza and to my Post-Graduation colleagues at INPE.

ACKNOWLEDGEMENTS

First al all, I would like to thank my dear Professor Dr. Marcelo Lopes de Oliveira e Souza for his most valuable contribution to the construction of this thesis. To my colleague of many years, Rui Nelson Almeida, many thanks for the guidance in the development of applications in C Language. To Doctors Paulo Giacomo Milani, Alírio Brito and Fabrício Kucinskis, my gratitude for the revising my texts until the final presentation. To my dearest Post-Graduation colleagues, Paula Renata Aranha, Graziela Maia, Roberta Porto and Ana Paula Rabello, my eternal thanks for supporting me during the whole process. To my first Mentor in R&D, Dr. Atair Rios Neto, my recognition for having encouraged me in accepting the challenge of post-gradute studies.

ABSTRACT

Communication protocols are essential components in complex and highly integrated systems onboard aerospace vehicles. The implementation of such type of component in software may demand a high processing cost, should itself being of high complexity, therefore choosing simpler protocols that allow tasks which use it execute with the desired efficiency must be part of a good system development process. This work presents a new Data Link Layer communication protocol to operate over Ethernet physical medium, and a new virtual communication resource called “channel” for accessing the protocol directly from the Application Layer, illustrated by a case study demonstrating the most relevant aspects of the proposal..

Keywords: communication protocol, Data Link Layer, Application Layer, Ethernet, embedded systems, aerospace vehicles.

ESPECIFICAÇÃO DE UM NOVO PROTOCOLO DE CAMADA DE ENLACE E SUA INTERFACE DE PROGRAMAÇÃO “CHANNEL” PARA REDES EMBARCADAS EM VEÍCULOS AEROESPACIAIS

RESUMO

Protocolos de comunicação são componentes essenciais em sistemas complexos e altamente integrados embarcados em veículos aeroespaciais. A implementação deste tipo de componente em software pode demandar um alto custo de processamento, caso ele próprio seja de alta complexidade, portanto buscar protocolos mais simples para que as tarefas que o utilizam executem com a eficiência desejada faz parte de um bom de projeto de desenvolvimento de sistemas. Este trabalho apresenta um novo protocolo de comunicação para a Camada de Enlace que usa o meio-físico Ethernet, com recursos adicionais de seqüenciamento de transmissão, temporização e integridade de dados, e um novo recurso virtual de comunicação denominado de “canal” para acesso a este protocolo a partir da Camada de Aplicação, ilustrado por um caso de estudo demonstrando os aspectos mais relevantes da proposta.

Palavras-chave: protocolo de comunicação, Camada de Enlace, Camada de Aplicação, Ethernet, sistemas embarcados, veículos aeroespaciais.

LIST OF FIGURES

Page

LIST OF TABLES

Page

LIST OF ACRONYMS AND ABBREVIATIONS

AFDX™	Avionics Full-Duplex Switched Ethernet (Airbus Industries)
API	Application Programming Interface
ARINC	Aeronautical Radio Incorporated
ARPA	Advanced Research Project Agency
ANSI	American National Standards Institute
BAG	Bandwidth Allocation Gap (defined in ARINC-664 Part 7 – AFDX™)
BSD	Berkeley Software Distribution
CAN	Controller Area Network (Robert Bosch)
CCSDS	Consultative Committee for Space Data Systems
CBERS	China-Brazil Earth Resource Satellite
CRC	Cyclic Redundancy Check
DARPA	Defense Advanced Research Project Agency
DCP	Digital Communication Protocol
DoD	Department Of Defense (USA)
DSAP	Destination Service Access Point
ECSS	European Cooperation for Space Standardization
ESA	European Space Agency
FIFO	First-In-First-Out
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
INPE	*Instituto Nacional de Pesquisas Espaciais* (Brazil)
IMA	Inegrated Modular Avionics
ID	Identification
IP	Internet Protocol
ISO	International Standards Organization
JAXA	Japan Aerospace Exploration Agency
LAN	Local Area Network
LLC	Logical Link Control (part of the ISO/OSI Data Link Layer)
MIL-STD	Military Standard
MAC	Media Access Control (part of the ISO/OSI Data Link Layer)

MTU	Maximum Transfer Unit (defined by the Internet Protocol – IP)
NASA	National Aeronautics and Space Administration (USA)
OBDH	On-Board Data Handling
OSI	Open Systems Interconnect (affiliated to ISO)
OUI	Organizationally Unique Identifier
PARC	Palo Alto Research Center (part of XEROX Corporation)
PCAP	Packet Capture (defined by the Wireshark application)
PDU	Protocol Data Unit (defined by IEEE 802.2)
RFC	Request For Comments (to the Internet Task Force)
RS	Recommended Standard
SAE	Society of Automotive Engineers
SAP	Service Access Point
SN	Sequence Number
SNAP	Subnetwork Access Protocol
SSAP	Source Service Access Point
TCP	Transmission Control Protocol
TTP	Time-Triggered Protocol
UDP	User Datagram Protocol
UI	Unnumbered Information (type of PDU)
VL	Virtual Link (part of ARINC-664 Part 7 – AFDX™)
VLAN	Virtual Local Area Network

SUMMARY

Page

1 INTRODUCTION

1.1 Context and motivation

Connecting people seems to be the most important consequence of a technology asset which began its development back in the 19th century with the telegraph. The ability of communicating facts over a physical medium beyond line-of-sight changed the face of the world.

“Communication”, according to a web dictionary (MERRIAN-WEBSTER, 2019), is “a process by which information is exchanged between individuals through a common system of symbols, signs, or behavior”. However, to what purpose one or more individuals would use communication? According to other web dictionary (LEXICO, 2019), here is a very good reason: “the successful conveying or sharing of ideas and feelings”.

Therefore, communication needs not only to be effective allowing two parties to connect, but it also needs to convey the correct fact or data.

Surprisingly at first, for electronic control systems embedded in modern vehicles, be it a car, bus, train, aircraft or spacecraft, communication is not only essential, but vital to their safe operation. Communication is what binds devices together forming a complex network of specialized functions.

In the early stages of the development of electronic control systems, processing was done by a single complex device, such as the trajectory control system of the V2 rocket developed by the Germans during the World War II. Every step of processing and resulting action on V2’s rocket engine and tail fins was performed by a unit called LEV-3 (STAKEM, 2010) and an analog computer designed by Helmut Hoelzer, an electrical engineer (EDISON TECH CENTER, 2019). Very little communication was necessary, all of it in the analog world.

Communication using analog signals (current or voltage) prevailed until the advent of the microprocessor. An early case of data being communicated using digital signals was in the AGC - Apollo Guidance Computer (HALL, 1996).

Digital Communication Protocols (DCP) for the aerospace industry started being standardized in the beginning of the 70’s and were firstly used in military aircraft, namely the F-16 Falcon, then lately in space applications as well, such

as the Space Shuttle and the International Space Station (GOFORTH et al., 2014).

The specification of a DCP usually describes the way digital information, that is, binaries 1 and 0, are encoded in the transmission medium and, at a higher level, how a finite collection of binary digits is to be interpreted by the communicating parties. The term “protocol” refers to the rules that apply for interpreting binary information transported by a DCP.

DCPs started being used in commercial aircraft which flew for the first time in the beginning of the 80’s, Boeing 767 and Airbus A320, to name two pioneer users, with the introduction of the Integrated Modular Avionics (IMA) concept (TAGAWA et al., 2011).

DCPs proved very important as electronic control systems became more and more complex, as more complex functions could be accommodated because of more and more powerful microprocessors. They evolved, as the topology of these systems changed for every new vehicle depending on how control functions were physically allocated in electronic units.

More recently, changing from more concentrated to more distributed allocation of functions in electronic control systems (WATKINS, 2007) has driven very important changes in DCP technologies.

There was also an interesting migration of DCPs from one industry field to another: from aircraft to automotive, from automotive to aircraft, from aircraft to space, from space to aircraft, from Information Technology to manufacturing floor.

The aerospace industry has become particularly sensitive to time and costs of developing and launching a new artifact. This is aligned with recent trends in the industry in general with the “Lean Production System” (KRAFCIK, 1988), and in the space industry with the “faster-better-cheaper” initiative pushed by Daniel Goldin (NASA, 2009).

In the space industry, the traditional demand for highly reliable space systems had driven high costs, longer life cycles and fewer missions. This tendency was described by Wertz (2010) as the “Space Spiral”.

The “New Space” approach, as reviewed by Koechel et al. (2018), now favors less reliable space systems, but with more missions of lower costs and shorter life cycles. In short, “New Space” expects to provide the same service to customers spending fewer resources in its development and in its operational life. Migrating DCPs matured in other industry fields to the space industry seems to follow this approach.

DCP evolution does not show signs of interruption, as new data processing and data communication scenarios are created for new vehicles and new industry fields. The very evolution of the architectures used in systems designed for aerospace vehicles toward a more integrated modular architecture is a key driver for the introduction of new DCPs (FUCHS, 2012).

Even after so many years of developing communication solutions, there is still room for innovation.

The core motivations of this work are:

1) There is still space for new ways of combining characteristics that are present in one and absent in another DCP to better serve systems installed onboard of aerospace vehicles of small and medium sizes;
2) The software interface for accessing a DCP is often proprietary and has a high acquisition cost associated with it;
3) The most frequently used communication protocol in our days, the Ethernet, has found its way as a DCP in space applications (SEPHY, 2015) after being under study since as early as 2002 (WEBB, 2002).

1.2 Objective

The main objective of this work is the specification of a new Data Link Layer protocol and a new virtual communication resource called “channel” for accessing the protocol from the Application Layer for embedded networks onboard of aerospace vehicles

This new Data Link Layer is an extension of the IEEE-802.2 protocol presenting the following additional qualities:

- It accepts mixed network topologies;
- It provides transmission sequencing and timing information as part of the Protocol Data Unit;
- It has extended data integrity information when compared to other known protocols;
- It provides validation of physical routes in mixed network topologies.

This new “channel” virtual communication resource presents the following qualities:

- It defines a standard software interface for accessing the new Data Link Layer protocol with a high level of abstraction;
- It provides data flow control supported by a means of estimating end-to-end delay experienced by a protocol data unit while traversing the network from its origin to its final destination.

1.3 Originality, generality and usefulness

The new Data Link Layer protocol presented in this work is original in the way it combines characteristics that are present in some communication protocols and absent in others to better serve systems installed in aerospace vehicles, supporting mixed topologies, providing additional transmission sequencing and timing information, as well as additional data integrity information. The associated “channel” resource offers a standard application programming interface for accessing the new protocol.

It is generic enough to be implemented on top of any hardware and software environment supporting the Ethernet IEEE-802.3 physical medium.

Its usefulness comes from offering a higher speed data communication means when compared with other more traditional communication protocols used in space applications, such as the MIL-STD-1553B (MIL STANDARD, 2020), and from being less costly to implement than ARINC-664 Part 7 (ARINC, 2009) in smaller aircraft.

1.4 Organization

The organization of this work is as follows:

- Chapter 2 introduces the basic concepts and associated literature required for the correct understanding of this text; it also reviews the characteristics of digital communication protocols which are relevant to this work and introduces important characteristics of networks serving embedded systems;
- Chapter 3 formulates the problem and presents one approach to its solution;
- Chapter 4 introduces the key objectives of the new Data Link Layer protocol;
- Chapter 5 details the specification of the new Data Link Layer protocol and the application programming interface for the “channel” virtual communication resource;
- Chapter 6 presents the implementation of a relevant case study and its results;
- Chapter 7 presents the conclusions and suggestions for further studies;
- The Appendices present complementary materials, such as details of the algorithm used for traffic control, configuration files, source code listings and the results generated by the case study in text format.

2 BASIC CONCEPTS AND LITERATURE REVIEW

2.1 Basic concepts and industry standards

The next five sections introduce basic networking concepts, associated industry standard literatures which are relevant to this work.

2.1.1 Computer network architectures

The correct understanding of this work requires some knowledge of architectures used in building computers networks (TANNENBAUM, 1989). In this section, the term “node” will be used freely to represent a single computer in a computer network.

Connecting computers became a necessity in the mid 70’s for a few reasons, but one very important: computers were very expensive, therefore sharing resources became strategic. If you needed to expand, it made more sense to acquire another computer tailored to your needs than to replace the one you already had by a bigger model. Luckily, computers those days enjoyed a quite long operational life: they remained operating for several years (quite commonly for 5 to 10 years).

Large computer networks appeared in the mid 80’s, when computers became smaller in size and less expensive.

In the early 70’s, it was already possible to connect two geographically separated computers using a private channel. The most common realization of this means of communication was over a telephone line. Binary digits were transformed into electric signals by a device called “modem”, which basically modulated an electric signal on transmission and demodulated it on reception (hence the name “modem” – agglutination of “modulator/demodulator”).

This “point-to-point” communication was enough for connecting two computers. If a third or fourth computer were involved, data had to be received and retransmitted to the next network node.

Even today, point-to-point communication is still used, in particular in the aerospace domain. In the late 70’s, a large computer network using point-to-point communication was created by the Advanced Research Projects Agency (ARPA) of the United States’ Department of Defense (DoD).

In 1973, XEROX Corporation developed a technical solution for connecting multiple computers influenced by a computer networking experiment conducted at the University of Hawaii: the “ALOHAnet”. Using a coaxial cable, the “Ethernet” cable, multiple computers at XEROX PARC laboratory could share a common physical medium, allowing any node to directly communicate with any other node in the network, a significant improvement over point-to-point communication (PERLMAN, 1999).

This shared medium arrangement had already posed a challenge to the creators of ALOHAnet: the recovery from the event of two computers “colliding” as they tried to transmit data at the same time. The researchers at the University of Hawaii came up with a simple protocol which inspired XEROX in the implementation of similar technique for properly handling these events: after failing to transmit by detecting a “collision”, the computers would have to wait a random chosen time interval before retrying.

This shared medium networking technology developed by XEROX attracted the attention of Digital Equipment Corporation and Intel. These three companies formed a consortium referred to as “DIX” and eventually introduced commercially the Ethernet technology as we know today.

In the mid 80’s, IBM adopted a design developed by researchers at the Cambridge University which arranged computers in a “ring”. This ring was formed by connecting computers point-to-point – the first to the second, the second to the third and so on – and closing the ring by connecting the last node to the first node (PERLMAN, 1999).

The transmission of data in this ring required the ownership of a special piece of data called “token”: only the computer in the possession of the token was allowed to transmit. After transmitting, the computer would then pass the token to its neighboring computer in the ring (IBM, 2013).

Ethernet evolved and managed to move away from the coaxial cable because it became a technical issue as computers in a network grew in number: cable lengths were limited and one would need eventually to replace the cable when the number of computers connected exceeded the limit allowed by a particular cable length.

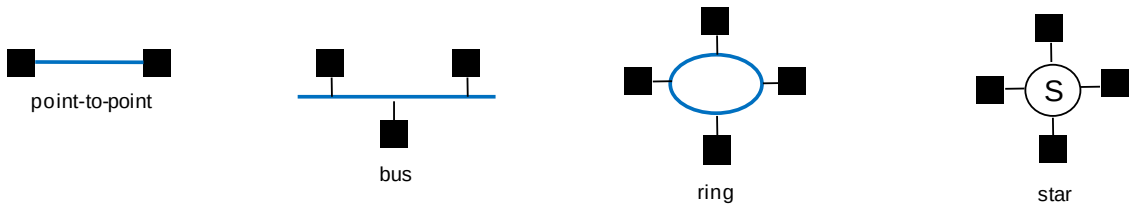
Devices called Ethernet “hubs” were developed, partly introducing a return to the point-to-point scheme of earlier computer networking days. Instead of connecting to a coaxial cable, computers using Ethernet would connect point-to-point to this Ethernet hub. This hub worked as a collapsed form of the traditional Ethernet cable. Installation and reinstallation of computers in an Ethernet network were facilitated and Ethernet hubs became commercially available with 8, 16 or more connection ports.

Ethernet hubs eventually turned into the Ethernet switches we recognize today in almost every household as part of the access-point hardware delivered by Internet service providers.

Occasionally, hubs were also called “star couplers” to denote the physical resemblance of computers connected to a hub and a planetary system, where planets are tied to a star by gravitational forces.

Today, the term “topology” is commonly used to describe how computers are arranged in networks. Computer network topologies were a consequence of technical decisions made by academic researchers and engineers trying to find solutions for real problems. Figure 2.1 summarizes elementary computer network topologies: a) Point-to-point; b) Shared Bus or simply Bus; c) Ring; d) Star. Naturally, more complex topologies can be obtained by combining one or more of these four.

Figure 2.1 – Elementary computer network topologies.



Another important component of computer network architectures is the communication medium access control. It was mentioned before that Ethernet, a bus topology in its origin, allowed for any two computers to start transmitting at any time and that a token was used to grant the privilege of transmitting in a ring topology to the computer which owns it. In short, access control to the

physical communication medium in computer networks can be done either: a) by using an arbitration protocol, which allows for one and only one transmission at any given point in time; b) by using no arbitration protocol, but providing a recovery mechanism in the event of a failed transmission.

Most computer networks operate over metallic of fiber-optic cables, but electromagnetic waves are also another viable medium. While the first can be constructed tolerant to electronic noise and harsh environments, open-air – or "wireless" – transmissions suffer greatly in the presence of natural phenomena, such as atmospheric discharges and heavy rain, and other radio transmissions from nearby sources.

The physical nature of the communication medium drives variations on the way access to the medium is controlled. For instance, wireless transmissions tend to avoid "collisions" as observed in the original Ethernet technology instead of reacting to them.

More recently, data security has become a great concern in computer networks operating in commercial aircraft because of the fear of "hacking", that is, a malicious attack which may result in loss of property and/or human lives. Networks which operate over cables, metallic or non-metallic, are more immune to attacks because it is necessary to have physical access to the network hardware. Networks which operate wireless can be protected against hacking by using encryption of data, but may still suffer in the presence of high-power electromagnetic transmissions causing what it is commonly called "denial-of-service".

One could argue that the term "architecture" should be used exclusively when speaking about "form", that is, what can be observed by the naked human eye. The term architecture in computer networks could limit the discussion around topologies only, which indeed define the form of a computer network. The careful reader will note however that this section, besides enumerating topologies, addresses also the physical medium and the type of control used in accessing it. The reason is simple: computer networks were conceived, implemented and perfected by combining these three elements: a) topology; b) physical medium; c) access control to physical medium. While the first one

provides a high-level perspective of the network, as it dictates its form, the latter two elements are its lowest levels.

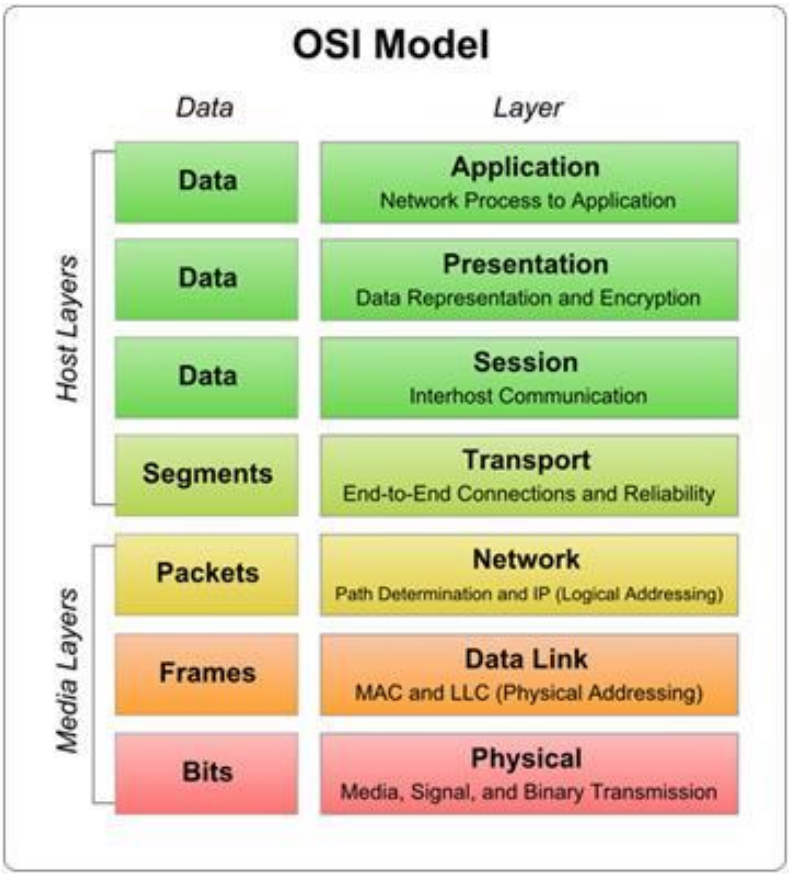
Researchers in Academia and Industry wrote the history of computer networks in the 70's and in the 80's. Today, we benefit from their hard pioneer intellectual work.

2.1.2 The ISO/OSI layered communication model

The Open Systems Interconnection model is a product of a project conducted by the International Standards Organization (ISO) and was published as a standard (7498) in 1984 (ISO, 1994).

It describes a seven-layer abstraction communication model, where one layer has to be concerned only with the interface to the layer immediately above it, which it serves, and the interface with the layer immediately below it, which it is served by.

Figure 2.2 – ISO/OSI Layered Communication Model.



Source: Adapted from ISO (1994).

The seven layers illustrated by Figure 2.2 are:

Layer 1: Physical Layer (lowest)

The physical layer is responsible for the transmission and reception of encoded binary digits over a transmission medium. Examples of Layer 1 protocols are IEEE 802.3 and Ethernet physical layers, serial transmission protocols such as RS-232, Universal Serial Bus (USB), IBM's Bluetooth, among others.

Layer 2: Data Link Layer

The data link layer provides actual data transfer between two directly connected nodes. Examples of Layer 2 protocols are IEE 802.3 (combined with IEEE 802.2 LLC) and Ethernet data link layers, Asynchronous Transfer Mode (ATM) for audio and video streaming, among others.

Layer 3: Network Layer

The network layer provides the transferring of variable length data structures (usually called "packets") from one node to another. The most famous example of a Layer 3 protocol is the Internet Protocol (IP), but others can be accounted for, such as Apple's Appletalk, Novell's Internetwork Packet Exchange (IPX) and Digital Equipment Corporation's DECnet.

Layer 4: Transport Layer

The transport layer provides the transferring of arbitrary length data sequences adding extra services such as segmentation (dividing a sequence into smaller pieces for transmission), error detection and recovery. Examples of Layer 4 protocols are Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Novell's Sequenced Packet Exchange (SPX).

Layer 5: Session Layer

The session layer provides establishing, managing and terminating virtual "connections" between a local and a remote application. Real-time Transport Protocol (RFC 3550) developed for audio and video streaming over IP is one of the few true Layer 5 protocols.

Layer 6: Presentation Layer

The presentation layer helps bridging different syntax and semantics between two Application Layer applications. One of the few Layer 6 protocol examples is the Multi-purpose Internet Mail Extensions (MIME), which allows for sending non-textual attachments over e-mail.

Layer 7: Application Layer (highest)

The application layer is the OSI layer closest to an end user software application. There is a multitude of Layer 7 protocols in use today: Hypertext Transfer Protocol (HTTP) used in the World-Wide-Web, File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), Simple Network Management Protocol (SNMP) and Telnet, all of them operating over IP.

The most frequent implementations of network protocols usually concentrate in the first four lower layers and the uppermost layer.

Physical Layer protocols implementing serial point-to-point communication, such as those belonging to the “Recommended Standard” (RS) family, are used for a multitude of upper layer protocols. Universal Serial Bus (USB) is another example rich example of flexibility, used for connecting portable storage devices, microphones, loudspeakers, video cameras, keyboards, pointing devices to personal computers, cell phones and modern TV sets.

For most applications, it is sufficient connecting directly the Data Link Layer to the Application Layer, that is, once the application receiving the data is identified, it should get it with as little processing delay as possible.

2.1.3 IEEE 802.3 standard for ethernet

The Ethernet protocol for networking communication developed by XEROX in 1973 was embodied by the IEEE 802.3 standard published in 1985 (IEEE, 2012). As Ethernet, it covers the first two layers of the ISO/OSI Layered model:

Layer 1 “Physical Layer” – standardizes all sorts of physical medium, from copper cables in various forms to fiber-optic cables, from transmission speeds starting at 10 megabits per second to 200 gigabits per second.

Layer 2 “Data Link Layer” – standardizes two sub-layers, the “Media Access Control” (MAC), basically the “Carrier Sense Multiple Access with Collision

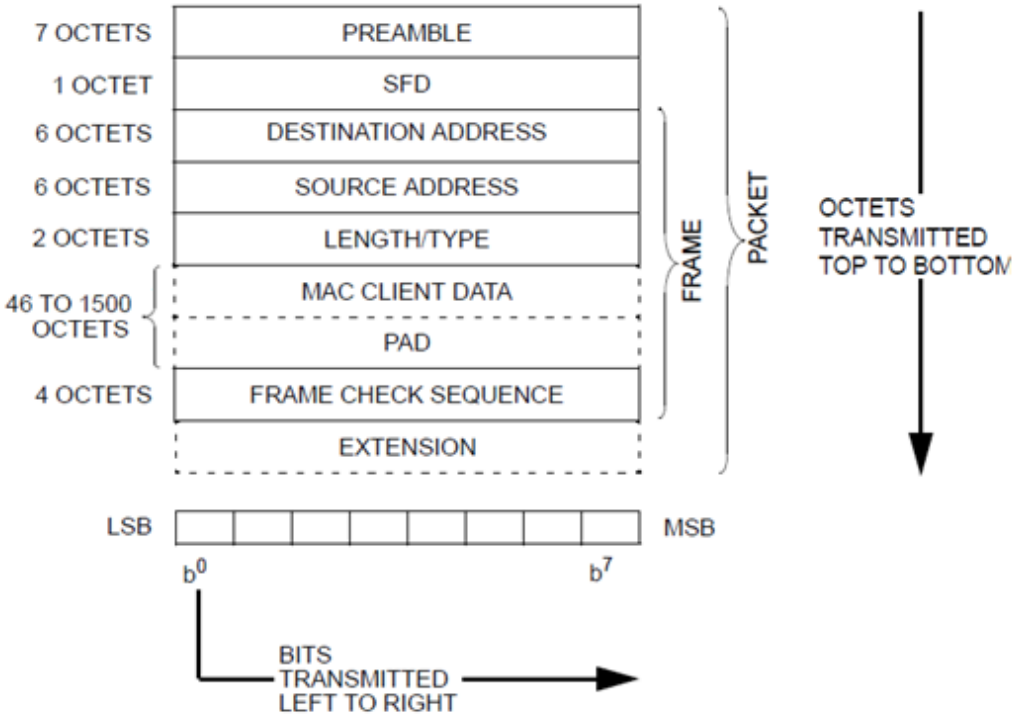
Detection” (CSMA/CD) method devised by XEROX for Ethernet, and the “Logical Link Control” (LLC) subject of the IEEE 802.2 standard (IEEE, 1998).

The IEEE 802.3 standard introduced a few changes in the original Ethernet data packet layout as illustrated in Figure 2.3. This new layout of the IEEE 802.3 data packet was a result of the standardization committee efforts in adding certain connection services directly into the Data Link Layer instead of leaving it to upper layers as Ethernet does (see next section).

The first 8 bytes of the Ethernet preamble, a fixed sequence of bits used to identify the beginning of a valid data packet after an idle period in the transmission medium, was divided in the IEEE 802.3 standard into two fields: a 7-byte “Preamble” copied from the first 7 bytes of the original Ethernet preamble and a 1-byte “Start-of-Frame-Delimiter” copied from the last 1 byte of the original Ethernet preamble.

The next 12 bytes following the Preamble, however, remained the same. The first 6 bytes are used by Ethernet and IEEE 802.3 as the “Destination Address” of the targeted MAC sub-layer, as the next 6 bytes, which are used as the “Source Address” of the sending MAC sub-layer.

Figure 2.3 – IEEE 802.3 Frame format.



Source: IEEE (2012).

The bit transmitting order in both Ethernet and IEEE 802.3 is least-significant bit first and the first 2 bits to be decoded by the receiving end have a special meaning in both standards. The first bit determines whether the MAC Destination Address is an “individual” (unicast transmission) or “group“ (multicast transmission). The second bit determines whether the MAC Destination Address is “locally administered” or “globally administered”. An all-1s MAC Destination Address (0xFFFFFFFFFFFF) is interpreted as a broadcast transmission.

It is important to point out that MAC addresses have a building rule according to IEEE, which includes a leading 3-byte field called “Organizationally Unique Identifier” (OUI). Each company manufacturing devices that can be attached to an Ethernet or IEEE 802.3 network uses its own OUI to uniquely identify each piece of equipment produced. Since OUI occupies the first 3 bytes of the MAC address, it is usually a number multiple of 4 (with a few exceptions). This is rather convenient, for it leaves untouched the first 2 LSBs which have the special use just described.

The following 2-byte field used by Ethernet as “Type” (the “Ethertype”) to define what sort of upper layer protocol was carried by the data packet was used in the IEEE 802.3 standard either as “Length” or as “Type” in a clear attempt to reconcile the intention of the IEEE standardization committee in embedding the identification of the protocol carried by the data packet into the Data Link Layer and the already existing large customer base Ethernet in the beginning of the 80’s.

This “reconciliation” rule is simple, as stated in the IEEE 802.3 standard document (IEEE, 2012):

a) If the value of this field is less than or equal to 1500 decimal (0x05DC hexadecimal), then the Length/ Type field indicates the number of MAC client data octets contained in the subsequent MAC Client Data field of the basic frame (Length interpretation).

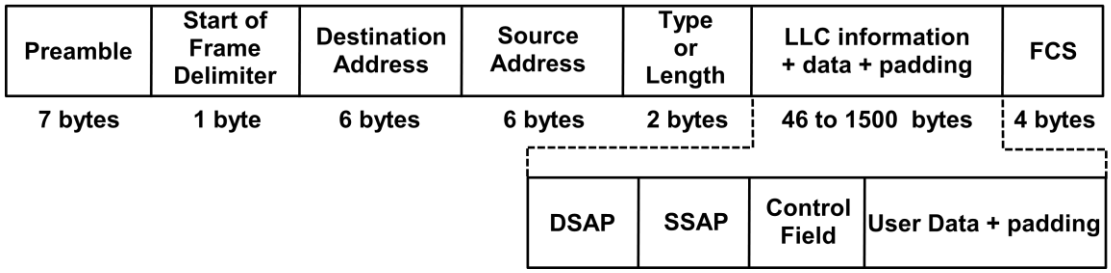
b) If the value of this field is greater than or equal to 1536 decimal (0x0600 hexadecimal), then the Length/Type field indicates the Ethertype of the MAC client protocol (Type interpretation).

The Length and Type interpretations of this field are mutually exclusive.

As a measure of how infrequent is the use of the Length interpretation, suffice it to say that the Internetworking Protocol known to us as IP has a Type of value 0x0800 hexadecimal, therefore falling in the case (b) above.

The embedding of the identification of the protocol carried by IEEE 802.3 packets when the Type/Length field is interpreted as Length is covered by the IEEE 802.2 standard addressed in the next section.

Figure 2.4 – IEEE 802.3 Frame with IEEE 802.2 LLC field.



2.1.4 IEEE 802.2 logical link control

This IEEE 802.2 standard (IEEE, 1998) is not new. Its first supplements were published in 1993. The last and final version was published in 1998. It covers the “Logical Link Control” (LLC) sub-layer of the Data Link Layer of the IEEE 802.3 standard.

As stated in the standard’s text:

“*This International Standard provides a description of the peer-to-peer protocol procedures that are defined for the transfer of information and control between any pair of data link layer service access points on a LAN*.”

The standard describes “service” as a means of accessing capabilities provided by upper communication layers. Using a “Service Access Point” (SAP) is how one reaches a particular service. A SAP can be understood as a logical construct managed by a software component belonging to an upper network layer.

The Logical Link Control (LLC) is defined as the upper sub-layer of the Data Link Layer, where Media Access Control (MAC) is the lower sub-layer. LLC describes the connection services available to SAPs.

The data structure used in LLC is called “Protocol Data Unit” (PDU). The PDU has following fields, as illustrated in Figures 2.5 and 2.6:

Address Fields:

Destination Service Access Point (DSAP) – Contains the destination SAP of the PDU.

Source Service Access Point (SSAP) – Contains the source SAP of the PDU.

Control Field - Designate command and response functions, may contain sequence numbers when required.

Information Field – Contains zero or more bytes of information.

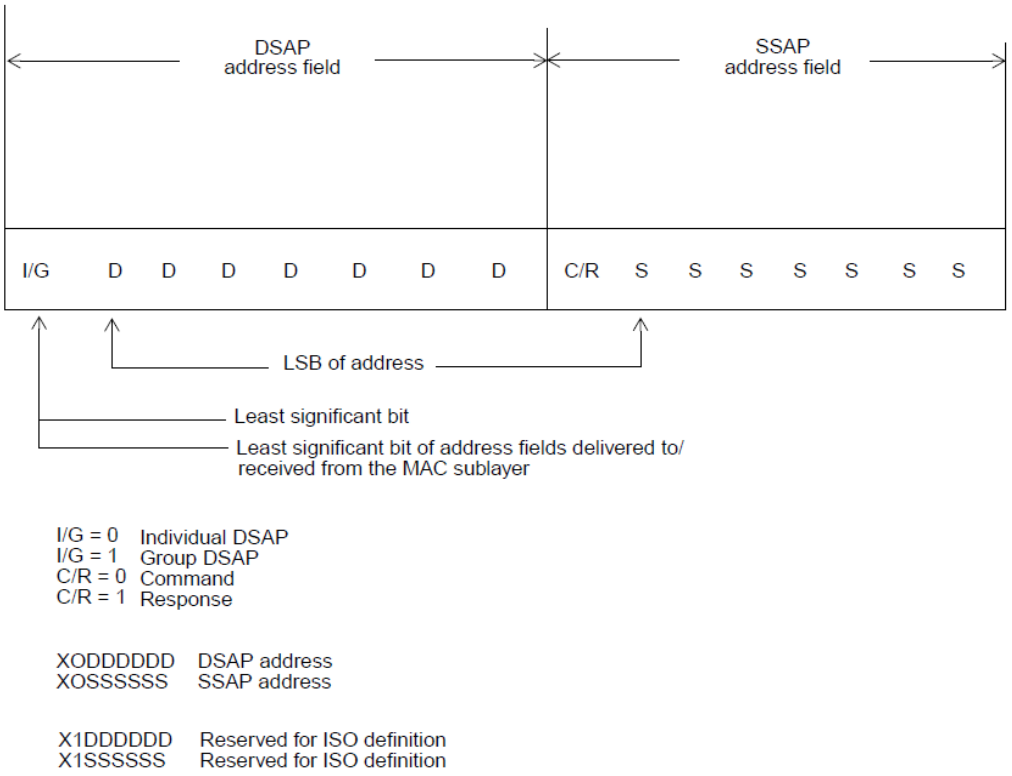
Figure 2.5 – IEEE 802.2 LLC Fields.

DSAP address	SSAP address	Control	Information
8 bits	8 bits	8 or 16 bits	M*8 bits

DSAP address	=	Destination service access point address field
SSAP address	=	Source service access point address field
Control	=	Control field [16 bits for formats that include sequence numbering, and 8 bits for formats that do not (see 5.2)]
Information	=	Information field
*	=	Multiplication
M	=	An integer value equal to or greater than 0. (Upper bound of M is a function of the medium access control methodology used.)

Source: IEEE (1998).

Figure 2.6 – IEEE 802.2 Address Fields.



Source: IEEE (1998).

Each SAP address field has seven bits of actual address and one least significant bit used in the DSAP address field to identify the DSAP address as either an individual (bit set to 0) or a group address (bit set to 1) and in the SSAP address field to identify that the LLC PDU is a command (bit set to 0) or a response (bit set to 1). The second least significant bit set to 1 indicates a reserved address.

All 1's in the DSAP address field is said to be the "Global" DSAP address and all 0's in the DSAP or SSAP address field is said to be the "Null" address.

In this work, both DSAP and SSAP addresses will use the least significant bit set to 0 (thus following a 0xxxxxxx format), giving 126 non-null different SAP addresses (all even decimal numbers).

The standard defines three different types of services:

Type 1 Operation: PDUs shall be exchanged between two LLC layers without the need for the establishment of a data link connection.

Type 2 Operation: A data link connection shall be established between two LLC layers prior to any exchange of information-bearing PDUs.

Type 3 Operation: PDUs shall be exchanged between two LLC layers without the need for the establishment of a data link connection, but the receiving side has to acknowledge the reception of any data back to the sending side.

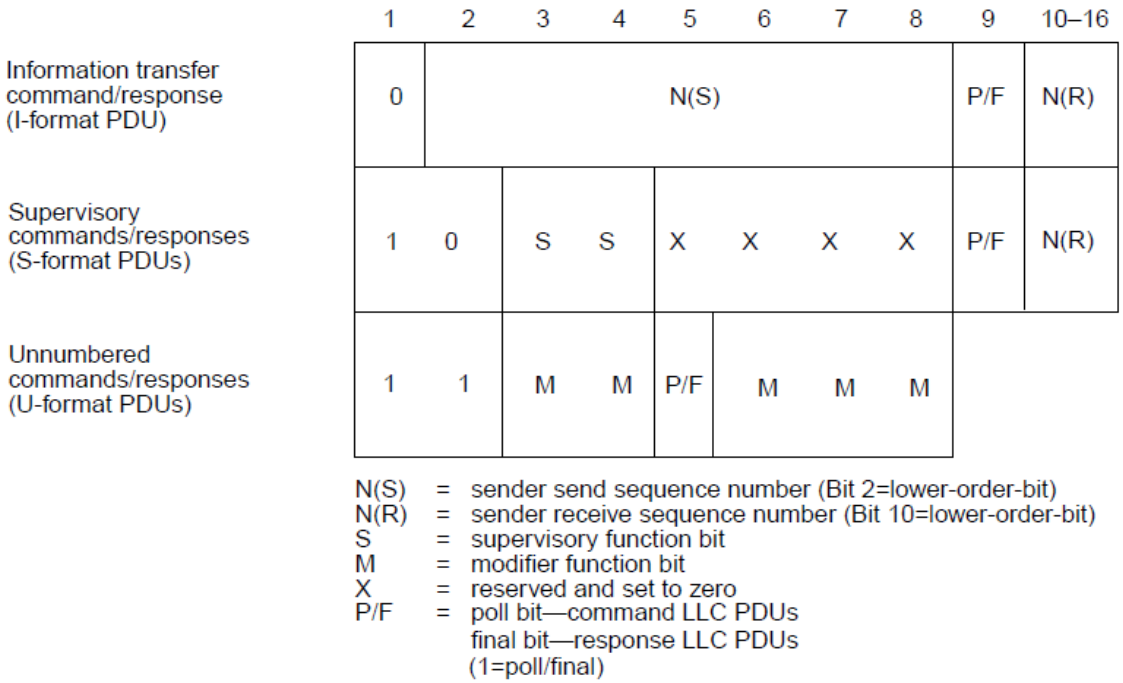
There are three different Control Field formats, as illustrated in Figure 2.7:

Information transfer format: The I-format PDU shall be used to perform numbered information transfer in Type 2 operation.

Supervisory format: The S-format PDU shall be used to perform data link supervisory control functions in Type 2 operation.

Unnumbered format: The U-format PDUs shall be used in Type 1, Type 2, or Type 3 operation to provide additional data link control functions and to provide unsequenced information transfer.

Figure 2.7 – IEEE 802.2 PDU Control Field.



Source: IEEE (1998).

To this work, only Type 1 Operation and Unnumbered Command/Response (U-format PDUs) will be relevant.

There are three types of U-format Commands and Responses PDUs in Type 1 Operation, as illustrated in Figure 2.8:

Unnumbered information (UI) Command

The UI command PDU shall be used to send information to one or more LLCs. There is no LLC response PDU to the UI command PDU.

Exchange identification (XID) Command/Response

The XID command PDU shall be used to convey the types of LLC services supported (for all LLC services. The XID response PDU shall be used to reply to an XID command PDU at the earliest opportunity

Test (TEST) Command/Response

The TEST command PDU shall be used to cause the destination LLC to respond with the TEST response PDU at the earliest opportunity, thus performing a basic test of the LLC to LLC transmission path.

Figure 2.8 – Type 1 operation command control field bit assignments.

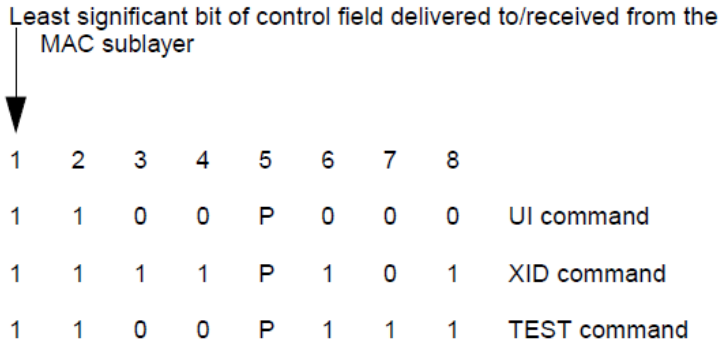
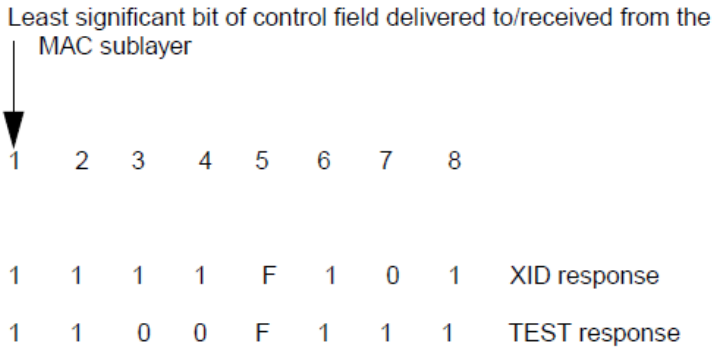


Figure 2.9 – Type 1 operation response control field bit assignments.



Source: IEEE (1998).

To this work, only UI and TEST PDUs will be relevant.

The only difference between Type 1 Operation Command and Response PDUs is on the fifth least-significant bit of the Control Field named “Poll/Final” (P/F) bit. On a XID or TEST Command and Response PDUs, the P/F bit shall be set to 1 and on a UI Command it shall be set to 0.

In the IEEE 802.2 standard there is an extension called “Subnetwork Access Protocol” (SNAP) which was created to provide to upper layer protocols the same Ethertype field used in Ethernet.

The SNAP header consists of the 3-byte Organizationally Unique Identifier (OUI) followed by a 2-byte Protocol ID. If the OUI is value 0x000000 hexadecimal, the protocol ID is the Ethernet Type field value for the protocol running on top of SNAP. If the OUI has a non-zero value, the 2-byte Protocol ID field points to a vendor-specific protocol.

The SNAP header is to be found only in UI PDUs which have both DSAP and SSAP fields filled with value 0xAA hexadecimal and Control Field set to value 0x03 hexadecimal (the first two least-significant bits set as required for U-format PDU and P/F bit set to 0).

The historic reason for this apparently unnecessary complication, which in fact reduces the number of usable data bytes in the IEEE 802.3 network packet, was that in its design the LLC Service Access Point (SAP) is just 7 bits long, allowing for at most 128 different combinations. As vendors started registering more and more different communication protocols, it became clear to IEEE that soon 8 bits would not be sufficient. As a result, the SAP value 0xAA hexadecimal was reserved and the SNAP extension created.

2.1.5 Internet Protocol (IP)

In May 1974, the Institute of Electrical and Electronics Engineers (IEEE) published a paper entitled "A Protocol for Packet Network Intercommunication" The paper's authors, Vinton G. Cerf and Robert E. Kahn (CERF, 1974), described an network protocol for sharing resources using packet switching among network nodes. A central control component of this model was the "Transmission Control Program" that incorporated both connection-oriented links and datagram services between nodes. The monolithic Transmission

Control Program was later divided into a modular architecture consisting of the “Transmission Control Protocol” (TCP) at the Transport Layer and the “Internet Protocol” (IP) at the Network Layer. This layered model became known as the United States’ Department of Defense (DoD) “Internet Model and Internet protocol suite”, and informally as “TCP/IP”.

The Advanced Research Projects Agency Network (ARPANET) initially funded by the Advanced Research Projects Agency (ARPA) of the DoD (DARPA, 1981) was the first network to implement the TCP/IP protocol suite.

IP versions 0 to 3 were experimental versions developed between 1977 and 1979. The protocol version in use today is version 4 (IPv4) introduced in September of 1978 and “4” is the protocol version number carried in every IP packet connecting computers, mobile phones and other communication devices all over the world.

The successor to IPv4 is IPv6. Its most prominent difference from version 4 is the size of the addresses. While IPv4 uses 32 bits for addressing, yielding 4.3 billion (4.3×10^{9}) different addresses, IPv6 uses 128-bit addresses providing 3.4×10^{38} different addresses.

Another Transport Layer protocol operating over IP is worth mentioning: the “User Datagram Protocol” (UDP) which was introduced in 1980 (COMER, 1995). Unlike TCP, which implements a reliable connection-oriented communication between network nodes, UDP is suitable for purposes where error checking and recovery are either not necessary or are performed at the Application Layer. UDP avoids the overhead of such processing in the protocol stack. Time-sensitive applications often use UDP because dropping packets is preferable over waiting for packet retransmission, which may not be an option in systems operating under real-time constraints.

2.2 Embedded networks

Networks designed to be installed in a completely segregated environment such as in aerospace vehicles present particularities that differentiate them from those originally developed for commercial networks.

Embedded networks serve the purpose of connecting functions hosted by electronics modules that interact to serve a greater purpose, for instance, providing global communication or Earth climate survey.

These systems need to have their behavior predicted during their design; therefore networks need to present a level of deterministic behavior while connecting functional elements of such systems.

These elements can be data producers, data consumers or both, and the relations between them are in general defined quite early in the system design phase.

Further, certain functions which give an embedded network the desired deterministic behavior are created and deployed at network elements as needed.

The next three sections explore aspects of embedded networks and their relevance to the design of the system and to the network protocol serving it.

2.2.1 Data producers and data consumers

In a complex and high integrated distributed processing system, in particular those present in aerospace vehicles, it is essential for a proper design to identify how engineering data flows from one part of the system to other parts of the system.

For instance, positioning data produced by a sensor installed in a satellite which tracks the Sun needs to flow to the energy supply subsystem which is responsible for moving the solar panels for optimal electric power generation. In a “fly-by-wire” flight control system present in modern aircraft, data must flow from pressure sensors calibrated for indicating altitude and airspeed, from accelerometers calibrated for indicating body acceleration, from the engines and from the pilot command inceptors to a central computer which is responsible for properly moving flight control surfaces ensuring a smooth flight path.

The communication paths connecting parts of a distributed system are the result of an analysis identifying Data Producers and Data Consumers.

The important questions that need to be answered are:

- Which information is required for the system to operate as designed?
- Which parts of the system produce what information?
- Which parts of the system consume what information?

Once Data Producers and Data Consumers are connected, a basic system topology emerges. It may indicate that point-to-point, star or a mix of the two topologies may seem more appropriate. However, other aspects of the communication infrastructure need to be addressed, such as physical distance between transmitters and receivers and any timing requirements that may affect how well Data Consumers process received data. These two aspects and perhaps others may alter the initial perception of the most suitable network topology for a given system and may limit the choice of the physical transmission medium.

The format in which data is produced and consumed is also very relevant. Sensors most commonly convert a physical quantity, such as air pressure, into a voltage level which can be calibrated to express a measure of altitude in meters or feet. Modern sensors can provide digital data, but it is not uncommon that their output also need some form of calibration. Further, if a system using a sensor for producing pressure altitude in meters needs to send this data to a system which consumes pressure altitude in feet (such as the Multi-function Display in the airplane cockpit), it must be converted before it is consumed. If mathematical operations are required for data formatting, care must be taken not to deteriorate the resolution required for the proper use of the data.

2.2.2 Essential services in embedded networks

Letting devices communicate over a network in a complex and highly integrated processing environment onboard aerospace vehicles is quite an engineering challenge.

Unlike a network in a household where any configuration is almost never required, every aspect of the data exchange between any two participants in such embedded network has to be identified and documented. For this task, it is usual to produce “Interface Control Documents” (ICD) describing messages being transmitted by one software application in one network node and being received by an application (or applications) in one or more network nodes.

In general, networks connecting devices in aerospace vehicles have neither spurious messages nor unplanned communication paths: everything is pre-planned and rigorously tested before entering operation.

Certain pieces of software, such as Attitude Control in satellites or Flight Controls in aircraft are very sensitive to unplanned data transport delays while crossing a communication channel. In such cases, system designers strive to ensure communication determinism, that is, the behavior of the network when in operation can be predicted while the whole system is still in its design phase.

Networking in closed environments such as in aerospace vehicles involves aspects other than simply transmitting and receiving binary data. For instance, a system designer may want to restrict the amount of data one communication is supposed to carry per unit of time, or may want to make sure that one particular message goes to one node and not to any other node. These and other design concerns have driven the implementation of certain services present in complex networking scenarios, in particular those found in modern aerospace vehicles.

These services serving different network layers are listed in Table 2.1:

Table 2.1 – Services at different ISO/OSI protocol layers.

ISO/OSI Layer	**Services**
Physical Layer	Data Encoding Data Decoding
Data Link Layer	Media Access Control Data Validation Routing Traffic Shaping Traffic Policing
Network Layer	Data Validation Fragmentation Defragmentation Routing
Transport Layer	Data Validation Error Recovery

Data Encoding and Data Decoding at the Physical Layer can represent more or less binary data transmitted per a complete sine wave, while different Media Access Control strategies at the Data Link Layer may represent more or less transport delay in case of a failure accessing the physical medium.

Different checksum algorithms may represent higher or lower statistical probability of accepting corrupted data as valid at the Data Link Layer, and different message routing implementation may introduce more or less transport delay when data has to be retransmitted to another network node. One must note that Data Validation is not the sole privilege of the Data Link Layer.

At the Data Link Layer it is also possible to protect a communication path from a misbehaving node by implementing Traffic Shaping and Traffic Policing, that is, “shaping” or constraining outgoing traffic and “policing” or forbidding incoming traffic according to some mathematical rule.

Fragmentation and Defragmentation (or reassembly) are usual at the Network Layer, because Transport Layer protocols tend to be agnostic of the limitations imposed by the physical medium with respect to the quantity of data transmitted in a single operation.

Some form of Error Recovery is more common at the Transport Layer, whereby any inconsistency found processing the received data is communicated back to the transmitting node. At the Transport Layer is also where the upper layer protocols using the Internet Protocol (IP) are identified, for instance, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). However, features similar to those provided by IP, TCP and UDP are available in IEEE 802.2 specification at the Logical Link Control (LLC) part of this standard’s Data Link Layer.

Depending on a design decision, these services can be allocated at different hardware and software elements involved in data communication across a network. For instance, a commercial Ethernet network card could implement all the above services of the Data Link Layer, but usually restricts itself to do Media Access Control (it makes sure that physical network addresses are properly managed) and Data Validation (it makes sure that the received data is checked against the data checksum present in the last 32 bits of the received frame).

One node in a more complex network such as those following a multi-star topology using network switches should implement Traffic Shaping by design for restricting the bandwidth on a transmitting node. By the same token, a network switch in such topology scenario should implement Traffic Policing for protecting the network from a node that does not perform the required Traffic Shaping properly.

2.2.3 Essential characteristics of embedded network protocols

Historically, network protocols were developed as a response to a real life need.

The original Ethernet was conceived inside XEROX Palo Alto Research Center (PARC) laboratory because there was a need for connecting workstations to a very expensive high-speed laser printer. The development of the Transmission Control Protocol (TCP) and the Internet Protocol (IP) was sponsored by the United States' Department of Defense (DoD) within the scope of a project conducted by the Defense Advanced Research Projects Agency (DARPA) for connecting geographically separated networks.

Design of networks expected to be installed in aerospace vehicles may benefit from the fact that nodes are just meters apart in an aircraft and confined in a less than one cubic meter space in a satellite, if we consider only installation issues. However, characteristics such as reliability and flexibility must be offered by embedded networks similarly to any other higher scale network.

Following the most important industry standards that support the development of this work, ARINC-429, ARINC-664 Part 7 and SpaceWire, all described in the next sections, the following high-level characteristics derived from system design concerns should be present in any embedded network protocol:

STATICALLY CONFIGURED

> The configuration of the protocol layers shall be statically defined and shall not change while the network is in operation.

FLEXIBILITY

> The services expected to be performed at the protocol layers shall be allocated on network elements where they can best preserve data integrity

without penalizing the end-to-end transport delay experienced while crossing the network from a source to a destination.

RELIABILITY

In the absence of any physical or electromagnetic interference, the protocol layers shall preserve data integrity as data flows from one source to one or more destinations.

MULTICASTING

The protocol layers shall permit one-to-one as well as one-to-many communication paths.

FORWARDING

The protocol layers shall permit a node to forward data to a destination other than the node itself.

ERROR DETECTION

Each protocol layer shall provide a means of detecting errors when validating data received from the protocol layer immediately below.

FLOW CONTROL

There shall be a form of limiting the data flow going out of or coming in to any network element at a protocol layer level according to fixed design parameters.

Other concerns related to the Physical Layer such as coding efficiency (number of significant binary digits transmitted per unit of time) and transmission rate (raw binary digits transmitted per unit of time) are not listed because they exceed the scope of this work, but they are not less important in the implementation of any network.

These characteristics are appropriate to the protocol levels at the Data Link and Physical Layers. However, to an application belonging to the highest level of the 7-layer ISO/OSI model, it is essential to have a standard interface for accessing the desired protocol. The Ethernet MAC Layer allows for encapsulating multiple protocols, but only the IP protocol provides similar means over a standard software interface called “sockets”.

This approach was used by the creators of the IP protocol, is relevant to this work and will be introduced in the next chapter. There, a new protocol and a programming interface will be proposed, which combined provide a network communication service to high-level applications not similar in the protocols reviewed.

2.3 Digital Communication Protocols

This section describes characteristics of the three Digital Communication Protocols (DCP) which became an inspiration for this work, namely ARINC-429, ARINC-664 Part 7 (AFDX™) and SpaceWire. In addition, because of its wide spread use in military aircraft and in spacecraft applications, the characteristics of the military standard MIL-STD-1553B are described in separate, followed by other DCPs which find application the aerospace industry grouped by three categories: serial communication, shared medium and time-triggered.

2.3.1 ARINC-429

“ARINC” stands for Aeronautical Radio Incorporated and “429” is the associated number to the specification document named “Mark 33 – Digital Information Transfer System (DITS)”. The ARINC-429 specification (ARINC, 2001) was officially published in 1978 and usually referred to as a “unidirectional”, “multi-drop” serial bus.

ARINC-429 allows point-to-point communication with the addition of an important feature: it allows for multiple receivers (up to 20) for one transmitter (hence the “multi-drop” attribute).

ARINC-429 physical medium is a 78Ω twisted-pair shielded copper cable. Data transmissions use Bipolar-Return-to-Zero (BP-RZ) encoding (transition from high-to-low or low-to-high voltage levels at half bit-time). To protect data transmissions from interference, ARINC-429 uses two wires mirror-imaging voltage levels on them between -10V and +10V. Allowed transmission speeds are 12.5kbps or 100kbps with 4 bit-times bus idle (at 0 Volts) between two consecutive data words.

Since ARINC-429 allows for only one transmitter on the physical medium, it does not require any access control to it. If Node A needs to communicate with Node B it uses one cable; if Node B needs to communicate back with Node A, it must use a second cable (hence the “unidirectional” attribute). Normally, devices communicating over ARINC-429 have separate circuitries for transmitting and receiving.

ARINC-429 words are 32 bits long with up to 19 bits of data, 1 bit of odd-parity and 12 bits overhead, including an 8-bit frame identifier (the “Label”).

Mostly because of its simplicity and reliability, communication links following the ARINC-429 specification are in current use, unmodified since its formal publication.

2.3.2 ARINC-664 Part 7 (AFDX™)

The 7th part of the ARINC-664 specification was developed around a concept created inside Airbus called “AFDX™” for the A380 project (FUCHS, 2012). AFDX is now a brand name which restricts its use in commercial products.

The ARINC-664 Part 7 specification received the title “AVIONICS FULL-DUPLEX SWITCHED ETHERNET” (ARINC, 2009). It describes what was called “Determinist Networks” within the more general concept of Aircraft Data Networks (ADN) introduced by the ARINC-664 specification, which now has 8 parts:

Part 1 - Systems Concepts and Overview

Part 2 - Ethernet Physical and Data-Link Layer Specifications

Part 3 - Internet-based Protocols and Services

Part 4 - Internet-based Address Structure and Assigned Numbers

Part 5 - Network Domain Characteristics and Functional Elements

Part 6 - Reserved;

Part 7 - Deterministic Networks

Part 8 - Upper Layer Protocol Services

According to the text, the Part 7 describes a special case of what the ARINC-664 specification calls “profiled networks”, which in turn is a special case of “compliant networks” (refer to ARINC-664 Part 1). The networks which are compliant with ARINC-664 are IEEE 802.3 and IP. The term “profiled” refers to some restrictions imposed to both IEEE 802.3 and IP networks, for instance, network addresses shall be fixed in each specific installation.

The term “SWITCHED” in the title of the ARINC-664 Part 7 (A664-P7) specification suggests that this type of network requires a switching hardware; therefore it follows a star topology which can be combined into a multi-star topology. Nodes in an A664-P7 network are called “End-Systems” (E/S).

Also according to the text, the most important feature of A664-P7 networks is “Quality-of-Service” (QoS), in particular timely delivery of data. To achieve this goal, several special elements were introduced, modifying how data packets are assembled and delivered throughout the network.

The first important element introduced is called “Virtual Link” (VL). The VL is a unidirectional logical communication link with guaranteed bandwidth. Being unidirectional has its consequences in an A664-P7 network: if E/S X needs to communicate to E/S Y it needs one VL and if E/S Y needs to communicate back to E/S X it needs a second VL.

It is interesting to note that this unidirectional characteristic of the VL is implemented in a full-duplex IEEE 802.3 physical medium. This means that an End-System can simultaneous transmit and receive data using a single cable but using two VLs. If one compares this situation with the one described in the previous section, he or she will immediately find similarities – at least from a logical perspective – with two nodes communication via ARINC-429. No surprises here, because of one of the key issues driving Airbus toward AFDX was precisely the virtualization of an ARINC-429 point-to-point network on an IEEE 802.3 infrastructure with additional benefits, such as electrical cabling simplification and a 100-fold increase in transmission speed.

The A664-P7 specification itself recognizes this by saying: "*In a system with many end points, point-to-point wiring is a major overhead. Ethernet networks can offer significant advantages and a suitable model for a deterministic network can be obtained through emulating the point-to-point connectivity.*"

The multi-drop attribute of ARINC-429 data bus is usually realized by splitting cables or working on cable connectors. The same multi-drop attribute applies to A664-P7 networks, however through another important element: the A664-P7 Switch. The technical specifications of this type of equipment were derived from those found in Ethernet Switches, but with special attention to the restrictions imposed by the ARINC-664 specification.

Another consequence of the multi-drop attribute of A644-P7 networks is that VLs must support multicast transmissions. This is realized by using special classes of IEEE 802.3 and IP network addresses built around VLs.

The A664-P7 Switch can provide Traffic Policing as any commercially available Ethernet Switch (CISCO, 2020), and policing is essential to the deterministic nature of A664-P7 networks. However, a A664-P7 Switch is not allowed to "auto-discover" routing paths for network data packets as any Ethernet Switch does: routing paths must be statically configured for each VL and made effective at A664-P7 Switch power-on.

While A664-P7 Switches are expected to do Traffic Policing on incoming network traffic, another element is required to secure bandwidth to VLs. An End-System that transmits data on A664-P7 networks need to provide Traffic Shaping on each VL, that is, no VL is allowed to transmit more than it is expected to. Traffic Shaping on commercial networks is present only on Ethernet Switches (CISCO, 2020).

On A664-P7 networks, two parameters are used for defining the allowed bandwidth of a VL:

Lmax: the maximum packet size a VL can transmit expressed in bytes;

Bandwidth Allocation Gap (BAG): the minimum amount of time separating two consecutive data packets transmitted on the VL expressed in milliseconds.

The bandwidth for a VL is defined by the quantity (Lmax+20)/BAG.

While performing Traffic Shaping on each VL it uses, an A664-P7 End-System needs to take into account an effect called “transmission jitter. VLs are streams of data that share the same physical network port; therefore the transmission carried out on one VL may suffer interference from transmissions from other VLs, since data packets line up for reaching the physical medium.

This transmission jitter is the maximum amount of time one expects to affect the BAG of a particular VL. If the maximum and the minimum amount of time observed on a VL separating two consecutive data packet transmissions are BAG plus X and BAG minus Y milliseconds respectively, the transmission jitter is the quantity X plus Y for that particular VL.

The transmission jitter is an important quantity for the Traffic Policing function performed by the A664-P7 Switch. If an A664-P7 End-System is responsible for shaping network traffic on each VL, the A664-P7 Switch is responsible for policing the incoming traffic in for each VL. Without transmission jitter, policing is simple and it is sufficient to verify that the bandwidth associated to a VL is not exceeded. With transmission jitter, a VL is allowed an overdraft to compensate for oscillations in the data packet transmission period represented by the parameter BAG.

Traffic Shaping and Traffic Policing working together should give A664-P7 networks its “deterministic” behavior, although it would be more appropriate to describe this quality as “bounded data delivery”. After all, what analytic methods permit in A664-P7 networks is the estimation of a bound for the arrival pattern of network data packets.

A third important element in A664-P7 networks is the introduction of the concept of “ports” through the use of the User Datagram Protocol (UDP) over IP. A port is a virtual construct that allows data exchange between applications running in

different network nodes and is present both in UDP and TCP protocols. However, the usage of a port in A664-P7 forces the operating system environment in an End-System to be compliant with the ARINC-653 specification (ARINC, 2015), which introduces the concepts of “sampling port” and ”queuing port” originally intended for process-to-process communication within the same operating system environment (ALENA, 2007). Binding ARINC-653 queuing and sampling ports to A664-P7 UDP ports is realized through proprietary solutions which do not share a common software interface.

Since A664-P7 use IEEE 802.3 and IP, fragmentation of data packets is supported. On IP, fragmentation is governed by the quantity Maximum Transfer Unit (MTU) expressed in bytes: any data packet with size bigger than MTU is split in two or more fragments reassembled at the receiving end. With IP over IEEE 802.3, the value of MTU is 1500 bytes, on A664-P& networks; MTU is equal to the parameter Lmax for each VL. That is, on an A664-P7 network, a data packet with size bigger than Lmax for a particular VL is split in two or more fragments. According to the ARINC-664 Part 7 specification, the maximum packet size allowed is 8192 bytes.

Data encoding on A664-P7 networks follows the IEEE 802.3 paradigm at 100 megabits per second (Manchester encoding also called Phase Encoding or PE). Data packets follow the UDP/IP over IEEE 802.3 paradigm with special rules for assembling IEEE 802.3 and IP destination and source addresses. And one important, 1-byte sized exception: A664-P7 data packets are numbered from 0 to 255 using a field located at the very end of each data packet called “Sequence Number” (SN). Because of this SN field, the maximum payload size of an A664-P7 data packet is one byte less than that of a normal UDP/IP over IEEE 802.3 data packet.

The SN field is the resource chosen for implementing two special layers into the otherwise IEEE 802.3/Ethernet standard compliant A664-P7network. These are called “Redundancy Management” (RM) and “Integrity Checking” (IC). Data packet transmissions on A664-P7 networks occur using two redundant physical links which transport two identical copies of each data packet. The IC layer

checks whether data packets have consecutive SN and the RM layer discards the second copy once it receives and validates the first copy.

The ARINC-664 Part 7 specification became the “de facto” standard for large avionics networks since its formal publication in 2005.

2.3.3 SpaceWire

Since the SpaceWire standard was publishedby the European Cooperation for Space Standardization in January 2003 (under the reference number ECSS-E-50-12A), it has been adopted by ESA, NASA, JAXA and ROSCOSMOS (the State Space Corporation of the Russian Federation). It is being used today on many high-profile scientific, Earth observation and commercial missions, including Gaia, ExoMars, BepiColombo, the James Webb Space Telescope, GOES-R, Lunar Reconnaissance Orbiter and ASTRO-H (PARKES et al., 2005).

In its own words, “*SpaceWire links are full-duplex, point-to-point, serial data communication links*” (ECSS, 2008). However, it actually supports multi-star topologies with the introduction of “routing switches” and an associated routing protocol.

The SpaceWire standard is divided into “clauses”, six of them dedicated to protocol levels:

Clause 5 (Physical Level) covers cables, connectors, cable assemblies and printed circuit board tracks.

Clause 6 (Signal Level) deals principally with electrical characteristics, and coding and signal timing.

Clause 7 (Character Level) describes how data and control characters are encoded.

Clause 8 (Exchange Level) presents the way in which a SpaceWire link operates including link initialization, normal operation, error detection and error recovery.

Clause 9 (Packet Level) describes the way in which data is encapsulated in packets for transfer across a SpaceWire network.

Clause 10 (Network Level) deals with the structure and operation of a SpaceWire network.

There is no official pairing of SpaceWire clauses to the ISO/OSI 7-layer network reference model.

SpaceWire was designed for moving large amounts of data reliably between two electronic units installed in a spacecraft. It provides mechanisms for securing link stability and link recovery following detection of an error condition and also a mechanism for finding alternate data traffic routes to overcome occasional link congestion. It also provides flow control on both transmitting and receiving sides of each node.

The packet structure is very simple: it defines a header which contains the routing information, a payload and an end-of-packet marker.

Data inside the packet is encapsulated as “Characters”. They can be either 10-bit “Data Characters” or 4-bit “Control Characters”. One particularly important Control Character is the “Flow Control Token” (FCT), used in regulating traffic between two nodes.

If one node is prepared for receiving data from other node, that is, it has enough memory space on its receiver electronics for admitting data characters, it must transmit a packet containing an FCT. Receiving an FCT authorizes the transmitting node to send 8 characters and sending an FCT sets the receiving node to expect 8 characters. The transmitting node keeps a credit count of how many characters it is allowed to send and the receiving node likewise keeps a credit count of how many characters it has allowed to receive. Each time the transmitting node sends a data character, it decrements the transmit credit count by one. Each time the receiving node receives a data character, it decrements the receive credit count by one. The standard specifies that the maximum number of outstanding data characters on either the transmitting or receiving side is 56.

Routing in SpaceWire deserves special attention due to its clever implementation. To support multi-star topologies by cascading routing switches, enough routing information is inserted in the beginning of the data packet as a sequence of 8-bit fields informing the switch output port whereto the data packet should be forwarded. As the data packet crosses a routing switch, the first leading character is removed and only remaining characters are forwarded to the output port.

Another important feature in SpaceWire is what the standard calls “wormhole routing”, described in the standard’s text as follows (ECSS, 2008):

“*As soon as the header for a packet is received the switch determines the output port to route the packet to by checking the destination address. If the requested output port is free then the packet is routed immediately to that output port. That output port is now marked as busy until the last character of the packet has passed through the switch*”

This mechanism is not new and a similar approach called “cut-through” was used in the first commercially available Ethernet switches (CISCO, 2004).

Broadcast and multicast are also supported by the standard, but these forms of packet distribution are treated as particular cases in the routing switch programming, unlike IEEE 802.3 Ethernet which use special network addresses for the same purpose.

SpaceWire physical medium operates with Low Voltage Differential Signaling (LVDS) using a low voltage swing (from -400mV to +400mV over 1.2V level). Data encoding is Data-Strobe (DS) with one line for Data and one line for Strobe. The data is transmitted Non-Return-to-Zero (high voltage level is interpreted as 1 and low voltage level is interpreted as 0) and the strobe signal changes state whenever the data remains constant from one data bit time to the next.

SpaceWire cables comprise four twisted pair wires with a separate shield around each twisted pair and an overall shield. The standard provides detailed

information not only about the cable construction, but also about connector types and other wiring requirements.

Supported data transmission speeds range from 2 megabits per second to 400 megabits per second, what places SpaceWire on the top of the list of DCPs for this particular attribute.

2.3.4 MIL-STD-1553B

MIL-STD-1553 (MIL STANDARD, 2019) was first published as a U.S. Air Force standard in 1973, and first was used on the F-16 Falcon fighter aircraft. It was originally designed as an avionic data bus for use with military avionics, but has also become commonly used in spacecraft on-board data handling (OBDH) subsystems.

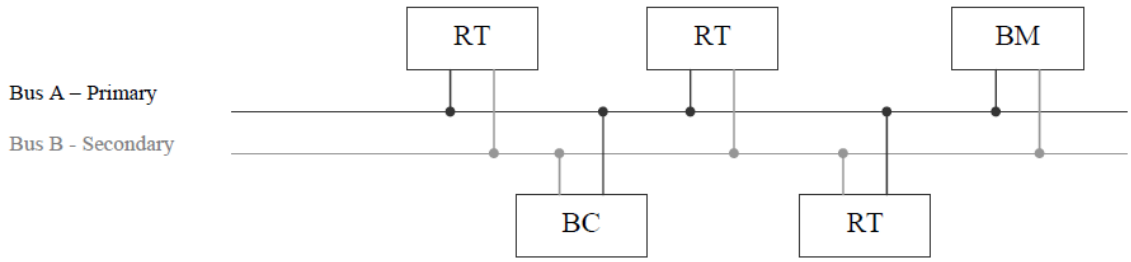
With the introduction of this standard, the point-to-point topology commonly used in previous on-board electronic systems was replaced by a bus topology, with immediate benefits in simplification of cabling design.

The hardware components in a MIL-STD-1553B bus assume three different roles:

- Bus Controller (BC): a unit responsible for controlling all message transmissions on the physical bus;
- Remote Terminal (RT): a unit connected to the physical bus responsible for transmitting or receiving messages as dictated by the Bus Controller;
- Bus Monitor (BM): a passive unit that can be used to monitor the bus data flow for monitoring purposes.

The physical layer is a 1 megabit per second, Manchester II encoded, dual-redundant serial bus. All communications go over the primary bus unless it becomes unavailable. In this event, a secondary bus is used. Figure 2.10 illustrates the bus topology and its components.

Figure 2.10 – Typical MIL-STD-1553B bus topology.



Source: Adapted from AIM GmbH (2010).

There can be up to 31 RT units connected to a single physical bus. In general, the BC has another unit that can be used as a backup in case of the BC becoming unavailable.

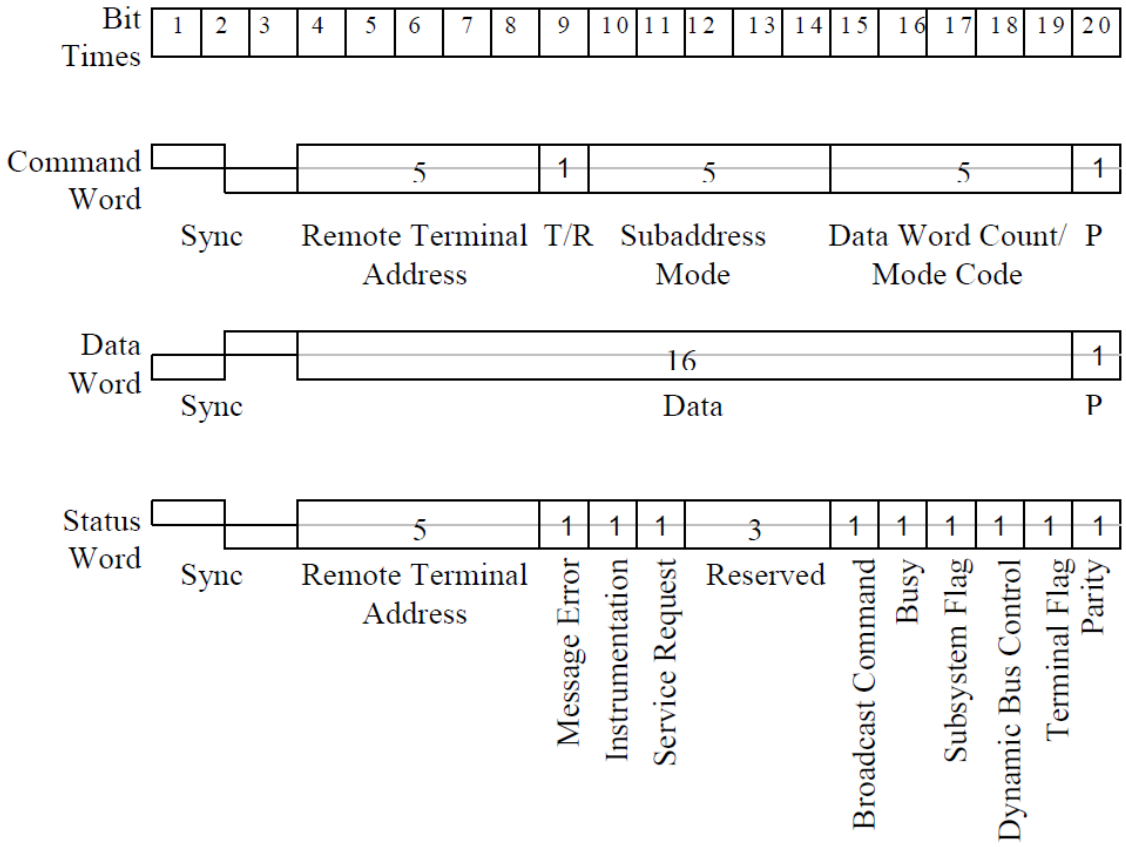
No RT can transmit on the bus unless instructed to by the BC. The system integrator responsible for the design of the on-board system has to program the BC for issuing commands to RT for receiving and transmitting messages with pre-defined number of data words.

The BC can issue commands to the RTs for data transmissions: a) from RT to BC; b) from BC to RT; and c) from RT to RT.

The commands sent by the BC to the RTs take 20 bit-times, 3 bits of a “Sync” field, 16 data bits and 1 odd-parity bit

There are three different types of words: command; data; and status, as illustrated in Figure 2.11.

Figure 2.11 – MIL-STD-1553B word formats



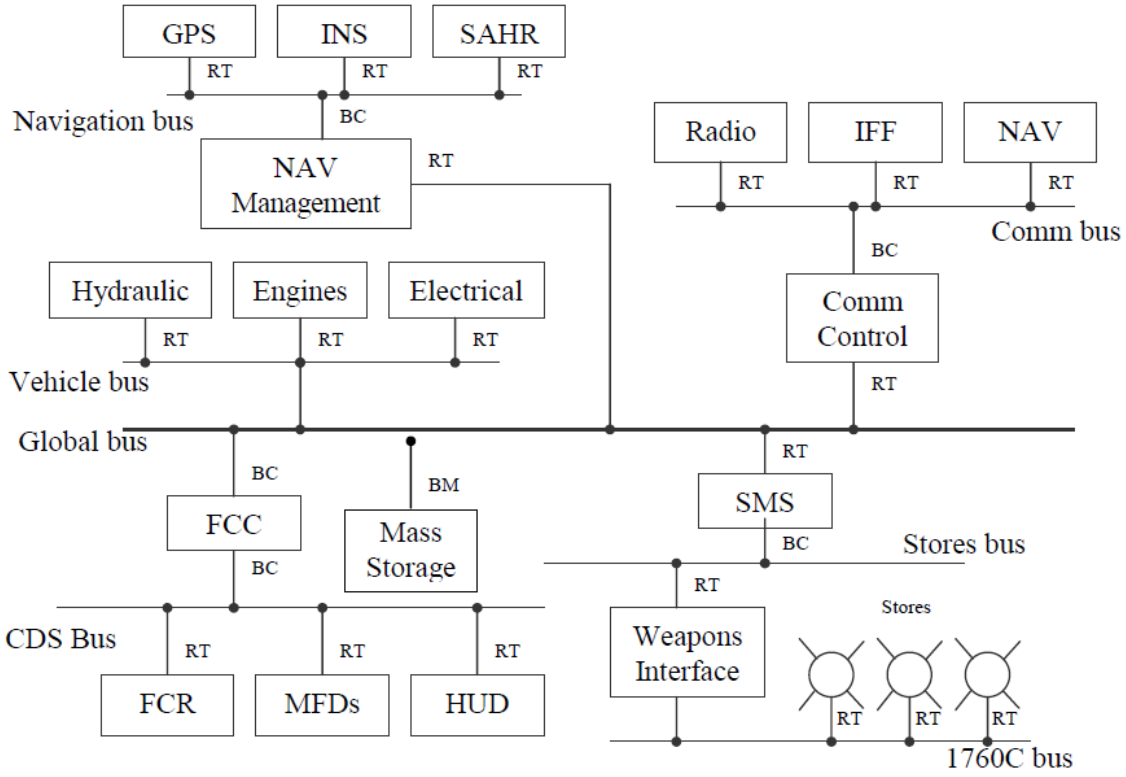
Source: Adapted from AIM GmbH (2010).

The sequence of words for an RT-to-RT communication usually goes as follows:

- The BC sends a Command Word instruction to the receiving RT with the T/R bit set to R and the Data Word Count set to the number of Data Words to be received;
- The BC sends a Command Word instruction to the transmitting RT with the T/R bit set to T and the Data Word Count set to the number of Data Words to be transmitted;
- The transmitting RT sends a Status Word for notifying the BC of its functional state, then it transmits the exact programmed number of Data Words.

The design of complex avionics systems often requires multiple hierarchically arranged physical buses connected via relay units, as illustrated in Figure 2.12.

Figure 2.12 – Typical MIL-STD-1553 bus topology in a military aircraft.

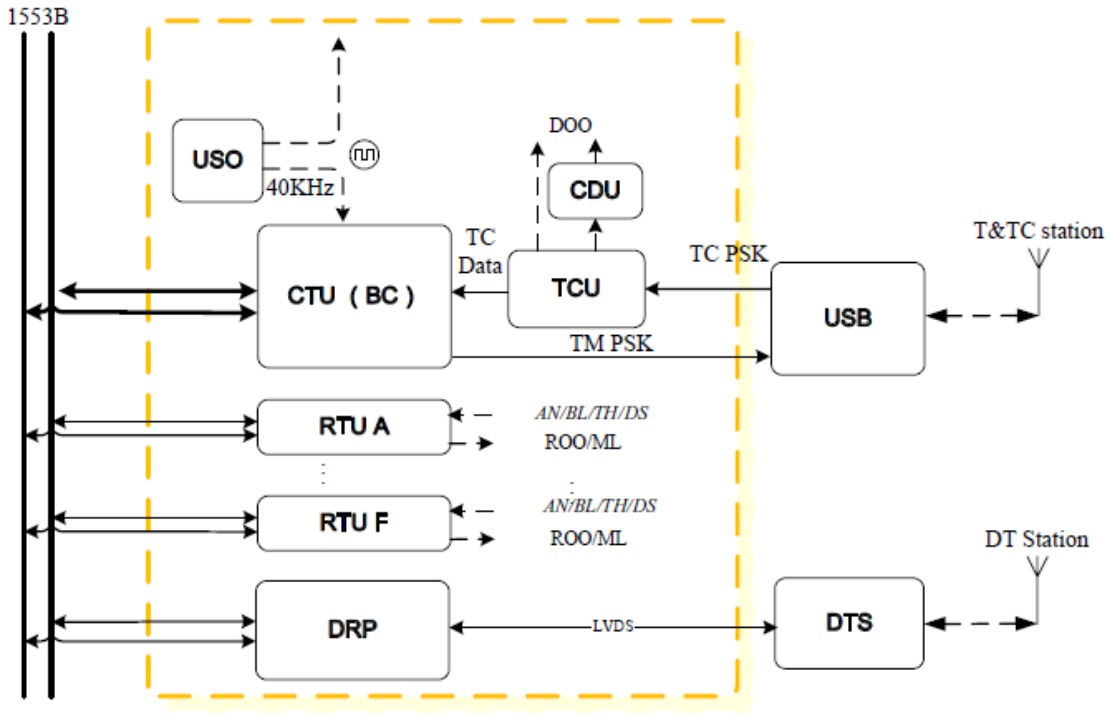


Source: Adapted from AIM GmbH (2010).

Note in Figure 2.11 that the “Stores Management System” (SMS) is RT to the Global bus and BC to the Stores bus and the “Flight Control Computer” (FCC) is BC for both Global bus and the “Cockpit Display System” (CDS) bus.

An example of the MIL-STD-1553B topology in a space vehicle is illustrated in Figure 2.13.

Figure 2.13 – MIL-STD-1553B bus in the CBERS Satellite.



Source: Adapted from Wang et al. (2017).

Figure 2.13 shows the topology of the On-Board Data Handling (OBDH) subsystem of the China-Brazil Earth Resources Satellite (CBERS), including a Central Terminal Unit (CTU), Remote Terminal Units (RTU A to F), a Tele-Command Unit (TCU), a Command Decode Unit (CDU), a Ultra-Stable Oscillator (USO) and a Data Recording and Processing unit (DRP). Note that the CTU plays the role of BC for the OBDH subsystem bus.

The MIL-STD-1553B bus is recognized by its reliability in communication networks in submarines, tanks, target drones, missiles, launch vehicles and larger space systems, including the International Space Station and Space Shuttle programs, and more recently in the Spacebus family of geostationary satellites (BOURGUIGNON, 2013).

2.3.5 Serial communication

2.3.5.1 RS-232

In telecommunications, the “Recommended Standard” 232 or RS-232 (EIA STANDARD, 1969), refers to a standard originally introduced in 1960 for serial communication transmission of data. It formally defines signals connecting a DTE (Data Terminal Equipment)m such as a computer terminal, and a DCE (Data Communication Equipment), such as a modem.

2.3.5.2 RS-422

RS-422 is a technical standard originated by the Electronic Industries Alliance (TIA/EIA STANDARD, 1994) that specifies electrical characteristics of a differential signaling that can transmit data at rates as high as 10 Mbit/s, or may be sent on cables as long as 1,500 meters. Some systems directly interconnect using RS-422 signals, or RS-422 converters may be used to extend the range of RS-232 connections.

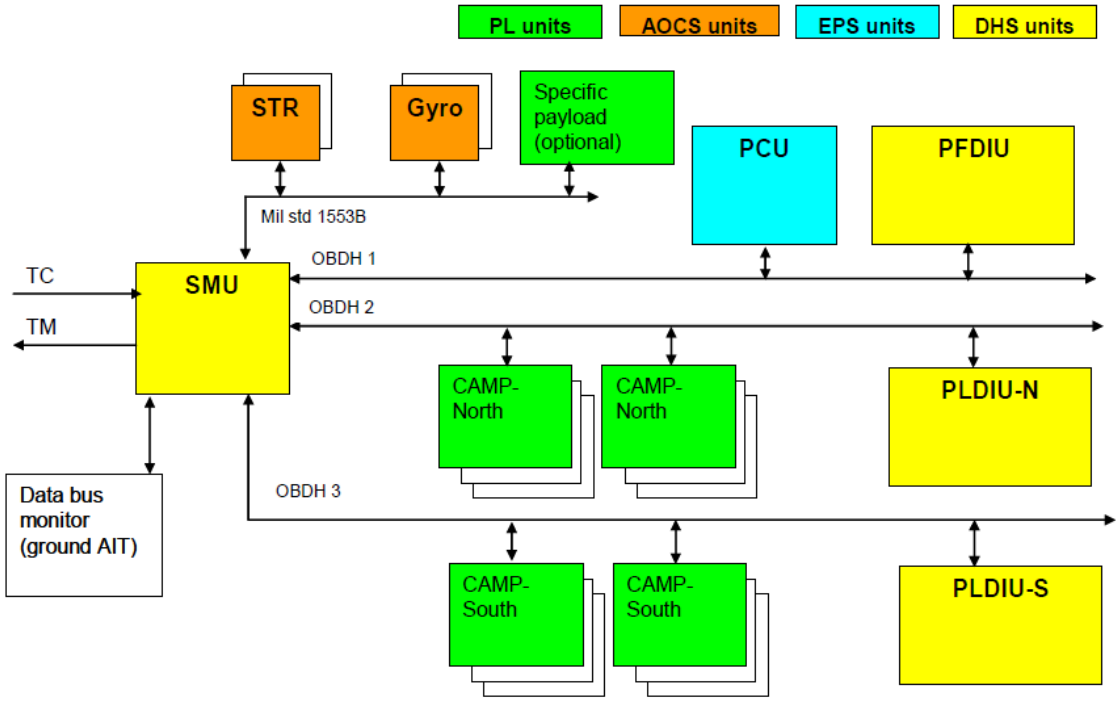
2.3.5.3 RS-485

RS-485 (EIA STANDARD, 1983) supports inexpensive local networks and multi-drop communications links, using the same differential signaling over twisted pair as RS-422. These characteristics made RS-485 attractive for industrial control systems and for aerospace applications.

RS-485 found its way as Physical Layer into space applications such as the OBDH-485 bus introduced by Thales-Alenia Space in the Spacebus family of geostationary satellites, in particular for the Spacebus 4000 (PETIT, 2012).

Figure 2.14 shows three OBDH buses connecting the Satellite Management Unit (SMU) of the Spacebus 4000 to other electronic units.

Figure 2.14 – OBDH-485 buses for the SMU of the Spacebus 4000 satellite.



Source: Adapted from Caramia (2016).

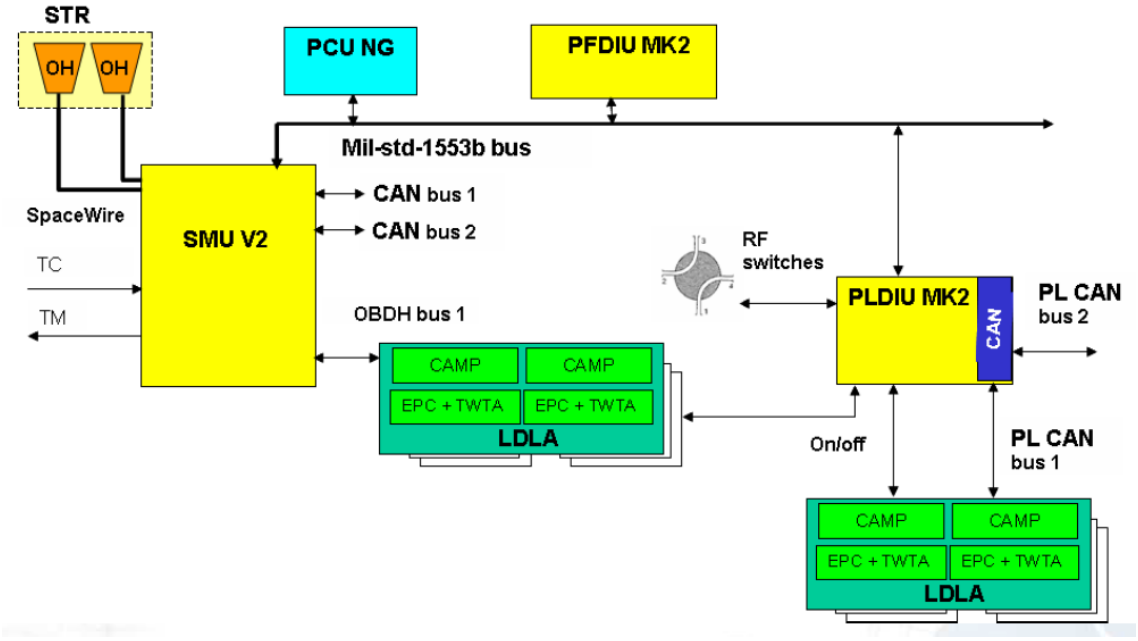
2.3.6 Shared medium

2.3.6.1 CAN

A Controller Area Network (CAN bus) is a robust vehicle bus standard designed to allow microcontrollers and devices to communicate with each other in applications without a host computer (BOSCH, 1991). It is a message-based protocol, designed originally for multiplex electrical wiring within automobiles to save on copper, but is also be used in many other contexts. Development of the CAN bus started in 1983 at Robert Bosch GmbH. The protocol was officially released in 1986 at the Society of Automotive Engineers (SAE) conference in Detroit, Michigan. Bosch published several versions of the CAN specification and the latest is CAN 2.0 published in 1991.

CAN also found its way into space applications replacing proprietary, RS-485 based multi-drop buses (CARAMIA, 2016), as illustrated in Figure 2.15 (compare with Figure 2.14).

Figure 2.15 – CAN-bus upgrade proposal for the Spacebus 4000 satellite.



Source: Adapted from Caramia (2016).

The CAN bus specification covers the MAC and the LLC sub-layers of the Data Link Layer and is not specific about the Physical Layer. Curiously, the suggested Physical Layer for the CAN bus for space applications is RS-485 (PETIT, 2012).

2.3.6.2 ARINC-629

The ARINC-629 computer bus was introduced in May 1995 and was first used on the Boeing 777 (SAE-ITC STANDARD, 2019). The ARINC-629 bus operates as a multiple-source, multiple-sink system, where each terminal can transmit data to, and receive data from, every other terminal on the data bus. While some people expected that the Boeing 777 would be the first and last aircraft to use ARINC-629 data bus, it is also used on the Boeing 737 MAX and Airbus A330 and A340.

2.3.7 Time-triggered

2.3.7.1 TTP

The Time-Triggered Protocol (TTP) is an open computer network protocol for control systems (TTTECH, 2003). It was designed as a time-triggered field bus for vehicles and industrial applications and standardized in 2011 as SAE

AS6003 (TTP Communication Protocol). TTP was originally designed at the Vienna University of Technology in the early 1980s. In 1998 TTTech Computertechnik AG took over the development of TTP, providing software and hardware products.

The TTP physical medium is not specified in the SAE standard. The commercially available hardware either relies on RS-485 at 4 megabit per second or on Ethernet at 25 megabit per second.

TTP was selected by NASA's Marshall Space Flight Center for implementing the Integrated System Health Management (ISHM) within the scope of the Propulsion High-Impact Avionics Technology (PHIAT) project (GWALTNEY et al., 2006).

2.3.7.2 TTEthernet

The Time-Triggered Ethernet (SAE AS6802) standard defines a fault-tolerant synchronization strategy for building and maintaining synchronized time in Ethernet networks, and outlines mechanisms required for synchronous time-triggered packet switching for critical integrated applications, such as integrated modular avionics architectures (TTTECH, 2008).

As the name indicates, TTEthernet uses Ethernet technology. More specifically, it implements a multi-star topology; therefore the use of a switch is mandatory to this networking architecture.

TTEthernet technology implemented by TTTech was used in the Orion Multipurpose Crew Vehicle (MPCV), a NASA spacecraft designed to take a crew of up to six astronauts to destinations beyond Low Earth Orbit including the Moon and Mars (NASA, 2020). The brains of the Orion spacecraft (NASA, 2019) is the Vehicle Management Computer (VMC), a single electronics unit consisting of four independent modules that deliver the processing capability for the Orion spacecraft and communicate with the other avionics via redundant Ethernet connections using the TTEthernet Network Interface Controllers (NIC) and network switches (GOFORTH et al., 2014).

TTEthernet was also proposed to future integrated modular spacecraft architectures as part of the Avionics and Software (A&S) project chartered by NASA's Advanced Exploration Systems (AES) program (LOVELESS, 2015).

2.3.7.3 FlexRay

FlexRay is an automotive network communications protocol developed by the FlexRay Consortium to govern on-board automotive computing (FLEXRAY, 2005). It was designed to be faster and more reliable than CAN and TTP. The FlexRay consortium disbanded in 2009, but the FlexRay standard is now a set of ISO standards, ISO 17458-1 to 17458-5.

There is no public reference about the use of FlexRay in the aerospace industry. However, FlexRay was considered a candidate by NASA in the same study for selecting a digital communication protocol for the Integrated System Health Management (ISHM), for which TTP was considered the best choice (GWALTNEY et al., 2006).

2.4 Comparing digital communication protocols

Table 2.2 lists the following high-level features for the industry standards that are most relevant to this work, namely: 1) the IEEE 802.3 Ethernet; 2) the IEEE 802.2 Logical Link Control (LLC); 3) the Internet Protocol (IP); 4) the ARINC-429 serial communication protocol; 5) the ARINC-664 Part 7 specification for the full-duplex switched Ethernet used with Integrated Modular Avionics (IMA); and 6) SpaceWire.

These high-level features are:

- Origin: which industry field introduced the standard;
- Topology: which network topologies the standard supports;
- ISO/OSI Layers: which layers of the 7-layer ISO/OSI reference model is covered by the standard;
- Transmission speed: for standards including the Physical Layer, which transmit speeds are available;

- Message sizes: which message sizes in bytes the standard is capable of transmitting.

Table 2.3 lists the same features for the digital communication protocols listed in the previous sections that are found in space applications, namely: 1) the RS-485 multi-drop serial communication protocol; 2) the Controller Area Network (CAN); 3) the MIL-STD-1553B; and 4) the TTEthernet, only recently listed among those suited for space applications. For these, Table 2.3 also lists their most prominent use in space applications.

It is important to mention that the transition of one digital communication protocol to the space industry requires further development particularly in microprocessor hardware. As an example, the Space Ethernet PHYsical Layer project (SEPHY, 2015) developed a radiation tolerant Ethernet transceiver as part of the initiative for introducing the TTEthernet technology in space applications.

It is also important to mention that the mechanisms used by a digital communication protocol for transporting data are well defined in the applicable standard, but the programming interface for accessing it from the Application Layer rely on proprietary solutions that highly depend on the operating system used for hosting the application.

For instance, the ARINC-664 Part 7 specification explicitly indicates that applications should be hosted by an ARINC-653 specification compliant operating system, but does not indicate what programming interface shall be used. The consequence of this absence is that different ARINC-653 implementations offer a different, non-standard application programming interface.

Filling this particular gap is one important objective of this work, as detailed in the next chapters.

Table 2.2 – Industry standard digital communication protocols.

Feature	**Ethernet IEEE-802.3**	**LLC IEEE-802.2**	**Internet Protocol**	**ARINC-429**	**ARINC-664 Part 7**	**SpaceWire**
Origin	Computer industry	Computer industry	Telecommunication industry	Aircraft industry	Aircraft industry	Space industry
Topology	star	n/a	n/a	multi-drop bus	multi-star	point-to-point multi-star
ISO/OSI layers	Physical Data Link	Data Link	Network	Physical Data Link	Physical Data Link Network Transport	Physical Data Link Network
Transmission speed	10 or 100 Mbit/s 1 or 10 Gbit/s	not applicable	not applicable	100 kbit/s	100 Mbit/s	2 Mbit/s up to 400 Mbit/s
Message sizes	46 to 1500 bytes	43 to 1487 bytes	MTU*	32 bits	18 bytes up to 8192 bytes	n x 8-bit

*the Maximum Transfer Unit (MTU) is defined for each individual Physical Layer.

Table 2.3 – Digital communication protocols for space applications.

Feature	RS-485	CAN	MIL-STD-1553B	TTEthernet
Origin	Telecommunication industry	Automotive industry	Military aircraft industry	Aircraft industry
Topology	multi-drop bus	shared bus	shared bus	multi-star
ISO/OSI layers	Physical	Data Link	Physical Data Link	Physical Data Link
Transmission speed	up to 4Mbit/s	1 Mbit/s	1 Mbit/s	100 Mbit/s
Message sizes	application defined	0 to 8 bytes	up to 32 data words of 16-bits	46 to 1500 bytes
Typical space application	OBDH connection to sensors and payload units	OBDH connection to sensors and payload units	spacecraft communication backbone	spacecraft communication backbone

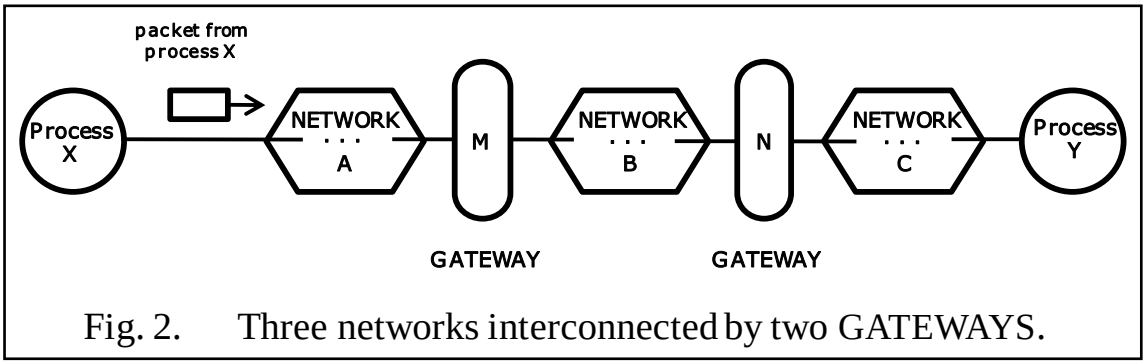
3 PROBLEM STATEMENT AND APPROACH TO A SOLUTION

3.1 High-level approach used for developing the Internet Protocol

Vinton G. Cerf and Robert E. Kahn in their article to the IEEE Transactions on Communications (CERF, 1974) called “A Protocol for Packet Network Intercommunication” described functionalities of what was called “Transmission Control Program” (TCP) allowing hosts connected to different networks to communicate through a “gateway”.

In the article, this TCP *“…handles the transmission and acceptance of messages on behalf of the processes it serves*”. The authors made clear their vision on “Process Level Communication” in their Figure 2 reproduced below in Figure 3.1:

Figure 3.1 – “Process Level Communication” according to Cerf and Kahn.



Source: Adapted from Cerf and Kahn (1974).

The authors also describe the data packet format and several protocol features, such as message segmentation, packet sequencing and retransmission to overcome potential incomplete information transfer and flow control (by using the “window” concept). There is no indication of a preferred physical layer.

Later, in the conclusion section, the text reads: “*The next important step is to produce a detailed specification of the protocol so that some initial experiments with it can be performed*”.

In December of the same year, Vinton Cerf and two other colleagues (CERF et al., 1974) submitted a “Request For Comments” (RFC) number 765 to the International Packet Network Working Group (INWG) of ARPANET’s (DARPA, 1981) Network Information Center (NIC), This RFC “*describes the functions to*

be performed by the internetwork Transmission Control Program [TCP] and its interface to programs or users that require its services".

In this RCF, the term "connection" is used to associate two "sockets" (a 3-tuple used to uniquely identify a transmitting or receiving end), for which user calls "*specify the basic functions the TCP will perform to support interprocess communication*". These user calls were: OPEN, SEND, RECEIVE, CLOSE, INTERRUPT and STATUS.

Vinton Cerf and his colleagues at the ARPANET project created a communication infrastructure that consisted not only in a lower (not lowest) level protocol, but also in a collection of specific functions that would allow a process executing in one host to communicate with a process executing in another host, even if the two hosts were connected to dissimilar networks (hence the term "internetworking").

For a process incorporating its functionality through software, it is convenient that the specifics of the physical connection to a communication link are hidden. This facilitates keeping the focus in developing the process functionality itself and assures portability, should the underlying software (most likely part of an operating system) and hardware (most likely a network port) have to be changed. This can be described as a protocol serving a data transfer service.

3.2 Problem statement

The specification ARINC-664 Part 7 (ARINC, 2009) uses the Ethernet Media Access Control and Physical Layer, requires a star (or multi-star) topology and, consequently, does not support a point-to-point topology.

SpaceWire supports both star (or multi-star) and point-to-point topologies but it has its own (other than Ethernet) Physical Layer.

Supporting simpler and more complex network topologies allows scaling up digital data communications in electronic systems onboard of small to medium size aerospace vehicles, without necessarily changing the communication protocol, and follows recent and current trends: "faster-better-cheaper", "Lean Production System" and the "New Space" approaches.

ARINC-664 Part 7 uses the Internet Protocol (IP) as the Network Layer and the User Datagram Protocol (UDP) as the Transport Layer, therefore network data frames have to be validated over two protocol layers before data is finally moved to the Application Layer.

Eliminating layers between the Application Layer and the Data Link Layer shortens the process of getting usefull data delivered to the application, reducing and potentially speeding up the processing steps required to this operation.

ARINC-664 Part 7 also does not include an Application Programming Interface (API) for applications wishing to use this protocol. The ARINC-664 Part 7 relies on commercial implementations of another specification ARINC-653 (ARINC, 2015) for connecting an application to an UDP port using proprietary middleware.

SpaceWire (ECSS, 2008) also does not provide an API, only suggests what services should be available.

Providing a standard API ensures software portability by decoupling the access to a “channel” from operating system specifics, thus eliminating proprietary solutions for sending and receiving application data.

These particular aspects of two very important industry standard protocols present an opportunity for improvement as developed in this work.

3.3 Approach to solving the current problem

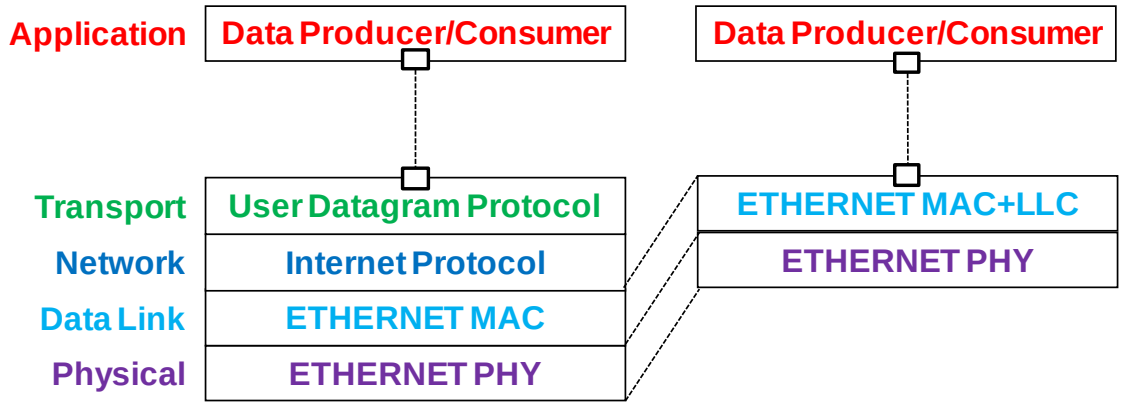
As Vinton Cerf’s and Robert Kahn’s contribution to the “Transmission Control Protocol” over the “Internet Protocol” (the “TCP/IP”), which was only formalized in 1980 (MCKENZIE, 2011), the contribution of this work is two-fold:

1) It provides a Data Link Layer Protocol that reduces the number of software layers that data received from an Ethernet Media Access Control (MAC) layer have to cross for reaching a top layer application by dispensing the Network and the Transport layers (the protocol);

2) It provides an Application Programming Interface (API) that uses a new concept called “channel” for simplifying the way an application uses the Data Link Layer Protocol for communicating with its remote peers (the service to the protocol).

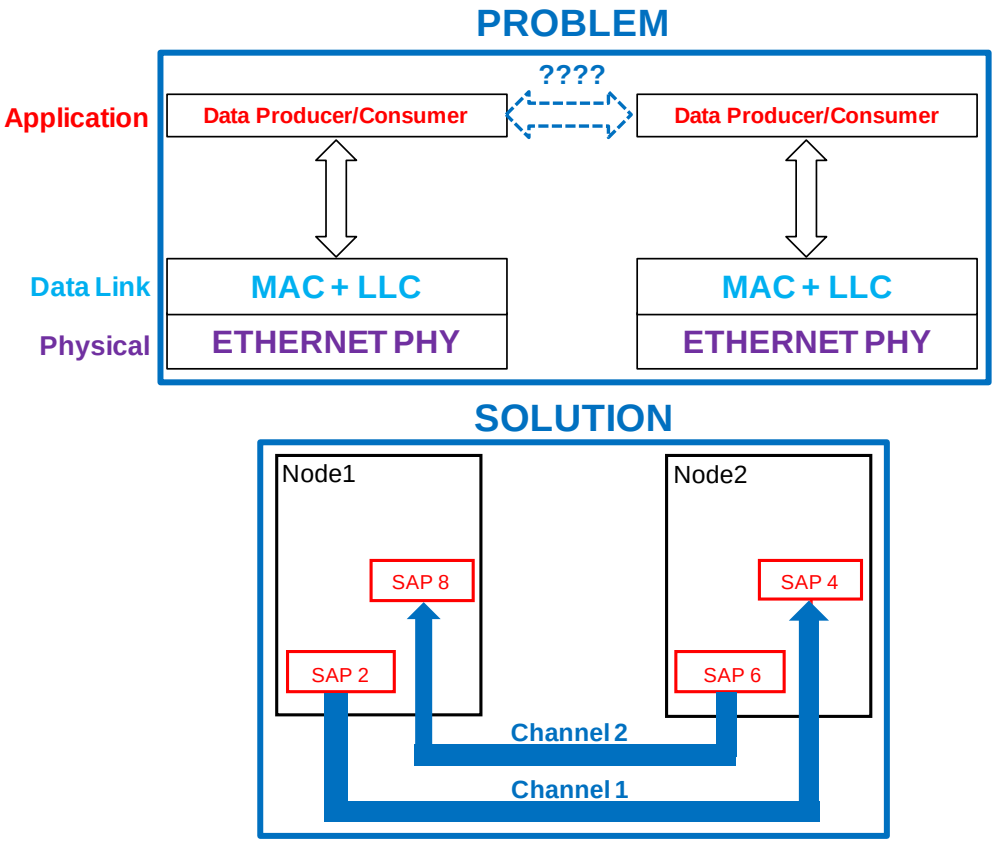
In currently existing solutions, data producer and data consumer processes execute at the topmost Application Layer relying on protocols at the Transport Layer, such as the User Datagram Protocol (UDP) for the ARINC-664 Part 7 specification. Figure 3.2 illustrates how a direct path to the Data Link Layer simplifies the access to data from the Application Layer by eliminating the need of interfacing with the Transport and Network Layers.

Figure 3.2 – A direct path from Application Layer to Data Link Layer.



The “channel” API introduced in this work, allowing a data producer process to communicate with a data consumer process in a reliable, controlled and flexible way, is the very reason for the development of the new Data Link Layer protocol. With the “channel” concept, applications abstract the access to the lower level supporting protocol, as illustrated in Figure 3.3.

Figure 3.3 –The “channel” concept connecting Application Layers.



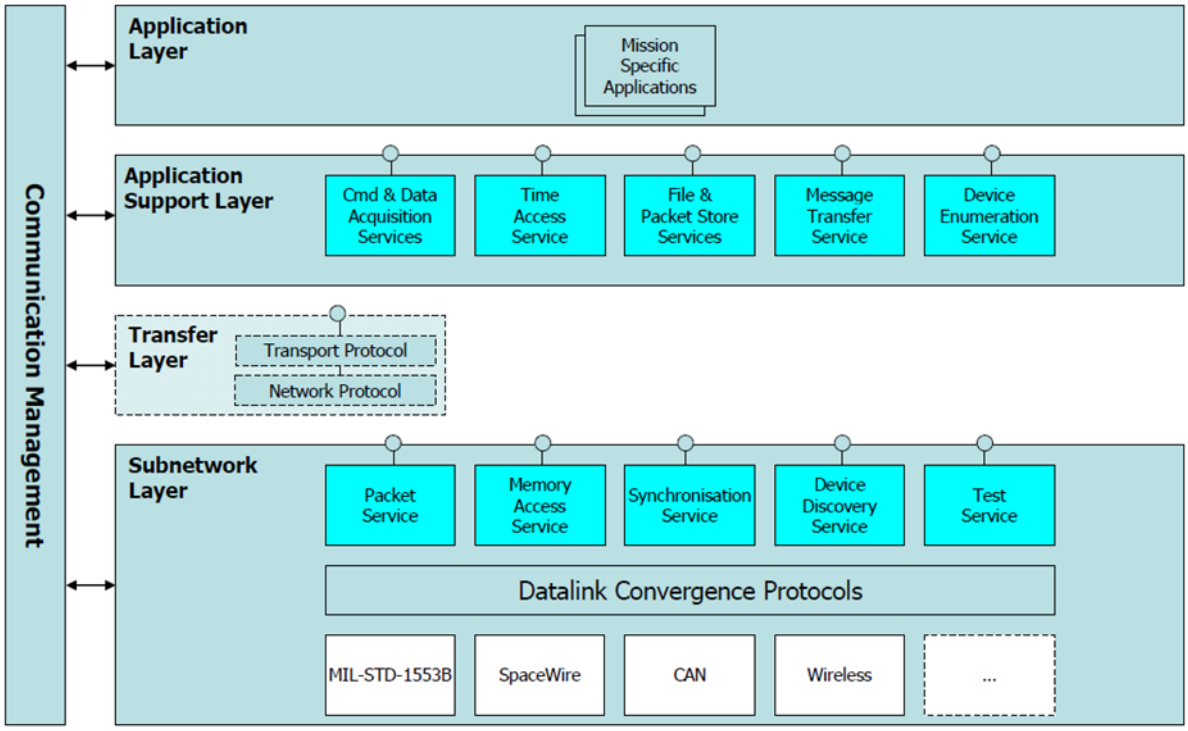
Without the “channel” concept and the associated supporting API, this new protocol is simply another player in the digital communication protocol arena.

Although the new Data Link Layer protocol relies on Ethernet as the Physical Layer, it does not impose any specific network topology. As in ARINC-664 Part 7, two of the services that the new “channel” concept provides to applications are Traffic Shaping and Traffic Policing for controlling how much data per unit of time an application is allowed to send or to receive over a “channel”. For them, a low computational cost method for estimating the worst-case transmission delay for a data frame crossing a traffic switching device, usually required in – but not limited to – network star topologies, is also provided.

The virtualization of the data communication infrastructure arises naturally when there is a high diversity in the nature of the devices operated by a system onboard of an aerospace vehicle which need to be accessed by software applications.

Figure 3.4 shows the architecture proposed by the Consultative Committee for Space Data Systems (CCSDS) called Spacecraft Onboard Interface Services (SOIS). Between the highest level, the Application Layer, and the lowest level, the Subnetwork Layer, different services are interposed for hiding Transport Layer, Network Layer protocols and Physical Layer communication protocols, such as MIL-STD-1553B, CAN and SpaceWire, from the application software (SOIS, 2013).

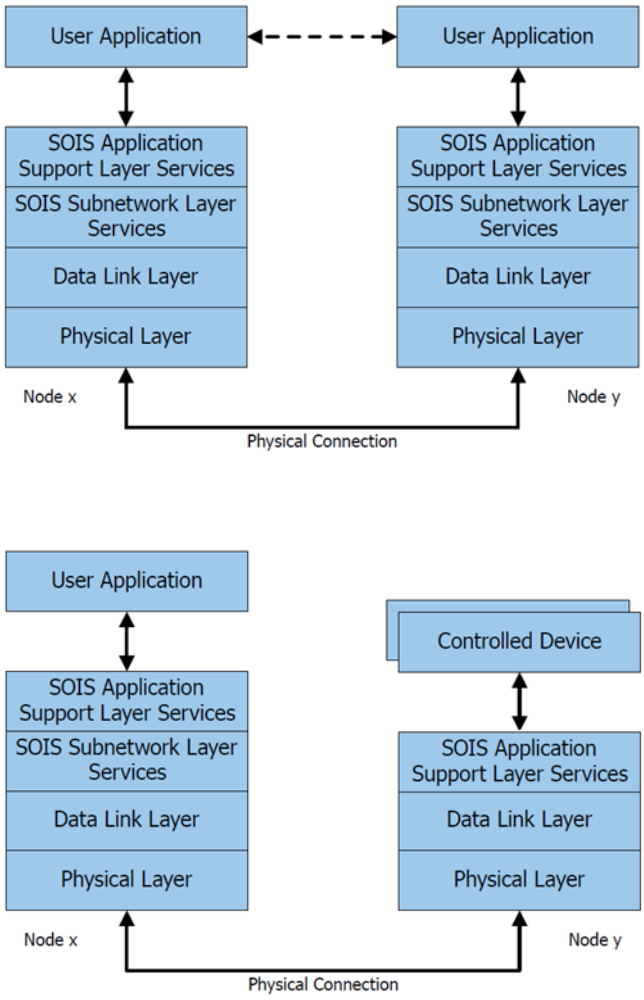
Figure 3.4 –The SOIS communication architecture.



Source: Adapted from CCSDS (2013).

Figure 3.5 shows two SOIS deployment schemes for communication between applications (top) and for communication between an application and a device (bottom).

Figure 3.5 –SOIS deployment schemes.



Source: Adapted from CCSDS (2013).

However, SOIS does not provide an application programming interface similar to the one provided in this work for supporting the “channel” concept.

The key objectives and the detailed specification of the new Data Link Layer protocol, associated services and the “channel” concept are detailed in the next chapter.

4 KEY OBJECTIVES FOR THE NEW PROTOCOL AND SERVICES

The next few sections present the key objectives orienting the specification of a new Data Link Layer, IEEE-802.2 extended protocol and associated services.

4.1 Connect data producers to data consumers

Among the Physical (Layer 1) and Data Link Layers (Layer 2) implementations developed in the last few decades, Ethernet and its IEEE standardization 802.3 is by far the most frequently used. In any household, wireless access points route network traffic to commercial Internet service providers over Ethernet. In the factory floor, several implementations allow automated manufacturing of consumer electronics and cars. In commercial and military aircraft, Ethernet is present since the ARINC-664 specification Parts 1 and 2 were published in 2002.

However, Ethernet implementations used in aerospace vehicles also imply in using other Network and Transport Layers, being the most frequent the Internet Protocol (IP) as the Network Layer and User Datagram Protocol (UDP) or Transmission Control Protocol (TCP) as the Transport Layer.

The reason is that Ethernet, being a Data Link Layer protocol, does not provide a means of linking two application instances running in different network nodes. For that, a virtual construct needs to be defined and supported by associated services. In UDP and TCP over IP, this virtual construct is named “port”.

Therefore, for connecting a Data Producer to a Data Consumer in an embedded network, such those present in modern aerospace onboard electronics, apparently requires a Transport Layer protocol and associated services to transmit and receive data.

This means processing three network layers before being able to access data needed by an application for its continuing operation, which in time-critical situations, such as in controlling flight, may represent simply consuming extra processing time with no actual work being done. Note that this is the case of ARINC-664 Part 7, which relies on the combination of Ethernet, IP and UDP.

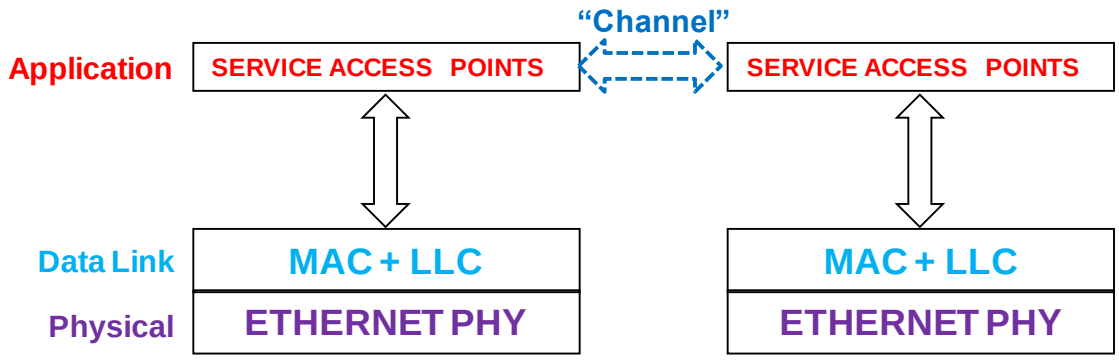
To better serve time-critical applications, shortening the processing time required for extracting relevant data from a network transmission is very important characteristic of a network protocol.

In fact, the network standard IEEE 802.2 Logical Link Control (LLC) provides precisely this feature by specifying Service Access Points (SAP) at the Data Link Layer.

The specification of a new Data Link Layer protocol and the “channel” concept proposed in this work take advantage of this feature introduced by the IEEE 802.2 LLC protocol.

The approach developed in this work for simplifying the communication between data producers and consumers using SAPs is illustrated in Figure 4.1.

Figure 4.1 –The “channel” concept connecting Service Access Points.



4.2 Support mixed topologies

Ethernet can be used in different network topologies, from its initial design as a shared bus to its current and most frequent star shaped, mix of these two and point-to-point, even when the latter seems limited to network maintenance scenarios.

The protocol described by the ARINC-664 Part 7 specification was introduced in the industry during the development program of a very large aircraft (the Airbus A380) and in this scenario, a multi-star Ethernet topology was by far the most appropriated, because of the drastic reduction of cabling expected when compared to any other possible topology arrangement, even considering that

the actual network had to be duplicated for the sake of reliability (ARINC-664 Part 7 requires dual-redundant Ethernet physical medium).

For smaller vehicles, such as satellites, a multi-star topology may represent in fact extra weight, space and power, because it requires the introduction of one or more network switches.

A point-to-point Ethernet topology designed similarly to a wire-mesh can be simpler to implement in such scenario, even considering that any network node should be reachable over at least two independent paths.

Ideally, a network protocol should allow mixing topologies depending on the target vehicle and the type of applications requiring data exchange. SpaceWire is a good example of this approach.

The specification of a new Data Link Layer protocol proposed in this work provides support for both star and point-to-point topologies and any combination of the two.

4.3 Provide timing information

Time-sensitive systems often require a deterministic behavior of the software applications involved.

Most of the time, software applications executing in a real-time operating systems are strictly periodic, If such applications are data producers or data consumers, it might be beneficial to applications which are data consumers to evaluate the periodic behavior of associated data producers.

Therefore, time-stamping data packets transmitted over a network may provide this means to any data consumer.

SpaceWire provides a means of propagating time, but neither it does it for time-stamping messages, nor it relies on it to ensure global time coherency across the whole network.

The specification of a new Data Link Layer protocol proposed in this work introduces a time-stamp field, which is to be filled at the time of the network data frame transmission.

4.4 Provide payload and header data integrity

Ethernet frames include a 32-bit Frame Check Sequence (FCS) field, a CRC-32 calculated over the entire Ethernet frame and transmitted after the last byte of the payload, while IP and UDP protocols provide 16-bit checksums for their respective headers. ARINC-664 Part 7 does not enforce any other form for protecting its payload from occasional bit-flips. SpaceWire transmissions are character-based and rely on a single parity bit for checking data integrity.

Phillip Koopman in 2002 presented a study on the performance of different CRC-32 polynomials introducing a new one (KOOPMAN, 2002), and in 2004 presented a performance evaluation on a series of CRC polynomials ranging from 3-bit to 16-bit (KOOPMAN, 2004).

The new Data Link Layer protocol proposed in this work introduces a new CRC field to protect the header information and an extra CRC field to protect the payload data, both taken from Koopman's studies.

4.5 Provide routing validation

In closed operational environments, such as those present in aerospace vehicles, it is vital for a reliable network operation that the communication paths defined by the person in charge of the design of the network topology are validated during the start-up phase of the embedded system which the network serves.

The new Data Link Layer protocol introduces a service for validating routing paths across the network independent of its topology, be it star or point-to-point or a combination of the two.

The routing validation is performed using a new field that extends the original IEEE 802.2 header in TEST PDUs.

4.6 Provide an operating system interface

The ARINC-664 Part 7 specification takes advantage of entities defined in the ARINC-653 "Avionics Application Software Interface" specification (ARINC, 2015). It inherits the concept of "ports" defined in ARINC-653 Part 1 for providing a path from UDP Source and Destination Port header fields to a logical construct accessible by applications hosted by an ARINC-653 compliant

operating system. It also uses a 5-bit field in the ARINC-664 Part 7 IP Source Address to communicate the numeric identification of the virtual-machine, named “partition” in the ARINC-653 specification, which hosts the application accessing the port.

Therefore, it is implicit that an ARINC-664 Part 7 requires an operating system which is ARINC-653 compliant to host its End-System. However, neither ARINC specification handles how data is passed from one partition executing in one equipment unit to a partition executing in another equipment unit. In essence, ARINC-653 ports are means of exchanging data between partitions like similar process-to-process (or task-to-task) data exchange features available in commercial operating systems. The consequence of this omission is that commercial implementations of the ARINC-653 specification have their own and proprietary way of performing data input and output over a hardware communication interface, damaging the first expected benefit of this specification: portability.

SpaceWire does not provide an application programming interface, only suggests that, should one exist, it shall support at least following services:

- ✓ Open link: Starts a link interface and attempts to establish a connection with the link interface at the other end of the link.
- ✓ Close link: Stops a link and breaks the connection.
- ✓ Write packet: Sends a packet out of the link interface.
- ✓ Read packet: Reads a packet from the link interface.
- ✓ Status and configuration: Reads the current status of the link interface and sets the link configuration.

These service definitions model the programming interface for the new Data Link Layer, IEEE-802.2 extended protocol, which introduces the “channel” concept, a virtual connection between a Source Service Access Point (SSAP) and a Destination Service Access Point (DSAP).

Following services will be defined:

- ✓ Register an operating system process, task or thread to a SAP

- ✓ Open a channel
- ✓ Send data to a channel
- ✓ Receive data from a channel
- ✓ Return status of a channel
- ✓ Close a channel
- ✓ Unregister an operating system process, task or thread from a SAP

For each configured channel, Traffic Shaping is provided for data sending operations, as Traffic Policing is provided for data receiving operations. These flow control features are put in place for protecting the integrity of the network in the event of an abnormal behavior of a hosted application.

4.7 Protocol specification breakdown

The development of the new Data Link Layer (Layer 2), IEEE 802.2 extended protocol specification involves:

- Specification of the new Protocol Data Units (UI and TEST PDUs) and all its fields:
  - ✓ IEEE 802.3 MAC Destination/Source Addresses
  - ✓ IEEE 802.3 Length Field
  - ✓ IEEE 802.2 DSAP and SSAP Fields
  - ✓ IEEE 802.2 Control Field
  - ✓ Header extension for Sequence Number on UI PDUs
  - ✓ Header extension for Hop Count on TEST PDUs
  - ✓ Header extension for Time-Stamping and Header-CRC on UI PDUs
  - ✓ New payload CRC
- Specification of the associated services for UI PDUs:
  - ✓ Data validation (via CRC on receiving end)
  - ✓ Introduction of the concept of “channel”
  - ✓ Traffic shaping (via token-bucket on transmitting end)

- ✓ Traffic policing (via token-bucket on receiving end)

- Route validation using TEST PDUs
- Specification of the operating system interface to the protocol layers:
  - ✓ Network configuration files
  - ✓ In-memory data structures
  - ✓ Application Programming Interface paradigm
- Specification of a method for estimating UI PDU forwarding latency.

This work also provides a sample implementation of the protocol layers, associated services and the “channel” concept using commercial off-the-shelf software tools.

4.8 Side-by-side comparison

The Table 4.1 shows a side-by-side comparison of features presented in ARINC-664 Part 7, SpaceWire and the proposed Data Link Layer, IEEE 802-2 extended protocol.

Table 4.1 – Feature comparison.

Feature	**ARINC-664 P7**	**SpaceWire**	**Proposal**
Physical Layer	Ethernet IEEE-802.3	LVDS	Ethernet IEEE-802.3
Speed	10Mbps –100Mbps	2Mbps to 400Mbps	not specified
Encoding	Manchester	Data-Strobe (DS)	not specified
Data Link Layer	Ethernet IEEE-802.3	n/a	Ethernet IEEE-802.3 IEEE-802.2
Network Layer	IP	n/a	n/a
Transport Layer	UDP	n/a	n/a
Topologies	Multi-star	Point-to-point Multi-Star	Point-to-point Multi-Star
Transmission unit	packet	character /packet	packet
Payload sizes	17-1471 bytes	10-bit for data 4-bit for control	34-1488 bytes
Time-stamping	no	no	20-bit field (count of microseconds)
Payload data validation	CRC-32	parity bit	Koopman-32
Header validation	IP/UDP header checksum	n/a	Koopman-12
Flow control	Traffic Shaping and Policing via token-bucket	specific protocol	Traffic Shaping and Policing via token-bucket
Link recovery	n/a	specific protocol	n/a
Routing validation	n/a	n/a	via TEST PDU
OS Interface	ARINC-653 ports	suggested	SSAP-DSAP channel

5 PROTOCOL SPECIFICATION

Once the LLC sub-layer of the new Data Link Layer protocol derives from IEEE 802.2, it is required that the underlying MAC sub-layer of the IEEE 802.3 is formatted accordingly, in particular by setting the interpretation of the Type/Length field to Length.

MAC addresses provided by Ethernet device manufacturers are very hard to track visually, so the next sections establish MAC address formatting rules that help the network integrator by embedding data into MAC addressing which reflects the network topology. This initiative can prove itself very useful when verifying network integrity during its operation.

5.1 Specification of the new UI and TEST Protocol Data Units (PDUs)

The construction of the complete network data frame includes formatting the IEEE 802.3 header and the new, extended IEEE 802.2 header.

5.1.1 IEEE 802.3 MAC source and unicast destination address formatting

The MAC Source and unicast MAC Destination addresses shall be formatted according to the following rules (Figure 5.1 and Figure 5.2):

- The three high-order octets, bit positions from 25 to 48, shall use value 0xAA0005 hexadecimal, as this Organizationally Unique Identifier (OUI) has not been assigned to any organization;

- The two mid-order octets, bit positions from 9 to 24, shall contain the *Equipment ID*, as defined by the ARINC-429 specification (Attachment 1-2 for Equipment Codes); bit positions 21 to 24 shall contain 0000 (only binary zeros);

- The first low-order octet of the MAC Source address, bit positions from 1 to 8, shall be composed by the physical *Port* number in the four low-order bits, bits positions from 1 to 4, and the *Unit* number in the four high-order bits, bit positions from 5 to 8.

- The first low-order octet of the unicast MAC Destination address, bit positions from 1 to 8, shall contain the *Unit* number in the four high-order

bits, bit positions from 5 to 8, followed by trailing zeros in bit positions from 1 to 4.

Figure 5.1 – MAC Source address formatting.

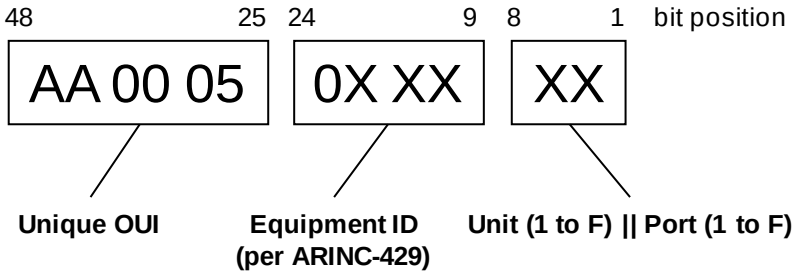
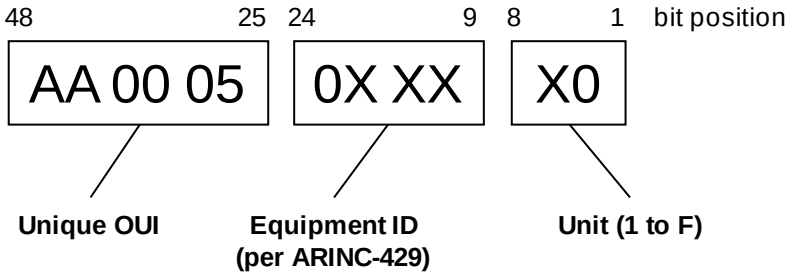


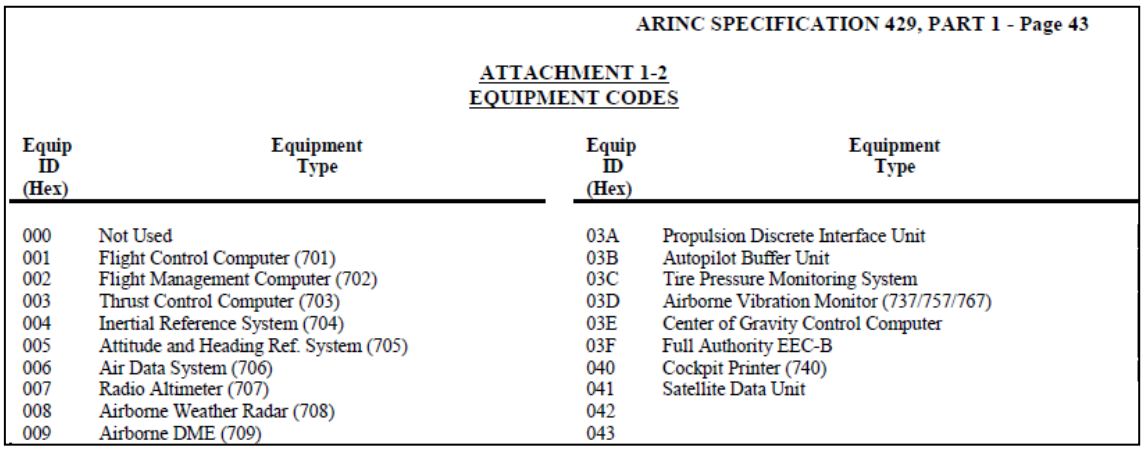
Figure 5.2 – Unicast MAC Destination address formatting.



The first and most significant octet of the OUI field is binary “10101010”, indicating that this family of MAC address is “locally administered” (second low-order bit is 1) and it is an individual address, not a group address (first low-order bit is 0), as dictated by the IEEE 802.3 standard.

The ARINC-429 specification defines a list of equipment hexadecimal codes (three hexadecimal digits) that shall be transmitted by the Label 377 (octal) in bit positions from 11 to 22 (twelve bits). Figure 5.2 illustrates part of the EQUIPMENT CODES table found in ATTACHMENT 1-2.

Figure 5.3 – Equipment Codes per ARINC-429 specification (extract).



ARINC SPECIFICATION 429, PART 1 - Page 43

ATTACHMENT 1-2
EQUIPMENT CODES

Equip ID (Hex)	Equipment Type	Equip ID (Hex)	Equipment Type
000	Not Used	03A	Propulsion Discrete Interface Unit
001	Flight Control Computer (701)	03B	Autopilot Buffer Unit
002	Flight Management Computer (702)	03C	Tire Pressure Monitoring System
003	Thrust Control Computer (703)	03D	Airborne Vibration Monitor (737/757/767)
004	Inertial Reference System (704)	03E	Center of Gravity Control Computer
005	Attitude and Heading Ref. System (705)	03F	Full Authority EEC-B
006	Air Data System (706)	040	Cockpit Printer (740)
007	Radio Altimeter (707)	041	Satellite Data Unit
008	Airborne Weather Radar (708)	042	
009	Airborne DME (709)	043	

The high-order hexadecimal digit in the low-order octet defines the Unit of the same equipment type (thus having the same Equipment ID). There can be 15 different units of the same equipment (from 1 to F hexadecimal). Unit number 0 is reserved and shall not be used in MAC Source and unicast MAC Destination addresses.

Since equipment units can have more than one connection to the network physical medium, a Port field was introduced at the low-order hexadecimal digit in the low-order octet of the MAC Source address. Each unit can have a total of 15 ports (from 1 to F hexadecimal). The Port field has no meaning in unicast MAC Destination address and shall be filled with binary zeroes.

5.1.2 IEEE 802.3 MAC destination multicast address formatting

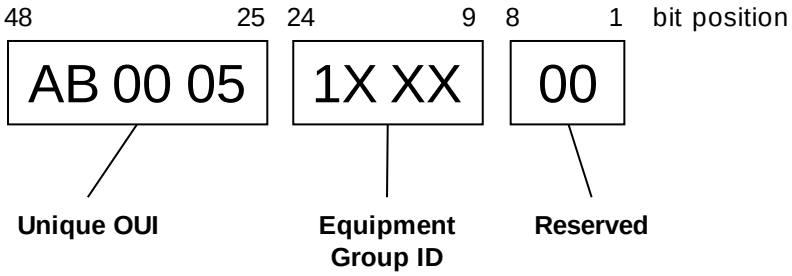
The destination MAC multicast addresses shall be formatted according to the following rules (Figure 5.3):

- The three high-order octets, bit positions from 25 to 48, shall use value 0xAB0005 hexadecimal, as this Organizationally Unique Identifier (OUI) has not been assigned to any organization;

- The two mid-order octets, bit positions from 9 to 23, shall contain the Equipment Group ID, an extension to the Equipment ID defined by the standard ARINC-429 (ATTACHMENT 1-2 for Equipment Codes); bit positions 21 to 24 shall contain 0001 (binary one at bit position 21);

- The first octet, bit positions from 1 to 8, is reserved and shall contain only binary zeroes.

Figure 5.4 – MAC destination multicast address formatting.



The first byte of the OUI field is binary 10101011, indicating that this family of MAC address is a “group address” (first low-order bit is 1), as dictated by the IEEE 802.3 standard.

The field Equipment Group ID is new and it is an extension to the Equipment ID as originally defined by the ARINC-429 specification. It facilitates the construction of MAC multicast addresses destined to reach a collection of equipments having the same Equipment ID.

5.1.3 IEEE 802.3 MAC destination broadcast address

The use of the IEEE 802.3 MAC broadcast address FF-FF-FF-FF-FF-FF is discouraged, but allowed and shall be recognized by the device attached to the network physical medium.

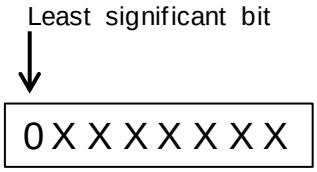
5.1.4 IEEE 802.3 length field

The IEEE 802.3 Type/Length field (2 octets following the MAC Source Address) shall be interpreted as Length. The valid values for the *Length* field start at decimal 46 (bytes) up to decimal 1500 (bytes) inclusively.

5.1.5 IEEE 802.2 DSAP and SSAP fields

The IEEE 802.2 DSAP and DSAP 8-bit fields shall have their least significant bit set to binary 0, leaving a total of 126 non-zero possible SAP numbers (all even) available (Figure 5.5):

Figure 5.5 – IEEE 802.2. DSAP and SSAP numbers formatting.



SAP numbers shall be associated to specific service implementations at the transmitting node (SSAP) and at the receiving node (DSAP). SAP number 1 (decimal) is not used in IEEE 802.2 and shall be reserved to TEST PDUs (DSAP = SSAP = 1).

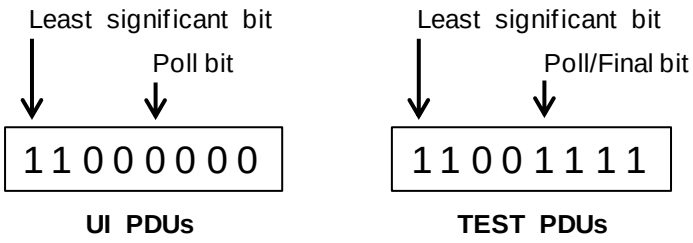
According to IEEE 802.2, having 0 as least significant bit at a DSAP number means that this DSAP is an “individual SAP” (not a “group SAP”), and having 0 as least significant bit at a SSAP number means that this SSAP is a “Command” (not a “Response”). According to IEEE 802.2, the bit following the least significant bit should be 0 for ordinary SAP numbers and should be 1 for “reserved” SAP numbers. This protocol specification deviates from IEEE 802.2 in this aspect to allow SAP numbers in a quantity that typically satisfies embedded network installations.

5.1.6 IEEE 802.2 control field

The IEEE 802.2 Control field shall be used unmodified (Figure 5.6):

- The Poll bit shall be set to 0 in UI PDUS;
- The Poll/Final bit shall be set to 0 in TEST PDUS.

Figure 5.6 – IEEE 802.2. Control field formatting.



5.1.7 Extended header for DSAP sequence number (UI PDUs)

The original IEEE 802.2 header shall be extended in UI PDUs to include a Sequence Number (SN) 8-bit field right after the Control field in the transmission order. This SN field shall be incremented by 1 every time a UI PDU is transmitted to a specific combination of DSAP and SSAP numbers. The transmitting node shall keep track of separate Sequence Numbers associated for each different combination of DSAP and SSAP numbers (see definition and utilization of “channel” in the next sections).

The SN field shall be set according to the following rules:

- The SN shall count up starting from 1 up to 255 and reset to 1 after reaching 255; the SN set to 0 shall be used by the sending node only on the first transmission after a system power-on (hard restart) or system reset (soft restart).

5.1.8 Extended header for hop count (TEST PDUs)

The original IEEE 802.2 header shall be extended in TEST PDUs to include a Hop Count (HC) 8-bit field right after the Control field in transmission order.

The HC shall be initialized and modified according to the following rules:

- The HC field shall be set by the transmitting node to the number of network nodes that a PDU has to cross until it reaches the receiving node;

- The HC field shall not exceed the value 63 decimal (hexadecimal 0x3F or binary 0011111);

- The HC field shall be decremented by 1 each time a PDU crosses a node on its path to the next node following a specific route defined by the network topology.

The receiving node shall verify that the HC in the TEST PDU is set to zero and report to the proper error handling software layer if not.

Upon receiving a TEST PDU with HC set to zero, the network node shall retransmit the TEST PDU back to the node of origin with the HC field set to value 0xC0 hexadecimal (one's complement of 0x3F or 63 decimal).

At the startup of an embedded network operation, sending a TEST PDU for validating network routes is essential to its continued correct operation. The process of transmitting, forwarding and receiving a TEST PDU allows each network node to build a vision of which network topology branches are valid at a certain point in time.

5.1.9 Extended header for time-stamping and header-CRC (UI PDUs)

The header extensions in the previous sections result in a 32-bit long header for both UI and TEST PDUs, while in the original IEEE 802.2 standard for Type 1 operation and U-format (unnumbered) PDUs the header is 24-bit long.

In this new specification, an extra 32-bit long header shall be built and transmitted right after the IEEE-802.2 extended header exclusively for UI PDUs consisting of the following fields:

- The 20 most significant bits, bit positions from 13 to 32, shall contain a the count of microseconds passed the second in normal operation (SN counting from 1 to 255); after a system power-on or system reset (SN set to 0), this field shall contain the number of seconds passed after midnight;

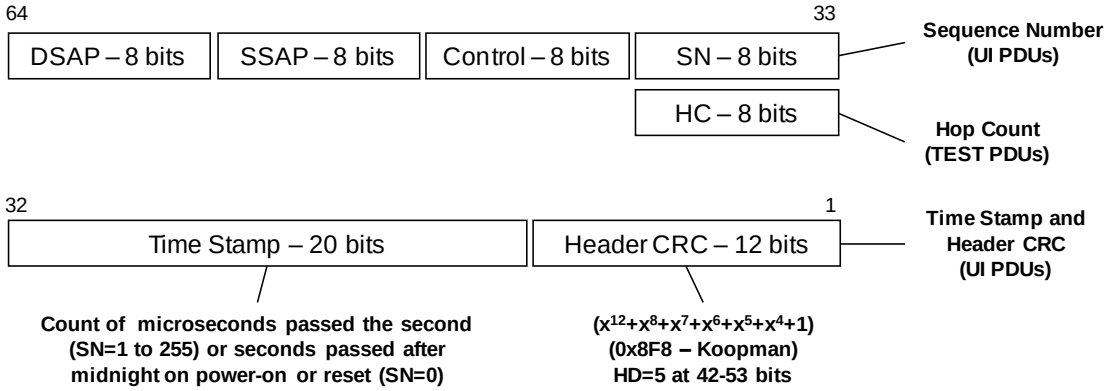
- The 12 least significant bits, bit positions from 1 to 12, shall contain a Cyclic Redundancy Check (CRC) calculated over the 32-bit IEEE 802.2 extended header and the 20-bit time-stamp using the polynomial x12+x8+x7+x6+x5+x4+1, equivalent to value hexadecimal 0x8F8 (KOOPMAN et al., 2004); according to the information published, this polynomial requires 5 bit inversions in order to create an error that is undetectable by the CRC (the "Hamming Distance" or simply HD) for data field lengths from 42 to 53 bits.

This Header CRC shall protect vital information at the receiving node, namely the DSAP and SSAP numbers, the Sequence Number (SN) and the 20-bit long time-stamp. For instance, a "good" time-stamp value shall allow the receiving

node to verify the stability of the transmitting node in producing information at the expected frequency within a confidence margin, whereas a “bad” time-stamp is of no use at all.

The 32-bit extended header and the extra 32-bit header making a new 64-bit header are illustrated in Figure 5.7:

Figure 5.7 – IEEE 802.2 extended 32-bit header and new 32-bit header.



5.1.10 New payload CRC (UI PDUs)

Nesting data frames for encapsulating different network protocols usually requires that each protocol specific data frame is protected by a separate CRC.

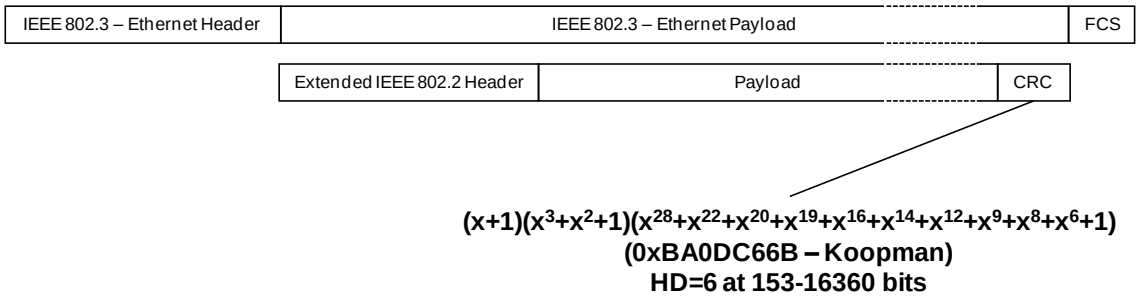
The IEEE 802.3 data packet is protected by a 32-bit CRC transmitted in the Frame Check Sequence (FCS) field (IEEE, 2012). In this specification, the application specific data transported by an UI PDU, that is, the IEEE 802.2 PDU excluding the extended IEEE 802.2 header and the new 32-bit header (a grand total of 64 bits) shall be protected by a CRC calculated using the following polynomial equivalent to value 0xBA0DC66B hexadecimal:

$$(x+1)(x^3+x^2+1)(x^{28}+x^{22}+x^{20}+x^{19}+x^{16}+x^{14}+x^{12}+x^9+x^8+x^6+1)$$

This 32-bit polynomial provides a Hamming Distance of 6 in data fields lengths from 153 to 16360 bits (KOOPMAN, 2002).

The IEEE 802.2 frame encapsulated in the IEEE 802.3 frame is illustrated in Figure 5.8:

Figure 5.8 – IEEE 802.2 PDU encapsulated in IEEE 802.3 data packet.



5.1.11 Unique characteristic of the new data link layer protocol

The most frequent form of IEEE 802.2 encapsulation of an industry standard protocol (IP for instance) implies in using the “Subnetwork Access Point” (SNAP) extension to the original header, explicitly:

- Both SSAP and DSAP fields shall contain value 0xAA hexadecimal (SNAP SAP);
- The Control field shall contain value 0x03 hexadecimal (required for UI PDU);
- The header shall be extended by extra 5 bytes:
    - 3 bytes containing binary zeroes or the OUI (for protocols not listed by ISO);
    - 2 bytes containing the Ethernet Type or the SNAP ID (for protocols not listed by ISO).

By using the Type/Length of the IEEE 802.3 header as Length and by using different numbers for the SSAP and DSAP fields of the IEEE 802.2 header (SSAP ≠ DSAP), both non-SNAP (≠ 0xAA) e non-zero, this new Data Link Layer protocol becomes an implementation of the IEEE 802.2 standard as it was originally conceived, more specifically by using DSAP and SSAP as an alternative to the Ethernet Type field.

The person in charge of the embedded network integration has the privilege of using all 126 non-zero numbers as SSAP and DSAP, however the use of already registered SAP numbers currently as listed in Table 5.1 (IEEE, 2020) is discouraged and shall be avoided:

Table 5.1 – LLC registered (SAP) numbers.

LLC address value (SSAP & Indiv. DSAP) Hexadecimal/Binary	**Organisation responsible for the document**	**Document references**
00 Z000 0000	ISO/IEC JTC1/SC6	ISO/IEC 8802-2 (1)
02 Z100 0000	ANSI	IEEE 802.1B (2)
06 Z110 0000	ANSI	ARPANET/IP (5)
0A Z101 0000	ANSI	IEEE 802.10B (11)
0E Z111 0000	IEC	IEC 955 (6)
42 Z100 0010	ISO/IEC JTC 1/SC6	ISO/IEC 10038 (3)
4E Z111 0010	ISO	ISO 9506 (8)
E6 Z110 0111	IEC TC13	IEC62056-46 (14)
7E Z111 1110	ISO/IEC JTC 1/SC6	ISO/IEC 8208 (9)
82 Z100 0001	ASHRAE	ANSI/ASHRAE 135-1995 (13)
8E Z111 0001	IEC	IEC 955 (7)
A6 Z110 0101	ISO/IEC JTC1/SC6	ISO/IEC 8802-2 (12)
AA Z101 0101	ANSI	IEEE 802 (4)
FE Z111 1111	ISO/IEC JTC 1/SC6	ISO/IEC TR 9577 (10)
REMARKS	*The bit marked 'Z' is the least significant bit and represents the command/response identifier bit in an SSAP field; or the address type designation bit (set to the value '0' - Individual) in a DSAP field. LSAP values that are neither assigned nor identified for unreserved use are reserved. The following numbers correspond to the numbers in parenthesis shown in the document reference column: 1) Used in ISO/IEC 8802-2 as the Null Address. 2) Used by IEEE 802.1b (IEEE 802.1b: IEEE Standard for LAN/MANs Network Management) to indicate LLC Sublayer Management. 3) Used in ISO/IEC 10038 (ISO/IEC 10038: 1993, Information technology - Telecommunications and information exchange between systems - LANs - Media Access Control (MAC) bridges) to identify the Bridge Spanning Tree Protocol. 4) Used in IEEE 802 (IEEE Std 802-1990, IEEE Standard for LAN/MANs: Overview and Architecture of Network Standards) to identify the SNAP SAP. 5) Used in ARPANET (RFC 791: ARPANET/IP, Internet	

	Protocol, DARPA Internet Program Protocol Specification) to identify the Internet Protocol. 6) Used in IEC 955 (IEC 955: 1989, Process Data Highway, Type C (Proway C), for Distributed Process Control Systems) to identify Network Management Maintenance and Initialization. 7) Used in IEC 955 (IEC 955: 1989, Process Data Highway, Type C (Proway C), for Distributed Process Control Systems) to identify Active station list Maintenance. 8) Used in ISO/IEC 9506 (ISO 9506: 1990, Industrial Automation Systems - Manufacturing Message Specification - Part 1: Service Definition 1st Edition; Part 2: Protocol Specification 1st Edition; Part 3: Robot Specific Message Systems) to identify Manufacturing Message Service. 9) Used to identify ISO/IEC 8208 (ISO/IEC 8208: 1995, Information technology - Data Communication - X.25 packet layer protocol for data terminal equipment) as the Network Layer Protocol. 10) Used to identify ISO/IEC TR 9577 (ISO/IEC TR 9577: 1993, Information technology - Telecommunications and information exchange between systems - Protocol identification in the network layer). 11) Used by IEEE 802.10B (IEEE 802.10B: IEEE Standard for LAN/MANs for Interoperable LAN Security (SILS) Part B) to identify the Secure Data Exchange Protocol. 12) Used in ISO/IEC 8802-2 to identify the Source Routing Route Determination Entity. 13) Used by ASHRAE (American Society for Heating, Refrigeration, and Air Conditioning Engineering) in BACnet - A Data Communication Protocol for Building Automation andControl Networks (ANSI/ASHRAE 135-1995). 14) Used in IEC62056-46 (Data exchange for meter reading, tariff and load control – Part 46: Data Link Layer using HDLC Protocol) to identify the Service user layer entity.

5.2 Specification of the associated services on UI PDUs

5.2.1 Data validation

The IEEE 802.3 standard defines a Cyclic Redundancy Check (CRC) to be used by the transmitting and receiving network nodes to generate and validate the Frame Check Sequence (FCS) field, a 32-bit value. This value is computed as a function of the contents of the protected fields of the network frame: MAC Destination and Source addresses, Length/Type field, user data and padding (if

any), that is, all frame fields except the FCS itself. The CRC polynomial for generating and verifying the FCS fields is the following (HAMMOND, 1975):

x32 + x26 + x23 + x22 + x16 + x12 + x11 + x10 + x8 + x7 + x5 + x4 + x2 + x + 1

This CRC polynomial is informally designated as “CRC-32” in the literature.

A received IEEE 802.3 frame is valid only if all its bits, exclusive of the FCS field itself, do generate a CRC value identical to the one received.

Once the IEEE 802.3 frame is validated, the IEEE 802.2 payload must be also validated before the application specific data is passed to the application. This new Data Link Layer protocol requires two separate CRC validations:

- Firstly, using the same polynomial which generated the 4-octet CRC field in the IEEE 802.2 extended frame, a 32-bit CRC shall be calculated over the application specific data and padding (if any);
- Secondly, using the same polynomial which generated the 12-bit CRC field in the IEEE 802.2 extended header, a 12-bit CRC shall be calculated over the contents of the extended header including DSAP, SSAP, Control, Sequence Number and 20-bit Time-Stamp fields, that is, all fields except the 12-bit CRC field itself.

The IEEE 802.2 extended frame shall be considered valid only if both CRC calculations are verified. Once the extended frame is validated, the application specific data shall be extracted and passed to the application.

5.2.2 Introducing the concept of “channel”

The next sections on network traffic conformance depend on the introduction of a new concept essential to the implementation of the new Data Link Layer protocol: the “channel”.

A channel represents a unique connection between a SSAP and a DSAP, therefore establishing a “producer-consumer” relationship between a SSAP (the “producer”) and a DSAP (the “consumer”).

The SSAP and DSAP numbers in a channel shall not be the same. Even when one considers that a channel is used to transport one particular data set, it is important to distinguish the producer of this data set from its consumer.

A channel can be “single-point”, when a SSAP connects to a DSAP in a single network node as in Figure 5.9, or “multi-point”, when a SSAP connects to the same DSAP number in multiple network nodes as in Figure 5.10.

Figure 5.9 – Example of single-point channels.

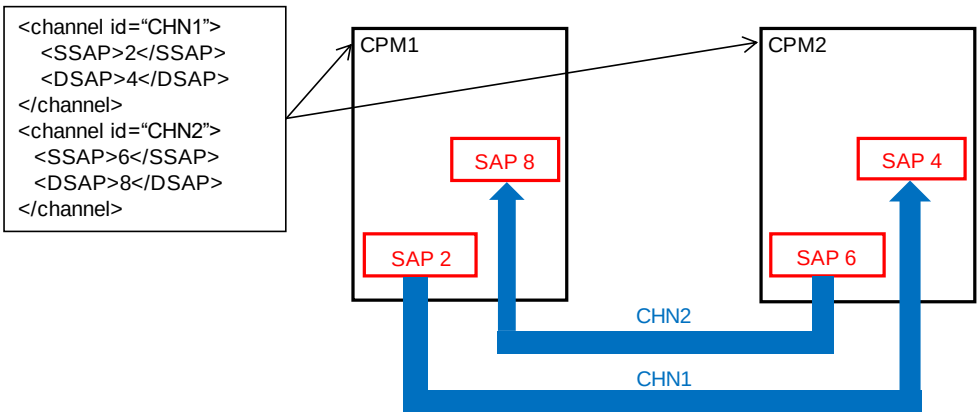
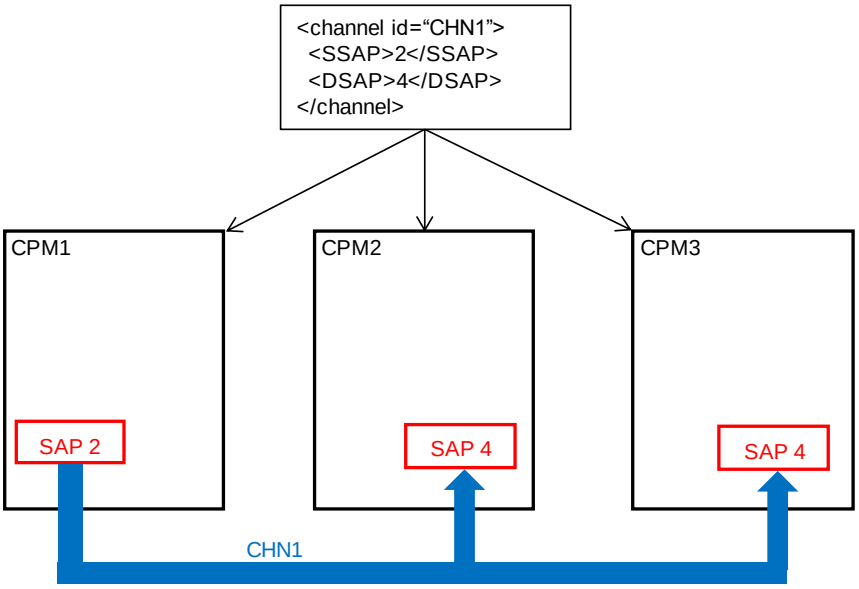


Figure 5.10 – Example of a multi-point channel.



The XML (W3C, 2006) texts in Figures 5.9 and 5.10 suggest how a channel can be defined in a network node configuration file:

- In Figure 5.9, the single-point channel CHN1 establishes a connection from SSAP 2 to DSAP 4 and the single-point channel CHN2 establishes

a connection from SSAP 6 to DSAP 8; note that the both channel definitions shall be part of the configuration of both networks nodes CPM1 and CPM2, such that either node recognize and further validate only PDUs received with a particular SSAP and only allow transmission of frames destined to a particular DSAP;

- In Figure 5.10, the multi-point channel CHN1 establishes a connection from SSAP 2 to DSAP 4; note that the Channel definition shall be part of the configuration of networks nodes CPM1, CPM2 and CPM3, such that the last two nodes recognize and further validate only PDUs received with a particular SSAP, and the first node only allows transmission of frames destined to a particular DSAP.

In essence, there is no distinction between single and multi-point channels. The difference resides in how the same channel configuration is deployed in network configuration files (in two nodes or in multiple nodes).

However, the person in charge of the network integration may choose to pick a certain range of DSAP numbers to identify them as belonging to a multi-point channel. For instance: multi-point channels should take DSAP numbers above 100. This can be very helpful when verifying proper network operation, because a particular DSAP number can be easily associated to a single or to a multi-point channel.

5.2.3 Traffic shaping

The purpose of shaping network traffic is to force the flow of network data frames transmitted by a node to be constrained – or submitted to “shaping” – to protect the network from an abnormal behavior of a node. Shaping network traffic is the responsibility of the transmitting node and shall be exercised by a specialized component before a frame passes from the LLC sub-layer to the MAC sub-layer within the new Data Link Layer protocol.

The algorithm known as “Token Bucket” (TANENBAUM, 2002) has been well adapted to embedded network environments (ARINC, 2009) as well as to commercial networks (CISCO, 2020) for performing both traffic shaping and policing.

In this specification, traffic shaping shall be solely used in the transmission of UI PDUs and shall use the Token Bucket algorithm separately for each configured channel, as follows:

- The channel bucket is initially filled with *T* tokens and can hold at the most *C* tokens;
- A token is added to the channel bucket every *1*/*R* seconds; if a token arrives when the channel bucket is full, then it is discarded;
- When a UI PDU of *N* bytes is ready to be transmitted, ask:
    - If at least *N* tokens exist in the channel bucket, then *N* tokens are removed from the channel bucket, and the UDI PDU is passed to the MAC sub-layer;
    - If fewer than *N* tokens are available, no tokens are removed from the channel bucket, the UI PDU is not passed to the MAC sub-layer, an error condition shall be raised and passed to the application associated to the channel for proper handling.

5.2.4 Traffic policing

The purpose of policing network traffic is to force the flow of network data frames received by a node to be constrained – or submitted to “policing” – to protect the node from an abnormal behavior of another node. Policing network traffic is the responsibility of the receiving node and shall be exercised by a specialized component before a frame passes from the MAC sub-layer to the LLC sub-layer within the new Data Link Layer.

In this specification, traffic policing shall be solely used in the reception of UI PDUs and shall use the Token Bucket algorithm separately for each configured channel, as follows:

- The channel bucket is initially filled with *T* tokens and can hold at the most *C* tokens;
- A token is added to the channel bucket every *1*/*R* seconds; if a token arrives when the channel bucket is full, then it is discarded;

- When a UI PDU of *N* bytes is passed from the MAC sub-layer to the LLC sub-layer, ask:
    - If at least *N* tokens exist in the channel bucket, then *N* tokens are removed from the channel bucket, and the UDI PDU is passed to the application;
    - If fewer than *N* tokens are available, no tokens are removed from the channel bucket, the UI PDU is not passed to the application, and an error condition shall be raised and passed to the application associated to the channel for proper handling.

5.2.5 Taking into account transmission delays

Traffic Policing is particularly important in network star or multi-star topologies.

In aerospace applications, the process of producing data is strictly periodic. However, data has to be encapsulated in network frames which need to travel from its origin node to its destination node crossing one or more traffic switching points, for example Ethernet switches, if this is the chosen network infrastructure.

Crossing switches imposes transmission delays which have to be taken into account in the Token Bucket algorithm; otherwise a valid data frame periodically generated by a transmitting node could be discarded by a receiving node because it did not pass Traffic Policing. To account for these anomalies observed at a receiving node in the periodicity of otherwise perfectly periodic transmissions, the Token Bucket algorithm needs to be programmed with an “overdraft”.

To assess this overdraft, it is essential to estimate transmission delays for a particular network data frame while traversing the network infrastructure.

For this purpose, there were several studies conducted by a research group located in Toulouse, France. Bauer published a method called “Trajectory Approach” in 2011 (BAUER, 2011), which was perfected by Kemayo in 2013 (KEMAYO et al., 2013). In 2014, Kemayo introduced another method called “Forward Analysis” (KEMAYO et al., 2014), which was perfected in 2015 (KEMAYO et al., 2015) and extended in 2017 (BENAMMAR et al., 2017).

The Appendix A introduces a new method for estimating transmission delays of a network data frame while crossing an Ethernet switch which is simpler than the other referenced methods.

5.2.6 Summary of protocol services for UI PDUs on network nodes

The next two figures illustrate the protocol services associated to the LLC sub-layer and to the MAC sub-layer of the new Data Link Layer protocol on a transmitting node (Figure 5.11) and on a receiving node (Figure 5.12).

It is each network node's responsibility to map a channel to one (on single-point channels) or more network nodes (on multi-point channels).

Figure 5.11 – Protocol services on a transmitting node.

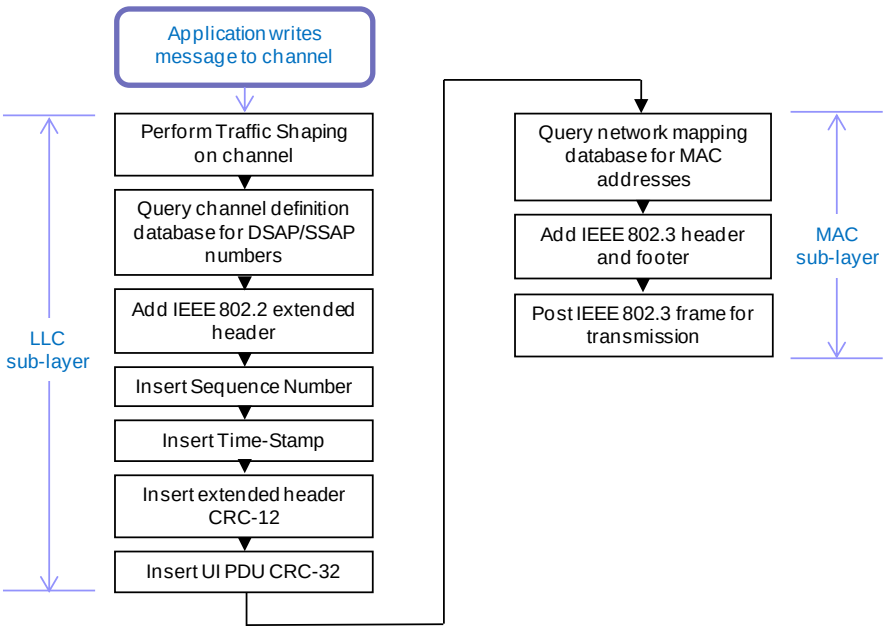
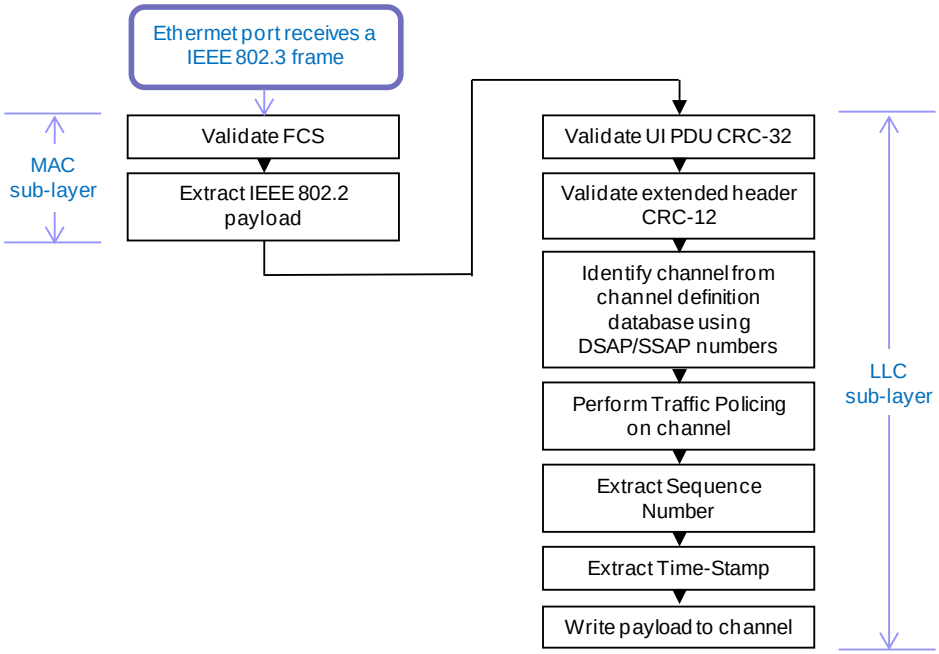


Figure 5.12 – Protocol services on a receiving node.



5.3 Routing validation using TEST PDUs

The exact network topology shall be defined during the design phase of the embedded system it serves.

A route shall be understood as the communication path from a node to another node across the network. A frame transmitted by a node may find its way to the intended destination node without crossing any other node or it might have to be forwarded by one or more intermediate nodes until it reaches its final destination.

Routes in this specification shall be used in the transport of UI PDUs. However, the route validation shall be performed using a TEST PDU and shall be performed at network startup once all nodes announce themselves as capable of transmitting and receiving data over their physical network connections.

5.3.1 Definition of static routes

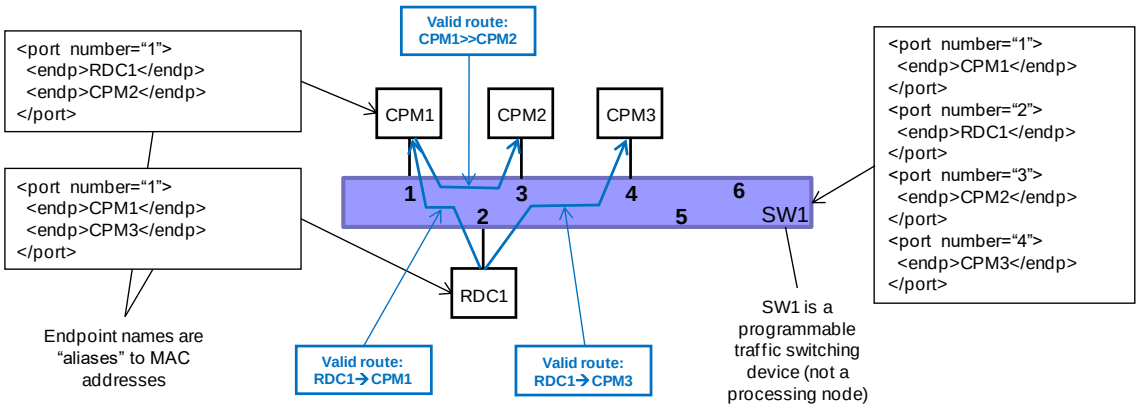
For a safe network operation, in particular in aerospace applications where tolerating a communication failure may be vital to the safety of vehicle, it is desirable to have more than one route from a node to another node (at least two).

Tolerating communication failures also means to be able to recognize abnormal behavior. For this reason, all network routes shall be statically defined, validated and verified before the network is deemed operational once it presents the expected behavior. No other routes shall be recognized by the network nodes and any identified deviations from the original design shall be reported as an error to the proper software layer.

Network nodes in static route definitions are referred to as “endpoints” and the physical connection points to the network are referred to as “ports”.

Figure 5.13 illustrates static route definitions on a star topology.

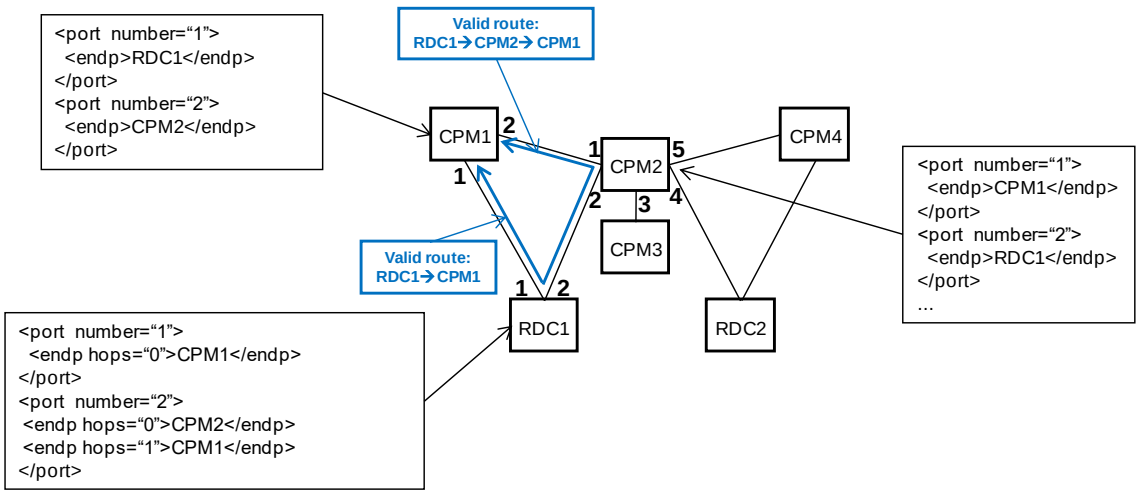
Figure 5.13 – Static route definitions on a star topology.



Note on Figure 5.13 that the element SW1 performs the role of a network traffic switching device, or simply a switch, and is not required to participate explicitly in a route definition. However, network traffic switches are programmable devices and as such need to be properly configured as part of the network integration process. Traffic switches are essential to star network topologies.

Figure 5.14 illustrates static route definitions on a point-to-point topology

Figure 5.14 – Static route definitions on a point-to-point topology.



The XML texts in Figures 5.13 and 5.14 suggest how routes can be defined in a network node configuration file. There shall be a separate section in the configuration file for assigning ports to endpoints.

Note that the XML texts in Figure 5.13 for nodes CPM1 and RDC1 do not mention SW1, but there is a XML text for SW1 suggesting its programming, for the SW1 configuration shall be also statically defined. The actual configuration of a traffic switch highly depends on its manufacturer and its format is beyond the scope of this specification.

Particular attention shall be given to the definition of ports when a node is only reachable over another network node. In Figure 5.14, the assignment of ports to endpoints for node RDC1 indicates a route to node CPM1 over two different ports: 1) via port number 1 directly (attribute “hops” equal 0); 2) via port number 2 indirectly (attribute “hops” equal to 1).

Node CPM2 is directly connected to node RDC1 via port number 2, as indicated by an entry in the port to endpoint assignment section for node RDC1 (attribute “hops” equal 0). Therefore, the path from node RDC1 to node CPM1 via its port number 2 shall cross node CPM2 and only node CPM2. As consequence, the port to endpoint assignment section in node CPM2 indicates that there is a direct route to node CPM1 via port number 1 (the attribute “hops” defaults to 0 and can be omitted for simplification).

5.3.2 Route validation

During the start-up phase of the network operation, routes statically defined during the design phase of the embedded system it serves shall be validated using the 8-bit Hop Count (HC) field present in the extended IEEE 802.2 header of TEST PDUs.

A network node shall validate a particular route by transmitting a TEST PDU over a physical network port associated to a route endpoint. The HC field shall be initialized with the number of network nodes that the PDU has to cross until it reaches its final destination. Each node forwarding a TEST PDU shall decrement the HC field by 1.

The node receiving the TEST PDU with HC set to zero shall return it to the network node which originally transmitted it by setting the HC field to value 0xC0 hexadecimal (one's complement of 0x3F hexadecimal or 63 decimal), swapping the MAC Source and Destination addresses in the IEEE 802.3 header and modifying the Port number field in the low order octet of the latter.

A network node shall recognize itself as being a route endpoint by comparing its own Equipment ID and Unit Number with these fields in the MAC Destination address of the TEST PDU it has just received:

- If these two fields match, the network node is a route endpoint;
- If these two fields do not match, the node is a forwarding node in the route from the network node associated with the MAC Source address to the network node associated with the MAC Destination address.

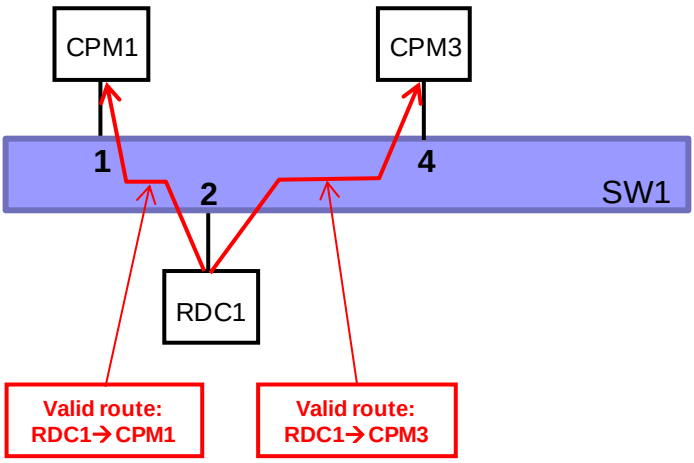
Any network node forwarding a TEST PDU shall verify that the HC field is not set to zero, for it should be zero only when received by the endpoint of a route. Should a network node detect such event when forwarding a TEST PDU, it shall report it to the proper error handling software layer.

A TEST PDU being returned to the network node which transmitted it originally shall be identified by the HC field set to value 0xC0 hexadecimal.

5.3.3 Sample route validation on a star network topology

Figure 5.15 illustrates a simple scenario on a network following a star topology, where network node RDC1 has static routes to nodes CPM1 and CPM3 over a network traffic switching device SW1 (reference Figure 5.13).

Figure 5.15 – Sample routing scenario on a star topology.



For network node RDC1, the sequence of steps for validating the route to CPM1 are as follows:

1) RDC1 sends a TEST PDU with the MAC Destination set to CPM1 and HC=0;
2) In SW1, CPM1 is connected to port 1;
3) SW1 receives the TEST PDU and forwards the TEST PDU to CPM1 over port 1; SW1 does not actively participate in the route validation;
4) CPM1 receives and validates the TEST PDU (HC must be zero) from RDC1, swaps MAC Source and Destination addresses and fixes the Port number field in both, sets HC to 0xC0 and sends the TEST PDU back to RDC1;
5) SW1 receives the TEST PDU with HC set to 0xC0 returning from CPM1 and verifies that the destination is RDC1;
6) In SW1, RDC1 is connected to port 2;
7) SW1 forwards the TEST PDU to RDC1 over port 2;

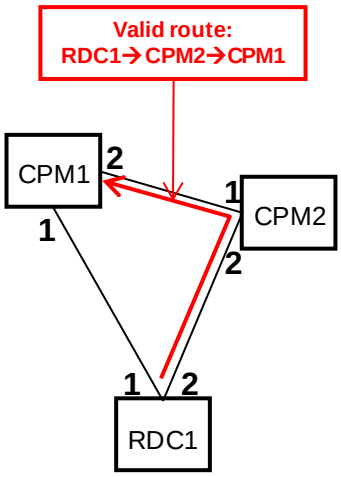
8) RDC1 receives the TEST PDU from CPM1 with HC set to 0xC0 and validates the route from RDC1 to CPM1.

Similar sequence of steps shall be followed by RDC1 for validating the route to CPM3.

5.3.4 Sample route validation on a point-to-point network topology

Figure 5.16 illustrates a simple scenario on a network following a point-to-point topology, where network node RDC1 has a static route to node CPM1 over another network node CPM2 (reference Figure 5.14).

Figure 5.16 – Sample routing scenario on a point-to-point topology.



For network node RDC1, the sequence of steps for validating the route to CPM1 are as follows:

1) RDC1 sends a TEST PDU with MAC Destination set to CPM1 e HC=1 to CPM2 over port 2;
2) In CPM2, CPM1 is connected to port 1;
3) CPM2 receives the TEST PDU from RDC1 on port 2, decrements HC by 1 and forwards the TEST PDU to CPM1 over port 1;
4) In CPM1, CPM2 is connected to port 2;
5) CPM1 receives and validates the TEST PDU (HC must be zero) from RDC1 on port 2, swaps MAC Source and Destination addresses and

fixes the Port number field in both, sets HC to 0xC0 and sends the TEST PDU back to RDC1 over port 2;

6) CPM2 receives the TEST PDU with HC set to 0xC0 returning from CPM1 on port 1 and verifies that the destination is RDC1;
7) CPM2 forwards the TEST PDU to RDC1 over port 2;
8) RDC1 receives the TEST PDU from CPM1 with HC set to 0xC0 and validates the route from RDC1 to CPM1 over CPM2.

RDC1 shall also validate the direct route to CPM1 by sending a TEST PDU with MAC Destination address set to CPM1 and HC set to 0 over port 1.

5.4 Specification of the operating system interface to the protocol layers

The implementation of a network protocol in a software environment requires a more sophisticated structure implemented by an operating system matching the underlying hardware.

The traditional approach is to implement what is loosely called “protocol stack”, a designation motivated by the ISO/OSI layered model (a “stack” of layers).

A protocol stack implementation requires the construction of operating system data structures for supporting the operating system services which actually do the job.

These operating system services shall offer a software interface to the applications willing to use the network protocol.

The responsibility of configuring a node to operate a network protocol belongs to a person who performs the role of the network integrator, which has to make sure that the network operates as specified by the clients using this network as a means of communication.

The simplest way of defining the configuration of protocol stack for a particular network node is to use a text file. A text file can be directly read (or printed) by humans, although a more formal validation of its correctness usually requires a software tool.

The “eXtensible Markup Language”, XML for short (W3C, 2008), is a very convenient means for defining a network node configuration in text format, for it

is supported by a wide variety of open-source software tools for writing, reading and contents validation.

XML file contents validation may require another text file called “XML schema” (W3C, 1998), which is complementary to the original XML text file.

The next sections describe the set of XML “tags” required to create the network node configuration file, the data structures to be built by the operating system and the application programming interface for supporting the new Data Link Layer protocol.

5.4.1 Network node configuration file

5.4.1.1 Configuration identification

The identification of a given network node configuration requires the *configuration* tag pair.

This tag admits two attributes:

- *host*: the identification of the node for which the configuration is intended, a single text word with minimum 4 and maximum 8 characters containing any combination of letters from A to Z and numeric digits from 0 to 9; no other special characters are allowed (see next section).
- *name*: the identification of the configuration, a single text word of minimum 4 and maximum 8 characters containing any combination of letters from A to Z and numeric digits from 0 to 9; no other special characters are allowed.

The person in charge of the network integration shall decide whether the identification of a configuration is unique to the whole network or specific to particular network node.

The *configuration* tag pair encapsulates all the XML tag pairs required in a network node configuration file.

A sample XML text for the *configuration* tag pair looks as follows (texts in red font are user entries or comments):

```
<configuration name="sample" host="CPM1">
  <! all other tag pairs shall be inserted here>
</configuration>
```

5.4.1.2 Node identification

The formatting of MAC addresses associated to a particular network node and to all other nodes it communicates with requires the *nodes* tag pair, which includes a collection of *node* tag pairs, one for each network node's Equipment ID and Unit number.

The *node* tag admits a single attribute *name*, a single text word with minimum 4 and maximum 8 characters containing any combination of letters from A to Z and numeric digits from 0 to 9. No other special characters are allowed.

Each *node* tag pair requires two other tag pairs for defining the network node's own identity and the identity of all other network nodes it shall communicate with:

- *hexid*: the three hexadecimal digit Equipment ID associated with the function executed by the network node per the ARINC-429 standard (ARINC, 2001);
- *unit*: a decimal number from 1 to 15 representing the instance of the present node within the collection of identical Equipment ID in the network.

A sample XML text for the node identification tag pairs looks as follows:

```
<nodes>
  <node name="CPM1">
    <hexid>004</hexid>
    <unit>1</unit>
  </node>
  <node name="CPM2">
    <hexid>00B</hexid>
    <unit>1</unit>
  </node>
</nodes>
```

5.4.1.3 Service identification

The services (SAP numbers) hosted by a particular network node for communicating with a counterpart implemented by other network node require the *SAPs* tag pair, which in turn require a collection of *number* tag pairs, one for each implemented SAP.

The *number* tag admits even decimal numbers from 2 to 254. The SAP number 170 decimal (0xAA hexadecimal) is reserved for the SNAP SAP.

A sample XML text for the service identification tag pair looks as follows:

```
<SAPs>
  <number>2</number>
  <number>8</number>
</SAPs>
```

5.4.1.4 Channel identification

The channels used by a particular network node for connecting a SSAP to a DSAP require the *channels* tag pair, which in turn requires a collection of *channel* tag pairs, one for each configured channel.

The *channel* tag admits a single attribute *id*, a single text word with 4 characters containing any combination of letters from A to Z and numeric digits from 0 to 9. No other special characters are allowed.

The *channel* tag pair requires two other tag pairs for declaring the SSAP and the DSAP numbers defining the channels through which data is expected to be transmitted and received by the presently configured network node:

- *SSAP*: the SAP number through which data is produced at the source node;
- *DSAP*: the SAP number through which data is to be consumed at the destination node;
- *capacity*: the maximum number of tokens in the channel bucket used for Traffic Shaping and Policing;
- *tokens*: the initial number of tokens in the channel bucket;
- *rate*: the number of tokens per unit of time to fill the channel bucket.

The *SSAP* and *DSAP* tags shall admit the same valid inputs as the *number* tag in the previous section on service identification.

A sample XML text for the channel identification tag pairs looks as follows:

```
<channels>
  <channel id="CHN1">
    <SSAP>2</SSAP>
    <DSAP>4</DSAP>
    <capacity>90</capacity>
    <tokens>60</tokens>
    <rate>30<rate/>
  </channel>
  <channel id="CHN2">
    <SSAP>6</SSAP>
    <DSAP>8</DSAP>
    <capacity>60</capacity>
    <tokens>60</tokens>
    <rate>60<rate/>
  </channel>
</channels>
```

5.4.1.5 Service to host configuration

The services (SAP numbers) hosted by other network nodes require the *services* tag pair, which in turn requires a collection of *SAP* tag pairs, one for each hosted service.

The *SAP* tag admits a single attribute *number*, the SAP number hosted by the node.

The *SAP* tag pair requires one other tag pair:

- *host*: the name of the network node hosting the SAP from a node defined in the previous node identification section.

A sample XML text for the service to host tag pairs looks as follows:

```
<services>
  <SAP number="4">
    <host>CPM2</host>
  </SAP>
  <SAP number="6">
    <host>CPM2</host>
  </SAP>
</services>
```

5.4.1.6 Port to endpoint configuration

Each network node needs one or more physical access to the network. In the network node configuration file, this physical connection is named “port”.

Each port in a network node is a path to one or more route endpoints in one of two ways:

- Directly, when the route endpoint is a network node connected to the port;
- Indirectly, when the route endpoint is reached over another network node.

The configuration of ports requires the *ports* tag pair, which includes one or more *port* tag pairs, one for each port.

The *port* tag admits a single attribute *number*, a decimal number from 1 to 15.

The *port* tag pair includes one or more *endp* tag pairs, one for each connected endpoint.

The *endp* tag admits a single attribute *hops*, a decimal number from 0 to 63 representing the number of network nodes that a PDU transmitted by the present network node has to cross until it reaches the route endpoint. This attribute shall be directly used as the initial Hop Count (HC) field value in TEST PDUs used for route validation.

When the route endpoint can be reached directly over a port, the *hops* attribute has a value of 0 and can be omitted (the value 0 becomes the default value for the attribute *hops*).

A sample XML text for the port to endpoint tag pairs looks as follows:

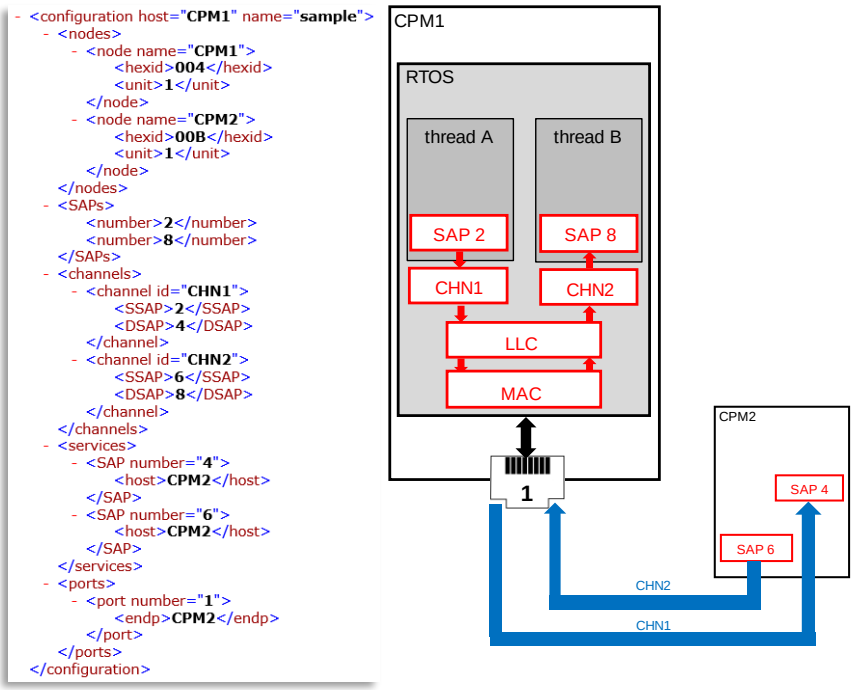
```
<ports>
  <port number="1">
    <endp hops="0">CPM1</endp>
  </port>
  <port number="2">
    <endp hops="0">CPM2</endp>
    <endp hops="1">CPM1</endp>
  </port>
</ports>
```

5.4.1.7 Sample configuration files

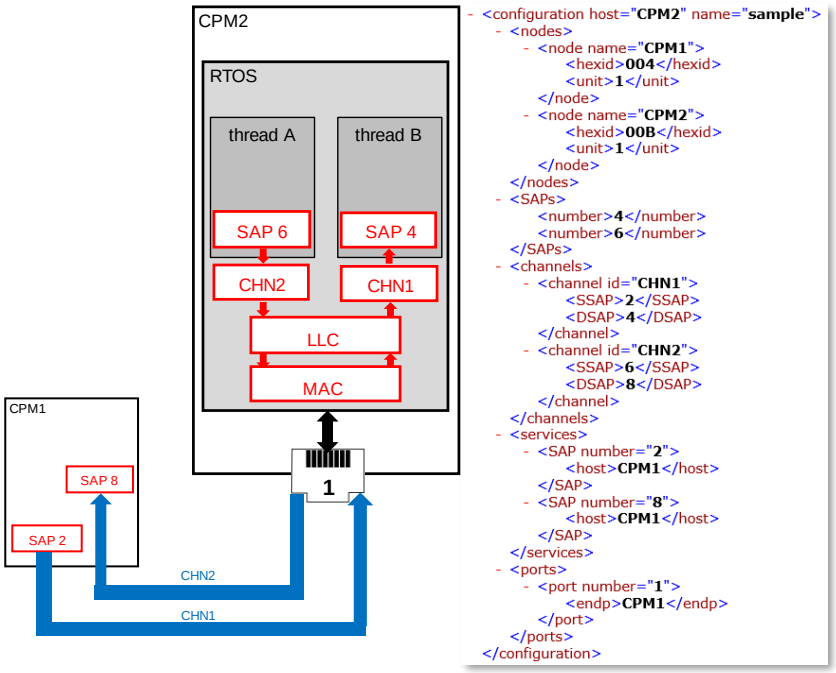
Figures 6.17 (a) and (b) illustrate network configuration files formatted in XML for the network nodes CPM1 and CPM2 previously shown in Figure 5.9, (partly repeated in miniature for reference). The XML tag pairs defining the traffic flow control parameters for each channel were omitted for simplicity.

Nodes CPM1 and CPM2 have defined channels CHN1 and CHN2 connecting SAP 2 to SAP 4 and SAP 6 to SAP 8 respectively. Note that channel identification sections shall be the same in both configuration files.

Figure 5.17 – Sample network configuration files.



(a) for CPM1



(b) for CPM2

From a node identification section, CPM1 can build the MAC addresses for itself and for CPM2. From a SAP identification section, CPM1 can start the proper operating system schedulable entities (thread or process) for each configured SAP. From a channel identification section, CPM1 can establish how each configured SAP will interact with a SAP implemented by other network node. From services to host section, CPM1 can find which SAP is hosted by which network node. From a port to endpoint assignment section, CPM1 can finally find which network port it shall use for communicating with CPM2.

The network configuration file deployed in CPM2 must be consistent with the network configuration file deployed in CPM1. If these two files are inconsistent, a PDU transmitted by CPM1 will be reported as invalid at reception on CPM2 if the DSAP field in the IEEE 802.2 LLC header is not a SAP hosted by CPM2.

The network configuration file deployed in CPM1 must be also consistent in itself. For instance, an endpoint referred to in the port to endpoint assignment section must be a network node (other than CPM1) defined in the node identification section.

Consistency in network node configuration files is mandatory; therefore they should be generated by a software application qualified for assisting in the network integration process.

5.4.2 In-memory data structures

Once all network node configuration files are created and verified, each one shall be deployed to the corresponding network node.

These configuration files shall be used to manage the services put in place by the node’s operating system for implementing the protocol layers involved in the transmission and reception of network data frames built following the IEEE 802.3 MAC and the extended IEEE 802.2 LLC layers introduced by this specification.

The information elements contained in each section of a network node configuration file shall be extracted and used for building operating system in-memory data structures to be accessed by the protocol layer services responsible for assembling, transmitting, receiving, parsing and validating network data frames.

The next section describes naming conventions used in the after next sections for describing each one of the data structures.

5.4.2.1 Naming conventions

The naming of the fields composing in-memory data structures shall obey the following conventions:

- The general format of a structure field shall be <*structure_name*>**$**<*data_type*>_<*field_name*>;
- The <*structure_name*> shall be a three letters mnemonic name of the parent data structure;
- The <*data_type*> shall be one of the following primitive data types:
    - B – for an 8-bit byte unsigned
    - W – for a 16-bit word unsigned
    - L – for a 32-bit longword unsigned
    - S – for an 8 byte, 64-bit long field of unspecified data type
    - R – for processor specific, privileged access memory address
- The <*field_name*> shall be a short field description with maximum 8 letters.

5.4.2.2 Node Identification Block (NIB)

The essential information for identifying a network node or a family of network nodes shall be stored in a structure called *Node Identification Block* (NIB).

The NIB is composed by the following fields (longword aligned):

- NIB$S_NNAME – the mnemonic name of an individual network node or a family of nodes;
- NIB$S_MACB – the MAC address of the node or family of nodes, including the MAC prefix for unicast (individual node) or multicast (family of nodes), the Equipment ID and the applicable Unit number;
- NIB$W_HEXID – the Equipment ID assigned to the node or family of nodes;

- NIB$B_UNIT – the Unit number of the node's Equipment ID for MAC unicast or 0 for MAC multicast.

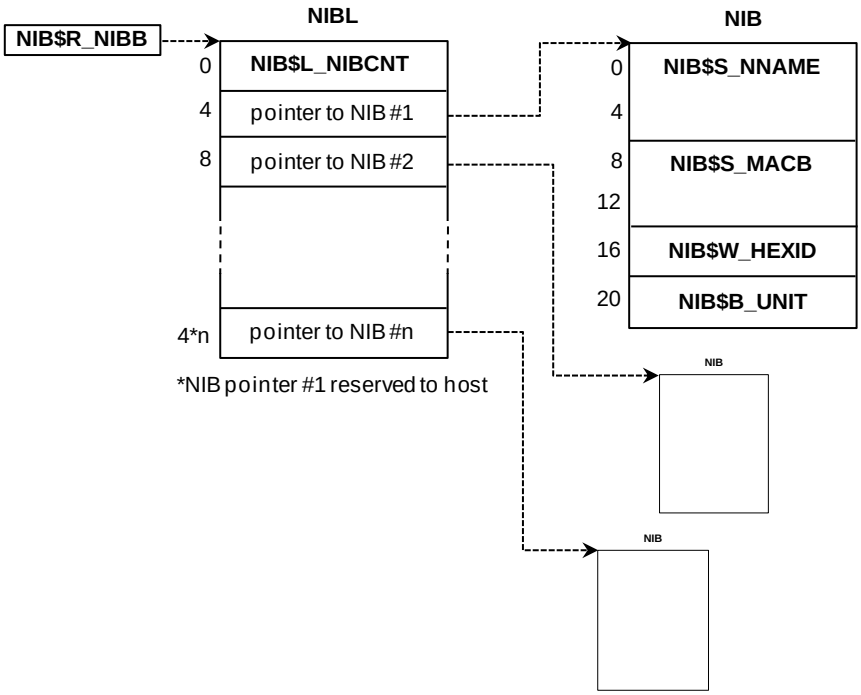
The NIBs configured for a particular node shall be pointed to by another structure named *Network Identification Block List* (NIBL).

The NIBL is composed by following fields (longword aligned):

- NIB$L_NIBCNT – the number *n* of NIBs pointed to by the NIBL;
- one or more memory addresses (pointers) of up to *n* NIBs.

The memory address of the NIBL shall be pointed to by a special register called *Network Identification Block Base* or NIB$R_NIBB. Figure 5.18 illustrates the NIB, the NIBL and its base register.

Figure 5.18 – Node Identification Block and associated structures.



Note that the first position in the NIBL right after the NIB$L_NIBCNT field shall be filled by the memory address of the NIB identifying the host node, that is, the node presently being configured.

5.4.2.3 Service Identification Block (SIB)

The identification of the services implemented by the host node associated to SAP numbers shall be stored in a structure called *Service Identification Block* (SIB).

The SIB is composed by the following fields (longword aligned):

- SIB$B_SAP – the SAP number;
- SIB$L_PID – the identification (usually a hexadecimal number) given by the node's operating system to the schedulable entity (process or thread) implementing the service.

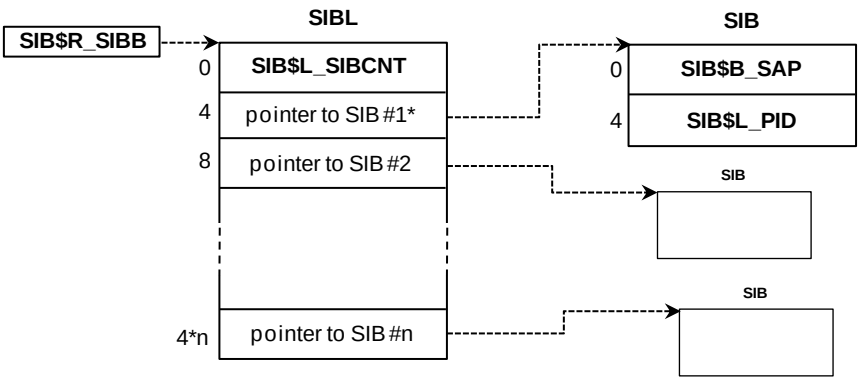
The SIBs configured for a particular node shall be pointed to by another structure named *Service Identification Block List* (SIBL).

The SIBL is composed by following fields (longword aligned):

- SIB$L_SIBCNT – the number *n* of SIBs pointed to by the SIBL;
- one or more memory addresses (pointers) of up to *n* SIBs.

The memory address of the SIBL shall be pointed to by a special register called *Service Identification Block Base* or SIB$R_SIBB. Figure 5.19 illustrates the SIB, the SIBL and its base register.

Figure 5.19 – Service Identification Block and associated structures.



5.4.2.4 Channel Control Block (CCB)

The identification of the channels implemented by the host node shall be stored in a structure called *Channel Control Block* (CCB).

The CCB is composed by the following fields (longword aligned):

- CCB$S_CHNID – the mnemonic identification of the channel;
- CCB$B_SSAP – the SAP source (producer) part of the channel;
- CCB$B_DSAP – the SAP destination (consumer) part of the channel;
- CCB$L_CAPACITY – the maximum count of tokens in the Token Bucket (to be used in Traffic Shaping and Traffic Policing);
- CCB$L_TOKENS – the initial count of tokens in the Token Bucket (ditto);
- CCB$L_RATE – the rate programmed for the channel in tokens per second (ditto);
- CCB$L_PID – the identification of the process owning the channel;
- CCB$B_SN – the Sequence Number of the last IEEE 802.2 PDU received or transmitted over the channel;
- CCB$L_TIMESTP – the Time Stamp of the last IEEE 802.2 PDU received or transmitted over the channel.

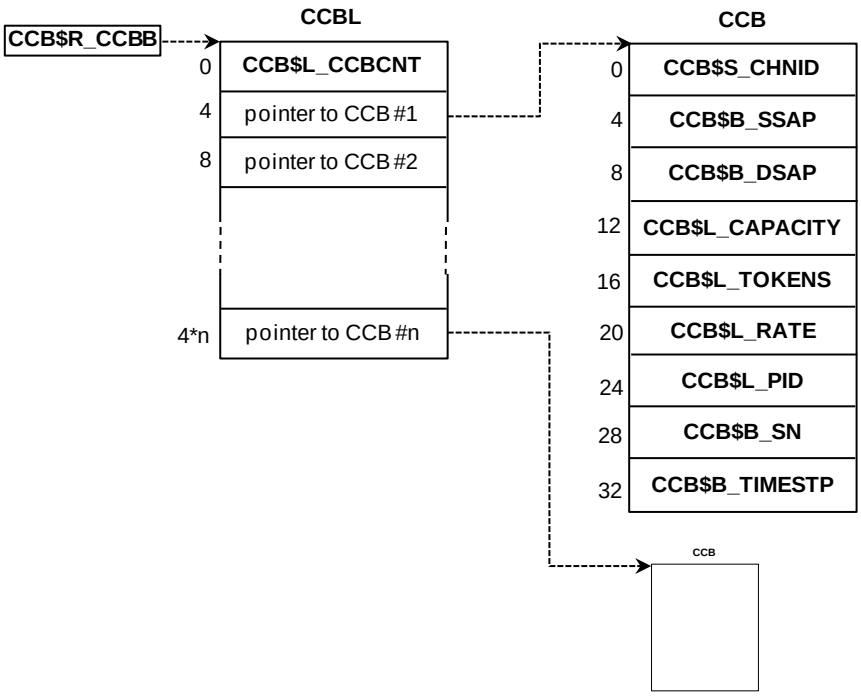
The CCBs configured for a particular node shall be pointed to by another structure named *Channel Control Block List* (CCBL).

The CCBL is composed by following fields (longword aligned):

- CCB$L_CCBCNT – the number *n* of CCBs pointed to by the CCBL;
- one or more memory addresses (pointers) of up to *n* CCBs.

The memory address of the CCBL shall be pointed to by a special register called *Channel Control Block Base* or CCB$R_CCBB. Figure 5.20 illustrates the CCB, the CCBL and its base register.

Figure 5.20 – Channel Control Block and associated structures.



5.4.2.5 Service Hosting Block (SHB)

The identification of the services implemented by network nodes other than the host node associated to SAP numbers shall be stored in a structure called *Service Hosting Block* (SHB).

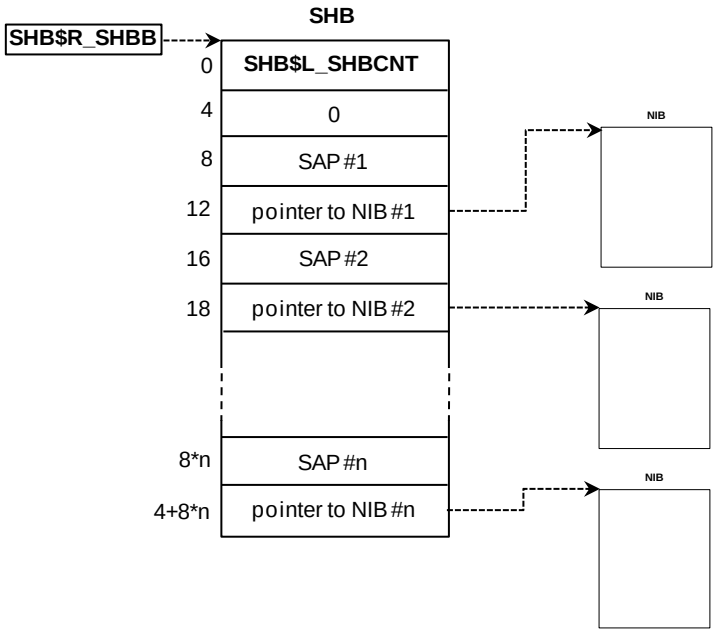
The SHB is composed by the following fields (longword aligned):

- SHB$L_SHBCNT – the number of longword pairs associating a SAP number to a network node pointed to by a *Network Identification Block* (NIB);
- a longword (zero) filler for aligning the SHB fields to double longwords;
- one or more longword pairs, each consisting of:
    - the SAP number associated to the service;
    - the memory address of the NIB of the network node implementing the service.

SAP numbers associated to multi-point channels shall point to a NIB holding a MAC multicast address.

The memory address of the SHB shall be pointed to by a special register called *Service Hosting Block Base* or SHB$R_SHBB. Figure 5.21 illustrates the SHB, and its base register.

Figure 5.21 – Service Hosting Block and base register.



5.4.2.6 Port Assignment Block (PAB)

The assignment of network node ports to route endpoints shall be stored in a structure called *Port Assignment Block* (PAB).

The PAB is composed by the following fields (longword aligned):

- PAB$L_NIBCNT – the number of longword pairs associating a route endpoint pointed to by a *Network Identification Block* (NIB) to the number of network nodes that a PDU has to cross until it reaches its final destination, or number of “hops”;
- a longword (zero) filler for aligning the PAB fields to double longwords;
- one or more longword pairs, each consisting of:
    - the memory address of the NIB of the route endpoint;

- the number of hops to the route endpoint.

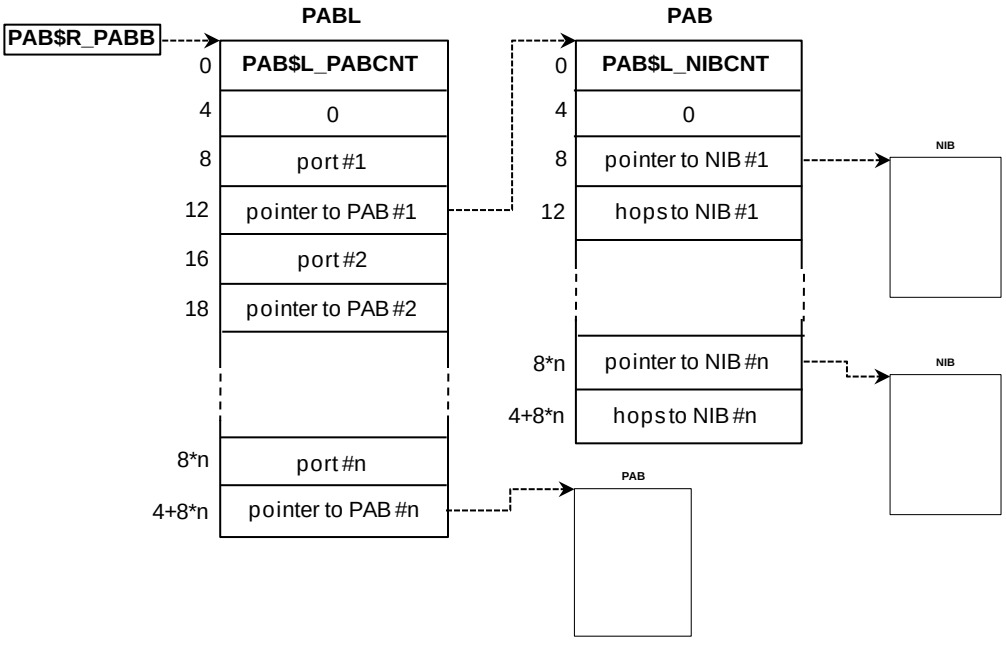
The PABs configured for a particular node shall be pointed to by another structure named *Port Assignment Block List* (PABL).

The PABL is composed by following fields (longword aligned):

- PAB$L_PABCNT – the number of longword pairs associating a port number to a PAB;
- a longword (zero) filler for aligning the PABL fields to double longwords;
- one or more longword pairs, each consisting of:
  - the number of the port from *1* to *n*;
  - the memory address of the PAB associated to the port.
- The memory address of the PABL shall be pointed to by a special register called *Port Assignment Block Base* or PAB$R_SIBB. Figure 5.22 illustrates the PAB, the PABL and its base register.

The memory address of the PABL shall be pointed to by a special register called *Port Assignment Block Base* or PAB$R_PABB. Figure 5.22 illustrates the PAB, the PABL and its base register.

Figure 5.22 – Port Assignment Block and associated structures.



5.4.2.7 Configuration Identification Block (CIB)

The identification of the configuration shall be stored in a structure called *Configuration Identification Block* (CIB).

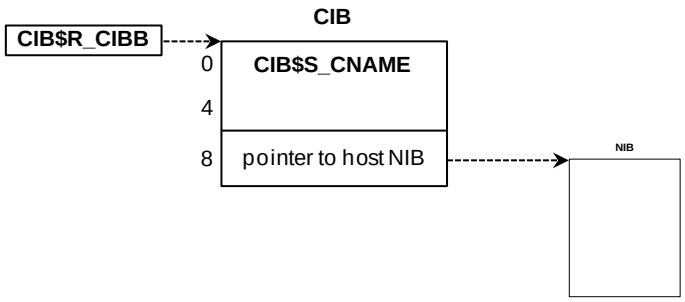
The CIB is composed by following fields (longword aligned):

CIB$S_CNAME – the configuration name, maximum 8 alpha-numeric characters long;

the memory address of the Node Identification Block (NIB) of the host node.

The memory address of the CIB shall be pointed to by a special register called *Configuration Identification Block Base* or CIB$R_CIBB. Figure 5.23 illustrates the CIB and its base register.

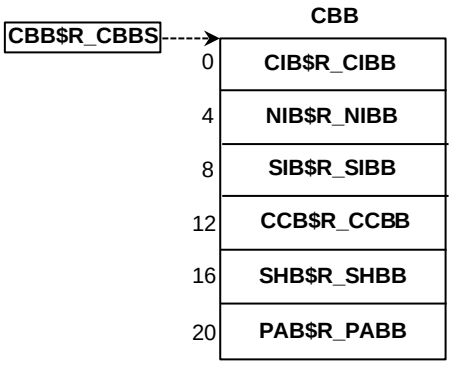
Figure 5.23 – Configuration Identification Block and its base register.



5.4.2.8 Configuration Base Block (CBB)

All the base registers shall be stored in a structure called *Configuration Base Block* (CBB) and pointed to by a special register called *Configuration Base Block Summary* or CCB$R_CBBS, as illustrated in Figure 5.24.

Figure 5.24 – Configuration Base Block and its summary register.



5.4.3 Channel Application Programming Interface

The introduction of the “channel” concept with the specification of the new Data Link Layer protocol shall require a programming interface to applications willing to use this communication resource in a network of embedded systems on board of aerospace vehicles.

Applications running in an embedded computing environment can be producers of a certain data set or consumers of a certain data set, but rarely assume both roles for the exact same data set. For instance, an application hosted by a network node in charge of producing vehicle attitude angles with respect to a reference system can be the consumer of pure sensor data produced by an inertial reference system and producer of the true attitude angles (usually after a coordinate transformation) to an attitude control system hosted by another network node.

The concept of channel frees producer and consumer applications from interfacing with lower levels of network protocol layers for transporting data from one node to another.

The next sections describe the functions that shall be made available to applications willing to use “channels” (and indirectly the new Data Link Layer protocol) for communicating with their counterparts within a network of embedded systems.

The behavior of each function is expressed using a pseudo-code inspired by the C Programming Language (ISO, 2011). The actual implementation shall be specific to each different software platform.

5.4.3.1 REGISTER

The purpose of the function REGISTER is to associate an operating system process (or equivalent task or thread) to a Service Access Point (SAP) pointed to by a Service Identification Block (SIB) accessed via the SIB$R_SIBB register.

The input argument is:

- `byte` – the SAP number to be registered for the caller process.

The output argument is:

- `SIB*` – the address of the SIB built for the caller process.

The return codes are:

- `SUCCESS` when successfully associating the process to the SAP;
- `NOTFOUND` when no matching SAP is found in the SIBs;
- `NOTFREE` when the SAP is already associated to another process.

The pseudo-code is:

```
int REGISTER(byte SAP,SIB* mySIB)
{
//get the address of SIB base register from CBB
 sib_r_sibb = cbb_r_ccbs[2];
//get the number of SIBs from SIBL
 sib_l_sibcnt = *sib_r_sibb;
//set SAP_FOUND to FALSE;
 SAP_FOUND = FALSE;
//search SIBL for matching SAP
 for(i=1; i<sib_l_sibcnt; i++)
 {
  sib = sib_r_sibb[i];
//check if SAP matches the one in the SIB
  if(sib[0] == (long)SAP)
  {
//check if SAP is already taken by other process
   if(sib[1] != 0) return NOTFREE;
//get process identification (operating system specific)
   SYS$GETPID(pid);
//set SIB$L_PID field to process ID
   sib[1] = pid;
```

```
//return the address of the SIB to the caller
   mySIB = sib;
   return SUCCESS;
  }
 }
 if(!SAP_FOUND) return NOTFOUND;
}
```

5.4.3.2 OPEN

The purpose of the function OPEN is to associate a process (or equivalent task or thread) registered as a Service Access Point (SAP) to a channel pointed to by a Channel Control Block (CCB) accessed via the CCB$R_CCBB register for either sending or receiving data. The address of the CCB is return to the caller process after checking the SAP against the SSAP for sending and the DSAP for receiving. The caller process identification is stored in the CCB to mark the channel status as opened.

The input arguments are:

- `byte*` – the address of the 4 characters channel identification;
- `int` – the desired access: `SEND` or `RECEIVE`.

The output argument is:

- `CCB*` – the address of the CCB built for the caller process.

The return codes are:

- `SUCCESS` when successfully associated the process to the channel;
- `NOTFOUND` when no matching channel is found in the CCBs;
- `UNKNOWN` when the access is neither `SEND` nor `RECEIVE`;
- `OPENED` when the channel has a process identification associated to it;
- `BADSSAP` when the SAP does not match the channel SSAP on `SEND`;
- `BASDSAP` when the SAP does not match the channel DSAP on `RECEIVE`.

The pseudo-code is:

```
int OPEN(byte* chanID,int access,CCB* myCCB)
{
//get the address of CCB base register from CBB
 ccb_r_ccbb = cbb_r_cbbs[3];
//get the number of CCBs from CCBL
 ccb_l_ccbcnt = *ccb_r_ccbb;
//set CHN_FOUND to FALSE;
 CHN_FOUND = FALSE;
//search CCBL for matching channel
 for(i=1; i<ccb_l_ccbcnt; i++)
 {
  ccb = ccb_r_ccbb[i];
//check if channel matches the channel in CCB
  if(ccb[0] == (long)chanID)
  {
   CHN_FOUND = TRUE;
//check if the channel in the CCB has a PID
   if(ccb[4] != 0) return OPENED;
//fetch SAP of the caller from SIB returned by REGISTER
   mySAP = mySIB[0];
//check SSAP for SEND or DSAP for RECEIVE against mySAP
   if(access == SEND)
   {
    if(ccb[1] == mySAP)
    {
//set the PID in CCB to the PID in the SIB of the caller
     ccb[4]= mySIB[1];
//return CCB to the caller
     myCCB = ccb;
     return SUCCESS;
    } else return BADSSAP;
   }else if(access == RECEIVE)
   {
    if(ccb[2] == mySAP)
    {
//set the PID in CCB to the PID in the SIB of the caller
     ccb[4]= mySIB[1];
//return CCB to the caller
     myCCB = ccb;
     return SUCCESS;
    } else return BADDSAP;
   }else return UNKNOWN;
  }
 }
 if(!CHN_FOUND) return NOTFOUND;
}
```

5.4.3.3 SEND

The purpose of the SEND function is to transmit data stored in a memory buffer over a channel previously opened by the caller process.

The input arguments are:

- `CCB*` – the address of the CCB to be used by the caller process.
- `byte*` – the address of the data buffer;
- `int` – the number of bytes to be sent.

The output argument is:

- `int` – the number of bytes actually sent.

The return codes are:

- `SUCCESS` when all data is successfully sent over the channel;
- `NOACCESS` when the PID of the caller process does not match the PID in the CCB passed as input argument;
- `BADSSAP` when the SAP of the caller process does not match the SSAP of the channel;
- `BADNUMBER` when the number of bytes passed as input argument is greater than the maximum number of bytes supported by the new Data Link Layer protocol (1488 bytes);
- `BADSEND` when the lower protocol layers return a number of bytes actually sent that is not equal to the number of bytes passed as input argument;
- `NOTOKENS` when the number of bytes to transmit violates the Traffic Shaping performed by the LCC layer on the channel.

The pseudo-code is:

```
int SEND(CCB* myCCB,byte* buffer,int nbytes,int xbytes)
{
//check if the PID in the CCB matches the PID of the caller
 SYS$GETPID(pid);
 if(pid != myCCB[4]) return NOACCESS;
//check SSAP against SAP from SIB returned by REGISTER
 if(myCCB[1] != mySIB[0]) return BADSSAP;
//check if the number of bytes is greater than MAXBYTES
 if(nbytes > MAXBYTES) return BADNUMBER
//call LLC for sending data (operating system specific)
 LLC$SEND(myCCB, buffer, nbytes, xbytes);
//check the number of bytes actually sent
 if(xbytes == nbytes) return SUCCESS;
// return NOTOKENS if Traffic Shaping violation
 if(xbytes < 0) return NOTOKENS;
 return BADSEND;
}
```

5.4.3.4 RECEIVE

The purpose of the RECEIVE function is to receive data over a channel previously opened by the caller process into a memory buffer.

The input arguments are:

- `CCB*` – the address of the CCB to be used by the caller process.
- `byte*` – the address of the data buffer;
- `int` – the number of bytes to be received.

The output argument is:

- `int` – the number of bytes actually received.

The return codes are:

- `SUCCESS` when all data is successfully received over the channel;
- `NOACCESS` when the PID of the caller process does not match the PID in the CCB passed as input argument;
- `BADDSAP` when the SAP of the caller process does not match the DSAP of the channel;

- `BADNUMBER` when the number of bytes passed as input argument is greater than the maximum number of bytes supported by the new Data Link Layer protocol (1488 bytes);
- `BADRECEIVE` when the lower protocol layers return a number of bytes actually received that is not equal to the number of bytes passed as input argument;
- `NOTOKENS` when the number of bytes to transmit violates the Traffic Policing performed by the LCC layer on the channel.

The pseudo-code is:

```
int RECEIVE(CCB* myCCB,byte* buffer,int nbytes,int xbytes)
{
//check if the PID in the CCB matches the PID of the caller
 SYS$GETPID(pid);
 if(pid != myCCB[4]) return NOACCESS;
//check DSAP against SAP from SIB returned by REGISTER
 if(myCCB[2] != mySIB[0]) return BADDSAP;
//check if the number of bytes is greater than MAXBYTES
 if(nbytes > MAXBYTES) return BADNUMBER
//call LLC for receiving data (operating system specific)
 LLC$RECEIVE(myCCB, buffer, nbytes, xbytes);
//check the number of bytes actually sent
 if(xbytes == nbytes) return SUCCESS;
// return NOTOKENS if Traffic Policing violation
 if(xbytes < 0) return NOTOKENS;
 return BAD RECEIVE;
}
```

5.4.3.5 STATUS

The purpose of the function STATUS is to return the current state of a Channel Control Block (CCB) for a particular channel.

The input arguments are:

- `byte*` – the address of the 4 characters channel identification.

The output arguments are:

- `byte` – the SSAP associated with the channel;
- `byte` – the DSAP associated with the channel;

- `long` – the Token Bucket capacity for the channel;
- `long` – the initial token count for the channel;
- `long` – the rate in tokens per second specified for the channel;
- `long` – the process identification currently owning the channel;
- `byte` – the Sequence Number of the last PDU transmitted or received over the channel;
- `long` – the Time-Stamp of the last PDU transmitted or received over the channel.

The return codes are:

- `SUCCESS` when successfully returned the contents of the CCB to the caller process;
- `NOTFOUND` when no matching channel is found in the CCBs;

The pseudo-code is:

```
int STATUS(byte* chanID,
           byte ssap,
           byte dsap,
           long capacity,
           long tokens,
           long rate,
           long pid,
           byte sn,
           long timestamp)
{
//get the address of CCB base register from CBB
 ccb_r_ccbb = cbb_r_cbbs[3];
//get the number of CCBs from CCBL
 ccb_l_ccbcnt = *ccb_r_ccbb;
//search CCBL for matching channel
 for(i=1; i<ccb_l_ccbcnt; i++)
 {
//check if channel matches the channel in CCB
  ccb = ccb_r_ccbb[i];
  if(ccb[0] == (long)chanID)
  {
   ssap = ccb[1];
   dsap = ccb[2];
   capacity = ccb[3];
   tokens = ccb[4];
```

```
   rate = ccb[5];
   pid = ccb[6];
   sn  = ccb[7];
   timestamp = ccb[8];
   return SUCCESS;
  }
 }
 return NOTFOUND;
}
```

5.4.3.6 CLOSE

The purpose of the function CLOSE is to remove the association of a process (or equivalent task or thread) to a channel pointed to by a Channel Control Block (CCB). The process identification is stored in the CCB is cleared to mark the channel status as closed.

The input arguments are:

- `CCB*` – the address of the CCB built for the caller process.

There are not explicit output arguments.

The return codes are:

- `SUCCESS` when successfully removed the association of process to the channel;
- `NOTFOUND` when no matching CCB is found in the CCBL;
- `NOACCESS` when the PID of the caller process does not match the PID in the CCB passed as input argument.

The pseudo-code is:

```
int CLOSE(CCB* myCCB)
{
//get the address of CCB base register from CBB
 ccb_r_ccbb = cbb_r_cbbs[3];
//get the number of CCBs from CCBL
 ccb_l_ccbcnt = *ccb_r_ccbb;
//set CCB_FOUND to FALSE;
//search CCBL for matching channel
 for(i=1; i<ccb_l_ccbcnt; i++)
 {
  if(ccb_r_ccbb[i] == myCCB)
  {
   ccb = ccb_r_ccbb[i];
   break;
```

```
  }
  else return NOTFOUND
 }
//check if the CCB has a matching PID
 SYS$GETPID(pid);
 if(myCCB[4] != pid) return NOACCESS;
//clear the PID in the CCB
 myCCB[4] = 0;
 return SUCCESS;
}
```

5.4.3.7 UNREGISTER

The purpose of the function UNREGISTER is to remove the association of a process (or equivalent task or thread) to a Service Identification Block (SIB) accessed via the SIB$R_SIBB register. The process identification stored in the SIB is cleared to free the SAP to another process.

The input argument is:

- `SIB*` – the address of the SIB built for the caller process.

There are not explicit output arguments.

The return codes are:

- `SUCCESS` when successfully removed the association of the process with the SAP pointed by the SIB;
- `NOTFOUND` when no matching SIB is found in the SIBL;
- `NOACCESS` when the PID of the caller process does not match the PID in the SIB passed as input argument.

The pseudo-code is:

```
int UNREGISTER(SIB* mySIB)
{
//get the address of SIB base register from CBB
 sib_r_sibb = cbb_r_ccbs[2];
//get the number of SIBs from SIBL
 sib_l_sibcnt = *sib_r_sibb;
//set SAP_FOUND to FALSE;
 SAP_FOUND = FALSE;
//search SIBL for matching SAP
 for(i=1; i<sib_l_sibcnt; i++)
 {
```

```
//check if SIB matches the one in the SIBL
  if(sib_r_sibb[i] == mySIB)
  {
   sib = sib_r_sibb[i];
   break;
  }
  else return NOTFOUND;
//check if the CCB has a matching PID
  SYS$GETPID(pid);
  if(mySIB[4] != pid) return NOACCESS;
//clear the PID in the SIB
 mySIB[1] = 0;
 return SUCCESS;
}
```

5.4.3.8 Operating system specific functions

The functions that build the Channel Application Programming Interface depend on lower level functions performed either natively by the operating system or added on by the LLC and MAC network layers.

One of the operating system function required by the functions involved in implementing the concept of channel retrieves the process (or equivalent task or thread) identification assigned to it by the operating system at its creation. This function is referred to in the previous sections as SYS$GETPID.

Two other important operating system functions are required by LLC$SEND and referred to as SYS$GETMICS, which retrieves the count of microseconds passed the second, and as SYS$GETSECS, which retrieves the number of seconds passed after midnight.

Two lower level LLC layer functions are referred to in the previous sections as LLC$SEND and LLC$RECEIVE. It is important to note that these functions handle only the transmission and reception of UI PDUs.

It is beyond the scope of this specification to write pseudo-code for these functions, for they highly depend on the operating system architecture, but they shall behave as follows.

```
LLC$SEND(myCCB, buffer, nbytes, xbytes)
// myCCB  – address of the Channel Control Block
// buffer – address of the data buffer
// nbytes – number of bytes to be transmitted
// xbytes – number of bytes actually transmitted
```

- perform Traffic Shaping on the channel and return error if the bucket does not have enough tokens to transmit all the data (`nbytes` plus the LLC overhead);
- get the DSAP and SSAP numbers from the CCB;
- get the Sequence Number (SN) from the CCB (may be 0 after system power-on or system reset), increment it by 1 or reset 1 to if the current SN is 255;
- build the 20-bit Time-Stamp by calling SYS$GETMICS if SN is not 0 or by calling SYS$GETSECS if SN is 0;
- build the UI PDU with the DSAP and SSAP, the Control field set to UI, the new SN, the fresh Time-Stamp and copying the application data;
- complete the UI PDU calculating the payload and header CRCs;
- call the MAC layer for transmitting the UI PDU;
- wait until the MAC layer completes transmission and returns the number of bytes actually transmitted;
- update the SN and Time-Stamp in the CCB of the caller;
- return the number of bytes actually transmitted to caller.

In turn, the MAC layer shall:

- use the DSAP saved in the CCB to search the Service Hosting Block list (SHBL) for a SHB with a matching SAP;
- get the Network Identification Block (NIB) of the node hosting the DSAP from the next position in the SHB;
- get the MAC address of the destination node the from the NIB;
- get the base of the MAC address of the host node from the Network Identification Block List (NIBL);
- if the MAC address is unicast, search the Port Assignment Block List (PABL) for all Port Assignment Blocks (PABs) that have a NIB matching the NIB of the destination node and save the Port numbers from the

PABs found to complete the MAC Destination address; if the MAC address is multicast, proceed to the next step

- complete the IEEE 802.3 data packets with the Length, Preamble and Start-of-Frame Delimiter, copying the UI PDU and calculating the Frame Check Sequence (FCS);
- call the PHY layer for transmitting the IEEE 802.3 data packets over the saved Port numbers if the Destination MAC address is unicast or over all ports if the Destination MAC address is multicast;
- wait until the PHY layer completes transmission and returns the number of bytes actually transmitted;
- return to LLC layer.

```
LLC$RECEIVE(myCCB, buffer, nbytes, xbytes)
// myCCB  – address of the Channel Control Block
// buffer – address of the data buffer
// nbytes – number of bytes to be received
// xbytes – number of bytes actually received
```

- use the SSAP saved in the CCB to search the Service Hosting Block list (SHBL) for a SHB with a matching SAP;
- get the Network Identification Block (NIB) of the node hosting the SSAP from the next position in the SHB;
- get the Source MAC base address of the source node the from the NIB;
- get the Destination MAC base address of the host node from the Network Identification Block List (NIBL);
- call the MAC layer for receiving data packets with the Source MAC and Destination MAC;
- wait until the MAC layer completes reception;
- check IEEE 802.2 extended header and payload CRC;
- compare the SN of the UI PDU received with the SN saved into the caller CCB;

- if the SN of the received UI PDU is less or equal to the saved SN, discard the packet and return error, otherwise proceed to next step;
- perform Traffic Policing on the channel and return error if the bucket does not have enough tokens to receive all the data (the complete UI PDU);
- copy the payload of the UI PDU into the buffer of the caller;
- save the SN and Time-Stamp from the IEEE 802.2 extended header into the caller CCB;
- return the number of bytes actually received to the caller.

In turn, the MAC layer shall:

- call the PHY layer for receiving IEEE 802.3 data packets on all physical network ports;
- wait until the PHY layer completes reception and returns the number of bytes actually received;
- check the FCS of the received IEEE 802.3 data packet;
- check the MAC Destination address of the IEEE 802.3 received data against the saved MAC host node base address;
- check the MAC Source address of the IEEE 802.3 received data against the saved MAC source node base address;
- save the Port field from the Source MAC address for possible further use;
- extract the UI PDU from the IEEE 802.3 received data packet;
- deduct the number of bytes of the IEEE 802.3 header and footer from the number of bytes received from the PHY layer;
- return to LLC layer.

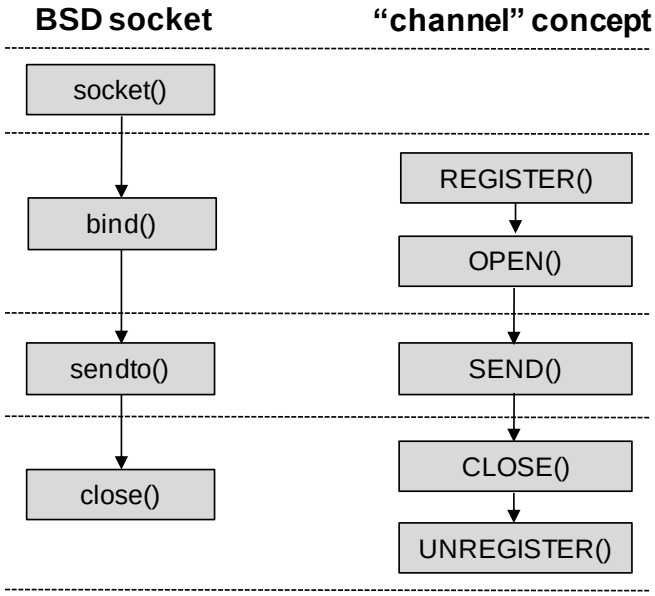
5.4.3.9 The “channel” concept and the BSD socket interface

The abstract construct called “socket” was introduced by the Berkeley Software Distribution (BSD) version 4.2 of the UNIX operating system as network connection end-point for sending and receiving data (COMER, 1995).

The communication model used with sockets between two network nodes is the “client-server” (FREEBSD, 2020), whereby the client sends and the server receives data using a digital communication protocol, most frequently the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP) over the Internet Protocol (IP). Under the perspective of the “channel” concept, a client plays the role of a data producer and a server plays the role of a data consumer.

Figure 5.25 places side-by-side the socket programming model for using UDP over IP and the channel Application Programming Interface (cAPI) for a typical client or data producer application.

Figure 5.25 – BSD socket and the “channel” API for a client application.



The *socket*() system call associates the socket with an “address family”, AF_INET for instance for IP, and a protocol, SOCK_DGRAM for instance for UDP. The cAPI does not require such operation, once the underlying protocol is completely hidden to the application.

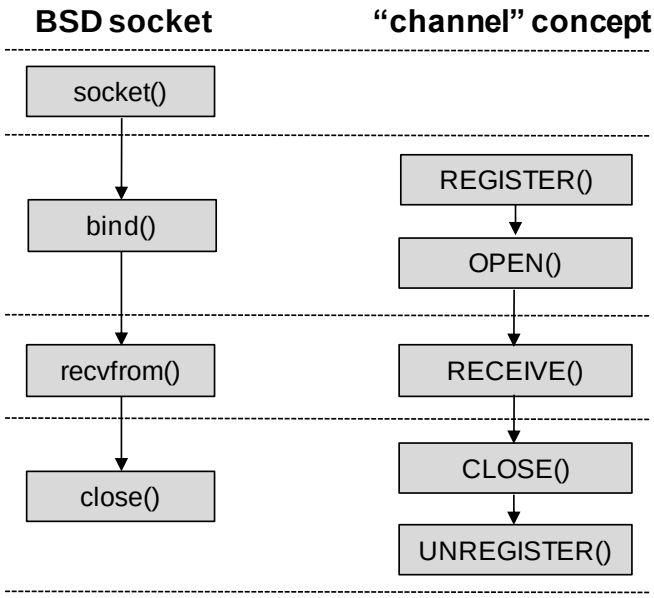
The *bind*() system call associates the client's IP address and UDP port number to the socket. The cAPI requires that the process (or task) registers itself to a Service Access Point (SAP) and opens a channel for sending. This last operation checks whether the SAP number registered to the process is a Source Service Access Point (SSAP) in the Channel Control Block (CCB) structure.

The *sendto*() system call requires the IP address and UDP port number of the server while the cAPI requires the address of the CCB structure besides the address and size of the data buffer.

The *close*() terminates the socket, while the cAPI requires the process closing the channel and unregistering itself from the SAP.

Figure 5.26 places side-by-side the socket programming model for using UDP over IP and the cAPI for a typical server or data consumer application.

Figure 5.26 – BSD socket and the "channel" API for a server application.



Conversely, the *bind*() system call associates the server's IP address and UDP port number to the socket. The cAPI requires that the process (or task) registers itself to a Service Access Point (SAP) and opens a channel for receiving. This last operation checks whether the SAP number registered to the process is a Destination Service Access Point (DSAP) in the Channel Control Block (CCB) structure.

For the server application, it is usual to call *recvfrom*() specifying an UDP port number leaving the socket open for receiving UDP datagrams from any IP address and the address and size of the data buffer. The cAPI requires the address of the CCB structure besides the address and size of the data buffer.

The socket interface requires previous knowledge of the IP addresses and UDP port numbers for both client and server applications.

The cAPI relies on structures that are built by the operating system from a configuration file, that is, a process is not free to choose registering to a SAP or opening a channel other than those previously configured by the system integrator.

5.4.4 Route testing programming model

The test for valid network routes supported by the introduction of the Hop Count (HC) field in the extended header of TEST PDUs shall also depend on a lower-level operating system routine that becomes part of the implementation of the new Data Link Layer protocol.

This routine highly depends on the operating system architecture, as it is the case for all routines handling input and output to a physical device, but its behavior shall be as follows:

```
LLC$TEST(nodename, portnumber, status)
// nodename   – address of the buffer containing the name
//              of the destination node
// portnumber – number of the physical port for which
//              the route will be tested
// status     – VALID/INVALID boolean for the route
//              to the destination node
```

- search the Node Identification Block List (NIBL) for a NIB with a matching the destination node name and save the NIB;
- search the Port Assignment Block list (PABL) for a PAB with a matching port number;
- search the PAB found for a matching NIB and save the Hop Count (HC) to the destination node;
- build the IEEE 802.2 PDU with the DSAP and SSAP set to 1, the Control field set for TEST PDU and the HC field;

- call the MAC layer for transmitting the TEST PDU;
- wait until the MAC layer completes transmission;
- call the MAC layer for receiving a TEST PDU;
- wait until the MAC layer completes reception of a TEST PDU;
- check HC and return VALID if HC is set to 0xC0 hexadecimal or return INVALID otherwise.

For transmitting a TEST PDU, the MAC layer shall:

- get the MAC address of the destination node the from the NIB;
- get the base of the MAC address of the host node from the Network Identification Block List (NIBL); the only part missing is the Port;
- use the Port from the PAB found to complete MAC Source address.
- complete the IEEE 802.3 data packet with the Length, Preamble and Start-of-Frame Delimiter, copying the TEST PDU and calculating the Frame Check Sequence (FCS);
- save the MAC Source and the MAC Destination base addresses;
- call the PHY layer for transmitting the IEEE 802.3 data packet;
- wait until the PHY layer completes transmission;
- return to LLC layer.

For receiving a TEST PDU, the MAC layer shall:

- call the PHY layer for receiving IEEE 802.3 data packets;
- wait until the PHY layer completes reception;
- check the FCS of the received IEEE 802.3 data packet;
- check the MAC Source address of the IEEE 802.3 received data against the saved MAC destination node base address;
- save the Port field from the Source MAC address for possible further use;

- check the MAC Destination address of the IEEE 802.3 received data against the saved MAC host node base address;
- check the IEEE 802.2 Control field for a TEST PDU;
- extract the TEST PDU from the IEEE 802.3 received data packet;
- return to LLC layer.

The MAC layer behavior on transmitting and receiving TEST PDUs shall be different from its behavior on transmitting and receiving UI PDUs in the sense that the LLC$TEST service requires the transmission and the reception of a TEST PDU in the same operation.

Testing network routes shall be performed in an orderly fashion by each network node as a mandatory step for its integration to an embedded network. It shall be considered “good practice” to let one and only one node at a time to validate routes to other nodes. The transmission and reception of TEST PDUs in the process of validating network routes shall not take more than a couple of milliseconds on a typical Ethernet physical medium.

5.4.5 UI and TEST PDU routing programming model

One important task that some of the nodes in a network need to perform, in particular those following the “point-to-point” topology, is to route IEEE 802.3 data packets containing either a TEST or an UI PDU if the node recognizes itself not being the end recipient by checking the Destination MAC address of a received data packet.

If the Destination MAC address does not correspond to the base MAC address of the node, the IEEE 802.3 data packet has to be retransmitted to the network using the proper physical port, as prescribed by the network configuration assigned to the node.

The process of recognizing the need of retransmitting a data packet must be as efficient as possible and shall consume only minimum computing resources of the software and hardware platforms.

The behavior of the MAC layer in this scenario shall be as follows for either a TEST or UI PDU:

- save the address of the NIB pointing to the host node in order to avoid searching for it every time a new data packet is received;
- compare the Destination MAC address with the host node MAC base address from the host NIB;
- search the Network Identification Block List (NIBL) for a NIB that has a base MAC address matching the Destination MAC address;
- check the Control field of the IEEE 802.2 PDU and proceed to the one of the next steps:
    - if the PDU is UI:
        - search the Port Assignment Block List (PABL) for one or more Port Assignment Blocks (PABs) that have a NIB matching the NIB of the destination node and save the Port numbers from the PABs;
        - call the PHY layer for transmitting the data packet over the saved Port numbers.
    - if the PDU is TEST and HC is not equal to 0xC0:
        - search the NIB List for a NIB having a matching base Source MAC address and report to the proper error handling software layer if none found;
        - save the Port number returned from the PHY layer over which the TEST PDU was received together with the base Source MAC address;
        - search the Port Assignment Block List (PABL) for one or more Port Assignment Blocks (PABs) that have a NIB matching the NIB of the destination node;
        - the HC field in the TEST PDU shall be 1 unit greater than the HC stored in the PAB for the NIB, otherwise report it to the proper error handling software layer;
        - subtract 1 unit from the HC field in the TEST PDU

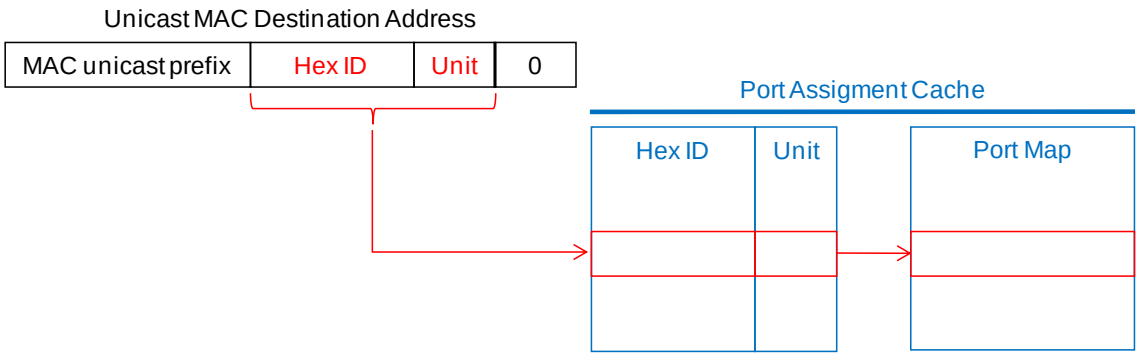
        - call the PHY layer for transmitting the data packet over the saved Port numbers.
    - If the PDU is TEST and HC is equal to 0xC0:
        - search the saved Port numbers and associated base Source MAC addresses for a matching MAC address and report to the proper error handling software layer if none found;
        - call the PHY layer for transmitting the data packet over the saved Port number.

Since this operation is likely to be repeated many times during the network operation, it is convenient to provide a cache memory for quickly retrieving the Port numbers for a particular Destination MAC address.

This caching can be done by simply associating the Hex ID and Unit fields of MAC addresses directly to one or more Port numbers as data packets are received and routed by the node. This cache memory shall be looked up first, for a likely match is expected to occur.

This cache memory layout for the purpose of routing is suggested in Figure 5.27.

Figure 5.27 – Cache layout for data packet routing.

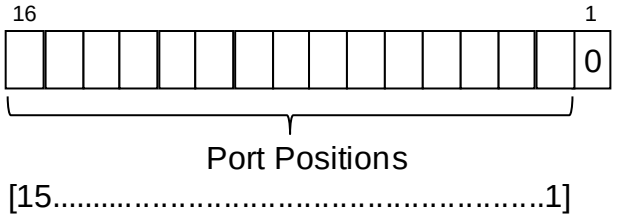


The “Port Map” is built as follows (see Figure 5.28):

- bit position 1 – must be clear (represents Port number 0 reserved for multicast transmissions);

- bit positions 2 to 16 – the Port numbers to be used for transmission represented by a bit set at the position equal to 1 plus the Port number (more than one bit can be set).

Figure 5.28 – Port Map for caching Destination MAC to port number.



The Port Assignment Cache shall be populated in two ways:

- by extracting the Hex ID and Unit fields of the Destination MAC address of the packet received after validating against the PABs, generating the Port Map with the Port numbers saved from the PABs and inserting a row in the cache;
- by extracting the Hex ID and Unit fields of the Source MAC address of the packet received, validating against the NIBs, generating the Port Map with the Port number returned from the PHY layer and inserting a row in the cache.

The operation of populating the cache will cost extra computing resources until all IEEE 802.3 data packets expected to cross the node have been received at least once.

Therefore, testing the valid routes using TEST PDUs as described in the previous sections shall produce this exact effect for the benefit of the overall performance of the network after startup.

5.4.6 Network traffic switching

On network “star” or “multi-star” topology the presence of one or more network traffic switching devices is essential.

They are specially designed to quickly route traffic from one switch port to another port or ports (when multicasting). The high-end products can be very sophisticated and capable of performing network traffic classification up to the

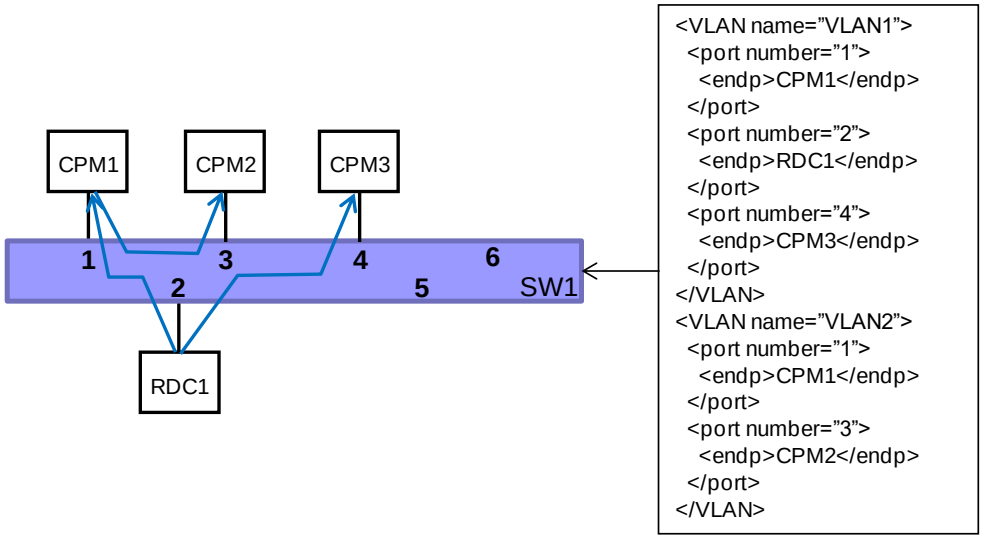
Layer 3 of the ISO-OSI model, traffic shaping and policing, and segregating broadcast domains by the means of “Virtual LANs”, or VLANs (SEIFERT, 2000).

Traffic switches shall be programmed as part of the network infrastructure. Although they do not run typical application code, their configuration shall be consistent with the configuration of the remaining network nodes.

For instance, the network illustrated in Figure 5.13 shows a switch SW1 routing traffic from node RDC1 to nodes CPM1 and CPM3 and traffic from node CPM1 to node CPM2. The XML text in the same figure illustrates how SW1 shall be configured with node CPM1 on port 1, RDC1 on port 2, CPM2 on port 3 and CPM3 on port 4.

However, a more appropriate XML text for programming a switch should define two VLANs, one for nodes RDC1, CPM1 and CPM3 and a second for nodes CPM1 and CPM2, as illustrated in Figure 5.29.

Figure 5.29 – VLAN programming example for the network in Figure 5.13.



The port 1 of the switch SW1 is called “multi-VLAN port”, once it is part of two different VLANs (CISCO, 2014).

The suggested XML text could be used to generate a switch configuration file following the format required by the device manufacturer.

6 EXPERIMENTAL RESULTS USING THE CONCEPT OF “CHANNEL”

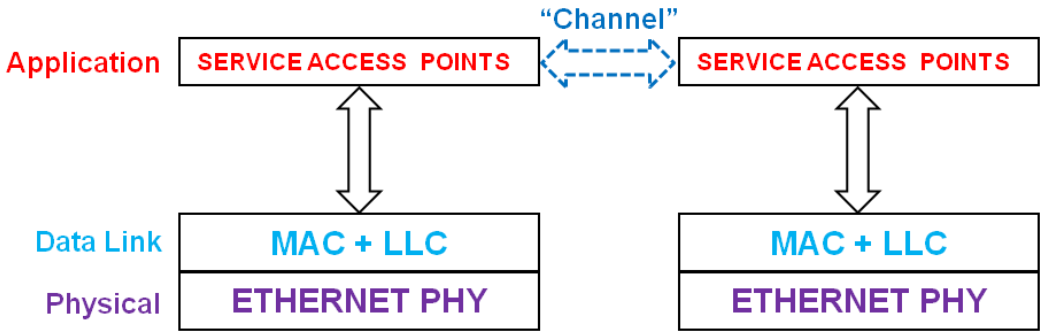
6.1 Introduction

Validating all the aspects involved in the introduction of a new network protocol involves resources and time that exceed the scope of this work.

Preparing and executing tests and validating test results in general add significant cost to any new system development program. Functional requirements have to be extracted from the protocol specification, representative software and hardware platform needs to be put in place and used to perform all the test cases created for validating the requirements.

However, validating the concept of “channel” introduced with the new Data Link Layer protocol is feasible under certain circumstances within the scope of this work. The channel is a new virtual entity that does not have similarity with other equivalent communication resource in current existing network protocol programming support. Therefore validating its use is an important contribution to the completeness of this work. A channel virtually connects two entities executing at the Application Layer, as illustrated in Figure 6.1 below.

Figure 6.1 – The “channel”: a virtual connection at the Application Layer.



6.2 Scenario for the test case

For the purpose, a simple “producer-consumer” scenario will be created with following objectives:

- Validate the in-memory structures described in Section 5.4.2;
- Validate the programming interface described in Section 5.4.3;

- Validate the format of the data frame built as described in Section 5.1;
- Validate the implementation of Traffic Shaping at the “producer” node and Traffic Policing at the “consumer” node as described in Sections 5.2.3 and 5.2.4.

A Data Link Layer protocol is not responsible for actually transmitting a data frame over a network wire. However, it is important to validate the integrity of the data frame to be passed to the Physical Layer for transmission. For that, an open-source software tool will be used: a Wireshark Generic Dissector (WSDG, 2020).

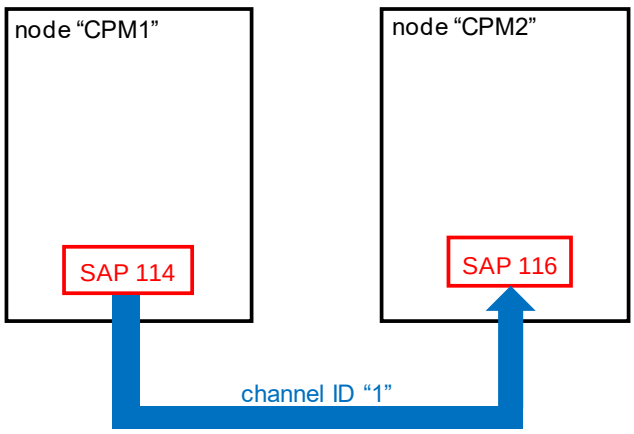
A piece of software emulating the lower-level system routine responsible for passing a data frame from the Data Link Layer to the Physical Layer (LLC$SEND described in Section 5.4.3.8) will write records to a disk file following the Wireshark Packet CAPture (PCAP) format. The counterpart emulating the receiving side (LLC$RECEIVE) will read the records and pass it to the Data Link Layer. The Wireshark tool itself will be used for validating the data frames written to the PCAP disk file.

The next sections describe the simple “producer-consumer” scenario: the network topology, the configuration of the two nodes involved; the configuration of the “channel” used to transmit and receive data, and the test case itself.

6.3 Network topology

The simple point-to-point network topology connecting nodes CPM1 and CPM2 is illustrated in Figure 6.2.

Figure 6.2 – Simple network topology for validating the “channel” concept.



6.4 Network nodes configuration

The network used for this demonstration has only two nodes named “CPM1” and “CPM2” with following configurations:

Node CPM1

- Equipment Hex ID: 0x341 (Satellite Attitude Control Unit - ACU)
- Unit number: 1
- Number of network ports: 1
- Host of SAP: 114 (0x72)

Node CPM2

- Equipment Hex ID: 0x341 (Satellite ACU)
- Unit number: 2
- Number of network ports: 1
- Host of SAP: 116 (0x74)

6.5 Channel configuration

There will be only one channel available for communication between nodes CPM1 and CPM2 with following configuration, assuming 1 token per byte for applying the Token Bucket algorithm:

- Channel ID: 1
- SSAP: 114 (0x72)
- DSAP: 116 (0x74)
- Capacity: 90 tokens (limit)
- Tokens: 60 tokens (initial)
- Rate: 30 tokens/second

6.6 Test case description

6.6.1 Role of node CPM1

The test case devised for experimenting with the concept of “channel” works as follows for CPM1:

- call REGISTER for registering itself to SAP 114 (0x72);
- call OPEN for opening channel 1;
- call SEND for sending a 34 byte message over channel 1;
- wait 3 seconds;
- call SEND for sending a second 34 byte message over channel 1;
- wait 1 second;
- call SEND for sending a third 34 byte message over channel 1;
- call STATUS for displaying the status of channel 1 after transmitting three messages;
- call CLOSE for closing the channel 1;
- call UNREGISTER for unregistering itself from SAP 114.

The expected results are as follows, taking into account the UI PDU format including its extended header, as described in Section 6.1, and the behavior of the Traffic Shaping at the transmitting end, as described in Section 6.2.3:

- CPM1 shall successfully register itself as SAP 114;
- CPM1 shall successfully open channel 1;
- CPM1 shall successfully send messages over channel 1 because its registered SAP matches the SSAP of channel 1;
- three records of length 60 bytes (minimum IEEE 802.3 packet length excluding 4 bytes of the Frame Check Sequence) shall be written to the PCAP disk file;
- all three SEND operations shall conclude successfully, that is, no operating shall be blocked by Traffic Shaping;
- the first record shall have the Sequence Number (SN) set to 0 and its Time-Stamp field shall be set to the correct number of seconds passed after midnight;

- the second record shall have the Sequence Number (SN) set to 1 and its Time-Stamp field shall be set to the correct number of microseconds passed after the second;
- the third record shall have the Sequence Number (SN) set to 2 and its Time-Stamp field shall be set to the correct number of microseconds passed after the second.

It is important to note that passing through Traffic Shaping validates the correct application of the Token Bucket algorithm at the transmitting end for channel 1 because of following reasons:

- the capacity of the Token Bucket is set to 90 and its initial content is set to 60;
- the first transmission has the cost of 60 tokens (equal to the size in bytes of the IEEE 802.3 data packet excluding 4 bytes of the Frame Check Sequence);
- after the first transmission, the balance of the Token Bucket is 60 – 60 = 0;
- after passing 3 seconds, the balance of the Token Bucket is 0 + 30x3 = 90, the maximum value equal to the Token Bucket capacity;
- the second transmission has the cost of 60 tokens (same packet length as the first);
- after the second transmission, the balance of the Token Bucket is 90 – 60 = 30;
- after passing 1 second, the balance of the Token Bucket is 30 + 30x1 = 60;
- the third transmission has the cost of 60 tokens (same packet length as the first);
- since the balance of the Token Bucket is 60, the third transmission is allowed to pass, even after passing only 1 second after the previous transmission.

If the capacity of channel 1 had been set to 60 (instead of 90), CPM1 would need to wait at least 2 seconds before transmitting the third 34-byte long message.

The choice of the “channel” Rate parameter in the order of tens of tokens and time intervals in the order of seconds is justified for two reasons: 1) the CPM1 and CPM2 applications depend of suspending themselves for a period of time and the *sleep*() system call used in this particular implementation does not guarantee millisecond granularity in its operation; 2) neither code was expected to run using real-time scheduling priority, for it was developed operating system independent and as a simple console application.

6.6.2 Role of node CPM2

The test case devised for experimenting with the concept of “channel” works as follows for CPM2:

- call REGISTER for registering itself to SAP 116 (0x74);
- call OPEN for opening channel 1;
- call RECEIVE for receiving a 34 byte message over channel 1;
- the RECEIVE operation shall complete immediately;
- call RECEIVE for receiving a second 34 byte message over channel 1;
- the RECEIVE operation shall complete after approximately 3 seconds;
- call RECEIVE for receiving a third 34 byte message over channel 1;
- the RECEIVE operation shall complete after approximately 1 second;
- call STATUS for displaying the status of channel 1 after transmitting three messages;
- call CLOSE for closing the channel 1;
- call UNREGISTER for unregistering itself from SAP 116.

The timing information in each PCAP record header will be used to mimic the time interval spent by CPM2 while waiting for CPM1 to transmit the second and third message, 3 seconds and 1 second respectively.

The expected results are as follows, taking into account the UI PDU format, including its extended header, as described in Section 6.1 and the behavior of the Traffic Policing at the receiving end, as described in Section 6.2.4:

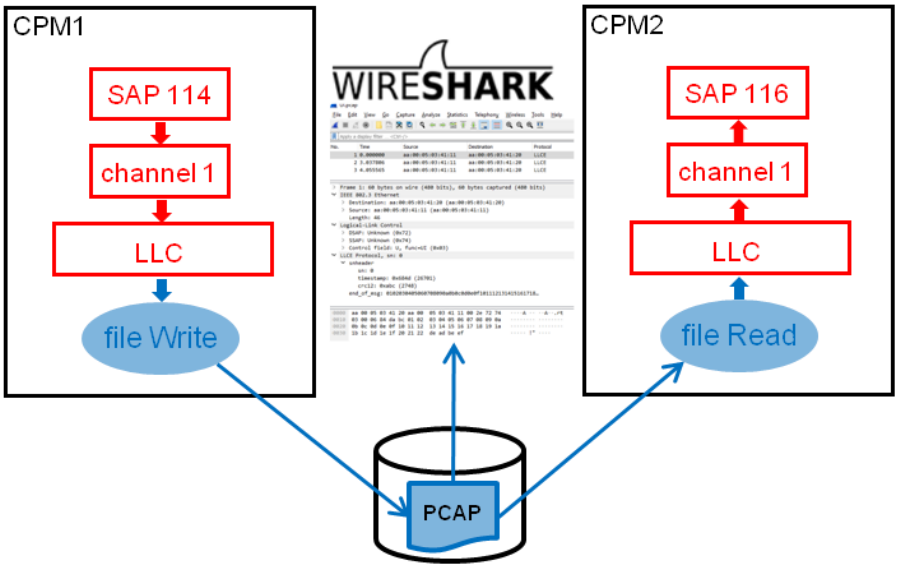
- CPM2 shall successfully register itself as SAP 116;
- CPM2 shall successfully open channel 1;
- CPM2 shall successfully receive messages over channel 1 because its registered SAP matches the DSAP of channel 1;
- three records of length 60 bytes (minimum IEEE 802.3 packet length excluding 4 bytes of the Frame Check Sequence) shall be read from the PCAP disk file;
- all three SEND operations shall conclude successfully, that is, no operating shall be blocked by Traffic Policing;
- the first record shall have the Sequence Number (SN) set to 0 and its Time-Stamp field shall be used to estimate the clock offset between the transmitting node and the receiving node;
- the second record shall have the Sequence Number (SN) set to 1 and its Time-Stamp field shall be used to estimate the time passed since previous transmission;
- the third record shall have the Sequence Number (SN) set to 2 and its Time-Stamp field shall be used to estimate the time passed since previous transmission.

It is important to note that passing through Traffic Policing validates the correct application of the Token Bucket algorithm at the receiving end for channel 1 for the same reasons detailed in previous section for node CPM1.

6.6.3 Test case illustrated

The test case devised for validating the concept of “channel” is illustrated in Figure 6.3:

Figure 6.3 – Test case for validating the “channel” concept.



6.6.4 Configurarion files

The Figure 6.4 shows the XML configuration files for nodes CPM1 and CPM2, whose values were used for fulfilling the in-memory data structures for the test case:

Figure 6.4 – XML configuration files for nodes CPM1 (left) and CPM2 (right).

```
<?xml version="1.0"?>
- <configuration host="CPM1" name="TEST001">
  - <nodes>
    - <node name="CPM1">
        <hexid>341</hexid>
        <unit>1</unit>
      </node>
    - <node name="CPM2">
        <hexid>341</hexid>
        <unit>2</unit>
      </node>
  </nodes>
  - <SAPs>
      <number>114</number>
  </SAPs>
  - <channels>
    - <channel id="1">
        <SSAP>114</SSAP>
        <DSAP>116</DSAP>
        <capacity>90</capacity>
        <tokens>60</tokens>
        <rate>30</rate>
      </channel>
  </channels>
  - <services>
    - <SAP number="116">
        <host>CPM2</host>
      </SAP>
  </services>
  - <ports>
    - <port number="1">
        <endp hops="0">CPM2</endp>
      </port>
  </ports>
</configuration>
```

```
<?xml version="1.0"?>
- <configuration host="CPM2" name="TEST001">
  - <nodes>
    - <node name="CPM2">
        <hexid>341</hexid>
        <unit>2</unit>
      </node>
    - <node name="CPM1">
        <hexid>341</hexid>
        <unit>1</unit>
      </node>
  </nodes>
  - <SAPs>
      <number>116</number>
  </SAPs>
  - <channels>
    - <channel id="1">
        <SSAP>114</SSAP>
        <DSAP>116</DSAP>
        <capacity>90</capacity>
        <tokens>60</tokens>
        <rate>30</rate>
      </channel>
  </channels>
  - <services>
    - <SAP number="114">
        <host>CPM1</host>
      </SAP>
  </services>
  - <ports>
    - <port number="1">
        <endp hops="0">CPM1</endp>
      </port>
  </ports>
</configuration>
```

6.6.5 Implementation details

The applications emulating the behavior of nodes CPM1 and CPM2 were developed in C language as console applications for the Microsoft Windows 10™ operating system using an Intel i3-7100™ CPU (@3.90GHz) with 8 gigabytes of memory.

The code for implementing the Token Bucket algorithm in Traffic Shaping for the SEND operation and in Traffic Policing for the RECEIVE operation was adapted from Thomas (2007). The code used for calculating the CRC12 for the UI PDU extended header and the extra CRC32 for the UI PDU payload was adapted from Reifegerste (2003).

Firstly the CPM1 Application was run, then an arbitrary number of seconds later the CPM2 application was run.

This test procedure has direct effect of the outputs provided by the applications, which are shown and commented in the next sections.

6.6.6 CPM1 application source code (extract)

The text below is an extract of the source code for the CPM1 application (the full text is provided in Appendix E). The function calls to the “channel” services are highlighted:

```
int main()
{
// calling "initialize()" not needed in real life..
   retcode = initialize();

   printf("\n\n>> Entering main()...");

   printf("\n>> Registering CPM1 to SAP %d", cpm1_SAP);

   retcode = REGISTER(cpm1_SAP, &cpm1_SIB);

   if(retcode == SUCCESS)
   {
      printf("\n\n>> CPM1 SAP %d is now registered to PID = 0x%x",
      cpm1_SIB->sap,
      cpm1_SIB->pid);
   } else
   {
      printf("\n>> bad code 0x%x", retcode);
   }

   printf("\n>> Opening channel ID %d for access 0x%x", cpm1_chn, cpm1_acc);

   retcode = OPEN(cpm1_chn, cpm1_acc, &cpm1_CCB);

   if(retcode == SUCCESS)
   {
      printf("\n\n>> Channel ID %d is now open",
```

```
   cpm1_CCB->chnid);
} else
{
   printf("\n>> bad code 0x%x", retcode);
}

for(i = 0; i<MINBYTES; i++)
{
   message[i] = i+1;
}

msize = strlen(message);
length = (int)msize;

printf("\n\n>> Sending message with %d bytes", length);
retcode = SEND(&cpm1_CCB, message, length, &xmited);

printf("\n   return code 0x%x, tokens = %d", retcode, cpm1_CCB->tokens);

sleep(3);

printf("\n\n>> Sending message with %d bytes", length);
retcode = SEND(&cpm1_CCB, message, length, &xmited);

printf("\n   return code 0x%x, tokens = %d", retcode, cpm1_CCB->tokens);

sleep(1);

printf("\n\n>> Sending message with %d bytes", length);
retcode = SEND(&cpm1_CCB, message, length, &xmited);

printf("\n   return code 0x%x, tokens = %d", retcode, cpm1_CCB->tokens);

retcode = STATUS(cpm1_chn, &chn_ssap, &chn_dsap, &chn_capacity, &chn_tokens,
                 &chn_rate, &chn_pid, &chn_sn, &chn_timestmp);

printf("\n\n>> Status of channel ID = %d, SSAP = %d, DSAP = %d,
                Token Bucket capacity/tokens/rate [in bytes/sec] = %d/%d/%d,
                PID = %x, SN = %d, Time-stamp = %d",
   cpm1_chn,
   chn_ssap,
   chn_dsap,
   chn_capacity,
   chn_tokens,
   chn_rate,
   chn_pid,
   chn_sn,
   chn_timestmp);

printf("\n\n>> Closing channel ID %d", cpm1_chn);

retcode = CLOSE(&cpm1_CCB);

if(retcode == SUCCESS)
{
   printf("\n\n>> Channel ID %d is now closed",
   cpm1_CCB->chnid);
} else
{
   printf("\n>> bad code 0x%x", retcode);
}

printf("\n\n>> Unregistering SAP %d from CPM1", cpm1_SAP);

retcode = UNREGISTER(&cpm1_SIB);

if(retcode == SUCCESS)
{
```

```
      printf("\n\n>> CPM1 SAP %d is now unregistered (PID = 0x%x)",
      cpm1_SIB->sap,
      cpm1_SIB->pid);
   } else
   {
      printf("\n>> bad code 0x%x", retcode);
   }

   return 0;
}
```

The call to the function `initialize()` is not needed in a production code, but it is necessary to the CPM1 application for initializing the in-memory data structures (see Section 5.4.2) specified for the test case.

6.6.7 CPM1 application output commented

The text output of the CPM1 application is shown in the next sections followed by comments explaining how it validates the expected results listed in the previous Section 6.4.1.

6.6.7.1 Initialization

```
>> Filling NIBs and CIB...
>> Number of nodes 2
>> Node name 'CPM1    ' Equipment ID = 0x341  Unit = 1 MAC Base = aa000503
                                                                  41100000
>> Node name 'CPM2    ' Equipment ID = 0x341  Unit = 2 MAC Base = aa000503
                                                                  41200000
>> Configuration name 'TEST001' Host NIB address = 62fcc0
   Host name from NIB List 'CPM1    ' Host name from CIB 'CPM1    '
>> Filling CCBs and SIBs...
>> Number of channels 1
>> Channel position 0, ID = 1, SSAP = 114, DSAP = 116,
   Token Bucket capacity/tokens/rate [in bytes/sec] = 90/60/30,
   PID = 0, SN = 0, Time-stamp = 0
>> Service position 0, SAP = 114, PID = 0

>> Filling SHBs...
>> Number of hosted services 1
>> Hosted service position 0, SAP = 116, Host name from NIB 'CPM2    '

>> Filling PABs and PAB List...
>> Port 1: hops = 0 to node 'CPM2    ' from NIB
...NIBs From PAB List...
>> Port 1: hops = 0 to node 'CPM2    '

>> Filling CBB...
>> Configuration name 'TEST001' Host name from CIB 'CPM1    '

>> End of initialize()...
```

Firstly, the Node Identification Blocks (NIB) for the two nodes are filled. Next, the Configuration Identification Block (CIB) is filled with the configuration name

‘TEST001’ and the first NIB pointing to the host node CPM1. The code correctly retrieves the name ‘CPM1’ after parsing the CIB.

The Channel Control Block List (CCBL) is filled with just one Channel Control Block (CCB) entry pointing to only one channel with “ID = 1” with SSAP 114 and DSAP 116, token bucket parameters capacity/tokens/rate set to 90/60/30 and the code correctly shows a PID (process identification), Sequence Number and Timestamp all set to zero (the PID shall be altered by the OPEN function call).

The Service Identification Block List (SIBL) is filled with just one Service Identification Block (SIB) for SAP 114 and the code correctly shows a zeroed PID (the PID shall be altered by the REGISTER function call).

The Service Hosting Block List (SHBL) is filled with just one service and node combination, correctly indicating the node name ‘CPM2’ as host of SAP 116 after parsing the NIB.

A Port Assignment Block (PAB) is filled for port number 1 and “hops = 0” pointing to the NIB built for node CPM2 and inserted in the Port Assignment Block List (PABL). The code correctly shows node name ‘CPM2’ after parsing the PABL.

Finally, the Configuration Base Block (CBB) is filled and the code correctly shows the configuration name ‘TEST001’ and the host node name ‘CPM1’ after parsing the CIB.

6.6.7.2 REGISTER and OPEN function calls

```
>> Registering CPM1 to SAP 114
[REGISTER] Service position 0, SAP = 114, PID = 0x0
[REGISTER] SAP 114 is now registered to PID = 0xcc1 at SIB address 0x40ab04

>> CPM1 SAP 114 is now registered to PID = 0xcc1
>> Opening channel ID 1 for access 0x10000051
[OPEN] channel ID 1 is now opened to PID = 0xcc1

>> Channel ID 1 is now open
```

The code shows the status of the CCB prior to the REGISTER function call correctly showing a PID field set to zero for SAP 114. After calling REGISTER, the PID of application CPM1 is copied into the SIB. The REGISTER function would have failed if the PID field of the CCB were non-zero, indicating that the service had been previously registered.

CPM1 application has to call the OPEN function for getting “SEND access” (indicated by the hex code 0x10000051) to channel ‘1’. The OPEN function checks whether node CPM1 holding SAP 114 is listed as SSAP in channel ‘1’, otherwise OPEN will fail. The OPEN function would have failed if the PID field in the CCB were non-zero, indicating that the channel had been previously opened.

6.6.7.3 SEND function calls

CPM1 calls SEND function three times, each one for sending a message of size 34 bytes over channel ‘1’.

Sending the first message goes as follows.

```
>> Sending message with 34 bytes
[SEND] Calling LLC_SEND

    magic number = 0xa1b2c3d4
    major version number = 2
    minor version number = 4
    GMT to local correction = -3
    accuracy of timestamps = 0
    max length of captured packets,in octets = 65535
    data link type = 1
[LLC_SEND] bytes written 24 for new PCAP file header
    tokens available = 60
    tokens left = 0 (bytes sent + LLCE header)

    timestamp seconds = 1606821301 (0x5fc625b5)
    timestamp microseconds = 200500 (0x30f34)
    number of octets of packet to be saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)
[LLC_SEND] bytes written 16 for new PCAP record header
    Host for service SAP = 116 has MAC 0xaa000503 41200000
[LLC_SEND} Port 1: hops = 0 to node 'CPM2' from PAB list and from SHB 'CPM2'

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length in network byte order = 0x2e00
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x0
    time stamp is seconds after midnight 29701
    UI time stamp = 0x7405
    UI CRC12      = 0xf8c
    Extended header = 0x7 40 5f 8c in network order
    UI CRC32      = 0x840feafa
[LLC_SEND] Channel ID = 1, SSAP = 114, DSAP = 116, capacity = 90, tokens = 0,
           rate = 30 bytes/sec, PID = 0xcc1, SN = 0, Time-stamp = 29701

[LLC_SEND] bytes written 60 for new PCAP record data
[LLC_SEND] length = 34, xmited = 34
    return code 0x10000001, tokens = 0
```

The SEND function validates whether the CCB pointed to by the first argument holds a PID matching the calling process and whether the number of bytes to be sent is valid (minimum 34 and maximum 1492 bytes).

Once arguments are validated, SEND calls LLC_SEND that effectively builds the network data frame for passing it to the Physical Layer. LLC_SEND is implemented in User Mode code, but in a production environment it would execute in Protected (or Kernel) Mode.

When called for the first time, LLC_SEND opens a PCAP file and write its 24 byte file header, including the type of network being monitored (Ethernet has a "data link type" of "1") and other information.

LLC_SEND then performs Traffic Shaping and checks whether the tokens remaining are sufficient for transmitting the 34-byte long message plus another 26 bytes of the IEEE 802.3 (14 bytes), the extended IEEE 802.2 LLC (8 bytes) headers and the extra CRC32 field (4 bytes).

Each network frame written to a PCAP file has a 16 byte record header, including a time stamp in seconds and microseconds and the length of the captured frame in bytes.

The individual fields of the IEEE 802.3 and of the extended IEEE 802.2 LLC headers are correctly listed:

- For finding which network node hosts DSAP 116 of channel '1', LLC_SEND has to search the Service Hosting Block (SHB) for this particular SAP number to find the associated NIB, which points to node CPM2;
- For assembling the MAC Source address, LLC_SEND has to search the Port Assignment Block List (PABL) looking for a Port Assignment Block that points to the NIB describing the destination node CPM2. The Port Number ('1' in this case) is the lowest 4-bit nibble of the IEEE 802.3 MAC Source Address and Unit Number the next high-order 4-bit nibble ('1' in this case).

LLC_SEND writes a record to the PACP file containing the network data frame of length 60 bytes and reports 34 bytes transmitted to the CPM1 application.

Before writing, LLC_SEND calculates the CRC12 for the UI PDU extended header and the extra CRC32 for the UI PDU payload and inserts them in the proper positions.

The calculated CRC12 is different for each UI PDU, since it includes in its calculation a different value for the Time-Stamp field. The extra CRC32 has always the same value (0x840FEAFA), since the data inserted in the payload data field of the UI PDU is always the same (a sequence of 34 values from 0x01 to 0x22). The CRC32 value for this simple data sequence was previously verified using an open-source application (REIFEGERSTE, 2003).

The contents of the CCB are also correctly listed after the LLC_SEND operation, the return code indicates success (hex code 0x10000001) and the remaining number of tokens is correctly indicated (0) after sending 60 bytes.

Sending a second message of 34 bytes follows the same path.

```
>> Sending message with 34 bytes
[SEND] Calling LLC_SEND
    tokens available = 90
    tokens left = 30 (bytes sent + LLCE header)

    timestamp seconds = 1606821304 (0x5fc625b8)
    timestamp microseconds = 231280 (0x38770)
    number of octets of packet to be saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)
[LLC_SEND] bytes written 16 for new PCAP record header
    Host for service SAP = 116 has MAC 0xaa000503 41200000
[LLC_SEND} Port 1: hops = 0 to node 'CPM2' from PAB list and from SHB 'CPM2'

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length in network byte order = 0x2e00
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x1
    time stamp is microseconds after second 231280
    UI time stamp = 0x38770
    UI CRC12      = 0x8b3
    Extended header = 0x38 77 8 b3 in network order
    UI CRC32      = 0x840feafa
[LLC_SEND] Channel ID = 1, SSAP = 114, DSAP = 116, capacity = 90, tokens = 30,
           rate = 30 bytes/sec, PID = 0xcc1, SN = 1, Time-stamp = 231280

[LLC_SEND] bytes written 60 for new PCAP record data
[LLC_SEND] length = 34, xmited = 34
    return code 0x10000001, tokens = 30
```

It is important to note that the Traffic Shaping worked properly, starting with a balance of 0 tokens, allowing sending 60 bytes and leaving a balance of 30 tokens after 3 seconds from the first SEND operation.

Sending a third message of 34 bytes after 1 second from the last SEND operation confirms the proper behavior of Traffic Shaping.

```
>> Sending message with 34 bytes
[SEND] Calling LLC_SEND
    tokens available = 60
    tokens left = 0 (bytes sent + LLCE header)

    timestamp seconds = 1606821305 (0x5fc625b9)
    timestamp microseconds = 246672 (0x3c390)
    number of octets of packet to be saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)
[LLC_SEND] bytes written 16 for new PCAP record header
    Host for service SAP = 116 has MAC 0xaa000503 41200000
[LLC_SEND} Port 1: hops = 0 to node 'CPM2    ' from PAB list and from SHB 'CPM2
'

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length in network byte order = 0x2e00
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x2
    time stamp is microseconds after second 246672
    UI time stamp = 0x3c390
    UI CRC12      = 0xd6c
    Extended header = 0x3c 39 d 6c in network order
    UI CRC32      = 0x840feafa
[LLC_SEND] Channel ID = 1, SSAP = 114, DSAP = 116, capacity = 90, tokens = 0,
rate = 30 bytes/sec, PID = 0xcc1, SN = 2, Time-stamp = 246672

[LLC_SEND] bytes written 60 for new PCAP record data
[LLC_SEND] length = 34, xmited = 34
    return code 0x10000001, tokens = 0
```

The token balance starts with 30 tokens, then 30 tokens are added after 1 second passed, and ending with a balance of 0 tokens after sending 60 bytes.

6.6.7.4 STATUS function call

```
>> Status of channel ID = 1, SSAP = 114, DSAP = 116,
   Token Bucket capacity/tokens/rate [in bytes/sec] = 90/0/30, PID = cc1,
  SN = 2, Time-stamp = 246672
```

After sending three messages over channel ‘1’, the STATUS function call correctly returns the token balance (0), which PID owns the channel, the last Sequence Number (2) and the last Time Stamp (246672 microseconds).

6.6.7.5 CLOSE and UNREGISTER function calls

```
>> Closing channel ID 1
[CLOSE] channel ID 1 is now closed, PID = 0x0

>> Channel ID 1 is now closed

>> Unregistering SAP 114 from CPM1
[UNREGISTER] Service position 0, SAP = 114, PID = 0xcc1
```

```
[UNREGISTER] SAP 114 is now unregistered, PID = 0x0

>> CPM1 SAP 114 is now unregistered (PID = 0x0)
```

Calling the CLOSE function forces the PID in the CCB to be cleared. The same applies to the PID in the SIB when calling the UNREGISTER function. Both functions verify whether the PID of the calling process matches the PID in the CCB and in the SIB.

6.6.8 CPM2 application source code (extract)

The text below is an extract of the source code for the CPM2 application (the full text is provided in Appendix E). The function calls to the “channel” services are highlighted:

```
int main()
{
// calling "initialize()" not needed in real life..
   retcode = initialize();

   printf("\n\n>> Entering main()...");

   printf("\n>> Registering CPM1 to SAP %d", cpm2_SAP);

   retcode = REGISTER(cpm2_SAP, &cpm2_SIB);

   if(retcode == SUCCESS)
   {
      printf("\n\n>> CPM2 SAP %d is now registered to PID = 0x%x",
      cpm2_SIB->sap,
      cpm2_SIB->pid);
   } else
   {
      printf("\n>> bad code 0x%x", retcode);
   }

   printf("\n>> Opening channel ID %d for access 0x%x", cpm2_chn, cpm2_acc);

   retcode = OPEN(cpm2_chn, cpm2_acc, &cpm2_CCB);

   if(retcode == SUCCESS)
   {
      printf("\n\n>> Channel ID %d is now open",
      cpm2_CCB->chnid);
   } else
   {
      printf("\n>> bad code 0x%x", retcode);
   }

   msize = strlen(message);
   length = (int)msize;

   printf("\n\n>> Receiving message with %d bytes", length);

   gettimeofday(&prvTime, NULL);
   retcode = RECEIVE(&cpm2_CCB, message, length, &receivd);

   printf("\n   return code 0x%x, tokens = %d", retcode, cpm2_CCB->tokens);

   printf("\n\n>> Receiving message with %d bytes", length);
```

```
   nanoTime.tv_nsec = 0;
   nanoTime.tv_sec = 3;
   nanosleep(&nanoTime, (struct timespec *)NULL);

   retcode = RECEIVE(&cpm2_CCB, message, length, &receivd);

   printf("\n   return code 0x%x, tokens = %d", retcode, cpm2_CCB->tokens);

   printf("\n\n>> Receiving message with %d bytes", length);

   nanoTime.tv_nsec = 0;
   nanoTime.tv_sec = 1;
   nanosleep(&nanoTime, (struct timespec *)NULL);

   retcode = RECEIVE(&cpm2_CCB, message, length, &receivd);

   printf("\n   return code 0x%x, tokens = %d", retcode, cpm2_CCB->tokens);

   retcode = STATUS(cpm2_chn, &chn_ssap, &chn_dsap, &chn_capacity, &chn_tokens,
                    &chn_rate, &chn_pid, &chn_sn, &chn_timestmp);

   printf("\n\n>> Status of channel ID = %d, SSAP = %d, DSAP = %d, Token Bucket
capacity/tokens/rate [in bytes/sec] = %d/%d/%d, PID = %x, SN = %d, Time-stamp
= %d",
      cpm2_chn,
      chn_ssap,
      chn_dsap,
      chn_capacity,
      chn_tokens,
      chn_rate,
      chn_pid,
      chn_sn,
      chn_timestmp);

   printf("\n\n>> Closing channel ID %d", cpm2_chn);

   retcode = CLOSE(&cpm2_CCB);

   if(retcode == SUCCESS)
   {
      printf("\n\n>> Channel ID %d is now closed",
      cpm2_CCB->chnid);
   } else
   {
      printf("\n>> bad code 0x%x", retcode);
   }

   printf("\n\n>> Unregistering SAP %d from CPM2", cpm2_SAP);

   retcode = UNREGISTER(&cpm2_SIB);

   if(retcode == SUCCESS)
   {
      printf("\n\n>> CPM2 SAP %d is now unregistered (PID = 0x%x)",
      cpm2_SIB->sap,
      cpm2_SIB->pid);
   } else
   {
      printf("\n>> bad code 0x%x", retcode);
   }

   return 0;
}
```

As for the CPM1 application, the call to the function `initialize()` is not needed in a production code, but it is necessary to the CPM2 application for initializing the in-memory data structures (see Section 5.4.2) specified for the test scenario

6.6.9 CPM2 application output commented

The text output of the CPM2 application is shown in the next sections followed by comments explaining how it validates the expected results listed in the previous Section 6.4.2.

6.6.9.1 Initialization

```
>> Filling NIBs and CIB...
>> Number of nodes 2
>> Node name 'CPM2    ' Equipment ID = 0x341  Unit = 2 MAC Base = aa000503
41200000
>> Node name 'CPM1    ' Equipment ID = 0x341  Unit = 1 MAC Base = aa000503
41100000
>> Configuration name 'TEST001' Host NIB address = 63fc90 Host name from NIB
List 'CPM2   ' Host name from CIB 'CPM2   '
>> Filling CCBs and SIBs...
>> Number of channels 1
>> Channel position 0, ID = 1, SSAP = 114, DSAP = 116,
   Token Bucket capacity/tokens/rate [in bytes/sec] = 90/60/30,
   PID = 0, SN = 0, Time-stamp = 0
>> Service position 0, SAP = 116, PID = 0

>> Filling SHBs...
>> Number of hosted services 1
>> Hosted service position 0, SAP = 114, Host name from NIB 'CPM1   '

>> Filling PABs and PAB List...
>> Port 1: hops = 0 to node 'CPM1   ' from NIB
...NIBs From PAB List...
>> Port 1: hops = 0 to node 'CPM1   '

>> Filling CBB...
>> Configuration name 'TEST001' Host name from CIB 'CPM2   '

>> End of initialize()...
```

As for the CPM1 application, the Node Identification Blocks (NIB) for the two nodes are filled, then the Configuration Identification Block (CIB) is filled with the configuration name ‘TEST001’ and the first NIB pointing to the host node CPM2. The code correctly retrieves the name ‘CPM2’ after parsing the CIB.

The Channel Control Block List (CCBL) is filled with the same values as for CPM1 appplication, since they both share the same channel ‘1’.

The Service Identification Block List (SIBL) is filled with just one Service Identification Block (SIB) for SAP 116 and the code correctly shows a zeroed PID (the PID shall be altered by the REGISTER function call).

The Service Hosting Block List (SHBL) is filled with just one service and node combination, correctly indicating the node name ‘CPM1’ as host of SAP 114 after parsing the NIB.

A Port Assignment Block (PAB) is filled for port number 1 and “hops = 0” pointing to the NIB built for node CPM1 and inserted in the Port Assignment Block List (PABL). The code correctly shows node name ‘CPM1’ after parsing the PABL.

Lastly, the Configuration Base Block (CBB) is filled and the code correctly shows the configuration name ‘TEST001’and the host name ‘CPM2’ after parsing the CIB.

6.6.9.2 REGISTER and OPEN function calls

```
>> Registering CPM2 to SAP 116
[REGISTER] Service position 0, SAP = 116, PID = 0x0
[REGISTER] SAP 116 is now registered to PID = 0xcc2 at SIB address 0x411c04

>> CPM2 SAP 116 is now registered to PID = 0xcc2
>> Opening channel ID 1 for access 0x10000053
[OPEN] channel ID 1 is now opened to PID = 0xcc2

>> Channel ID 1 is now open
```

The code shows the status of the CCB prior to the REGISTER function call correctly showing a PID field set to zero for SAP 116. After calling REGISTER, the PID of application CPM2 is copied into the SIB. The REGISTER function would have failed if the PID field of the CCB were non-zero, indicating that the service had been previously registered.

CPM2 application has to call the OPEN function for getting “RECEIVE access” (indicated by the hex code 0x10000053) to channel ‘1’. The OPEN function checks whether node CPM2 holding SAP 116 is listed as DSAP in channel ‘1’, otherwise OPEN will fail. The OPEN function would have failed if the PID field in the CCB were non-zero, indicating that the channel had been previously opened.

6.6.9.3 RECEIVE function calls

CPM2 calls RECEIVE function three times, each one for receiving a message of size 34 bytes over channel ‘1’.

```
>> Receiving message with 34 bytes
 [RECEIVE] Calling LLC_RECEIVE
[LLC_RECEIVE] bytes read 24 for PCAP file header

    magic number = 0xa1b2c3d4
    major version number = 2
    minor version number = 4
    GMT to local correction = -3
    accuracy of timestamps = 0
    max length of captured packets,in octets = 65535
    data link type = 1
    tokens available = 60
    tokens left = 0 (bytes sent + LLCE header)
[LLC_RECEIVE] bytes read 16 from PCAP record header

    timestamp seconds = 1606821301 (0x5fc625b5)
    timestamp microseconds = 200500 (0x30f34)
    number of octets of packet saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)

[LLC_RECEIVE] bytes read 60 from PCAP record data

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length = 0x2e
    UI CRC12      = 0xf8c
    UI CRC32      = 0x840feafa
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x0
    Extended header = 0x7 40 5f 8c in network order
    UI time stamp = 29701 (0x7405)
    time stamp is seconds after midnight (apparent offset to remote clock =
120 +ahead/-behind)
[LLC_RECEIVE] length = 34, received = 34
    return code 0x10000001, tokens = 0
```

The RECEIVE function validates whether the CCB pointed to by the first argument holds a PID matching the calling process and whether the number of bytes to be received is valid (minimum 34 and maximum 1492 bytes).

Once arguments are validated, RECEIVE calls LLC_RECEIVE that has to wait for the Physical Layer to pass a received network data frame with a Destination MAC address matching the one for node CPM2. In this test case, LLC_RECEIVE will simply open the PCAP file recorded by CPM1 application and process its contents.

When called for the first time, LLC_RECEIVE opens the PCAP file and reads its 24 byte file header. The code correctly lists the PCAP file header contents.

LLC_RECEIVE then performs Traffic Policing and checks whether the tokens remaining are sufficient for receiving the 34-byte long message plus another 26 bytes of the IEEE 802.3 (14 bytes), the extended IEEE 802.2 LLC (8 bytes) headers and the extra CRC32 field (4 bytes), a total of 60 bytes.

LLC_RECEIVE reads the first record of the PCAP file and correctly displays its contents, including the total frame length (60 bytes or hexadecimal 0x3C). LLC_RECEIVE also validates the CRC12 for the UI PU extended header and the CRC32 for the UI PDU payload (an error condition would have been raised in case of CRC mismatch).

LLC_RECEIVE validates that the received network frame is destined to channel '1' by examining the DSAP and SSAP against the CCB for this channel in the CCBL structure. The code correctly lists the fields of the IEEE 802.3 and the extended IEEE 802.2 headers.

The contents of the CCB are also correctly listed after the LLC_RECEIVE operation, the return code indicates success (hex code 0x10000001) and the remaining number of tokens is correctly indicated (0) after receiving 60 bytes.

Since the first network frame has Sequence Number 0 and carries a Time Stamp with the number of seconds passed after midnight, LLC_RECEIVE can estimate a time offset between the remote clock at CPM1 and the local clock at CPM2 by calculating the difference between the number of seconds passed after midnight locally and the received time stamp.

After the first call to RECEIVE, the CPM2 application waits 3 seconds before calling RECEIVE again.

```
>> Receiving message with 34 bytes
[RECEIVE] Calling LLC_RECEIVE
    tokens available = 90
    tokens left = 30 (bytes sent + LLCE header)
[LLC_RECEIVE] bytes read 16 from PCAP record header

    timestamp seconds = 1606821304 (0x5fc625b8)
    timestamp microseconds = 231280 (0x38770)
    number of octets of packet saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)

[LLC_RECEIVE] bytes read 60 from PCAP record data

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length = 0x2e
    UI CRC12      = 0x8b3
    UI CRC32      = 0x840feafa
```

```
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x1
    Extended header = 0x38 77 8 b3 in network order
    UI time stamp = 231280 (0x38770)
    delta time in seconds for traffic policing = 3.015157
[LLC_RECEIVE] length = 34, received = 34
    return code 0x10000001, tokens = 30
```

Receiving a second message of 34 bytes follows the same path.

The code also displays the number of microseconds passed since the last call to LLC_RECEIVE used by the Traffic Policing.

The Traffic Policing worked properly, starting with a balance of 90 tokens after 3 seconds passed from the first RECEIVE operation, receiving 60 bytes and leaving a balance of 30 tokens.

After the second call to RECEIVE, the CPM2 application waits 1 second before calling RECEIVE again.

```
>> Receiving message with 34 bytes
[RECEIVE] Calling LLC_RECEIVE
    tokens available = 60
    tokens left = 0 (bytes sent + LLCE header)
[LLC_RECEIVE] bytes read 16 from PCAP record header

    timestamp seconds = 1606821305 (0x5fc625b9)
    timestamp microseconds = 246672 (0x3c390)
    number of octets of packet saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)

[LLC_RECEIVE] bytes read 60 from PCAP record data

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length = 0x2e
    UI CRC12      = 0xd6c
    UI CRC32      = 0x840feafa
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x2
    Extended header = 0x3c 39 d 6c in network order
    UI time stamp = 246672 (0x3c390)
    delta time in seconds for traffic policing = 1.015389
    estimated delta time in seconds at origin  = 1.015392
[LLC_RECEIVE] length = 34, received = 34
    return code 0x10000001, tokens = 0
```

Since the Time Stamp carried by the third received frame has the number of microseconds passed after the second, LLC_RECEIVE can estimate the time interval taken by the transmitting party, CPM1, before transmitting the third message.

Traffic Policing worked properly, starting with a balance of 30 tokens, adding another 30 tokens after passing 1 second from the last RECEIVE operation, receiving 60 bytes and leaving a balance of 0.

6.6.9.4 STATUS function call

```
>> Status of channel ID = 1, SSAP = 114, DSAP = 116,
   Token Bucket capacity/tokens/rate [in bytes/sec] = 90/0/30, PID = cc2,
   SN = 2, Time-stamp = 246672
```

After receiving three messages over channel ‘1’, the STATUS function call correctly returns the token balance (0), which PID owns the channel, the last Sequence Number (2) and the last Time Stamp.

6.6.9.5 CLOSE and UNREGISTER function calls

```
>> Closing channel ID 1
[CLOSE] channel ID 1 is now closed, PID = 0x0

>> Channel ID 1 is now closed

>> Unregistering SAP 116 from CPM2
[UNREGISTER] Service position 0, SAP = 116, PID = 0xcc2
[UNREGISTER] SAP 116 is now unregistered, PID = 0x0

>> CPM2 SAP 116 is now unregistered (PID = 0x0)
```

Calling the CLOSE function forces the PID in the CCB to be cleared. The same applies to the PID in the SIB when calling the UNREGISTER function. Both functions verify whether the PID of the calling process matches the PID in the CCB and in the SIB.

6.6.10 Frame validation using wireshark generic dissector

The Wireshark tool (WSDG, 2020) can be configured for parsing records written to PCAP files by any Ethernet-based protocol such as the new Data Link Layer protocol.

More details in how to configure Wireshark are provided in Appendix B.

For the test case, it is relevant to show that Wireshark correctly displays the contents of the 3 records written the CPM1 application to a PCAP file.

The next figures are copies of the Wireshark screen, one for each record read.

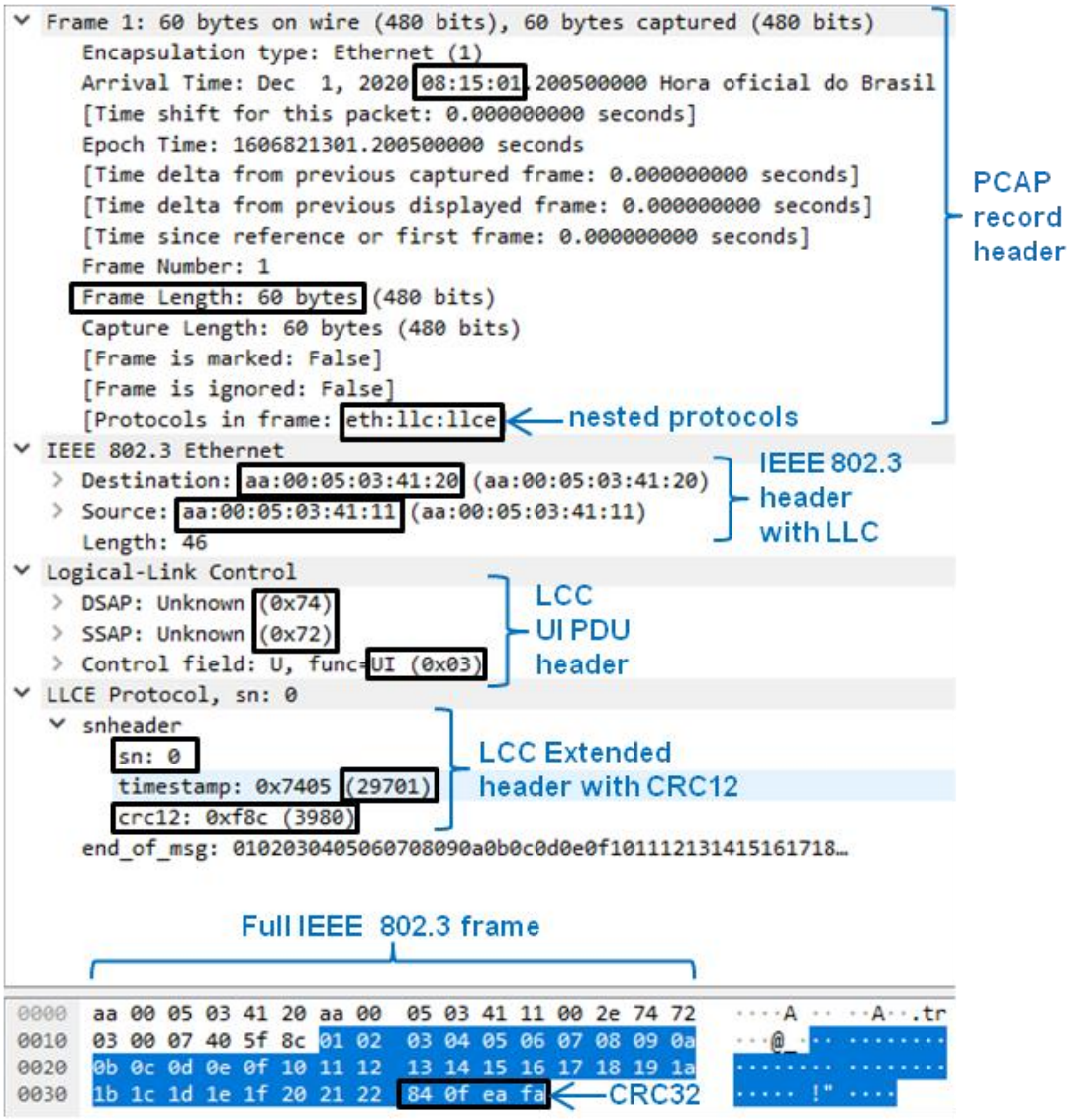
The important data for verifying the correctness of the network frame structure are the correct identification of the IEEE 802.3 header fields, namely the Destination and Source MAC addresses and the Length, and the extended IEEE 802.2 LLC fields, namely the DSAP, the SSAP, the Control for UI PDU (0x03), the Sequence Number, the Time-Stamp and the header CRC12.

The extra CRC32 at the end of the LLC payload also reads correctly (0x840FEAFA) at the last 4 bytes of the “end-of-msg” field.

6.6.10.1 First record: UI PDU sequence number 0

The first captured frame has a SN of 0 and carries a time stamp of 29701 seconds passed after midnight, which is in line with the record header's "Arrival Time" at 08:15:01 (29701 = 8x3600 + 15x60 + 1).

Figure 6.5 – Wireshark screen for first record with SN = 0.

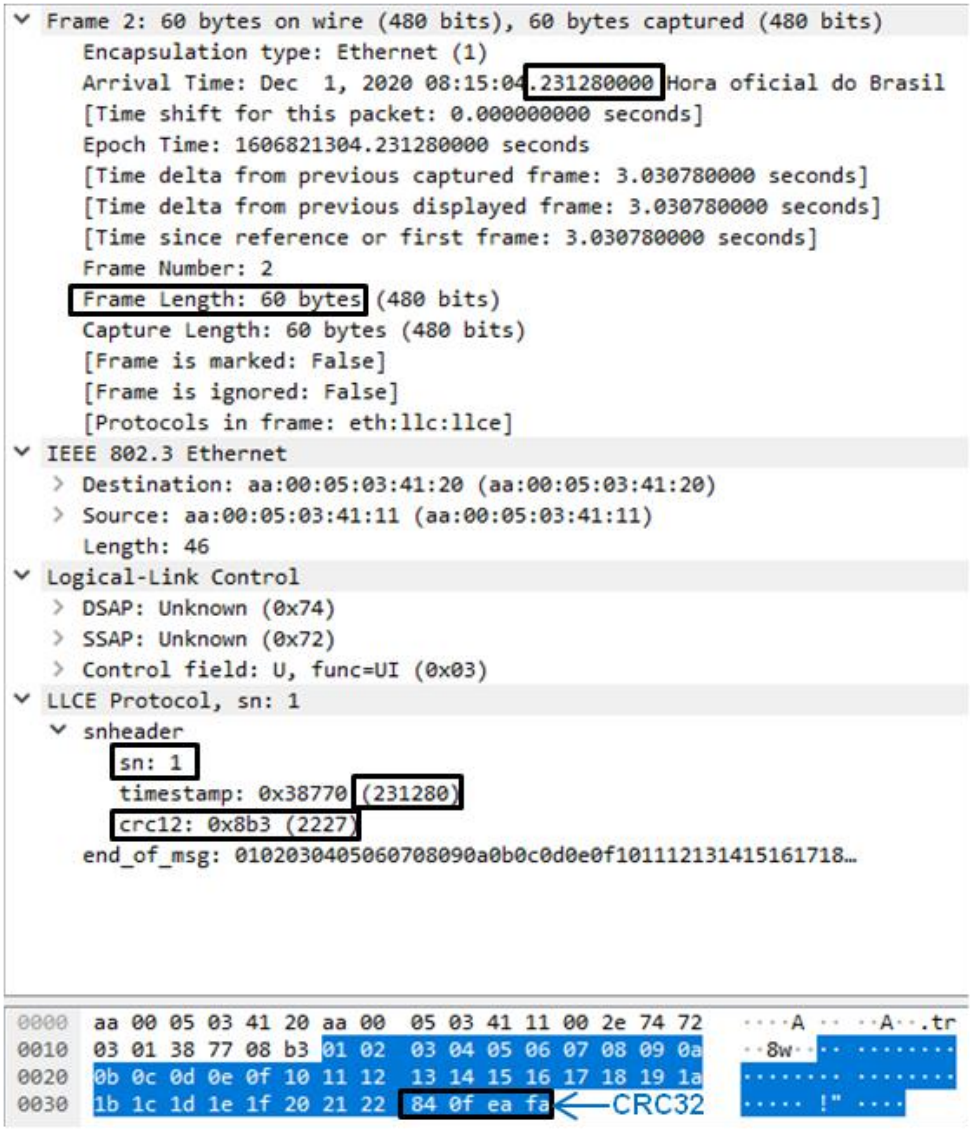


6.6.10.2 Second record: UI PDU sequence number 1

The second captured frame has a SN of 1 and carries a time stamp of 231208 microseconds after the second, the same time stamp inserted in the PCAP record header.

Wireshark also displays a “Time delta from previous captured frame” of 3.030780 seconds, the difference between the current and previous “Epoch Time” (seconds and nanoseconds after January 1st 1970), and close enough to the 3 seconds time interval passed after the first SEND operation.

Figure 6.6 – Wireshark screen for first record with SN = 1.

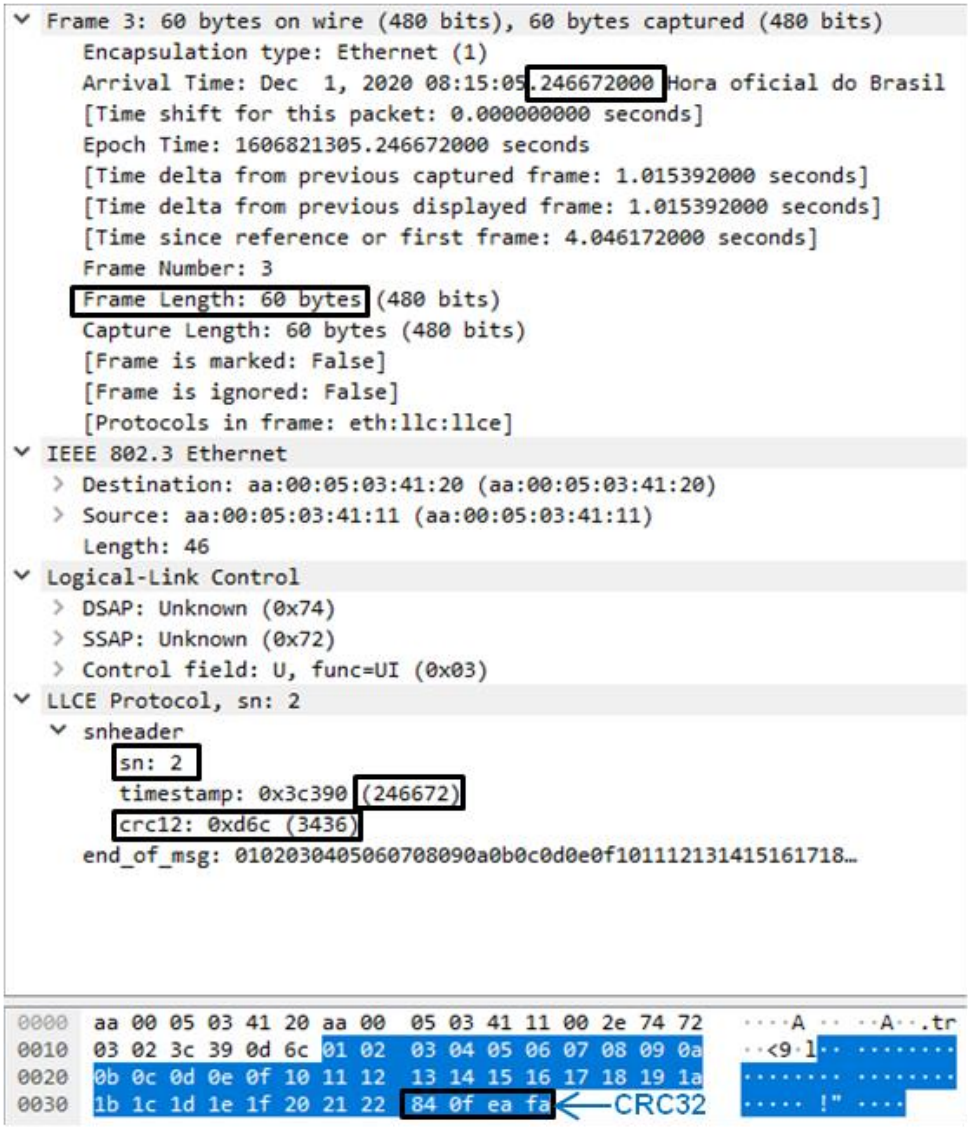


6.6.10.3 Third record: UI PDU sequence number 2

The second captured frame has a SN of 2 and carries a time stamp of 246672 microseconds after the second, the same time stamp inserted in the PCAP record header.

Wireshark also displays a “Time delta from previous captured frame” of 1.015392 seconds, the difference between the current and previous “Epoch Time”, close enough to the 1 second time interval passed after second SEND operation.

Figure 6.7 – Wireshark screen for first record with SN = 2.



6.7 Summary

The evidences showing that the test case in Sections 6.2 to 6.6 completed the objectives listed in Section 6.1 are:

- The collection of structures implemented in C Language allowed the “channel” API calls to correctly retrieve the necessary data for performing its function as designed, validating the in-memory structures described in Section 5.4.2;
- The sequence of “channel” API function calls correctly produced and consumed the simulated data as designed, validating the programming interface described in Section 5.4.3;
- The Wireshark correctly displayed the contents of the network data frame recorded in PCAP format at the MAC layer, according to IEEE 802.3, at the standard LLC layer, according to IEEE 802.2, and at the extended LLC layer, validating the format of the data frame built, as described in Section 5.1;
- Sending and receiving three network data frames separated by irregular time intervals validated the implementation of Traffic Shaping at the “producer” node (CPM1) and Traffic Policing at the “consumer” node (CPM2), as described in Sections 5.2.3 and 5.2.4.

A production version of the new Data Link Layer and services shall require a proper test suite for testing error conditions and evaluating the timing behavior of the protocol implementation. Such investigation is beyond the scope of this work and is left as a suggestion for future work.

7 CONCLUSIONS, CONTRIBUTIONS AND SUGGESTIONS

7.1 Conclusions

Seeking to fill a gap in existing network protocols, this work introduced a new, IEEE 802.2 Logical Layer Control (LLC) extended protocol to the IEEE 802.3 Data Link Layer protocol, and an associated software interface targeting the reduction the number of protocol layers required to connect a “data producer” process to a “data consumer” process over the network infrastructure.

A virtual entity called “channel” was introduced as the means of achieving this goal.

A software Application Programming Interface (API) accessible through the higher Application Layer was introduced, allowing portability across different software implementations of the channel.

Flow control services, namely Traffic Shaping and Traffic Policing, were introduced as an integral part of the system services involved in sending and receiving data over a channel, thus protecting an application from an occasional malfunction of another application.

The proposed test case was sufficient for validating the in-memory data structures that support the implementation of the protocol and the access to it over the channel API. The test case also demonstrated the correct behavior of the Traffic Policing and the Traffic Shaping services, by allowing sending and receiving messages separated by irregular, yet constrained, time intervals.

The timing information added to the IEEE 802.2 original Unnumbered Information (UI) of the “Type 1” Portable Data Unit (PDU) network data frames was proven useful for estimating the timing regularity of the “data producer” by the “data consumer”, even considering that, due to the limitations of the platform used for the test case implementation, the time intervals for validating Traffic Shaping and Policing were specified in seconds. In a production environment it would be typical to have time intervals specified in milliseconds.

The additional network data frame integrity was verified by validating calculations of the CRC12 for the extended IEEE 802.2 UI PDU header and of

the CRC32 for the user data by using an open-source application accessible via the Internet (REIFEGERSTE, 2003).

The results obtained so far are very encouraging and suggest that the proposed protocol and the associated software interface can play an important role in communications supporting complex distributed systems installed in aerospace vehicles.

Implementing route validation and testing it in a mixed network topology are left to a future study, because essential resources were not available to this work, namely the access to an open-source Ethernet network card Device Driver and a suitable laboratory environment configured with a few, multiple Ethernet port nodes and Ethernet switches.

7.2 Summary of contributions

1) Bibliographic review of the most relevant digital communication protocols used in aerospace applications, in particular MIL-STD-1553B, ARINC-664 Part 7 (AFDX™) and SpaceWire;

2) Detailed review of the “Type 1” operation of the IEEE 802.2 Logical Link Control (LLC) protocol to the IEEE 802.3 Data Link Layer protocol;

3) Introduction of a new, IEEE 802.2 LLC extended Data Link Layer protocol for connecting data producers to data consumers executing at the Application Layer without the need of the Transport and Network Layers through a new concept called “channel”;

4) Introduction of a route validation protocol using IEEE 802.2 TEST Protocol Data Units (PDU);

5) Introduction of a new method for estimating transmission delays of a network data frame while crossing an Ethernet switch, which is simpler than other referenced methods, allowing routing nodes to perform Traffic Policing for protecting the network against abnormal behavior of a node (PENNA et al., 2020); this method is briefly described in Appendix A.

7.3 Suggestions for further studies

1) To implement the “channel” API, as well as its supporting in-memory structures, in protected mode on a suitable operating system that supports

real-time applications; in particular, to investigate how an underlying operating system can handle shared access to protected memory structures;

2) To create a small laboratory environment for testing the route validation through the use of TEST PDUs by deploying a mixed-topology network using single Ethernet port nodes, multiple Ethernet port nodes and Ethernet switches;

3) Using the same laboratory, experiment with the new method of estimating transmission delays and compare results obtained by analysis with actual delay measurements; this experiment would require precise local clock synchronization on each node over the entire network.

BIBLIOGRAFIC REFERENCES

ALENA, R.; OSSENFORT, J.; LAWS, K.; GOFORTH, A.; FIGUEROA, F. Communications for integrated modular avionics. In: IEEE AEROSPACE CONFERENCE, 2007, Big Sky, Montana. **Proceedings…** IEEE, 2007. DOI: 10.1109/AERO.2007.352639.

AERONAUTICAL RADIO INCORPORATED - ARINC. **ARINC specification 429**: mark 33 Digital Information Transfer System (DITS) – part 1 functional description, electrical interface, label assignments and word formats. Riva Road, Annapolis, 2001. 628p.

AERONAUTICAL RADIO INCORPORATED – ARINC. **ARINC specification 653-1**: avionics application software standard interface. Bowie, Maryland, 2015. 285p.

AERONAUTICAL RADIO INCORPORATED - ARINC. **ARINC specification 664**: aircraft data networks – part 7, deterministic networks. Riva Road, Annapolis, 2009. 150p.

BENAMMAR, N.; RIDOUARD, F.; BAUER, H.; RICHARD, P. Forward end-to-end delay analysis extension for FP/FIFO policy in AFDX networks. In: IEEE INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION (ETFA), 22., 2017. **Proceedings…** IEEE, 2017. P.1-8. DOI: 10.1109/ETFA.2017.8247606.

BAUER, H. **Analyze pire cas de flux hétérogenes dans un réseaux embarqué avion**. 2011. 188p. Thesis (Doctorat) - Université de Toulouse, France, 2011.

BOSCH, R. **CAN bus specification version 2.0.** Stuttgart, Germany: Robert Bosch GmbH, 1991. 73p.

BOURGUIGNON, E.; FRASELLE, S.; SCALAIS, T. Power processing unit activities at Thales Alenia Space Belgium (ETCA). In: INTERNATIONAL ELECTRIC PROPULSION CONFERENCE, 33., 2013, Washington-DC. **Proceedings…** 2013.

CARAMIA, M. **CAN @ thales - beyond ExoMars**. In: ESA CAN WORKSHOP, 2016. **Proceedings…** 2016. Available from: https://indico.esa.int/event/120/contributions/480/attachments/658/704/ESA-CAN-WS-2016-TAS-I-Caramia_1.pdf. Access on: 01 feb. 2021.

CERF, V. G.; KAHN, R. E. A protocol for packet network inter-communication. **IEEE Transactions on Communicaitons**, v.22, n. 5, 1974. Available from: https://www.cs.princeton.edu/courses/archive/fall06/cos561/papers/cerf74.pdf. Access on: 01 feb. 2021.

CERF, V.; DALA, Y.; SUNSHINE, C. **Specification of internet transmission control program**. 1974. Available from: https://tools.ietf.org/html/rfc675. Access on: 01 feb. 2021.

CISCO. **Creating ethernet VLANs on catalyst switches**. 2014. Available from: https://www.cisco.com/c/pt_br/support/docs/lan-switching/vlan/10023-3.html. Access on: 01 feb. 2021.

CISCO. **How a LAN switch works**. 2004. Available from: https://www.ciscopress.com/articles/article.asp?p=357103&seqNum=4. Access on: 01 feb. 2021.

CISCO. **QoS: policing and shaping configuration guide**. 2020. Available from: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/qos_plcshp/configuration/xe-17/qos-plcshp-xe-17-book.html. Access on: 01 feb. 2021.

COMER, D. E. **Internetworking with TCP/IP:** principles, protocols, and architecture. 2.ed. Englewood Cliffs: Prentice-Hall, 1991. 547p.

CONSULTATIVE COMMITTEE FOR SPACE DATA SYSTEMS - CCSDS, **Spacecraft Onboard Interface Services (SOIS)**. 2013. Available from: https://public.ccsds.org/publications/SOIS.aspx. Access on: 01 feb. 2021.

CRUZ, R.L. A calculus for network delay: network elements in isolation. **IEEE Transaction on Information Theory**, v. 37, n. 1, p. 114-131, 1991.

CRUZ, R.L. A calculus for network delay: network analysis. **IEEE Transaction on Information Theory**, v. 37, n.1, p. 132-141, 1991.

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY. **The history of Arpanet:** the first decade. 1981. Available from: https://apps.dtic.mil/dtic/tr/fulltext/u2/a115440.pdf. Access on: 01 feb. 2021.

EUROPEAN AVIATION SAFTEY ADMINISTRATION - EASA. **Aircraft certification**. Available from: https://www.easa.europa.eu/easa-and-you/aircraft-products/aircraft-certification. Access on: 01 feb. 2021.

EDISON TECH CENTER. **The V2 rocket - how it works, guidance**. Available from: https://www.youtube.com/watch?v=Ph-npS29n9Q. Access on: 01 feb. 2021.

ELECTRONIC INDUSTRIES ALLIANCE - EIA. **RS-232-C**: interface between data terminal equipment and data communication equipment employing serial binary data interchange**.** Washington, USA, 1969. 29p.

ELECTRONIC INDUSTRIES ALLIANCE - EIA. **RS-485**: electrical characteristics of generators and receivers for use in balanced multipoint systems. Washington, USA,1983. 22p.

EUROPEAN COOPERATION FOR SPACE STANDARDIZATION - ECSS. **SpaceWire**: links, nodes, routers and networks. Noordwijk, The Netherlands: ECSS, 2008.

FEDERAL AVIATION ADMINISTRATIONS - FAA. **Airworthiness certificates overview**. Available from: https://www.faa.gov/aircraft/air_cert/airworthiness_certification/aw_overview/. Access on: 01 feb. 2021.

FRANCES, F.; FRABOUL C.; GRIEU, J. Using network calculus to optimize the AFDX network. In: ERTS, Toulouse, France, 2006. **Proceedings…** 2006. p. 1-8.

FREEBSD. **Essential socket functions**. Available from: https://docs.freebsd.org/en_US.ISO8859-1/books/developers-handbook/sockets-essential-functions.html. Access on: 01 feb. 2021.

FLEXRAY. **FlexRay communications system**: protocol specification - version 3.0.1. [S.l.]: FlexRay Consortium, 2010. 341p.

FUCHS, C. M. The evolution of avionics networks from ARINC 429 to AFDX. In: SEMINAR ON NETWORK ARCHITECTURES AND SERVICES, 2012. **Proceedings…** 2012. p.65-76. DOI:10.2313/NET-2012-08-1_10.

GASKA, T.; WATKIN, C. B.; CHEN, Y. Integrated modular avionics - past, present, and future. **IEEE Aerospace and Electronic Systems Magazine**, v. 30, n. 9, p. 12-23, 2015. DOI:10.1109/MAES.2015.150014.

GEORGES, J.-P.; DIVOUX, T.; RONDEAU, E. Confronting the perfomances of a switched ethernet network with industrial constraints by using the network calculus. **International Journal of Communication Systems**, v. 18, p. 877-903, 2005.

GOFORTH, M.; RATLIFF, J. E.; HAMES, K. L.; VITALPUR, S. V. Avionics architectures for exploration: building a better approach for (human) spaceflight avionics. In: AIAA SPACEOPS CONFERENCE, 2014, Pasadena, California. **Proceedings…** 2014.p.1-16. DOI:10.2514/6.2014-1604.

GRIEU, J. **Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systémes avioniques**. 2004. 145 p. Thesis (Docteur en Réseaux et Télécommunications) - Institut National Polytechnique de Toulouse, France, 2004.

GWALTNEY, D. A.; BRISCOE, J. M. **Comparison of communication architectures for spacecraft modular avionics systems**. Alabama: Marshall Space Flight Center, 2006. NASA Report TM-2006-214431.

HALL, E. C. **Journey to the Moon:** the history of the Apollo guidance computer. Reston, Virginia: AIAA, 1996. 196 p. ISBN 1-56347-185-X.

HAMMOND, J. L.; BROWN, J. E.; LIU, S. S. **Development of a transmission error model and error control model**. New York: RADC, 1975. Technical Report RADC-TR-75-138.

HARTER, P. K. **Response times in level-structured systems**. Boulder, Colorado: University of Colorado, 1984. Technical Report CU-CS-269-84.

IBM. **Networking token ring**. 2013. Available from: https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_72/rzaju/rzaju000pdf.pdf?view=kc. Access on: 01 feb. 2021.

INSTITUTE OF ELECTRICAL AND ELECTRONICS - IEEE. **IEEE standard 802.3**: standard for ethernet. New York, 2012. 634 p.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS - IEEE. **IEEE standard 802.2**: part 2: logical link control. New York, 1998. 253 p.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS - IEEE. **Logical Link Control (LLC) public listing**. Available from: https://standards.ieee.org/products-services/regauth/llc/public.html. Access on: 01 feb. 2021.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS - IEEE. **Ethertype public listing**. Available from: http://standards-oui.ieee.org/ethertype/eth.txt. Access on: 01 feb. 2021.

INTERNATIONAL STANDARDS ORGANIZATION - ISO. **ISO/IEC 7498-1**: information technology - open systems interconnection - basic reference model. 1994. Available from: https://www.iso.org/standard/20269.html. Access on: 01 feb. 2021.

INTERNATIONAL STANDARDS ORGANIZATION - ISO. **ISO/IEC 9899**: programming languages - C. 2011. Available from: http:// http://www.open-std.org/jtc1/sc22/wg14/www/projects#9899. Access on: 01 feb. 2021.

JOSEPH, M.; PANDYA, P. Finding response times in a real time system **The Computer Journal**, v. 29, n. 5, p. 390-395, 1986.

KEMAYO, G.; RIDOUARD, F.; BAUER, H.; RICHARD, P. Optimistic problems in the trajectory approach in FIFO context. In: IEEE CONFERENCE ON EMERGING TECHNOLOGIES FACTORY AUTOMATION (ETFA), 18., 2013. **Proceedings…** 2013. p.1-8. DOI: 10.1109/ETFA.2013.6648054.

KEMAYO, G.; RIDOUARD, F.; BAUER, H.; RICHARD, P. A Forward end-to-end delays Analysis for packet switched networks. In: INTERNATIONAL CONFERENCE ON REAL-TIME NETWORKS AND SYSTEMS (RTNS), 22., 2014, New York. **Proceedings…** 2014. p.65-74. DOI: 10.1145/2659787.2659801.

KEMAYO, G.; BENAMMAR, N.; RIDOUARD, F.; BAUER, H.; RICHARD, P. Improving AFDX end-to-end delays analysis. In: IEEE CONFERENCE ON EMERGING TECHNOLOGIES & FACTORY AUTOMATION (ETFA), 20., 2015. **Proceedings…** 2015. DOI: 10.1109/ETFA.2015.7301463.

KOECHEL, S.; LANGERB, M. New space: impacts of innovative concepts in satellite development on the space industry In: INTERNATIONAL ASTRONAUTICAL CONGRESS (IAC), 69., 2018, Bremen. **Proceedings…** 2018. IAC-18-E6.3.2.

KOOPMAN, P.; CHAKRAVARTY, T Cyclic Redundancy Code (CRC) polynomial selection for embedded networks. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS (DSN), 2004, Florence. **Proceedings…** 2004. DOI: 10.1109/DSN.2004.1311885.

KOOPMAN, P. 32-bit cyclic redundancy codes for internet applications. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS (DSN), 2002, Washington-DC. **Proceedings…** 2002. p. 459-472. DOI: 10.1109/DSN.2002.1028931.

KRAFCIK, J. F. Triumph of the lean production system. **SLOAN Management Review**, v. 30, n. 1, 1988.

LE BOUDEC, J.Y.; THIRAN, P. **Network calculus:** a theory of deterministic queuing systems for the internet. Berlin: Springer-Verlag, 2001. 114p.

LEXICO. **Definition of communication in English**. Available from: https://www.lexico.com/en/definition/communication. Access on: 01 feb. 2021.

LEXICO. **Definition of topology in English**. Available from: https://www.lexico.com/en/definition/topology. Access on: 01 feb. 2021.

LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. **Journal of the Association for Computing Machinery**, v. 20, n. 1, p. 46-61, 1973.

LOVELESS, A. On TTEthernet for integrated fault-tolerant spacecraft networks. In: AIAA SPACE CONFERENCE, 2015, Pasadena, California. **Proceedings…** 2015. DOI: 10.2514/6.2015-4526.

LUXI, Z.; QIAO, L.; YING, X.; ZHONG, Z.; HUAGANG, X. Using multi-link grouping technique to achieve tight latency in network calculus. In: IEEE/AIAA DIGITAL AVIONICS SYSTEMS CONFERENCE (DASC), 32., New York, 2013. **Proceedings…** IEEE, 2013. DOI:10.1109/DASC.2013.6712551.

MARTIN, J.; CHAPMAN, K. K.; LEBEN J. **Local area networks:** architectures and implementations. 2.ed. Saddle River: Prentice-Hall, 1994. 566p.

MCKENZIE, A. INWG and the conception of the internet: an eyewitness account. **IEEE Annals of the History of Computing**, v. 33, n. 1, p 66-71, 2011.

MERRIAN-WEBSTER. **Definition of communication**. Available from: https://www.merriam-webster.com/dictionary/communication. Access on: 01 feb. 2021.

MILITARY STANDARD. **Digital time division command/response multiplex data bus**. Available from: https://quicksearch.dla.mil/qsDocDetails.aspx?ident_number=275874. Access on: 01 feb. 2021.

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION - NASA. **Orion spacecraft**. Available from: https://www.nasa.gov/exploration/systems/orion/index.html. Access on: 01 feb. 2021.

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION - NASA. **Daniel S. Goldin**, Available from: https://www.hq.nasa.gov/office/pao/History/dan_goldin.html. Access on: 01 feb. 2021.

PARKES, S.; ARMBRUSTER, P.; SUESS, M. **The spacewire on-board data-handling network**. [S.l.]: ESA, 2005.

PENNA, S.; SOUZA, M.L.O. Estimating delays in switched ethernet networks on board of aerospace vehicles. **IEEE Latin America Transactions**, v.100, n. 1, 2020. Available from: https://latamt.ieeer9.org/index.php/transactions/article/view/3889. Access on: 01 feb. 2021.

PERLMAN, R. J. **Interconnections:** bridges, routers, switches, and internetworking protocols. 2.ed. Reading: Addison-Wesley Longman, 1992. 537p.

PETIT, J. L. **CAN bus for telecom avionics**. Torino: TAS, 2012. TAS-12-TL/IA/EA/D-29.

REIFEGERSTE, S. **CRC tester v1.2.** 2003. Available from: http://zorc.breitbandkatze.de/crctester.c. Access on: 01 feb. 2021.

SOCIETY OF AUTOMOTIVE ENGINEERS - SAE. **Digital time division command/response multiplex data bus AS15531**. Available from: https://www.sae.org/standards/content/as15531/. Access on: 01 feb. 2021.

SOCIETY OF AUTOMOTIVE ENGINEERS - SAE. **ARINC-629 multi-transmitter data bus parts 1 and 2**. Available from: https://www.aviation-ia.com/sae-search/content/629. Access on: 01 feb. 2021.

SPACE ETHERNETY PHYSICAL LAYER TRANSCEIVER - SEPHY. **Space Ethernet PHYsical Layer Transceiver**. Available from: http://www.sephy.eu/. Access on: 01 feb. 2021

SEIFERT, R. **The switch book**: the complete guide to LAN switching technology. Indianapolis: John Wiley & Sons, 2000. 698p.

STAKEM, P. H. **The history of spacecraft computers from the V-2 to the space station**. [S.l.]: Independent Publisher, 2017. 202 p.

TAGAWA, G. B. S.; SOUZA, M. L. O. An overview of the Intergrated Modular Avionics (IMA) concept. In: CONFERÊNCIA BRASILEIRA DE DINÀMICA, CONTROLE E APLICAÇÕES (DINCON). **Proceedings…** 2011. p. 277-280.

TANENBAUM, A. S. **Computer networks**. 4.ed. Englewod Cliffs: Prentice-Hall, 2002. 912p.

TELECOMMUNICATIONS INDUSTRY ASSOCIATION – TIA. **TIA/EIA RS-422-B**: electrical characteristics of balanced voltage digital interface circuits. Arlington, Virginia, 1994. 38p.

THOMAS, A. **An implementation of the token bucket algorithm in Python**. 2007. Available from: https://code.activestate.com/recipes/511490-implementation-of-the-token-bucket-algorithm/. Access on: 01 feb. 2021.

TTTECH. **Time-triggered protocol TTP/C high-level specification document protocol version 1.1**. Romenia, Nov. 2003.

TTTECH. **TTEthernet specification**. Romenia, Nov. 2008.

UNITED STATES DEPARTMENT OF DEFENSE. **AIM MIL-STD-1553 tutorial.** Freiburg, Germany: AIM GmbH, 2010.

VDOVIN, P. M.; KOSTENKO, V. A. Organizing message transmission in AFDX networks. **Programming and Computer Software**, v. 43, n. 1, p. 1–12, 2017.

VDOVIN, P. M. **AFDX designer**, Available from: https://github.com/PavelVdovin/AFDX_Designer. Access on: 01 feb. 2021.

W3C. **Extensible Markup Language (XML) 1.0**. 2008. Available from: https://www.w3.org/TR/2008/REC-xml-20081126/. Access on: 01 feb. 2021.

W3C. **XML schema**. 1998. Available from: https://www.w3.org/XML/Schema. Access on: 01 feb. 2021.

WATKINS, C. B.; WALTER, R. Transitioning from federated avionics architectures to Integrated Modular Avionics. In: AIAA/IEEE DIGITAL AVIONICS SYSTEMS CONFERENCE, 26., 2007, Dallas, Texas. **Proceedings…** 2007. DOI:10.1109/DASC.2007.4391842.

WEBB, E. Ethernet for space flight applications. In: IEEE AEROSPACE CONFERENCE, 2002, Big Sky, Montana. **Proceedings…** 2002. DOI: 10.1109/AERO.2002.1036905.

WENPING, W.; XIANGHUI, W.; BO, H.; YAHANG, Z.; YONG, L. Design of OBDH subsystem for remote sensing satellite based on onboard route architecture. In: MATEC WEB OF CONFERENCES, 139., 2017. **Proceedings…** 2017. DOI: 10.1109/TSSA48701.2019.8985466.

WERTZ, J. R. Assessment of smallsat utility and the need for dedicated, low-cost, responsive small satellite launch. In: AIAA RESPONSIVE SPACE CONFERENCE, 8., Los Angeles, 2010. **Proceedings…** 2010. AIAA-RS8-2010-5005.

WIRESHARK GENERIC DISSECTOR - WSDG. **Wireshark Generic Dissector.** Available from: http://wsdg.free.fr. Access on: 01 feb. 2021.

APPENDIX A – A NEW METHOD FOR ESTIMATING WORST CASE TRANSMISSION DELAY IN SWITCHED ETHERNET NETWORKS

Distributing computation and communication is now a common approach for simplifying the design of embedded systems used in aerospace vehicles (WATKINS et al, 2007). In distributed systems, similarly designed modules can be used to process data, as well as to capture data from sensors and to send commands to actuators. Such initiative can reduce the diversity of part-numbers and simplify parts replacement, just to mention two advantages (GASKA et al., 2015).

However, connecting modules in distributed systems can become quite complex with the increasing number of modules involved. In larger distributed systems, Ethernet physical medium has become a frequent choice (TTTECH, 2008) due to its high transmission speed and immense availability of off-the-shelf solutions, both in hardware (network transceivers) and in software (network protocols).

Ethernet can be used in point-to-point connections between modules, but its greatest advantage is to provide a very good means for implementing “star” topologies (MARTIN, 1994). In such topologies, the presence of one or more traffic switching device is essential for routing data transmissions from one module to another module or to many modules.

However, Ethernet does not provide any means of synchronizing data transmissions other than a physical medium access protocol called Carrier-Sense Multiple Access with Collision Detection, CSMA/CD for short (IEEE, 2012). This CSMA/CD protocol was essential when Ethernet offered only “bus” topologies (MARTIN, 1994) until mid 90’s. After the introduction of Ethernet switches(PERLMAN, 1999), the issue moved from controlling transmission collisions on a shared physical medium to the internal design of Ethernet switches and its advantages and disadvantages (SEIFERT, 2000).

If Ethernet itself does not provide a deterministic behavior on transmissions initiated by connected modules, a distributed processing system designer has to reach out for other means to make sure that the system will operate as required in all foreseeable conditions. For a viable design of such complex systems, it is

essential to estimate the worst possible end-to-end delay experienced by any data fragment traveling through the network, for it affects the nature and quantity of resources required by both hardware and software components involved in its processing and ultimately the behavior of aerospace vehicles.

The academic world has been providing such means since the beginning of the 2000's. Early studies used a method called "Network Calculus", which was published by René L. Cruz in 1991 (CRUZ, 1991) and was later formalized by Jean-Yves Le Boudec and Patrick Thiran in 2001 (LE BOUDEC et al., 2001).

Network Calculus provides a "fluid model" for the network traffic, whereby a data flow is assumed to be continuous in time as if it could be looked at as a liquid flowing inside a tube.

On most cases, Network Calculus principles provide a conservative means of estimating end-to-end delays in network data transmissions, but it is well accepted when distributed processing systems require certification by a designated authority, such as the Federal Aviation Administration (FAA, 2020) in the United States and the Europen Aviation Safety Administration (EASA, 2020) in Europe.

When crossing a traffic switch, Ethernet frame transmissions have to obey the switching fabric design. The most common commercial design is called store-and-forward (SEIFERT, 2000), in which incoming frames are received and stored up to their last bit before they get routed to one or more switch output ports. Taking into account that Ethernet transmissions, once started are not interrupted, two Ethernet frames going out through the same output port in a switch using store-and-forward fabric are transmitted one after the other. This effect was described as "serialization" by Henri Bauer (BAUER, 2011) and this is the effect used for constructing the scheduling criterion used by the new method.

Methods for estimating the amount of time a particular data frame takes to traverse a networked infrastructure has become a very rich field of research. A fairly large number of publications was produced since early 2000's (GRIEU, 2004) (GEORGE et al., 2005) (FRANCES et al., 2006); and even recently, in particular by a group linked to Henri Bauer (KEMAYO et al, 2013) (KEMAYO et

al, 2014) (KEMAYO et al, 2015). In fact, end-to-end network transmission delay estimation is one of the key design factors for developing modern distributed and highly integrated electronics on board of aerospace vehicles.

The search for a less conservative and mathematically less complex method for estimating the end-to-end delay for a particular message flow throughout a network infrastructure is the ground motivation of this work, especially if results by using such method can be obtained using commercial-off-the-shelf software tools.

The following sections present important remarks which are essential to the correct understanding of the new method, introduce two key propositions and two important definitions that form the basis of the method, present a simple procedure for implementing the transmission schedule generated by the method and compare the results obtained with the new method with a previously published work and those obtained using a publicly available software tool.

A.1 Remarks on network traffic pattern and bandwidth utilization

Network traffic

Network traffic in distributed systems used on board of aerospace vehicles tends to be rigorously constrained by design. That is, no transmitting node is expected to generate more network traffic than it is allowed to by design. Further, means of preventing a transmitting node to misbehave and of protecting the network from a misbehaving node are usually put in place through the implementation of network traffic shaping and policing algorithms (CISCO, 2020).

The network traffic presumed for this work does imply that: no spurious frame transmissions are expected and; all frame transmissions accounted for are those programmed by the designers of the distributed system.

Network utilization

Computer tasks developed for controlling flight of aerospace vehicles are usually periodic due to fact that the implemented algorithms work in discrete time. The majority of the tasks involved in input or output operations with an external device are also expected to be periodic, therefore there could have

been more than one instance of the same network frame in the time interval considered for the estimation of the longest delay of a particular frame. However, the scheduling criterion described in the next sections takes into account only one instance of each network frame in the analysis.

The concept of “utilization” in the context of schedulabilty analysis of a set of computer tasks was introduced by C. L. Liu and James W. Layland (LIU et al., 1973) and used by P.K. Harter (HARTER, 1984) and M. Joseph and P. Pandya (JOSEPH et al., 1986) for estimating the time a computer task needs to execute at least once, called “response time” of a task, in the presence of other higher priority tasks.

This utilization is expressed by the quotient C/T, where C is the time a task takes to execute without interference (“C” as in “Capacity”) and T is the task period. The maximum value of utilization for a schedulable computer task set is 1 (one).

When analyzing the response time of a computer task, the expression *ceiling*(Δt/T).C needs to be evaluated for each task with priority higher than the task being analyzed and added together, where Δt is the time interval being considered and *ceiling*() is the smallest integer equal or greater than its argument. The longest Δt is usually evaluated starting with the sum of the C of all tasks in the set and repeating the calculation iteratively until the value of Δt is stable.

In the context of transmitting network frames using Ethernet, if one considers C being the amount of time a frame takes to be transmitted on the physical medium, C will fall within a time interval from a few microseconds up to a few hundreds of microseconds. For instance, the minimum frame length admissible in Ethernet is 64 bytes and the maximum is 1518 bytes, and each frame transmission requires extra 20 bytes of overhead (IEEE, 2012). On a 100 megabit per second physical medium, the shortest frame take 6.72 microseconds and the longest frame take only 123.04 microseconds to be transmitted.

In the context of controlling flight of an aircraft, it is common to see tasks being executed 50 times per second, but in modern “fly-by-wire” systems this rate can

go up to 500 times per second. Controlling attitude and guidance in space vehicles requires much less frequent computations. So, task periods T for aircraft flight control applications usually vary from 2 to 20 milliseconds.

The time interval Δt in which network frames are expected to be transmitted by all tasks involved in flight control applications needs to be shorter than the shortest period of a typical task designed for the purpose, that is, less than its period T. In fact, Δt should not be more than a few hundreds of microseconds on a 100 megabit per second Ethernet physical medium, for Δt is in the order of magnitude of the sum of the C of all network frames involved.

If one rounds up the quotient of Δt/T to the first largest integer, in short the *ceiling*(Δt/T), using the orders of magnitude of Δt (in the order of 10^2 microseconds) and T (in the order of 10^3 microseconds) mentioned above, the expression *ceiling*(Δt/T) will be evaluated as 1. Therefore, it is reasonable to assume that one and only one instance of any network frame will be present within the time interval required for transmitting all network frames in a typical flight control application, which is the most critical application of all in a computer controlled aircraft or space vehicle.

A.1 Serialization of network traffic on a transmitting node

The first step into the estimation of end-to-end delay in a network such as those used in distributed systems on board aerospace vehicles is to bound the amount of interference one Ethernet frame transmission can cause to another Ethernet frame transmission going out of the same network node:

Proposition 1

On an Ethernet network node, any frame transmission can be delayed at most by the sum of the transmission times of all (including itself) Ethernet frames scheduled to be sent through the same Ethernet port.

Rationale

Ethernet devices, be it a network card or a switch, usually organize frames in a transmission queue and transmits one frame after the other until the queue is empty in a “fist-in-first-out” fashion.

Since Ethernet frames are transmitted serially, hence the term “serialization” coined by Bauer, each frame has to wait for the transmission of all other frames ahead of itself in the transmission queue and the last bit of the frame under observation has to wait until all other bits of the same frame go out into the physical medium.

For estimating frame transmission delays on Ethernet physical medium, it is important to account for the mandatory 8-byte long Preamble and Start-of-Frame-Delimiter (SFD) fields and the 12-byte long data transmission gap called Inter-Frame Gap (IFG) required by the Ethernet physical medium access protocol (IEEE, 2012), for these (8 + 12)x8 equal to 160 bit-times also occupy the physical medium exclusively.

For example, considering the transmission of two Ethernet frames of sizes 230 and 480 bytes on a physical medium speed of 100 megabits per second, any frame transmission is delayed by at most 60 microseconds, as follows:

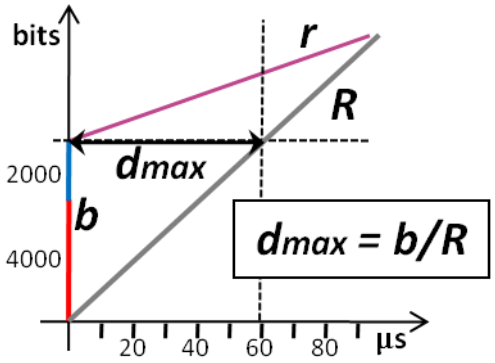
[(230 + 8 + 12 bytes) x 8 bit/byte]/100 bit/microsecond = 20 microseconds

[(480 + 8 + 12 bytes) x 8 bit/byte]/100 bit/microsecond = 40 microseconds

Note that the transmission order does not matter, since the last bit of any frame has to wait for the transmission of all other bits in the same frame and for the complete transmission of the frames ahead of itself in the transmission queue.

In this simple scenario, Network Calculus produces the same value of 60 microseconds for the “delay bound”. As defined in the work by Le Boudec and Thiran, this delay bound is the longest horizontal segment separating two functions as illustrated in Figure A.1: an “arrival curve”, modeled by a simple (b+r.t) linear function, and a “service curve”, modeled by a simple (R.t) linear function.

Figure A.1 – “Delay bound” as modeled by Network Calculus.



The delay bound for the sum of two flows, one for the 250 byte (= 2000 bits) long frame and the other for the 500 byte (= 4000 bits) long frame, is obtained by dividing the total length of the two frames (b=6000 bits) by the physical medium speed (R=100 bits per microsecond), as shown in Figure A.1, irrespective of the value of the constant rate of the flow (r) modeled by the arrival curve. In this simple case, the maximum calculated delay is 60 microseconds.

Some industry standards (ARINC, 2009) recommend adding a fixed delay to account for the latency that lower levels of software (or firmware) usually present for initiating and terminating an input or output operation on a physical device. The amount used as this fixed delay is completely arbitrary and, at best, should be the result of a careful analysis for each combination of software and hardware platforms.

To account for a fixed delay, this amount should be added to the delay calculated using Proposition 1 above

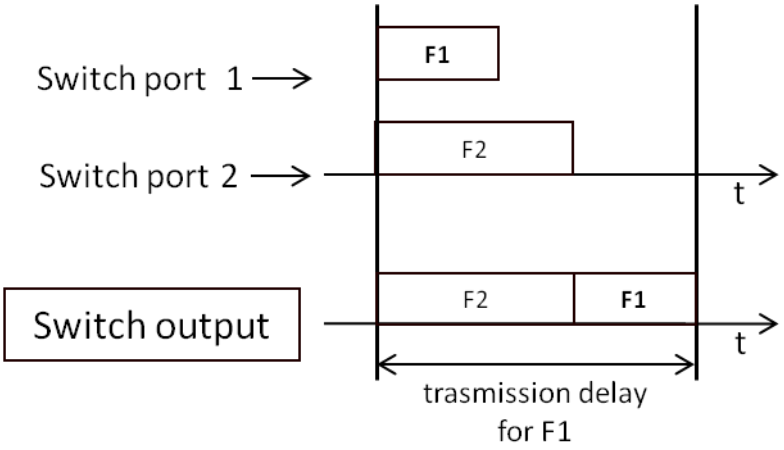
A.2 Serialization of network traffic on a switching device

The effect of serialization on a switching device is illustrated in Figures A.2, A.3 and A.4.

In Figure A.2, two incoming frames F1 and F2, received on switch input ports 1 and 2 respectively, are routed for going out through the same switch output port. This Figure A.2 depicts the time interval in which both frames are received and the time interval in which they are transmitted on the physical medium. For clarity, other latencies due to internal delays in the switching fabric are omitted. Note that the serialization effect in this case is the same described for a

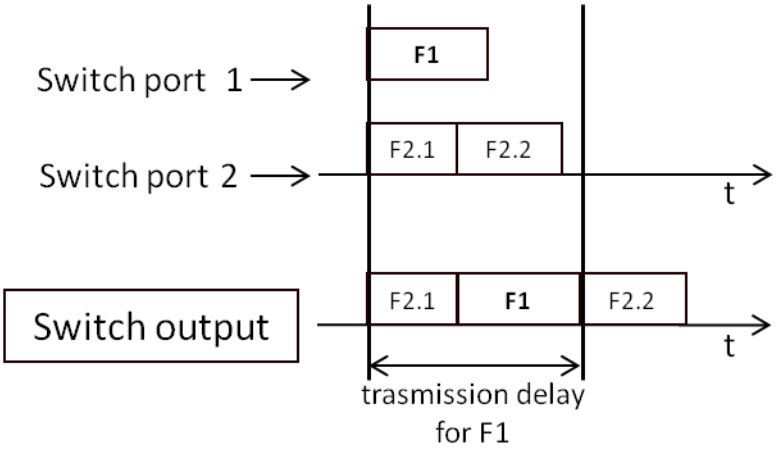
transmitting node in the previous paragraph and the transmission delay experienced by any frame follows Proposition 1.

Figure A.2 – Switch output for two incoming frames F1 and F2.



Assume that, instead of one single F2 frame coming on switch port 2, two shorter frames F2-1 and F2-2, for which the total transmission time is the same as for F2, come in as shown in Figure A.3. In this situation, frame F1 has apparently the same amount of traffic ahead in the switching fabric. However, due to serialization, frame F1 will suffer interference from either frame F2-1 or from frame F2-2, but not from both.

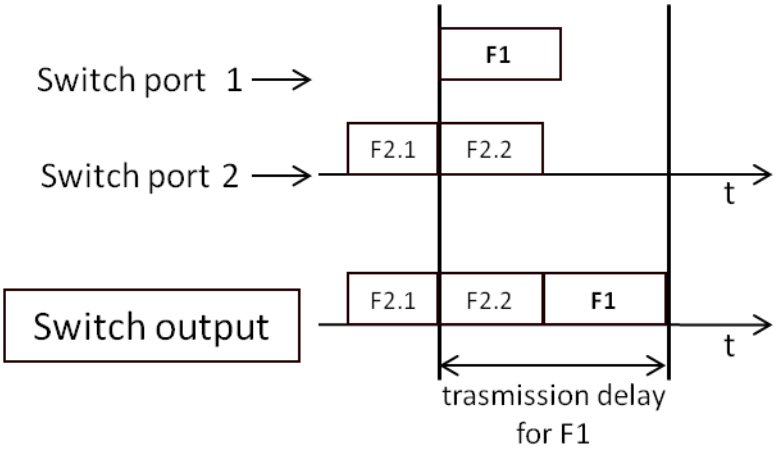
Figure A.3 – Switch output for F2 split in two shorter frames.



Frame F1 first bit arrival time may coincide with the first bit arrival time of either F2-1 as shown in Figure A.3 or F2-2 as shown in Figure A.4. In either case, frame F1 will have to wait until either frame F2-1 or frame F2-2 gets completely transmitted before the switching fabric starts its transmission. In this alternative

scenario, frame F1 will be delayed by at most its own frame transmission time and by the longest transmission time between frames F2-1 and F2-2.

Figure A.4 – Switch output for frame F2 (alternative scenario).



In this “split-frame” scenario, frame F1 benefits from the fact that frames F2-1 and F2-2 have been “serialized” by the node which originated them

A.3 Definition of the “critical instant” in a switching fabric

The term “critical instant” was originally coined by Liu and Layland (1973) in the context of schedulability analysis for a set of computer tasks:

A critical instant for a task is defined to be an instant at which a request for that task will have the largest response time.

For the context of scheduling frame transmissions in a switching fabric, the “critical instant” in Liu and Layland’s text is here slightly redefined:

Definition 1

The “critical instant” for a frame is defined as the arrival instant of its first bit at which the frame will experience its longest transmission delay.

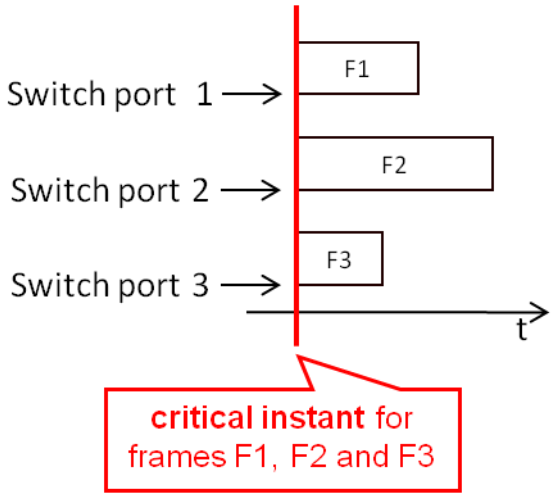
Rationale

It is important to note that: 1) computer tasks in aerospace applications are usually scheduled according to some priority assignment scheme and tasks with higher priority are scheduled first for execution; 2) the traditional Ethernet switching fabric would queue frames for transmission on a particular switch output port in the order they arrive on a “First-Come-First-Served” basis; 3) in the context of building the worst-case scenario for scheduling frames for

transmission, it is assumed here that the frame under observation has always the “lowest priority”, that is, it is scheduled to be transmitted after all other frames sharing the same switch output port are transmitted, despite their arrival order.

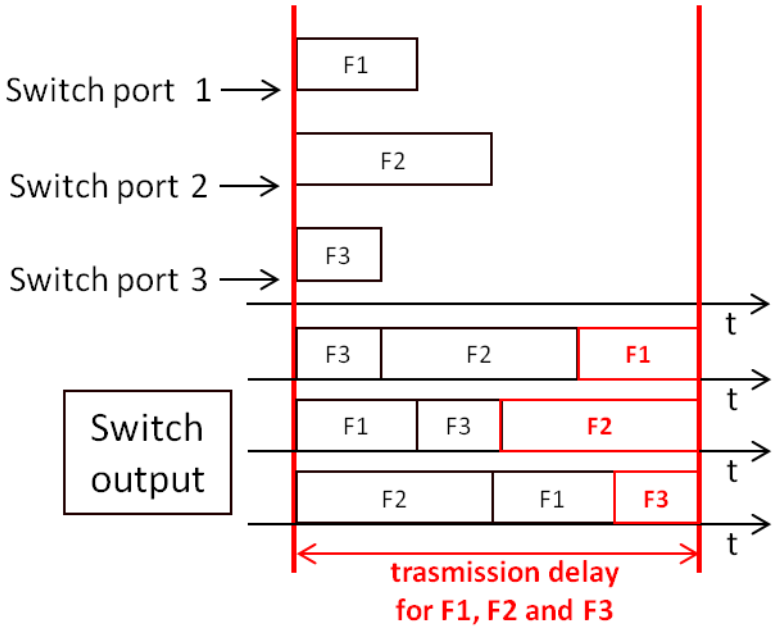
In Figure A.5, frames F1, F2 and F3 received on different switch input ports have their critical instant when the first bit of all frames are received at the same instant in time.

Figure A.5 – “Critical instant” for incoming frames F1, F2 and F3.



Once the critical instant is defined for this simple scenario, the longest transmission delay for either frame in Figure A.5 is simply the time interval taken for transmitting frames F1, F2 and F3 as partially illustrated in Figure A.6. Actually, there are (*n*-1)! possible permutations for each of the *n* frames chosen to be the last one transmitted.

Figure A.6 – Transmission delay for frames F1, F2 and F3.



A.4 Definition of “transmission backlog” in a switching fabric

Assume that multiple frames arrive on different switch input ports and on different instants in time and are scheduled to be transmitted through the same switch output port.

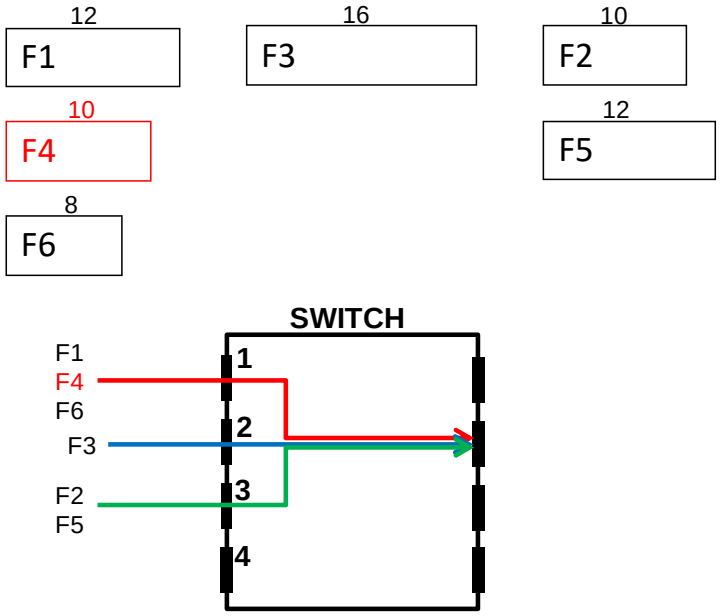
Definition 2

Once the “critical instant” for a particular frame is defined, the remaining frames not involved in its construction build the “transmission backlog” for the associated switch output port, given that particular “critical instant”.

Rationale

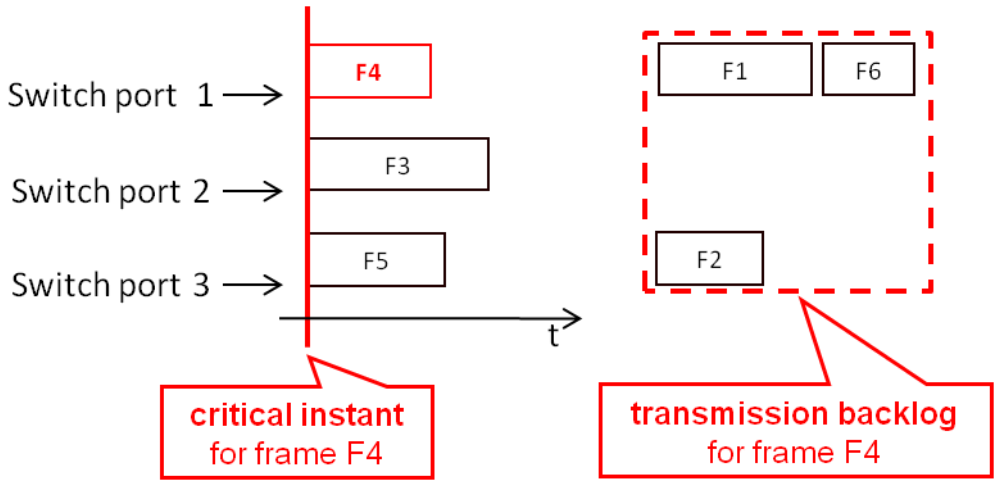
On Figure A.7, frames F1, F4 and F6 (length of F1 being the longest) are received on switch port 1, frame F3 is received on switch port 2 and frames F2 and F5 (length of F5 being the longest) are received on switch port 3, all scheduled for transmission through the same switch output port. Assume that the frame under observation is frame F4.

Figure A.7 – Frames F1, F2, F3, F4, F5 and F6 illustrated.



The critical instant for frame F4 is built by taking the longest frame from each of the other input ports and aligning in time the arrival instant of their first bits. The remaining frames F1, F6 and F2 represent the “transmission backlog” for frame F4 at that particular switch output port, given the particular critical instant built for this frame F4, as shown in Figure A.8.

Figure A.8 – “Critical instant” and “transmission backlog” for frame F4.



Should frames F4, F3 and F5 be the only frames being transmitted, once the critical instant is built for frame F4, the longest transmission delay for this frame F4 could be estimated as done in Figure A.8 above. However, the existence of

a transmission backlog affects the transmission order of the remaining frames in the set under observation and ultimately affects the transmission delay of frame F4.

The amount of interference caused by the frames in the transmission backlog is examined in the next section.

A.5 Worst-case scenario for the “transmission backlog”

Once the critical instant for frame F4 is built, the arrival of all other frames in the corresponding transmission backlog for the particular switch output port shall be arranged in a way that F4 will experience its longest transmission delay.

Proposition 2

Given a particular “critical instant”, the frame under observation will experience its longest transmission delay when the frames belonging to the associated “transmission backlog” are arranged arriving before the frames participating in the “critical instant” at each switch input port in such a way that the arrival instant of the last bit of any frame is immediately followed by the arrival instant of the first bit of the next frame.

Rationale

In Figure A.9, the transmission backlog for frame F4’s critical instant is built with frames F1 and F6 arriving before F4, and frame F2 arriving before frame F5. Frame F4 will experience its longest transmission delay when the transmission backlog is built as follows: a) the arrival instant of the last bit of frame F1 is immediately followed by the arrival instant of the first bit of frame F6; b) the arrival instant of the last bit of frame F6 is immediately followed the arrival instant of the first bit of frame F4; c) the arrival instant of the last bit of frame F2 is immediately followed by the arrival instant of the first bit of frame F5. Since no other frames arrive on the same input port of frame F3, there is no contribution to the transmission backlog from switch input port 2.

Figure A.9 – Worst case scenario for frame F4.

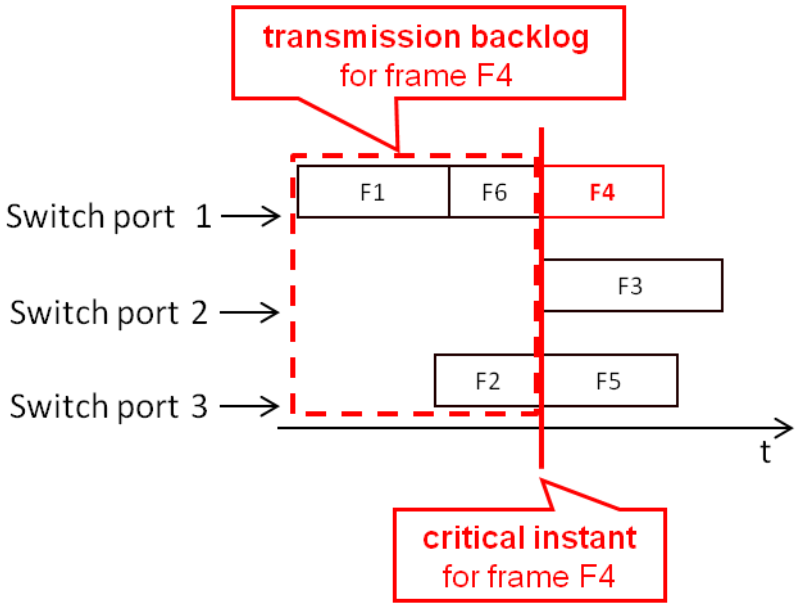


Figure A.9 illustrates how the “serialization” effect is materialized on switch input ports 1 and 3. In short, Proposition 2 says that frames belonging to the transmission backlog should be “serialized” with the frames participating in the critical instant at each switch input port.

A.6 Procedure for building the worst-case frame transmission schedule

Given a set of frames arriving on different switch input ports and scheduled to be output through the same switch output port, the following steps use the critical instant and the transmission backlog for a frame chosen from this set to build the transmission schedule in which that particular frame will experience its longest transmission delay:

NOTE: Once a frame is selected (the frame under observation), the switch input port on which the selected frame arrives is referred to as the “selected port” in this section.

1. The selected frame should be the last frame transmitted;

2. On all switch input ports other than the selected port, select the longest frames;

3. The selected frame and the other frames selected in the previous step shall have the arrival instant of their first bits aligned in time; this step builds the critical instant for the selected frame;

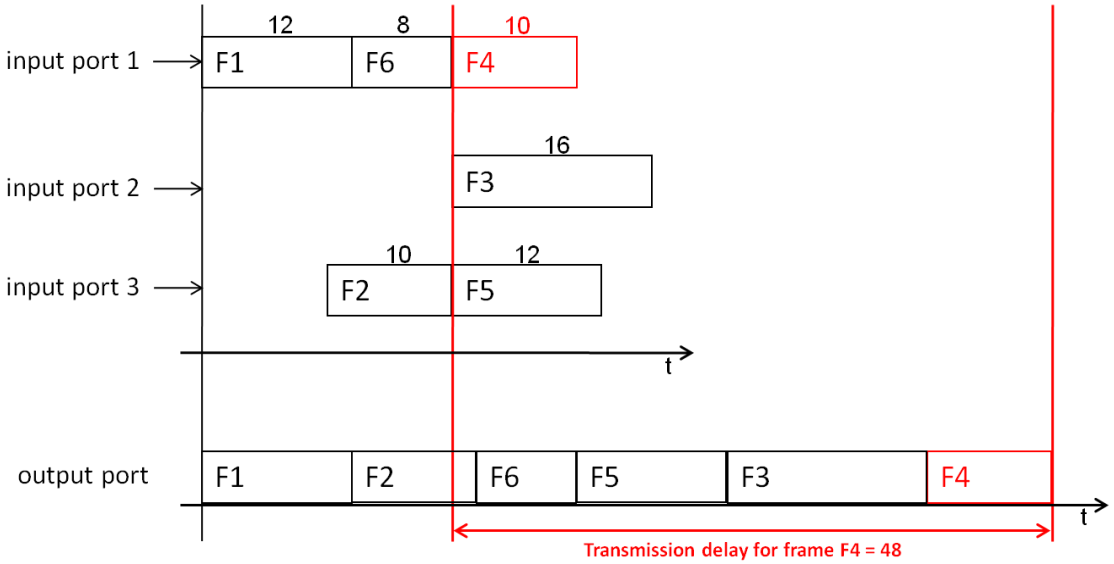
4. Frames remaining from steps 1 and 2 on the selected port and on each of the other switch input ports build the transmission backlog for the selected frame;

5. Frames are expected to arrive “back-to-back” on the same input port, that is, there is no transmission gap between the arrival instant of the last bit of a frame and the arrival instant of the first bit of the next frame;

6. Frames shall be scheduled for transmission starting with the frame with the earliest arrival instant, then picking frames from each of all switch input ports moving forward in time.

This step-by-step procedure was used for scheduling the transmission of frames F1, F2, F6, F5, F3 and finally F4 in such a way that frame F4 will experience its longest transmission delay, as shown in Figure A.10. It should be noted that in this scenario, the order of arrival of frames F1 and F6 is irrelevant to the composition of the transmission delay for frame F4 (F6 might as well have arrived before F1).

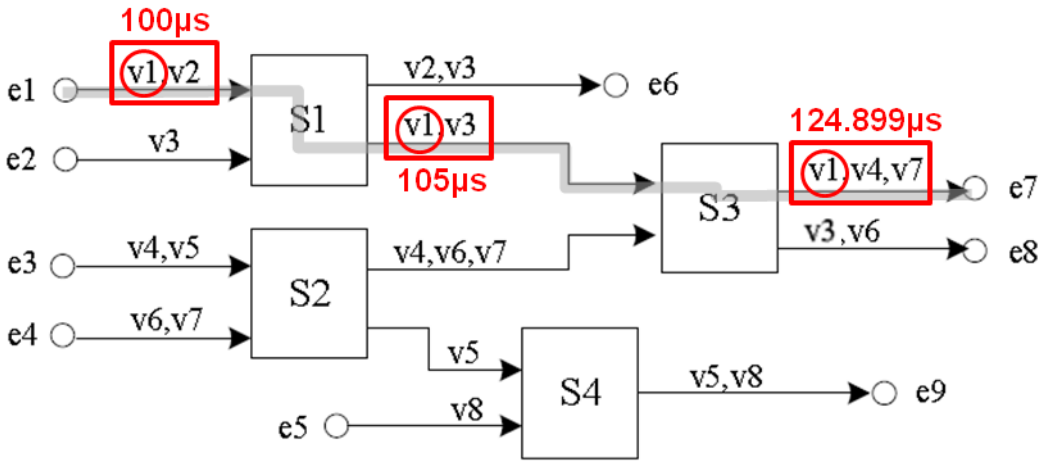
Figure A.10 – Transmission schedule for frame F4.



A.7 Comparing results with network calculus on a simple case

In a work published in 2013, Zhao, Li, Xiong, Zheng and Xiong, all from Beihang University, Beijing, P. R. of China, introduced a very interesting approach for taking “serialization” into account for the estimation of end-to-end transmission delays in complex networked systems when using Network Calculus (ZHAO et al., 2013). The authors indicate that their proposition reduces the overestimation produced by the original Network Calculus approach when a data flow crosses several network elements. In Figure A.11, the results published for the flow called v1 in its path through the network from node e1 to node e7 crossing switches S1 and S3 are reproduced. Frames in all flows have the same length 5000 bits, therefore the same transmission time of 50 microseconds on a 100 megabit per second physical medium. Flows have the same constant transmission rate of 10 bits per microsecond and can be expressed analytically, according to Network Calculus, as a linear function (5000 + 10t) where 5000 is a “burst” of the size of the longest frame.

Figure A.11 – Network analyzed by Zhao et al. (2013).

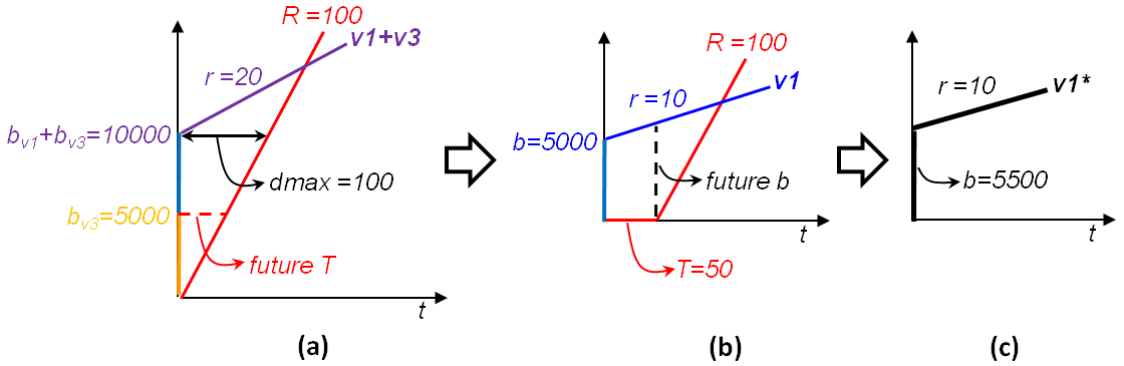


Note that v1 exiting node e1 suffers interference only from v2 and, as expected, the delay experienced by v1 is simply the time taken to transmit one frame of v1 and one frame of v2, 100 microseconds total as shown in Figure A.12(a). This is in-line with Proposition 1.

However, according to the authors when adopting principles of Network Calculus, flow v1 has to be modified when exiting e1, because it takes into account an increase in burst equal to the amount of v1 traffic accumulated

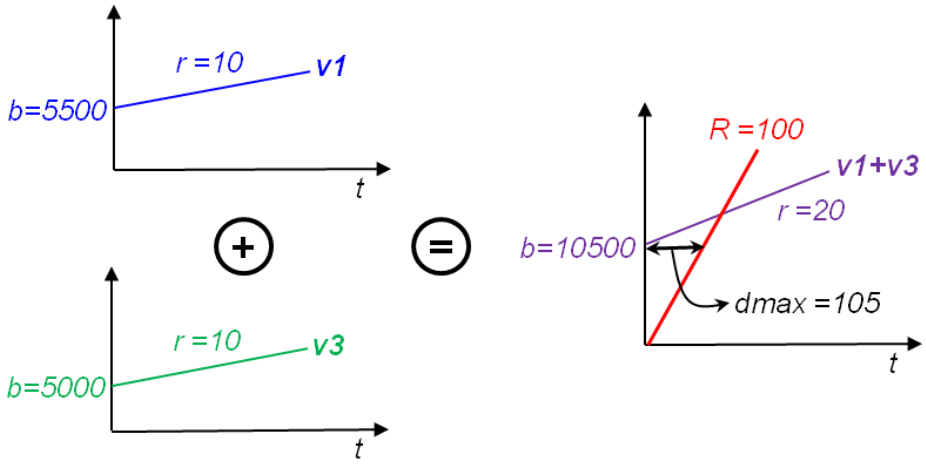
during the 50 microseconds node e1 takes transmitting a frame of v2. Figure 12(b) shows flow v1 modeled as (5000+10t) serviced by a function modeled as $100(t-50)^{+}$. This interference caused by v2 on v1 when exiting e1 results in an output flow v1* with a burst of 5500 bits (the original 5000 bits plus 10 bits per microsecond accumulated for 50 microseconds), as shown in Figure A.12(c).

Figure A.12 – Output flow v1* for v1 exiting e1 (not in scale).



This new flow v1* in turn suffers interference from flow v3 when exiting S1 and the delay of 105 microseconds calculated using Network Calculus is illustrated in Figure A.13.

Figure A.13 – Maximum delay of v1* exiting S1 (not in scale).



The authors calculate the delay for v1 exiting S1 to be 105 microseconds, even when one considers that v1 suffers interference only from v3. By Proposition 1, the delay experienced by v1 should be also 100 microseconds, since the scenario for v1 exiting S1 is similar to the scenario for v1 exiting e1.

The delay estimation for v1 exiting S3 is far more complex, because “serialization” comes into play. One has to model the flows of v4 exiting e3, v6

and v7 exiting e4, then “serializing” v6 and v7 before “adding” to v4 while crossing S2 towards S3, then “serializing” v4 and v7 before “adding” to v1 while crossing S3 towards e7. The result published by the authors is 124.899 microseconds.

Using Proposition 2 and noting that v3 and v6 are routed to a different switch output port, v1 should suffer interference either from v4 or from v7 (not from both!) as shown in Figure A.14, as they get “serialized” exiting S2. Therefore, Proposition 2 calculates 100 microseconds for the delay for v1 exiting S3, as shown in Figure A.15.

Figure A.14 – Transmission schedule scenarios for frame v1.

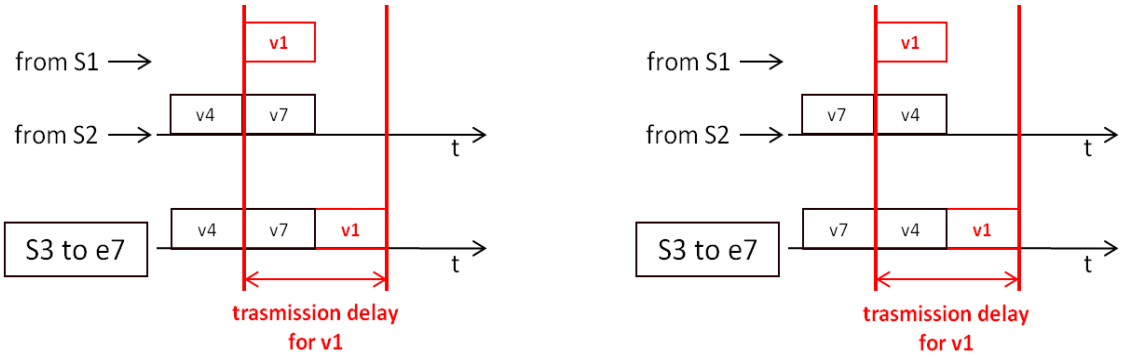
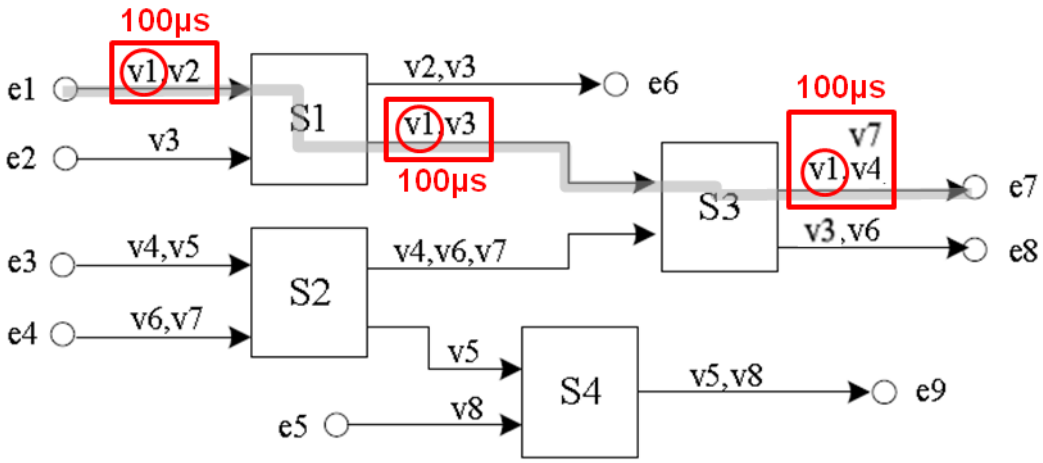


Figure A.15 – End-to-end delay for frame v1 under Proposition 2.



A.8 Comparing results with the “AFDX_Designer”

Tools that implement methods of estimating end-to-end delays in embedded networks are seldom made public. However, there is one called “AFDX_Designer” (VDOVIN et al., 2017) available through *github* (VDOVIN,

2020) which, according to the author, implement a variation of the Trajectory Approach introduced by Bauer (2011). The tool can be configured to mimic the same network analyzed in the previous section, in particular by setting the delay experienced when crossing a switch to 0.

The AFDX_Designer tool follows the ARINC-664 Part 7 standard (ARINC, 2009), which relies on a switched Ethernet/IP/UDP network topology and on a virtual, rate-constrained communication channel called “Virtual Link” (vl). A vl allows transmission of at most “Lmax” bytes every “Bandwidth Allocation Gap” (BAG) milliseconds.

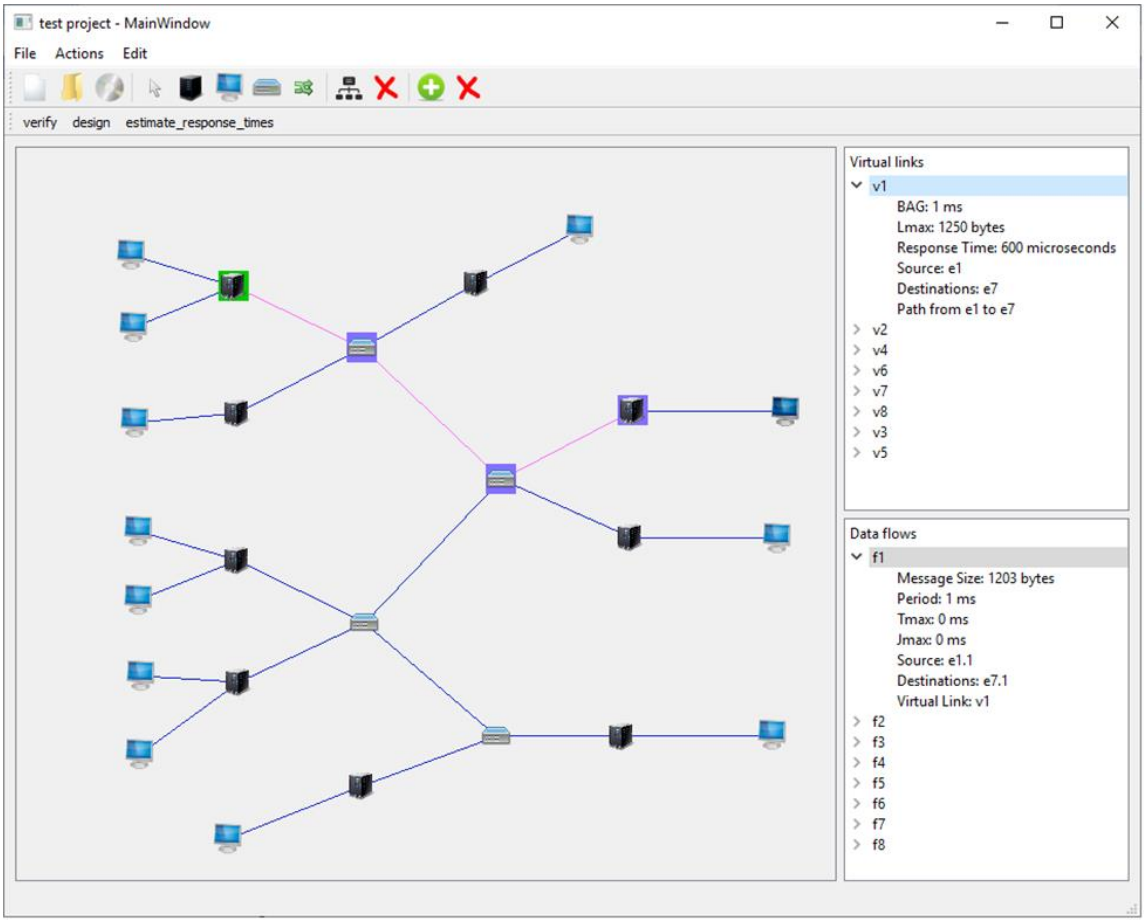
The ARINC-664 Part 7 standard does not allow for a BAG shorter than 1 millisecond, therefore a vl in the AFDX_Designer would need to be created with Lmax of 1250 bytes (twice the original length of 5000 bits per frame divided by 8) and a BAG of 1 millisecond (twice the period resulting from 5000 bits per frame divided by a constant transmission rate of 10 bits per microsecond). The result of this maneuver is that all calculated end-to-end transport delays (indicated as “Response time” in the “Virtual links” window in the tool) will appear twice the actual value.

The result obtained by the same flow v1 analyzed in previous section is shown in Figure A.16, a screen copy of the AFDX_Designer tool output with a few extra annotations to help correlating the elements in the graphic window of the tool with the network of Figure A.11.

The elements e1.1, e1.2, e2.1 and so on, are the “partitions”, operating system entities which produce or consume the messages transported by “flows”. The elements e1, e2, e3 and so on, are the “End-Systems”, ARINC-664 Part 7 network nodes which process the outgoing or incoming “flows”. The elements S1, S2, S3 and so on, are the network switching devices.

The “flow” f1 uses Virtual Link v1, which goes from e1 to e7 as in Figure A.11 carrying messages from e1.1 to e7.1.

Figure A.16 – End-to-end delay for flow v1 using “AFDX_Designer”.



The value 600 microseconds calculated by the tool as “Response Time” for v1 on its path from node e1 to node e7 is, as expected, twice the value obtained in the previous section using Proposition 1 and Proposition 2.

A.9 Closing remarks

The new method for estimating the longest transmission delay experienced by a data frame traveling across a switched network topology relies on a scheduling criterion that explores the “serialization” effect observed in frame based network transmissions

This criterion is materialized using two propositions: 1) the amount of delay experienced by any frame is simply the time interval necessary to transmit all frames in the transmitting queue onto the network physical medium; 2).network frames suffer the effect of “serialization” when the network element is a network traffic switching device.

The scheduling criterion depends on two definitions:

- The “critical instant” for a frame is defined as the arrival instant of its first bit at which the frame will experience its longest transmission delay;
- Once the “critical instant” for a particular frame is defined, the remaining frames not involved in its construction represent the “transmission backlog” for the associated switch output port, given that particular “critical instant”.

This new method for estimating the transmission delay for a network frame while crossing an Ethernet switch does not take into account other delays due to inner nature of the switching fabric. The only assumption is that the switch operates “store-and-forward”, as most commercial and industrial switches do, and not “cut-through” (SEIFERT, 2000).

Other details about the method can be found in reference Penna et al. (2020).

APPENDIX B – CONFIGURING A WIRESHARK GENERIC DISSECTOR

The Wireshark tool (WSGD, 2020) provide a means of parsing Ethernet frames using what is called a “dissector”. Some of these “dissectors” are standard, such as those which parse the IP protocol and the other Transport Layer protocols over IP.

The new Data Link Layer protocol is an extension of the IEEE 802.2 Logical Link Control, therefore its “dissector” can be built using the standard LLC “dissector” and configuring two text files for directing additional frame parsing.

These two text files are:

- `wsgd` – the “generic dissector” file;
- `fdesc` – the data format description file.

The text files specifically built for the test case of the new LLC extended, IEEE 802.2 Data Link Layer protocol, are displayed in the next sections. The `wsgd` file contains only the SAP numbers 114 and 116 for limiting the range of SAP numbers that should be identified as belonging to the new LLC extended protocol.

Directions on how to build these files can be found in the reference (WSGD, 2020). The installation of the “generic dissector” binary and text files depends on the type and version of the operating system used.

B.1 WSGD file for the extended LLC Data Link Layer protocol

(filename: `llce.wsgd`)

```
# Debug global flag
DEBUG

# Protocol's names.
PROTONAME          LLCE Protocol
PROTOSHORTNAME     LLCE
PROTOABBREV        llce

# Specify when the dissector is used.
PARENT_SUBFIELD           llc.dsap
PARENT_SUBFIELD_VALUES    114 116

# Message's header type.
# The message must begin by the header.
```

```
# The header must contains MSG_ID_FIELD_NAME and any
MSG_SUMMARY_SUBSIDIARY_FIELD_NAMES.
MSG_HEADER_TYPE       T_UI_header

# Field which permits to identify the message type.
# Must be part of MSG_HEADER_TYPE.
MSG_ID_FIELD_NAME     sn

# The main message type.
# Must begins by the header.
# This is a switch case depending on
#  the MSG_ID_FIELD_NAME field which is inside
MSG_HEADER_TYPE.
MSG_MAIN_TYPE T_UI_msg

# Definitions of the packet's format.
PROTO_TYPE_DEFINITIONS

# Debug flag only for interpreting the types
DEBUG

include  llce.fdesc;
```

B.2 FDESC file for the extended LLC Data Link Layer protocol

(filename: `llce.fdesc`)

```
############################################################
# Header
############################################################

struct  T_UI_header
{
  uint8 sn;
  uint20{d=hex}{byte_order=big_endian} timestamp;
  uint12{d=hex}{byte_order=big_endian} crc12;
}

############################################################
# Basic types
############################################################

struct T_UI_msg {
  T_UI_header snheader;
  raw(*) end_of_msg;
}
```

APPENDIX C – NODE CONFIGURATION FILES

The node configuration files used by the modules developed for the test case were composed using XML (W3C, 2020).

There are two XML files, one for node CPM1 and one for node CPM2. Their full texts are copied in the next two sections.

C.1 Configuration file for node CPM1

```
<configuration name="TEST001" host="CPM1">
<nodes>
  <node name="CPM1">
    <hexid>341</hexid>
    <unit>1</unit>
  </node>
  <node name="CPM2">
    <hexid>341</hexid>
    <unit>2</unit>
  </node>
</nodes>
<SAPs>
  <number>114</number>
</SAPs>
<channels>
  <channel id="1">
    <SSAP>114</SSAP>
    <DSAP>116</DSAP>
    <capacity>90</capacity>
    <tokens>60</tokens>
    <rate>30</rate>
  </channel>
</channels>
<services>
  <SAP number="116">
    <host>CPM2</host>
  </SAP>
</services>
<ports>
  <port number="1">
    <endp hops="0">CPM2</endp>
  </port>
</ports>
</configuration>
```

C.2 Configuration file for node CPM2

```
<configuration name="TEST001" host="CPM2">
<nodes>
  <node name="CPM2">
    <hexid>341</hexid>
    <unit>2</unit>
  </node>
  <node name="CPM1">
    <hexid>341</hexid>
    <unit>1</unit>
  </node>
</nodes>
<SAPs>
  <number>116</number>
</SAPs>
<channels>
  <channel id="1">
    <SSAP>114</SSAP>
    <DSAP>116</DSAP>
    <capacity>90</capacity>
    <tokens>60</tokens>
    <rate>30</rate>
  </channel>
</channels>
<services>
  <SAP number="114">
    <host>CPM1</host>
  </SAP>
</services>
<ports>
  <port number="1">
    <endp hops="0">CPM1</endp>
  </port>
</ports>
</configuration>
```

APPENDIX D – SOURCE CODE LISTINGS

The full source code listings in C Language for the modules CPM1 and CPM2 developed for the test case are copied the next two sections.

D.1 Source code for the CPM1 module

```
#include <stdio.h>
#include <stdlib.h>

#include <time.h>
#include <unistd.h>

#include <string.h>

#include "chndef.h"

#include "pcap.h"
#include "IEEE802UI.h"

NIBL myNIBL;
CIB myCIB;
NIB myNIB;

CCBL myCCBL;
SIBL mySIBL;

SHB mySHB;

PAB p1PAB;
PABL myPABL;

CBB myCBB;

pcap_hdr_t pcap_hdr;
pcaprec_hdr_t pcaprec_hdr;

IEEE802UI myUI;

char outfile[] = "UI.pcap";
FILE *poutFile = NULL;
```

```
struct timeval now;
struct timeval then;

#define FALSE 0
#define TRUE !(FALSE)

int powerUP = TRUE;

void bswap16(char *input)
{
  unsigned char tmp;

  tmp = input[1];
  input[1] = input[0];
  input[0] = tmp;
}

void bswap32(char *input)
{
  unsigned char tmp;

  tmp = input[0];
  input[0] = input[3];
  input[3] = tmp;

  tmp = input[1];
  input[1] = input[2];
  input[2] = tmp;
}

/* ------------------------------------------------------------------------------------------ */
/* ---------------------------------------- initialize()------------------------------------- */
int fillNIBCIB()
{
  int i;

  myNIBL.nibcnt = 2;
```

```
  myNIBL.p2nib[0].nname[0] = '1MPC';
  myNIBL.p2nib[0].nname[1] = 0x00202020;
  myNIBL.p2nib[0].macb[0] = 0x00000000;
  myNIBL.p2nib[0].macb[1] = 0xAA000500;
  myNIBL.p2nib[0].hexid = 0x341;
  myNIBL.p2nib[0].macb[0] |= 0x41000000;
  myNIBL.p2nib[0].macb[1] |= 0x00000003;
  myNIBL.p2nib[0].unit = 1;
  myNIBL.p2nib[0].macb[0] |= 0x00100000;

  myNIBL.p2nib[1].nname[0] = '2MPC';
  myNIBL.p2nib[1].nname[1] = 0x00202020;
  myNIBL.p2nib[1].macb[0] = 0x00000000;
  myNIBL.p2nib[1].macb[1] = 0xAA000500;
  myNIBL.p2nib[1].hexid = 0x341;
  myNIBL.p2nib[1].macb[0] |= 0x41000000;
  myNIBL.p2nib[1].macb[1] |= 0x00000003;
  myNIBL.p2nib[1].unit = 2;
  myNIBL.p2nib[1].macb[0] |= 0x00200000;

  printf("\n>> Number of nodes %d", myNIBL.nibcnt);
  for(i=0; i<myNIBL.nibcnt; i++)
  {
    printf("\n>> Node name '%s' Equipment ID = 0x%x   Unit = %d MAC Base = %x %x", myNIBL.p2nib[i].nname,
myNIBL.p2nib[i].hexid, myNIBL.p2nib[i].unit, myNIBL.p2nib[i].macb[1], myNIBL.p2nib[i].macb[0]);
  }

  myCIB.cfgname[0] = 'TSET';
  myCIB.cfgname[1] = 0x00313030;
  myNIB = myNIBL.p2nib[0];
  myCIB.p2nib = &myNIB;

  printf("\n>> Configuration name '%s' Host NIB address = %x Host name from NIB List '%s' Host name from CIB '%s'",
myCIB.cfgname, myNIB, myNIB.nname, myCIB.p2nib->nname);

  return 0;
```

```
}

int fillCCBSIB()
{
  int i;

  myCCBL.ccbcnt = 1;

  myCCBL.ccbi[0].chnid = 1;
  myCCBL.ccbi[0].ssap = 114; // 0x72
  myCCBL.ccbi[0].dsap = 116; // 0x74
  myCCBL.ccbi[0].capacity = 90; // token bucket limit with 50% overdraft
  myCCBL.ccbi[0].tokens = 60;   // token buckt initial filling equal 38 bytes of message + 22 bytes of LLCE header
  myCCBL.ccbi[0].rate = 30;     // token bucket fill rate equal [].tokens/2
  myCCBL.ccbi[0].pid = 0;
  myCCBL.ccbi[0].sn = 0;
  myCCBL.ccbi[0].timestmp = 0;

  printf("\n>> Number of channels %d", myCCBL.ccbcnt);

  for(i=0; i<myCCBL.ccbcnt; i++)
  {
    printf("\n>> Channel position %d, ID = %d, SSAP = %d, DSAP = %d, Token Bucket capacity/tokens/rate [in
bytes/sec] = %d/%d/%d, PID = %x, SN = %d, Time-stamp = %d", i,
      myCCBL.ccbi[0].chnid,
      myCCBL.ccbi[0].ssap,
      myCCBL.ccbi[0].dsap,
      myCCBL.ccbi[0].capacity,
      myCCBL.ccbi[0].tokens,
      myCCBL.ccbi[0].rate,
      myCCBL.ccbi[0].pid,
      myCCBL.ccbi[0].sn,
      myCCBL.ccbi[0].timestmp);
  }

  mySIBL.sibcnt = 1;
```

```
  mySIBL.sibi[0].sap = 114;
  mySIBL.sibi[0].pid = 0;

  for(i=0; i<mySIBL.sibcnt; i++)
  {
    printf("\n>> Service position %d, SAP = %d, PID = %x", i,
      mySIBL.sibi[0].sap,
      mySIBL.sibi[0].pid);
  }

  return 0;
}

int fillSHB()
{
  int i;

  mySHB.shbcnt = 1;
  mySHB.filler = 0;

  mySHB.shbi[0].sapnumber = 116;
  mySHB.shbi[0].hostp2nib = &myNIBL.p2nib[1]; //NIB pointing to CPM2

  printf("\n>> Number of hosted services %d", mySHB.shbcnt);

  for(i=0; i<mySHB.shbcnt; i++)
  {
    printf("\n>> Hosted service position %d, SAP = %d, Host name from NIB '%s'", i,
      mySHB.shbi[i].sapnumber,
      mySHB.shbi[i].hostp2nib->nname);
  }

  return 0;
}

int fillPABL()
{
```

```
int i;
int j;
PAB *tmpPAB;

p1PAB.nibcnt = 1;
p1PAB.filler = 0;

p1PAB.pabi[0].p2nib = &myNIBL.p2nib[1]; //NIB pointing to CPM2
p1PAB.pabi[0].hops = 0;

for(i=0; i<p1PAB.nibcnt; i++)
{
  printf("\n>> Port 1: hops = %d to node '%s' from NIB",
    p1PAB.pabi[i].hops,
    p1PAB.pabi[i].p2nib->nname);
}

myPABL.pabcnt = 1;
myPABL.filler = 0;

myPABL.pabli[0].port = 1;
myPABL.pabli[0].p2pab = &p1PAB; //Port 1 to CPM2

for(i=0; i<myPABL.pabcnt; i++)
{
  tmpPAB = myPABL.pabli[i].p2pab;
  printf("\n...NIBs From PAB List...");
  for(j=0; j<tmpPAB->nibcnt; j++)
  {
    printf("\n>> Port %d: hops = %d to node '%s'",
      myPABL.pabli[i].port,
      tmpPAB->pabi[j].hops,
      tmpPAB->pabi[j].p2nib->nname);
  }
}

return 0;
```

```
}

int fillCBB()
{
  myCBB.cibb = &myCIB;
  myCBB.nibb = &myNIBL;
  myCBB.sibb = &mySIBL;
  myCBB.ccbb = &myCCBL;
  myCBB.shbb = &mySHB;
  myCBB.pabb = &myPABL;

  printf("\n>> Configuration name '%s' Host name from CIB '%s'", myCBB.cibb->cfgname, (myCBB.cibb->p2nib)->nname);

  return 0;
}

int initialize()
{
  int retcode;

  printf(">> Filling NIBs and CIB...");
  retcode = fillNIBCIB();

  printf("\n>> Filling CCBs and SIBs...");
  retcode = fillCCBSIB();

  printf("\n\n>> Filling SHBs...");
  retcode = fillSHB();

  printf("\n\n>> Filling PABs and PAB List...");
  retcode = fillPABL();

  printf("\n\n>> Filling CBB...");
  retcode = fillCBB();

  printf("\n\n>> End of initialize()...");
  return retcode;
```

```
}
/* ---------------------------------------- initialize()---------------------------------------- */
/* ------------------------------------------------------------------------------------------- */

/* ------------------------------------------------------------------------------------------- */
/* ------------------------------------- CRC calculation ------------------------------------- */
// CRC parameters (values for CRC-32 Koopman G(x) = (x+1)(x3+x2+1)(x28+x22+x20+x19+x16+x14+x12+x9+x8+x6+1):
// CRC parameters (values for CRC-12 Koopman G(x) = x12 +x8 +x7 +x6 +x5 +x4 +1):

int order;
unsigned long polynom;
int direct;
unsigned long crcinit;
unsigned long crcxor;
int refin;
int refout;

// 'order' [1..32] is the CRC polynom order, counted without the leading '1' bit
// 'polynom' is the CRC polynom without leading '1' bit
// 'direct' [0,1] specifies the kind of algorithm: 1=direct, no augmented zero bits
// 'crcinit' is the initial CRC value belonging to that algorithm
// 'crcxor' is the final XOR value
// 'refin' [0,1] specifies if a data byte is reflected before processing (UART) or not
// 'refout' [0,1] specifies if the CRC will be reflected before XOR

// internal global values:

unsigned long crcmask;
unsigned long crchighbit;
unsigned long crcinit_direct;
unsigned long crcinit_nondirect;

// subroutines

unsigned long reflect (unsigned long crc, int bitnum)
{
```

```
  // reflects the lower 'bitnum' bits of 'crc'

  unsigned long i, j=1, crcout=0;

  for (i=(unsigned long)1<<(bitnum-1); i; i>>=1) {
    if (crc & i) crcout|=j;
    j<<= 1;
  }
  return (crcout);
}

unsigned long crcbitbybitfast(unsigned char* p, unsigned long len)
{

  // fast bit by bit algorithm without augmented zero bytes.
  // does not use lookup table, suited for polynom orders between 1...32.

  unsigned long i, j, c, bit;
  unsigned long crc = crcinit_direct;

  for (i=0; i<len; i++) {

    c = (unsigned long)*p++;
    if (refin) c = reflect(c, 8);

    for (j=0x80; j; j>>=1) {

      bit = crc & crchighbit;
      crc<<= 1;
      if (c & j) bit^= crchighbit;
      if (bit) crc^= polynom;
    }
  }

  if (refout) crc=reflect(crc, order);
  crc^= crcxor;
  crc&= crcmask;
```

```
  return (crc);
}

int setcrcparams(int input_order, unsigned long input_polynom, int input_direct, unsigned long input_crcinit,
unsigned long input_crcxor, int input_refin, int input_refout)
{
  // Test program for checking crcbitbybitfast().
  // Parameters are at the top of this program.
  // Result will be printed on the console.

  int i;
  unsigned long bit, crc;

  order = input_order;
  polynom = input_polynom;
  direct = input_direct;
  crcinit = input_crcinit;
  crcxor = input_crcxor;
  refin = input_refin;
  refout = input_refout;

  // at first, compute constant bit masks for whole CRC and CRC high bit

  crcmask = ((((unsigned long)1<<(order-1))-1)<<1)|1;
  crchighbit = (unsigned long)1<<(order-1);

  // compute missing initial CRC value

  if (!direct) {

    crcinit_nondirect = crcinit;
    crc = crcinit;
    for (i=0; i<order; i++) {

      bit = crc & crchighbit;
      crc<<= 1;
```

```
      if (bit) crc^= polynom;
    }
    crc&= crcmask;
    crcinit_direct = crc;
  }

  else {

    crcinit_direct = crcinit;
    crc = crcinit;
    for (i=0; i<order; i++) {

      bit = crc & 1;
      if (bit) crc^= polynom;
      crc >>= 1;
      if (bit) crc|= crchighbit;
    }
    crcinit_nondirect = crc;
  }

  return (0);
}
/* ---------------------------------------- CRC calculation ---------------------------------------- */
/* ------------------------------------------------------------------------------------------------- */

#defineSUCCESS  0x10000001

#defineNOTFOUND 0x10000010
#defineNOTFREE  0x10000020
#defineNOACCESS 0x10000030

#defineUNKNOWN  0x10000040
#defineOPENED   0x10000050
#defineOPENSND  0x10000051
#defineOPENRCV  0x10000053
#defineBADSSAP  0x10000060
#defineBADDSAP  0x10000070
```

```
#defineBADBYTES 0x10000080
#defineBADSEND  0x10000090
#defineBADRECV  0x100000A0
#defineNOTOKENS 0x100000B0

#defineBADCRC12 0x100000C0
#defineBADCRC32 0x100000D0

#defineMINBYTES 34
#define MAXBYTES 1488

#define IEEEHDSIZE 14
#define UIHDSIZE 8
#define UICONTRL 3
#define CRC32SIZE 4

int GETPID(void)
{
  return 0xCC1;
}

int REGISTER(int mySAP, SIB **mySIB)
{
  int  myPID;

  int i;

  for(i=0; i<mySIBL.sibcnt; i++)
  {
    printf("\n[REGISTER] Service position %d, SAP = %d, PID = 0x%x", i,
      mySIBL.sibi[i].sap,
      mySIBL.sibi[i].pid);

    if(mySIBL.sibi[i].sap == mySAP)
    {
      if(mySIBL.sibi[i].pid == 0)
      {
```

```
         myPID = GETPID();
         mySIBL.sibi[i].pid = myPID;
         *mySIB = &mySIBL.sibi[i];

         printf("\n[REGISTER] SAP %d is now registered to PID = 0x%x at SIB address 0x%x",
            (*mySIB)->sap,
            (*mySIB)->pid,
            *mySIB);

         return SUCCESS;
       } else
       {
         return NOTFREE;
       }
     }
   }
   return NOTFOUND;
}

int UNREGISTER(SIB **mySIB)
{
   int  myPID;

   int i;

   for(i=0; i<mySIBL.sibcnt; i++)
   {
     printf("\n[UNREGISTER] Service position %d, SAP = %d, PID = 0x%x", i,
        (*mySIB)->sap,
        (*mySIB)->pid);

     if(&mySIBL.sibi[i] == *mySIB)
     {
       myPID = GETPID();
       if(mySIBL.sibi[i].pid == myPID)
       {
         mySIBL.sibi[i].pid = 0;
```

```
          printf("\n[UNREGISTER] SAP %d is now unregistered, PID = 0x%x",
            mySIBL.sibi[i].sap,
            mySIBL.sibi[i].pid);

          return SUCCESS;
        } else
        {
          return NOACCESS;
        }
      }
    }
    return NOTFOUND;
}

int OPEN(int chanID, int access, CCB **myCCB)
{
    int  myPID;
    int mySAP;

    int i;
    int j;

    for(i=0; i<myCCBL.ccbcnt; i++)
    {
      if(myCCBL.ccbi[i].chnid == chanID)
      {
        if(myCCBL.ccbi[i].pid != 0) return OPENED;

        myPID = GETPID();

        for(j=0; j<mySIBL.sibcnt; j++)
        {
          if(mySIBL.sibi[j].pid == myPID)
          {
            mySAP = mySIBL.sibi[j].sap;
```

```
            if(access == OPENSND)
            {
              if(myCCBL.ccbi[j].ssap != mySAP) return BADSSAP;
            } else if(access == OPENRCV)
            {
              if(myCCBL.ccbi[j].dsap != mySAP) return BADDSAP;
            } else
            {
              return UNKNOWN;
            }

            myCCBL.ccbi[i].pid = myPID;
            *myCCB = &myCCBL.ccbi[j];

            printf("\n[OPEN] channel ID %d is now opened to PID = 0x%x",
               (*myCCB)->chnid,
               (*myCCB)->pid);

            return SUCCESS;
          }
        }
        return NOTFOUND;
      }
    }
    return NOTFOUND;
}

int CLOSE(CCB **myCCB)
{
    int  myPID;

    int i;

    for(i=0; i<myCCBL.ccbcnt; i++)
    {
      if(&myCCBL.ccbi[i] == *myCCB)
      {
```

```
        myPID = GETPID();
        if(myCCBL.ccbi[i].pid == myPID)
        {
          myCCBL.ccbi[i].pid = 0;

          printf("\n[CLOSE] channel ID %d is now closed, PID = 0x%x",
             (*myCCB)->chnid,
             (*myCCB)->pid);

          return SUCCESS;
        } else
        {
          return NOACCESS;
        }
      }
    }
    return NOTFOUND;
}

#define min(x,y) (x < y ? x : y)
#define max(x,y) (x > y ? x : y)

void LLC_SEND(CCB *llcCCB, char *record, int llcLen,int *llcSent)
{
    int wrsize;

    size_t nbytes;

    unsigned short hlength;
    unsigned long  hexheader;
    unsigned long  timestamp;
    unsigned long  CRC12;
    unsigned long  CRC32;

    double deltaT;
    unsigned long tokens;
    unsigned long deltaTokens;
```

```
struct tm* tmlocal;
time_t tlocal;
unsigned long secs_after_midnight;

PAB *tmpPAB;
unsigned long port;

int tmp;

int i,j;

if(poutFile == NULL)
{
  poutFile = fopen(outfile, "wba");

  pcap_hdr.magic_number = 0xa1b2c3d4;
  pcap_hdr.version_major = 2;
  pcap_hdr.version_minor = 4;
  pcap_hdr.thiszone = -3;
  pcap_hdr.sigfigs = 0;
  pcap_hdr.snaplen = 65535;
  pcap_hdr.network = 1;

  printf("\n\n    magic number = 0x%x", pcap_hdr.magic_number);
  printf("\n    major version number = %d", pcap_hdr.version_major);
  printf("\n    minor version number = %d", pcap_hdr.version_minor);
  printf("\n    GMT to local correction = %d", pcap_hdr.thiszone);
  printf("\n    accuracy of timestamps = %d", pcap_hdr.sigfigs);
  printf("\n    max length of captured packets,in octets = %d", pcap_hdr.snaplen);
  printf("\n    data link type = %d", pcap_hdr.network);

  wrsize = sizeof(pcap_hdr);
  nbytes = fwrite(&pcap_hdr, 1, wrsize, poutFile);
  printf("\n[LLC_SEND] bytes written %d for new PCAP file header", nbytes);

  gettimeofday(&then, NULL);
```

```
  }

  gettimeofday(&now, NULL);

// do traffinc shaping on channel
  llcLen = max(llcLen,MINBYTES);

  tokens = llcLen + IEEEHDSIZE + UIHDSIZE + CRC32SIZE;

  if(llcCCB->tokens < llcCCB->capacity)
  {
    deltaT = (now.tv_sec - then.tv_sec) * 1e6;
    deltaT = (deltaT + (now.tv_usec - then.tv_usec)) * 1e-6;

    deltaTokens = llcCCB->rate * ((long) deltaT);

    llcCCB->tokens = min(llcCCB->capacity, (llcCCB->tokens + deltaTokens));
  }

   then = now;

  printf("\n    tokens available = %d", llcCCB->tokens);
  if(tokens <= llcCCB->tokens)
  {
    llcCCB->tokens -= tokens;
    printf("\n    tokens left = %d (bytes sent + LLCE header)", llcCCB->tokens);
  } else
  {
    *llcSent = -1; // NOTOKENS
    return;
  }
// return negative bytes sent to indicate insufficient tokens in bucket

  pcaprec_hdr.ts_sec = now.tv_sec;
  pcaprec_hdr.ts_usec = now.tv_usec;
  pcaprec_hdr.incl_len = sizeof(IEEE802UI) - MAXBYTES + llcLen;
  pcaprec_hdr.orig_len = pcaprec_hdr.incl_len;
```

```
  printf("\n\n    timestamp seconds = %d (0x%x)", pcaprec_hdr.ts_sec, pcaprec_hdr.ts_sec);
  printf("\n    timestamp microseconds = %d (0x%x)", pcaprec_hdr.ts_usec, pcaprec_hdr.ts_usec);
  printf("\n        number of octets of packet to be saved in file = %d (0x%x)", pcaprec_hdr.incl_len,
pcaprec_hdr.incl_len);
  printf("\n    actual length of packet = %d (0x%x)", pcaprec_hdr.orig_len, pcaprec_hdr.orig_len);

  wrsize = sizeof(pcaprec_hdr);
  nbytes = fwrite(&pcaprec_hdr, 1, wrsize, poutFile);
  printf("\n[LLC_SEND] bytes written %d for new PCAP record header", nbytes);

  myUI.smac[0] = (unsigned char)(((myCBB.cibb->p2nib)->macb[1] & 0xFF000000) >> 24);
  myUI.smac[1] = (unsigned char)(((myCBB.cibb->p2nib)->macb[1] & 0x00FF0000) >> 16);
  myUI.smac[2] = (unsigned char)(((myCBB.cibb->p2nib)->macb[1] & 0x0000FF00) >>  8);
  myUI.smac[3] = (unsigned char)(((myCBB.cibb->p2nib)->macb[1] & 0x000000FF)      );
  myUI.smac[4] = (unsigned char)(((myCBB.cibb->p2nib)->macb[0] & 0xFF000000) >> 24);
  myUI.smac[5] = (unsigned char)(((myCBB.cibb->p2nib)->macb[0] & 0x00FF0000) >> 16);

  for(i=0; i<mySHB.shbcnt; i++)
  {
    if(mySHB.shbi[i].sapnumber == llcCCB->dsap)
      printf("\n    Host for service SAP = %d has MAC 0x%x %x",
        mySHB.shbi[i].sapnumber,
        mySHB.shbi[i].hostp2nib->macb[1], mySHB.shbi[i].hostp2nib->macb[0]);

      myUI.dmac[0] = (unsigned char)((mySHB.shbi[i].hostp2nib->macb[1] & 0xFF000000) >> 24);
      myUI.dmac[1] = (unsigned char)((mySHB.shbi[i].hostp2nib->macb[1] & 0x00FF0000) >> 16);
      myUI.dmac[2] = (unsigned char)((mySHB.shbi[i].hostp2nib->macb[1] & 0x0000FF00) >>  8);
      myUI.dmac[3] = (unsigned char)((mySHB.shbi[i].hostp2nib->macb[1] & 0x000000FF)      );
      myUI.dmac[4] = (unsigned char)((mySHB.shbi[i].hostp2nib->macb[0] & 0xFF000000) >> 24);
      myUI.dmac[5] = (unsigned char)((mySHB.shbi[i].hostp2nib->macb[0] & 0x00FF0000) >> 16);

      tmp = i;
  }

  port = 0;
  for(i=0; i<myPABL.pabcnt; i++)
```

```
{
  tmpPAB = myPABL.pabli[i].p2pab;
  for(j=0; j<tmpPAB->nibcnt; j++)
  {
    if(tmpPAB->pabi[j].p2nib == mySHB.shbi[tmp].hostp2nib)
    {
      printf("\n[LLC_SEND} Port %d: hops = %d to node '%s' from PAB list and from SHB '%s'",
        myPABL.pabli[i].port,
        tmpPAB->pabi[j].hops,
        tmpPAB->pabi[j].p2nib->nname,
        mySHB.shbi[tmp].hostp2nib->nname);

      port = myPABL.pabli[i].port;
      break;
    }
  }
  if(port != 0) break;
}

myUI.smac[5] = myUI.smac[5] | port;

printf("\n\n    Source MAC= ");
for(i = 0; i < 6; i++)
{
  printf(" %x", myUI.smac[i]);
}

printf("\n    Destination MAC =");
for(i = 0; i < 6; i++)
{
  printf(" %x", myUI.dmac[i]);
}

hlength = (short)(llcLen + UIHDSIZE + CRC32SIZE);
bswap16((char*)&hlength);
myUI.length = hlength;
printf("\n    UI length in network byte order = 0x%x", myUI.length);
```

```
myUI.dsap = (unsigned char)llcCCB->dsap;
myUI.ssap = (unsigned char)llcCCB->ssap;
myUI.ctrl = UICONTRL;

if(powerUP)
{
  powerUP = FALSE;
  llcCCB->sn = 0;
} else
{
  if(llcCCB->sn == 255)
  {
    llcCCB->sn = 1;
  } else
  {
    (llcCCB->sn)++;
  }
}
myUI.sn = (unsigned char)llcCCB->sn;

printf("\n    DSAP = 0x%x", myUI.dsap);
printf("\n    SSAP = 0x%x", myUI.ssap);
printf("\n    CTRL = 0x%x", myUI.ctrl);
printf("\n    SN = 0x%x", myUI.sn);

if(llcCCB->sn == 0)
{
  tlocal = time(NULL);
  tmlocal = localtime(&tlocal);
  secs_after_midnight = (unsigned long)(tmlocal->tm_hour*3600 + tmlocal->tm_min*60 + tmlocal->tm_sec);
  timestamp = (secs_after_midnight & 0x000FFFFF);
  printf("\n    time stamp is seconds after midnight %d", timestamp);
} else
{
  timestamp = (now.tv_usec & 0x000FFFFF);
  printf("\n    time stamp is microseconds after second %d", timestamp);
```

```
  }
  llcCCB->timestmp = timestamp;
  printf("\n    UI time stamp = 0x%x", timestamp);

  hexheader = timestamp << 12;
  bswap32((char*)&hexheader);
  memcpy(&myUI.exheader, &hexheader, 4);

  setcrcparams(12, 0x1F1, 1, 0xFFF, 0x0, 0, 0);
  CRC12 = crcbitbybitfast((unsigned char *)&(myUI.dsap), 7);
  printf("\n    UI CRC12      = 0x%x", CRC12);

  bswap32((char*)&hexheader);
  hexheader = (hexheader | CRC12);
  bswap32((char*)&hexheader);
  memcpy(&myUI.exheader, &hexheader, 4);
  printf("\n        Extended header = 0x%x %x %x %x in network order", myUI.exheader[0], myUI.exheader[1],
myUI.exheader[2], myUI.exheader[3]);

  memcpy(myUI.data, record, llcLen);

  setcrcparams(32, 0x741B8CD7, 1, 0xFFFFFFFF, 0x0, 0, 0);
  CRC32 = crcbitbybitfast((unsigned char *)record, llcLen);
  printf("\n    UI CRC32      = 0x%x", CRC32);

  bswap32((char*)&CRC32);
  memcpy(&(myUI.data[llcLen]), &CRC32, CRC32SIZE);

  printf("\n[LLC_SEND] Channel ID = %d, SSAP = %d, DSAP = %d, capacity = %d, tokens = %d, rate = %d bytes/sec, PID =
0x%x, SN = %d, Time-stamp = %d",
     llcCCB->chnid,
     llcCCB->ssap,
     llcCCB->dsap,
     llcCCB->capacity,
     llcCCB->tokens,
     llcCCB->rate,
     llcCCB->pid,
```

```
      llcCCB->sn,
      llcCCB->timestmp);

   nbytes = fwrite(&myUI, 1, pcaprec_hdr.incl_len, poutFile);
   printf("\n\n[LLC_SEND] bytes written %d for new PCAP record data", nbytes);

   *llcSent = llcLen;
   printf("\n[LLC_SEND] length = %d, xmited = %d", llcLen, *llcSent);
}

int SEND(CCB **myCCB, char* buffer, int nbytes, int *xbytes)
{
   int  myPID;

   int i;
   int goodCCB = 0;

   for(i=0; i<myCCBL.ccbcnt; i++)
   {
      if(&myCCBL.ccbi[i] == *myCCB)
      {
         myPID = GETPID();
         if(myCCBL.ccbi[i].pid != myPID) return NOACCESS;
         goodCCB = i;
      }
   }

   if(goodCCB == MAXCCB) return NOACCESS;

   if(nbytes > MAXBYTES) return BADBYTES;

   printf("\n[SEND] Calling LLC_SEND");
   LLC_SEND(&myCCBL.ccbi[goodCCB], buffer, nbytes, xbytes);

   if(*xbytes < 0) return NOTOKENS;

   if(*xbytes != nbytes) return BADSEND;
```

```
  return SUCCESS;
}

int STATUS(int chanID, unsigned long *chn_ssap, unsigned long *chn_dsap, unsigned long *chn_capacity, unsigned long
*chn_tokens, unsigned long *chn_rate,
            unsigned long *chn_pid, unsigned long *chn_sn, unsigned long *chn_timestmp)
{
  int i;

  for(i=0; i<myCCBL.ccbcnt; i++)
  {
    if(myCCBL.ccbi[i].chnid == chanID)
    {
      *chn_ssap = myCCBL.ccbi[i].ssap;
      *chn_dsap = myCCBL.ccbi[i].dsap;
      *chn_capacity = myCCBL.ccbi[i].capacity;
      *chn_tokens = myCCBL.ccbi[i].tokens;
      *chn_rate = myCCBL.ccbi[i].rate;
      *chn_pid = myCCBL.ccbi[i].pid;
      *chn_sn = myCCBL.ccbi[i].sn;
      *chn_timestmp = myCCBL.ccbi[i].timestmp;
      return SUCCESS;
    }
  }
  return NOTFOUND;
}

int main()
{
  int  cpm1_SAP;
  int cpm1_chn;
  int cpm1_acc;
  int  retcode;

  SIB  *cpm1_SIB;
  CCB *cpm1_CCB;
```

```
  cpm1_SAP = 114;
  cpm1_chn = 1;
  cpm1_acc = OPENSND;

  unsigned long chn_id;
  unsigned long chn_ssap;
  unsigned long chn_dsap;
  unsigned long chn_capacity;
  unsigned long chn_tokens;
  unsigned long chn_rate;
  unsigned long chn_pid;
  unsigned long chn_sn;
  unsigned long chn_timestmp;

  size_t msize;
  int  length;
  int  xmited = 0;
  char message[MINBYTES];

  int i;

// calling "initialize()" not needed in real life..
  retcode = initialize();

  printf("\n\n>> Entering main()...");

  printf("\n>> Registering CPM1 to SAP %d", cpm1_SAP);

  retcode = REGISTER(cpm1_SAP, &cpm1_SIB);

  if(retcode == SUCCESS)
  {
    printf("\n\n>> CPM1 SAP %d is now registered to PID = 0x%x",
    cpm1_SIB->sap,
    cpm1_SIB->pid);
  } else
```

```
{
  printf("\n>> bad code 0x%x", retcode);
}

printf("\n>> Opening channel ID %d for access 0x%x", cpm1_chn, cpm1_acc);

retcode = OPEN(cpm1_chn, cpm1_acc, &cpm1_CCB);

if(retcode == SUCCESS)
{
  printf("\n\n>> Channel ID %d is now open",
  cpm1_CCB->chnid);
} else
{
  printf("\n>> bad code 0x%x", retcode);
}

for(i = 0; i<MINBYTES; i++)
{
  message[i] = i+1;
}

msize = strlen(message);
length = (int)msize;

printf("\n\n>> Sending message with %d bytes", length);
retcode = SEND(&cpm1_CCB, message, length, &xmited);

printf("\n    return code 0x%x, tokens = %d", retcode, cpm1_CCB->tokens);

sleep(3);

printf("\n\n>> Sending message with %d bytes", length);
retcode = SEND(&cpm1_CCB, message, length, &xmited);

printf("\n    return code 0x%x, tokens = %d", retcode, cpm1_CCB->tokens);
```

```
  sleep(1);

  printf("\n\n>> Sending message with %d bytes", length);
  retcode = SEND(&cpm1_CCB, message, length, &xmited);

  printf("\n    return code 0x%x, tokens = %d", retcode, cpm1_CCB->tokens);

  retcode = STATUS(cpm1_chn, &chn_ssap, &chn_dsap, &chn_capacity, &chn_tokens, &chn_rate, &chn_pid, &chn_sn,
&chn_timestmp);

  printf("\n\n>> Status of channel ID = %d, SSAP = %d, DSAP = %d, Token Bucket capacity/tokens/rate [in bytes/sec] =
%d/%d/%d, PID = %x, SN = %d, Time-stamp = %d",
     cpm1_chn,
     chn_ssap,
     chn_dsap,
     chn_capacity,
     chn_tokens,
     chn_rate,
     chn_pid,
     chn_sn,
     chn_timestmp);

  printf("\n\n>> Closing channel ID %d", cpm1_chn);

  retcode = CLOSE(&cpm1_CCB);

  if(retcode == SUCCESS)
  {
    printf("\n\n>> Channel ID %d is now closed",
    cpm1_CCB->chnid);
  } else
  {
    printf("\n>> bad code 0x%x", retcode);
  }

  printf("\n\n>> Unregistering SAP %d from CPM1", cpm1_SAP);
```

```
  retcode = UNREGISTER(&cpm1_SIB);

  if(retcode == SUCCESS)
  {
    printf("\n\n>> CPM1 SAP %d is now unregistered (PID = 0x%x)",
    cpm1_SIB->sap,
    cpm1_SIB->pid);
  } else
  {
    printf("\n>> bad code 0x%x", retcode);
  }

  return 0;
}
```

D.2 Source code for the CPM2 module

```
#include <stdio.h>
#include <stdlib.h>

#include <time.h>
#include <unistd.h>

#include <string.h>

#include "chndef.h"

#include "pcap.h"
#include "IEEE802UI.h"

NIBL myNIBL;
CIB myCIB;
NIB myNIB;

CCBL myCCBL;
SIBL mySIBL;

SHB mySHB;

PAB p1PAB;
PABL myPABL;

CBB myCBB;

pcap_hdr_t pcap_hdr;
pcaprec_hdr_t pcaprec_hdr;

IEEE802UI myUI;

char infile[] = "UI.pcap";
FILE *pinFile = NULL;
```

```
struct timeval now;
struct timeval then;

double difsec;
long prvusec;

long difusec;

#define FALSE 0
#define TRUE !(FALSE)

void bswap16(char *input)
{
  unsigned char tmp;

  tmp = input[1];
  input[1] = input[0];
  input[0] = tmp;
}

void bswap32(char *input)
{
  unsigned char tmp;

  tmp = input[0];
  input[0] = input[3];
  input[3] = tmp;

  tmp = input[1];
  input[1] = input[2];
  input[2] = tmp;
}

/* ---------------------------------------------------------------------------------------------- */
/* ---------------------------------------- initialize()---------------------------------------- */
int fillNIBCIB()
{
```

```
  int i;

  myNIBL.nibcnt = 2;

  myNIBL.p2nib[0].nname[0] = '2MPC';
  myNIBL.p2nib[0].nname[1] = 0x00202020;
  myNIBL.p2nib[0].macb[0] = 0x00000000;
  myNIBL.p2nib[0].macb[1] = 0xAA000500;
  myNIBL.p2nib[0].hexid = 0x341;
  myNIBL.p2nib[0].macb[0] |= 0x41000000;
  myNIBL.p2nib[0].macb[1] |= 0x00000003;
  myNIBL.p2nib[0].unit = 2;
  myNIBL.p2nib[0].macb[0] |= 0x00200000;

  myNIBL.p2nib[1].nname[0] = '1MPC';
  myNIBL.p2nib[1].nname[1] = 0x00202020;
  myNIBL.p2nib[1].macb[0] = 0x00000000;
  myNIBL.p2nib[1].macb[1] = 0xAA000500;
  myNIBL.p2nib[1].hexid = 0x341;
  myNIBL.p2nib[1].macb[0] |= 0x41000000;
  myNIBL.p2nib[1].macb[1] |= 0x00000003;
  myNIBL.p2nib[1].unit = 1;
  myNIBL.p2nib[1].macb[0] |= 0x00100000;

  printf("\n>> Number of nodes %d", myNIBL.nibcnt);
  for(i=0; i<myNIBL.nibcnt; i++)
  {
    printf("\n>> Node name '%s' Equipment ID = 0x%x   Unit = %d MAC Base = %x %x", myNIBL.p2nib[i].nname,
myNIBL.p2nib[i].hexid, myNIBL.p2nib[i].unit, myNIBL.p2nib[i].macb[1], myNIBL.p2nib[i].macb[0]);
  }

  myCIB.cfgname[0] = 'TSET';
  myCIB.cfgname[1] = 0x00313030;
  myNIB = myNIBL.p2nib[0];
  myCIB.p2nib = &myNIB;
```

```
  printf("\n>> Configuration name '%s' Host NIB address = %x Host name from NIB List '%s' Host name from CIB '%s'",
myCIB.cfgname, myNIB, myNIB.nname, myCIB.p2nib->nname);

  return 0;
}

int fillCCBSIB()
{
  int i;

  myCCBL.ccbcnt = 1;

  myCCBL.ccbi[0].chnid = 1;
  myCCBL.ccbi[0].ssap = 114; // 0x72
  myCCBL.ccbi[0].dsap = 116; // 0x74
  myCCBL.ccbi[0].capacity = 90; // token bucket limit with 50% overdraft
  myCCBL.ccbi[0].tokens = 60;   // token buckt initial filling equal 38 bytes of message + 22 bytes of LLCE header
  myCCBL.ccbi[0].rate = 30;     // token bucket fill rate equal [].tokens/2
  myCCBL.ccbi[0].pid = 0;
  myCCBL.ccbi[0].sn = 0;
  myCCBL.ccbi[0].timestmp = 0;

  printf("\n>> Number of channels %d", myCCBL.ccbcnt);

  for(i=0; i<myCCBL.ccbcnt; i++)
  {
    printf("\n>> Channel position %d, ID = %d, SSAP = %d, DSAP = %d, Token Bucket capacity/tokens/rate [in
bytes/sec] = %d/%d/%d, PID = %x, SN = %d, Time-stamp = %d", i,
       myCCBL.ccbi[0].chnid,
       myCCBL.ccbi[0].ssap,
       myCCBL.ccbi[0].dsap,
       myCCBL.ccbi[0].capacity,
       myCCBL.ccbi[0].tokens,
       myCCBL.ccbi[0].rate,
       myCCBL.ccbi[0].pid,
       myCCBL.ccbi[0].sn,
       myCCBL.ccbi[0].timestmp);
```

```
  }

  mySIBL.sibcnt = 1;

  mySIBL.sibi[0].sap = 116;
  mySIBL.sibi[0].pid = 0;

  for(i=0; i<mySIBL.sibcnt; i++)
  {
    printf("\n>> Service position %d, SAP = %d, PID = %x", i,
      mySIBL.sibi[0].sap,
      mySIBL.sibi[0].pid);
  }

  return 0;
}

int fillSHB()
{
  int i;

  mySHB.shbcnt = 1;
  mySHB.filler = 0;

  mySHB.shbi[0].sapnumber = 114;
  mySHB.shbi[0].hostp2nib = &myNIBL.p2nib[1]; //NIB pointing to CPM1

  printf("\n>> Number of hosted services %d", mySHB.shbcnt);

  for(i=0; i<mySHB.shbcnt; i++)
  {
    printf("\n>> Hosted service position %d, SAP = %d, Host name from NIB '%s'", i,
      mySHB.shbi[i].sapnumber,
      mySHB.shbi[i].hostp2nib->nname);
  }

  return 0;
```

```
}

int fillPABL()
{
   int i;
   int j;
   PAB *tmpPAB;

   p1PAB.nibcnt = 1;
   p1PAB.filler = 0;

   p1PAB.pabi[0].p2nib = &myNIBL.p2nib[1]; //NIB pointing to CPM1
   p1PAB.pabi[0].hops = 0;

   for(i=0; i<p1PAB.nibcnt; i++)
   {
      printf("\n>> Port 1: hops = %d to node '%s' from NIB",
         p1PAB.pabi[i].hops,
         p1PAB.pabi[i].p2nib->nname);
   }

   myPABL.pabcnt = 1;
   myPABL.filler = 0;

   myPABL.pabli[0].port = 1;
   myPABL.pabli[0].p2pab = &p1PAB; //Port 1 to CPM1

   for(i=0; i<myPABL.pabcnt; i++)
   {
      tmpPAB = myPABL.pabli[i].p2pab;
      printf("\n...NIBs From PAB List...");
      for(j=0; j<tmpPAB->nibcnt; j++)
      {
         printf("\n>> Port %d: hops = %d to node '%s'",
            myPABL.pabli[i].port,
            tmpPAB->pabi[j].hops,
            tmpPAB->pabi[j].p2nib->nname);
```

```
    }
  }

  return 0;
}

int fillCBB()
{
  myCBB.cibb = &myCIB;
  myCBB.nibb = &myNIBL;
  myCBB.sibb = &mySIBL;
  myCBB.ccbb = &myCCBL;
  myCBB.shbb = &mySHB;
  myCBB.pabb = &myPABL;

  printf("\n>> Configuration name '%s' Host name from CIB '%s'", myCBB.cibb->cfgname, (myCBB.cibb->p2nib)->nname);

  return 0;
}

int initialize()
{
  int retcode;

  printf(">> Filling NIBs and CIB...");
  retcode = fillNIBCIB();

  printf("\n>> Filling CCBs and SIBs...");
  retcode = fillCCBSIB();

  printf("\n\n>> Filling SHBs...");
  retcode = fillSHB();

  printf("\n\n>> Filling PABs and PAB List...");
  retcode = fillPABL();

  printf("\n\n>> Filling CBB...");
```

```
  retcode = fillCBB();

  printf("\n\n>> End of initialize()...");
  return retcode;
}
/* ---------------------------------------- initialize()---------------------------------------- */
/* ------------------------------------------------------------------------------------------------ */

/* ------------------------------------------------------------------------------------------------ */
/* ---------------------------------------- CRC calculation ---------------------------------------- */
// CRC parameters (values for CRC-32 Koopman G(x) = (x+1)(x3+x2+1)(x28+x22+x20+x19+x16+x14+x12+x9+x8+x6+1):
// CRC parameters (values for CRC-12 Koopman G(x) = x12 +x8 +x7 +x6 +x5 +x4 +1):

int order;
unsigned long polynom;
int direct;
unsigned long crcinit;
unsigned long crcxor;
int refin;
int refout;

// 'order' [1..32] is the CRC polynom order, counted without the leading '1' bit
// 'polynom' is the CRC polynom without leading '1' bit
// 'direct' [0,1] specifies the kind of algorithm: 1=direct, no augmented zero bits
// 'crcinit' is the initial CRC value belonging to that algorithm
// 'crcxor' is the final XOR value
// 'refin' [0,1] specifies if a data byte is reflected before processing (UART) or not
// 'refout' [0,1] specifies if the CRC will be reflected before XOR

// internal global values:

unsigned long crcmask;
unsigned long crchighbit;
unsigned long crcinit_direct;
unsigned long crcinit_nondirect;

// subroutines
```

```
unsigned long reflect (unsigned long crc, int bitnum)
{

   // reflects the lower 'bitnum' bits of 'crc'

   unsigned long i, j=1, crcout=0;

   for (i=(unsigned long)1<<(bitnum-1); i; i>>=1) {
      if (crc & i) crcout|=j;
      j<<= 1;
   }
   return (crcout);
}

unsigned long crcbitbybitfast(unsigned char* p, unsigned long len)
{

   // fast bit by bit algorithm without augmented zero bytes.
   // does not use lookup table, suited for polynom orders between 1...32.

   unsigned long i, j, c, bit;
   unsigned long crc = crcinit_direct;

   for (i=0; i<len; i++) {

      c = (unsigned long)*p++;
      if (refin) c = reflect(c, 8);

      for (j=0x80; j; j>>=1) {

         bit = crc & crchighbit;
         crc<<= 1;
         if (c & j) bit^= crchighbit;
         if (bit) crc^= polynom;
      }
   }
```

```
  if (refout) crc=reflect(crc, order);
  crc^= crcxor;
  crc&= crcmask;

  return (crc);
}

int setcrcparams(int input_order, unsigned long input_polynom, int input_direct, unsigned long input_crcinit,
unsigned long input_crcxor, int input_refin, int input_refout)
{
  // Test program for checking crcbitbybitfast().
  // Parameters are at the top of this program.
  // Result will be printed on the console.

  int i;
  unsigned long bit, crc;

  order = input_order;
  polynom = input_polynom;
  direct = input_direct;
  crcinit = input_crcinit;
  crcxor = input_crcxor;
  refin = input_refin;
  refout = input_refout;

  // at first, compute constant bit masks for whole CRC and CRC high bit

  crcmask = ((((unsigned long)1<<(order-1))-1)<<1)|1;
  crchighbit = (unsigned long)1<<(order-1);

  // compute missing initial CRC value

  if (!direct) {

    crcinit_nondirect = crcinit;
    crc = crcinit;
```

```
    for (i=0; i<order; i++) {

      bit = crc & crchighbit;
      crc<<= 1;
      if (bit) crc^= polynom;
    }
    crc&= crcmask;
    crcinit_direct = crc;
  }

  else {

    crcinit_direct = crcinit;
    crc = crcinit;
    for (i=0; i<order; i++) {

      bit = crc & 1;
      if (bit) crc^= polynom;
      crc >>= 1;
      if (bit) crc|= crchighbit;
    }
    crcinit_nondirect = crc;
  }

  return (0);
}
/* ------------------------------------- CRC calculation ------------------------------------- */
/* ------------------------------------------------------------------------------------------- */

#defineSUCCESS  0x10000001

#defineNOTFOUND 0x10000010
#defineNOTFREE  0x10000020
#defineNOACCESS 0x10000030

#defineUNKNOWN  0x10000040
#defineOPENED   0x10000050
```

```
#defineOPENSND  0x10000051
#defineOPENRCV  0x10000053
#defineBADSSAP  0x10000060
#defineBADDSAP  0x10000070
#defineBADBYTES 0x10000080
#defineBADSEND  0x10000090
#defineBADRECV  0x100000A0
#defineNOTOKENS 0x100000B0

#defineBADCRC12 0x100000C0
#defineBADCRC32 0x100000D0

#defineMINBYTES 34
#define MAXBYTES 1488

#define IEEEHDSIZE 14
#define UIHDSIZE 8
#define UICONTRL 3
#define CRC32SIZE 4

int GETPID(void)
{
  return 0xCC2;
}

int REGISTER(int mySAP, SIB **mySIB)
{
  int  myPID;

  int i;

  for(i=0; i<mySIBL.sibcnt; i++)
  {
    printf("\n[REGISTER] Service position %d, SAP = %d, PID = 0x%x", i,
      mySIBL.sibi[i].sap,
      mySIBL.sibi[i].pid);
```

```
    if(mySIBL.sibi[i].sap == mySAP)
    {
      if(mySIBL.sibi[i].pid == 0)
      {
        myPID = GETPID();
        mySIBL.sibi[i].pid = myPID;
        *mySIB = &mySIBL.sibi[i];

        printf("\n[REGISTER] SAP %d is now registered to PID = 0x%x at SIB address 0x%x",
          (*mySIB)->sap,
          (*mySIB)->pid,
          *mySIB);

        return SUCCESS;
      } else
      {
        return NOTFREE;
      }
    }
  }
  return NOTFOUND;
}

int UNREGISTER(SIB **mySIB)
{
  int  myPID;

  int i;

  for(i=0; i<mySIBL.sibcnt; i++)
  {
    printf("\n[UNREGISTER] Service position %d, SAP = %d, PID = 0x%x", i,
      (*mySIB)->sap,
      (*mySIB)->pid);

    if(&mySIBL.sibi[i] == *mySIB)
    {
```

```
      myPID = GETPID();
      if(mySIBL.sibi[i].pid == myPID)
      {
        mySIBL.sibi[i].pid = 0;

        printf("\n[UNREGISTER] SAP %d is now unregistered, PID = 0x%x",
          mySIBL.sibi[i].sap,
          mySIBL.sibi[i].pid);

        return SUCCESS;
      } else
      {
        return NOACCESS;
      }
    }
  }
  return NOTFOUND;
}

int OPEN(int chanID, int access, CCB **myCCB)
{
  int  myPID;
  int mySAP;

  int i;
  int j;

  for(i=0; i<myCCBL.ccbcnt; i++)
  {
    if(myCCBL.ccbi[i].chnid == chanID)
    {
      if(myCCBL.ccbi[i].pid != 0) return OPENED;

      myPID = GETPID();

      for(j=0; j<mySIBL.sibcnt; j++)
      {
```

```
          if(mySIBL.sibi[j].pid == myPID)
          {
            mySAP = mySIBL.sibi[j].sap;

            if(access == OPENSND)
            {
              if(myCCBL.ccbi[j].ssap != mySAP) return BADSSAP;
            } else if(access == OPENRCV)
            {
              if(myCCBL.ccbi[j].dsap != mySAP) return BADDSAP;
            } else
            {
              return UNKNOWN;
            }

            myCCBL.ccbi[i].pid = myPID;
            *myCCB = &myCCBL.ccbi[j];

            printf("\n[OPEN] channel ID %d is now opened to PID = 0x%x",
               (*myCCB)->chnid,
               (*myCCB)->pid);

            return SUCCESS;
          }
        }
        return NOTFOUND;
      }
    }
    return NOTFOUND;
}

int CLOSE(CCB **myCCB)
{
    int  myPID;

    int i;
```

```
  for(i=0; i<myCCBL.ccbcnt; i++)
  {
    if(&myCCBL.ccbi[i] == *myCCB)
    {
      myPID = GETPID();
      if(myCCBL.ccbi[i].pid == myPID)
      {
        myCCBL.ccbi[i].pid = 0;

        printf("\n[CLOSE] channel ID %d is now closed, PID = 0x%x",
          (*myCCB)->chnid,
          (*myCCB)->pid);

        return SUCCESS;
      } else
      {
        return NOACCESS;
      }
    }
  }
  return NOTFOUND;
}

#define min(x,y) (x < y ? x : y)
#define max(x,y) (x > y ? x : y)

void LLC_RECEIVE(CCB *llcCCB, char *record, int llcLen,int *llcRecvd)
{
  int rdsize;

  size_t nbytes;

  unsigned short hlength;
  unsigned long  hexheader;
  unsigned char  fulheader[8];
  unsigned long  timestamp;
  unsigned long  CRC12;
```

```
unsigned long  ckdCRC12;
unsigned long  CRC32;
unsigned long  ckdCRC32;

double deltaT;
unsigned long tokens;
unsigned long deltaTokens;

struct tm* tmlocal;
time_t tlocal;
long secs_after_midnight;

PAB *tmpPAB;

int i,j;

if(pinFile == NULL)
{
  pinFile = fopen(infile, "rb");

  if(pinFile == NULL)
  {
    printf("\n[LLC_RECEIVE] PCAP file not found");
    exit(NOTFOUND);
  }

  rdsize = sizeof(pcap_hdr);
  nbytes = fread(&pcap_hdr, 1, rdsize, pinFile);
  printf("\n[LLC_RECEIVE] bytes read %d for PCAP file header", nbytes);

  printf("\n\n    magic number = 0x%x", pcap_hdr.magic_number);
  printf("\n    major version number = %d", pcap_hdr.version_major);
  printf("\n    minor version number = %d", pcap_hdr.version_minor);
  printf("\n    GMT to local correction = %d", pcap_hdr.thiszone);
  printf("\n    accuracy of timestamps = %d", pcap_hdr.sigfigs);
  printf("\n    max length of captured packets,in octets = %d", pcap_hdr.snaplen);
  printf("\n    data link type = %d", pcap_hdr.network);
```

```
      gettimeofday(&then, NULL);

      prvusec = 0;
    }

    gettimeofday(&now, NULL);

// do traffinc policing on channel
    tokens = llcLen + IEEEHDSIZE + UIHDSIZE + CRC32SIZE;

    if(llcCCB->tokens < llcCCB->capacity)
    {
      difsec = now.tv_sec - then.tv_sec;
      deltaT = (now.tv_sec - then.tv_sec) * 1e6;
      deltaT = (deltaT + (now.tv_usec - then.tv_usec)) * 1e-6;

      deltaTokens = llcCCB->rate * ((long) deltaT);

      llcCCB->tokens = min(llcCCB->capacity, (llcCCB->tokens + deltaTokens));
    }

     then = now;

    printf("\n    tokens available = %d", llcCCB->tokens);
    if(tokens <= llcCCB->tokens)
    {
      llcCCB->tokens -= tokens;
      printf("\n    tokens left = %d (bytes sent + LLCE header)", llcCCB->tokens);
    } else
    {
      *llcRecvd = -1; // NOTOKENS
      return;
    }
// return negative bytes sent to indicate insufficient tokens in bucket

    rdsize = sizeof(pcaprec_hdr);
```

```
nbytes = fread(&pcaprec_hdr, 1, rdsize, pinFile);
printf("\n[LLC_RECEIVE] bytes read %d from PCAP record header", nbytes);

printf("\n\n    timestamp seconds = %d (0x%x)", pcaprec_hdr.ts_sec, pcaprec_hdr.ts_sec);
printf("\n    timestamp microseconds = %d (0x%x)", pcaprec_hdr.ts_usec, pcaprec_hdr.ts_usec);
printf("\n    number of octets of packet saved in file = %d (0x%x)", pcaprec_hdr.incl_len, pcaprec_hdr.incl_len);
printf("\n    actual length of packet = %d (0x%x)", pcaprec_hdr.orig_len, pcaprec_hdr.orig_len);

nbytes = fread(&myUI, 1, pcaprec_hdr.incl_len, pinFile);
printf("\n\n[LLC_RECEIVE] bytes read %d from PCAP record data", nbytes);

printf("\n\n    Source MAC= ");
for(i = 0; i < 6; i++)
{
  printf(" %x", myUI.smac[i]);
}

printf("\n    Destination MAC =");
for(i = 0; i < 6; i++)
{
  printf(" %x", myUI.dmac[i]);
}

hlength = myUI.length;
bswap16((char*)&hlength);
printf("\n    UI length = 0x%x", hlength);

memcpy(&hexheader, &myUI.exheader, 4);
bswap32((char*)&hexheader);
CRC12 = hexheader & 0x00000FFF;

memcpy(&fulheader, &myUI.dsap, 8);
fulheader[6] = fulheader[6] & 0xF0;
fulheader[7] = 0;

setcrcparams(12, 0x1F1, 1, 0xFFF, 0x0, 0, 0);
ckdCRC12 = crcbitbybitfast(&fulheader, 7);
```

```
  if(CRC12 != ckdCRC12)
  {
    printf("\n[LLC_RECEIVE] bad CRC12");
    *llcRecvd = 0;
    return;
  }

  printf("\n    UI CRC12      = 0x%x", CRC12);

  setcrcparams(32, 0x741B8CD7, 1, 0xFFFFFFFF, 0x0, 0, 0);
  ckdCRC32 = crcbitbybitfast((unsigned char *)&(myUI.data[0]), (llcLen+CRC32SIZE));

  if(ckdCRC32 != 0)
  {
    printf("\n[LLC_RECEIVE] bad CRC32");
    *llcRecvd = 0;
    return;
  }

  memcpy(&CRC32, &(myUI.data[llcLen]),  CRC32SIZE);
  bswap32((char*)&CRC32);
  printf("\n    UI CRC32      = 0x%x", CRC32);

  memcpy(record, myUI.data, llcLen);

  printf("\n    DSAP = 0x%x", myUI.dsap);
  printf("\n    SSAP = 0x%x", myUI.ssap);
  printf("\n    CTRL = 0x%x", myUI.ctrl);
  printf("\n    SN = 0x%x", myUI.sn);

  printf("\n        Extended  header  =  0x%x  %x  %x  %x  in  network  order",  myUI.exheader[0],  myUI.exheader[1],
myUI.exheader[2], myUI.exheader[3]);

  timestamp = hexheader >> 12;
  printf("\n    UI time stamp = %d (0x%x)", timestamp, timestamp);
```

```
  if(myUI.sn == 0)
  {
    tlocal = time(NULL);
    tmlocal = localtime(&tlocal);
    secs_after_midnight = (long)(tmlocal->tm_hour*3600 + tmlocal->tm_min*60 + tmlocal->tm_sec);

    printf("\n     time stamp is seconds after midnight (apparent offset to remote clock = %d +ahead/-behind)",
(secs_after_midnight - (long)timestamp));
  } else
  {
    printf("\n    delta time in seconds for traffic policing = %f", deltaT);

    if(prvusec != 0)
    {
      difusec = timestamp - prvusec;
      difsec = difsec + (difusec)*1e-6;
      printf("\n    estimated delta time in seconds at origin  = %f", difsec);
    } else
    {
      prvusec = timestamp;
    }
  }
  llcCCB->timestmp = timestamp;
  llcCCB->sn = myUI.sn;

  *llcRecvd = llcLen;
  printf("\n[LLC_RECEIVE] length = %d, received = %d", llcLen, *llcRecvd);
}

int RECEIVE(CCB **myCCB, char* buffer, int nbytes, int *xbytes)
{
  int  myPID;

  int i;
  int goodCCB = 0;

  for(i=0; i<myCCBL.ccbcnt; i++)
```

```
  {
    if(&myCCBL.ccbi[i] == *myCCB)
    {
      myPID = GETPID();
      if(myCCBL.ccbi[i].pid != myPID) return NOACCESS;
      goodCCB = i;
    }
  }

  if(goodCCB == MAXCCB) return NOACCESS;

  if(nbytes > MAXBYTES) return BADBYTES;

  printf("\n[RECEIVE] Calling LLC_RECEIVE");
  LLC_RECEIVE(&myCCBL.ccbi[goodCCB], buffer, nbytes, xbytes);

  if(*xbytes < 0) return NOTOKENS;

  if(*xbytes != nbytes) return BADRECV;

  return SUCCESS;
}

int STATUS(int chanID, unsigned long *chn_ssap, unsigned long *chn_dsap, unsigned long *chn_capacity, unsigned long
*chn_tokens, unsigned long *chn_rate,
            unsigned long *chn_pid, unsigned long *chn_sn, unsigned long *chn_timestmp)
{
  int i;

  for(i=0; i<myCCBL.ccbcnt; i++)
  {
    if(myCCBL.ccbi[i].chnid == chanID)
    {
      *chn_ssap = myCCBL.ccbi[i].ssap;
      *chn_dsap = myCCBL.ccbi[i].dsap;
      *chn_capacity = myCCBL.ccbi[i].capacity;
      *chn_tokens = myCCBL.ccbi[i].tokens;
```

```
        *chn_rate = myCCBL.ccbi[i].rate;
        *chn_pid = myCCBL.ccbi[i].pid;
        *chn_sn = myCCBL.ccbi[i].sn;
        *chn_timestmp = myCCBL.ccbi[i].timestmp;
        return SUCCESS;
      }
   }
   return NOTFOUND;
}

int main()
{
   int  cpm2_SAP;
   int cpm2_chn;
   int cpm2_acc;
   int  retcode;

   SIB  *cpm2_SIB;
   CCB *cpm2_CCB;

   cpm2_SAP = 116;
   cpm2_chn = 1;
   cpm2_acc = OPENRCV;

   unsigned long chn_id;
   unsigned long chn_ssap;
   unsigned long chn_dsap;
   unsigned long chn_capacity;
   unsigned long chn_tokens;
   unsigned long chn_rate;
   unsigned long chn_pid;
   unsigned long chn_sn;
   unsigned long chn_timestmp;

   size_t msize;
   int  length;
   int  receivd = 0;
```

```
   char message[MINBYTES];

   struct timeval prvTime;
   struct timeval aftTime;
   struct timespec nanoTime;

   int i;

// calling "initialize()" not needed in real life..
   retcode = initialize();

   printf("\n\n>> Entering main()...");

   printf("\n>> Registering CPM1 to SAP %d", cpm2_SAP);

   retcode = REGISTER(cpm2_SAP, &cpm2_SIB);

   if(retcode == SUCCESS)
   {
     printf("\n\n>> CPM2 SAP %d is now registered to PID = 0x%x",
     cpm2_SIB->sap,
     cpm2_SIB->pid);
   } else
   {
     printf("\n>> bad code 0x%x", retcode);
   }

   printf("\n>> Opening channel ID %d for access 0x%x", cpm2_chn, cpm2_acc);

   retcode = OPEN(cpm2_chn, cpm2_acc, &cpm2_CCB);

   if(retcode == SUCCESS)
   {
     printf("\n\n>> Channel ID %d is now open",
     cpm2_CCB->chnid);
   } else
   {
```

```
      printf("\n>> bad code 0x%x", retcode);
    }

    for(i = 0; i<MINBYTES; i++)
    {
      message[i] = 0xFF;
    }

    msize = strlen(message);
    length = (int)msize;

    printf("\n\n>> Receiving message with %d bytes", length);

    gettimeofday(&prvTime, NULL);
    retcode = RECEIVE(&cpm2_CCB, message, length, &receivd);

    printf("\n    return code 0x%x, tokens = %d", retcode, cpm2_CCB->tokens);

    printf("\n\n>> Receiving message with %d bytes", length);

    gettimeofday(&aftTime, NULL);
    nanoTime.tv_nsec = 0;
    nanoTime.tv_sec = 3;
//  nanoTime.tv_nsec = (1005000L - aftTime.tv_usec + prvTime.tv_usec)*1000;
//  nanoTime.tv_sec = 2;
    nanosleep(&nanoTime, (struct timespec *)NULL);

    gettimeofday(&prvTime, NULL);
    retcode = RECEIVE(&cpm2_CCB, message, length, &receivd);

    printf("\n    return code 0x%x, tokens = %d", retcode, cpm2_CCB->tokens);

    printf("\n\n>> Receiving message with %d bytes", length);

    gettimeofday(&aftTime, NULL);
    nanoTime.tv_nsec = 0;
    nanoTime.tv_sec = 1;
```

```
  nanosleep(&nanoTime, (struct timespec *)NULL);

  retcode = RECEIVE(&cpm2_CCB, message, length, &receivd);

  printf("\n    return code 0x%x, tokens = %d", retcode, cpm2_CCB->tokens);

  retcode = STATUS(cpm2_chn, &chn_ssap, &chn_dsap, &chn_capacity, &chn_tokens, &chn_rate, &chn_pid, &chn_sn,
&chn_timestmp);

  printf("\n\n>> Status of channel ID = %d, SSAP = %d, DSAP = %d, Token Bucket capacity/tokens/rate [in bytes/sec] =
%d/%d/%d, PID = %x, SN = %d, Time-stamp = %d",
     cpm2_chn,
     chn_ssap,
     chn_dsap,
     chn_capacity,
     chn_tokens,
     chn_rate,
     chn_pid,
     chn_sn,
     chn_timestmp);

  printf("\n\n>> Closing channel ID %d", cpm2_chn);

  retcode = CLOSE(&cpm2_CCB);

  if(retcode == SUCCESS)
  {
    printf("\n\n>> Channel ID %d is now closed",
    cpm2_CCB->chnid);
  } else
  {
    printf("\n>> bad code 0x%x", retcode);
  }

  printf("\n\n>> Unregistering SAP %d from CPM2", cpm2_SAP);

  retcode = UNREGISTER(&cpm2_SIB);
```

```
  if(retcode == SUCCESS)
  {
    printf("\n\n>> CPM2 SAP %d is now unregistered (PID = 0x%x)",
    cpm2_SIB->sap,
    cpm2_SIB->pid);
  } else
  {
    printf("\n>> bad code 0x%x", retcode);
  }

  return 0;
}
```

APPENDIX E – TEST CASE FULL TEXT OUTPUTS

The full text output by the CPM1 and CPM2 modules developed while running the test case are copied in the next two sections.

E.1 Full text output for module CPM1

```
>> Filling NIBs and CIB...
>> Number of nodes 2
>> Node name 'CPM1    ' Equipment ID = 0x341  Unit = 1 MAC Base = aa000503
41100000
>> Node name 'CPM2    ' Equipment ID = 0x341  Unit = 2 MAC Base = aa000503
41200000
>> Configuration name 'TEST001' Host NIB address = 63fcc0 Host name from NIB
List 'CPM1   ' Host name from CIB 'CPM1   '
>> Filling CCBs and SIBs...
>> Number of channels 1
>> Channel position 0, ID = 1, SSAP = 114, DSAP = 116, Token Bucket
capacity/tokens/rate [in bytes/sec] = 90/60/30, PID = 0, SN = 0, Time-stamp =
0
>> Service position 0, SAP = 114, PID = 0

>> Filling SHBs...
>> Number of hosted services 1
>> Hosted service position 0, SAP = 116, Host name from NIB 'CPM2   '

>> Filling PABs and PAB List...
>> Port 1: hops = 0 to node 'CPM2   ' from NIB
...NIBs From PAB List...
>> Port 1: hops = 0 to node 'CPM2   '

>> Filling CBB...
>> Configuration name 'TEST001' Host name from CIB 'CPM1   '

>> End of initialize()...

>> Entering main()...
>> Registering CPM1 to SAP 114
[REGISTER] Service position 0, SAP = 114, PID = 0x0
[REGISTER] SAP 114 is now registered to PID = 0xcc1 at SIB address 0x40bb44

>> CPM1 SAP 114 is now registered to PID = 0xcc1
>> Opening channel ID 1 for access 0x10000051
[OPEN] channel ID 1 is now opened to PID = 0xcc1

>> Channel ID 1 is now open

>> Sending message with 34 bytes
[SEND] Calling LLC_SEND

    magic number = 0xa1b2c3d4
    major version number = 2
    minor version number = 4
    GMT to local correction = -3
    accuracy of timestamps = 0
    max length of captured packets,in octets = 65535
    data link type = 1
[LLC_SEND] bytes written 24 for new PCAP file header
    tokens available = 60
    tokens left = 0 (bytes sent + LLCE header)

    timestamp seconds = 1606821301 (0x5fc625b5)
    timestamp microseconds = 200500 (0x30f34)
    number of octets of packet to be saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)
```

```
[LLC_SEND] bytes written 16 for new PCAP record header
    Host for service SAP = 116 has MAC 0xaa000503 41200000
[LLC_SEND} Port 1: hops = 0 to node 'CPM2    ' from PAB list and from SHB 'CPM2
'

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length in network byte order = 0x2e00
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x0
    time stamp is seconds after midnight 29701
    UI time stamp = 0x7405
    UI CRC12      = 0xf8c
    Extended header = 0x7 40 5f 8c in network order
    UI CRC32      = 0x840feafa
[LLC_SEND] Channel ID = 1, SSAP = 114, DSAP = 116, capacity = 90, tokens = 0,
rate = 30 bytes/sec, PID = 0xcc1, SN = 0, Time-stamp = 29701

[LLC_SEND] bytes written 60 for new PCAP record data
[LLC_SEND] length = 34, xmited = 34
    return code 0x10000001, tokens = 0

>> Sending message with 34 bytes
[SEND] Calling LLC_SEND
    tokens available = 90
    tokens left = 30 (bytes sent + LLCE header)

    timestamp seconds = 1606821304 (0x5fc625b8)
    timestamp microseconds = 231280 (0x38770)
    number of octets of packet to be saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)
[LLC_SEND] bytes written 16 for new PCAP record header
    Host for service SAP = 116 has MAC 0xaa000503 41200000
[LLC_SEND} Port 1: hops = 0 to node 'CPM2    ' from PAB list and from SHB 'CPM2
'

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length in network byte order = 0x2e00
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x1
    time stamp is microseconds after second 231280
    UI time stamp = 0x38770
    UI CRC12      = 0x8b3
    Extended header = 0x38 77 8 b3 in network order
    UI CRC32      = 0x840feafa
[LLC_SEND] Channel ID = 1, SSAP = 114, DSAP = 116, capacity = 90, tokens = 30,
rate = 30 bytes/sec, PID = 0xcc1, SN = 1, Time-stamp = 231280

[LLC_SEND] bytes written 60 for new PCAP record data
[LLC_SEND] length = 34, xmited = 34
    return code 0x10000001, tokens = 30

>> Sending message with 34 bytes
[SEND] Calling LLC_SEND
    tokens available = 60
    tokens left = 0 (bytes sent + LLCE header)

    timestamp seconds = 1606821305 (0x5fc625b9)
    timestamp microseconds = 246672 (0x3c390)
    number of octets of packet to be saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)
[LLC_SEND] bytes written 16 for new PCAP record header
    Host for service SAP = 116 has MAC 0xaa000503 41200000
```

```
[LLC_SEND} Port 1: hops = 0 to node 'CPM2    ' from PAB list and from SHB 'CPM2
'

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length in network byte order = 0x2e00
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x2
    time stamp is microseconds after second 246672
    UI time stamp = 0x3c390
    UI CRC12      = 0xd6c
    Extended header = 0x3c 39 d 6c in network order
    UI CRC32      = 0x840feafa
[LLC_SEND] Channel ID = 1, SSAP = 114, DSAP = 116, capacity = 90, tokens = 0,
rate = 30 bytes/sec, PID = 0xcc1, SN = 2, Time-stamp = 246672

[LLC_SEND] bytes written 60 for new PCAP record data
[LLC_SEND] length = 34, xmited = 34
    return code 0x10000001, tokens = 0

>>  Status  of  channel  ID  =  1,  SSAP  =  114,  DSAP  =  116,  Token  Bucket
capacity/tokens/rate [in bytes/sec] = 90/0/30, PID = cc1, SN = 2, Time-stamp =
246672

>> Closing channel ID 1
[CLOSE] channel ID 1 is now closed, PID = 0x0

>> Channel ID 1 is now closed

>> Unregistering SAP 114 from CPM1
[UNREGISTER] Service position 0, SAP = 114, PID = 0xcc1
[UNREGISTER] SAP 114 is now unregistered, PID = 0x0

>> CPM1 SAP 114 is now unregistered (PID = 0x0)
```

E.2 Full text output for module CPM2

```
>> Filling NIBs and CIB...
>> Number of nodes 2
>> Node name 'CPM2    ' Equipment ID = 0x341   Unit = 2 MAC Base = aa000503
41200000
>> Node name 'CPM1    ' Equipment ID = 0x341   Unit = 1 MAC Base = aa000503
41100000
>> Configuration name 'TEST001' Host NIB address = 63fc90 Host name from NIB
List 'CPM2    ' Host name from CIB 'CPM2    '
>> Filling CCBs and SIBs...
>> Number of channels 1
>>  Channel  position  0,  ID  =  1,  SSAP  =  114,  DSAP  =  116,  Token  Bucket
capacity/tokens/rate [in bytes/sec] = 90/60/30, PID = 0, SN = 0, Time-stamp =
0
>> Service position 0, SAP = 116, PID = 0

>> Filling SHBs...
>> Number of hosted services 1
>> Hosted service position 0, SAP = 114, Host name from NIB 'CPM1    '

>> Filling PABs and PAB List...
>> Port 1: hops = 0 to node 'CPM1    ' from NIB
...NIBs From PAB List...
>> Port 1: hops = 0 to node 'CPM1    '

>> Filling CBB...
>> Configuration name 'TEST001' Host name from CIB 'CPM2    '
```

```
>> End of initialize()...

>> Entering main()...
>> Registering CPM1 to SAP 116
[REGISTER] Service position 0, SAP = 116, PID = 0x0
[REGISTER] SAP 116 is now registered to PID = 0xcc2 at SIB address 0x412c44

>> CPM2 SAP 116 is now registered to PID = 0xcc2
>> Opening channel ID 1 for access 0x10000053
[OPEN] channel ID 1 is now opened to PID = 0xcc2

>> Channel ID 1 is now open

>> Receiving message with 34 bytes
[RECEIVE] Calling LLC_RECEIVE
[LLC_RECEIVE] bytes read 24 for PCAP file header

    magic number = 0xa1b2c3d4
    major version number = 2
    minor version number = 4
    GMT to local correction = -3
    accuracy of timestamps = 0
    max length of captured packets,in octets = 65535
    data link type = 1
    tokens available = 60
    tokens left = 0 (bytes sent + LLCE header)
[LLC_RECEIVE] bytes read 16 from PCAP record header

    timestamp seconds = 1606821301 (0x5fc625b5)
    timestamp microseconds = 200500 (0x30f34)
    number of octets of packet saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)

[LLC_RECEIVE] bytes read 60 from PCAP record data

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length = 0x2e
    UI CRC12      = 0xf8c
    UI CRC32      = 0x840feafa
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x0
    Extended header = 0x7 40 5f 8c in network order
    UI time stamp = 29701 (0x7405)
    time stamp is seconds after midnight (apparent offset to remote clock =
120 +ahead/-behind)
[LLC_RECEIVE] length = 34, received = 34
    return code 0x10000001, tokens = 0

>> Receiving message with 34 bytes
[RECEIVE] Calling LLC_RECEIVE
    tokens available = 90
    tokens left = 30 (bytes sent + LLCE header)
[LLC_RECEIVE] bytes read 16 from PCAP record header

    timestamp seconds = 1606821304 (0x5fc625b8)
    timestamp microseconds = 231280 (0x38770)
    number of octets of packet saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)

[LLC_RECEIVE] bytes read 60 from PCAP record data

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length = 0x2e
    UI CRC12      = 0x8b3
```

```
    UI CRC32      = 0x840feafa
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x1
    Extended header = 0x38 77 8 b3 in network order
    UI time stamp = 231280 (0x38770)
    delta time in seconds for traffic policing = 3.015157
[LLC_RECEIVE] length = 34, received = 34
    return code 0x10000001, tokens = 30

>> Receiving message with 34 bytes
[RECEIVE] Calling LLC_RECEIVE
    tokens available = 60
    tokens left = 0 (bytes sent + LLCE header)
[LLC_RECEIVE] bytes read 16 from PCAP record header

    timestamp seconds = 1606821305 (0x5fc625b9)
    timestamp microseconds = 246672 (0x3c390)
    number of octets of packet saved in file = 60 (0x3c)
    actual length of packet = 60 (0x3c)

[LLC_RECEIVE] bytes read 60 from PCAP record data

    Source MAC=  aa 0 5 3 41 11
    Destination MAC = aa 0 5 3 41 20
    UI length = 0x2e
    UI CRC12      = 0xd6c
    UI CRC32      = 0x840feafa
    DSAP = 0x74
    SSAP = 0x72
    CTRL = 0x3
    SN = 0x2
    Extended header = 0x3c 39 d 6c in network order
    UI time stamp = 246672 (0x3c390)
    delta time in seconds for traffic policing = 1.015389
    estimated delta time in seconds at origin  = 1.015392
[LLC_RECEIVE] length = 34, received = 34
    return code 0x10000001, tokens = 0

>>  Status  of  channel  ID  =  1,  SSAP  =  114,  DSAP  =  116,  Token  Bucket
capacity/tokens/rate [in bytes/sec] = 90/0/30, PID = cc2, SN = 2, Time-stamp =
246672

>> Closing channel ID 1
[CLOSE] channel ID 1 is now closed, PID = 0x0

>> Channel ID 1 is now closed

>> Unregistering SAP 116 from CPM2
[UNREGISTER] Service position 0, SAP = 116, PID = 0xcc2
[UNREGISTER] SAP 116 is now unregistered, PID = 0x0

>> CPM2 SAP 116 is now unregistered (PID = 0x0)
```