TECHNICAL RESEARCH REPORT

Advanced Orbiting Systems Data Generator/Simulator: A Functional Description of the Software (Version 3)

by J. Baras, G. Atallah, T. Fuja, A. Murad, K.D. Jang

CSHCN T.R. 94-1
(ISR T.R. 94-40)

The Center for Satellite and Hybrid Communication Networks is a NASA-sponsored Commercial Space Center also supported by the Department of Defense (DOD), industry, the State of Maryland, the University of Maryland and the Institute for Systems Research. This document is a technical report in the CSHCN series originating at the University of Maryland.

Web site http://www.isr.umd.edu/CSHCN/

Advanced Orbiting System Data Generator/Simulator: A Functional Description of the Software (Version 3)

Corresponds to CCSDS 701.0-B-2 Blue Book and associated Recommendations for Advanced Orbiting Systems

Dated: May 30, 1994

Ahsun H. Murad, Kap Do Jang,
Dr. John Baras, Dr. George Atallah and
Dr. Thomas E. Fuja

Center for Commercial Development of Space
Institute for Systems Research
University of Maryland
College Park, MD 20742

Abstract

The Advanced Orbiting System (AOS) Data Generator/Simulator is a software implementation of the transmitter (data generation) section of the CCSDS Recommendation 701.0-B-2 for Advanced Orbiting Systems: Networks and Data Links. An object-oriented approach to the simulation of a complex, high-performance communication protocol, it makes full use of the concepts of data-encapsulation and inheritance to ease implementation. The backbone of the software is a general-purpose packet description and generation module that may be used as part of any packet-based simulation software. The user-interface to the program is in the form of a command-language, designed to ease the process of generation of large, multiple data-streams. The output of the program may be configured for interpretation by a graphical user interface (for visual inspection of the data), or as a bit-stream suitable for further processing.

This paper consists of three sections. The first two sections provide a brief, yet comprehensive description of the above CCSDS Recommendation. The various kinds and qualities of user-services, data units involved, and data-paths defined by the protocol are discussed. The different qualities of service (in terms or data reliability) available to the user (and the error-control schemes used to provide them) are also discussed. The last section describes the structure and user-interfaces of the AOS Data Generator/Simulator.

1. Introduction

Between 1982 and 1986, in response to space mission requirements of that period, the *Consultative Committee for Space Data Systems* (CCSDS) developed a series of technical recommendations for the standardization of the common data system functions associated with *conventional* space missions. These recommendations provided a broad basis for the standardization of data communications for many types of unmanned spacecrafts.

To meet the needs for the technologically *Advanced Orbiting Systems* (AOS) of the '90s as evidenced by the international Space Station, the CCSDS decided to extend (in an upward compatible manner) its original recommendations for conventional space-systems to a more diverse and flexible set of data-handling services. Typical AOSs include manned and man-tended space transportation systems, unmanned space platforms, free-flying spacecraft and advanced space transportation systems. To serve the needs of this wide variety of space data-communication applications, the AOS recommendation has provisions for concurrently transmitting multiple classes of digital data (including real-time audio and video) through space/space, space/ground and ground/space data channels at relatively high combined data rates, with high protocol efficiency. The AOS recommendations also provide the capability to interface with, and exploit the rich service environment of worldwide Open Systems Interconnection, allowing for the first time, support for commercially derived network protocols over the space segment.

The principal difference between the conventional and AOS recommendations by the CCSDS is the much wider range of services provided for Advanced Orbiting Systems. Advances in technology (leading to increased on-board processing power) have now made it possible to consider the space segment

as a conceptually symmetric counterpart of its supporting ground network, allowing the provision of symmetric services and protocols for AOS, so that bidirectional exchange of video, audio, high-rate telemetry and low-rate transaction data etc., is possible over space-links.

Because of the varied nature of the data to be transmitted, and the transmission requirements of each over a common link, different transmission schemes (e.g., asynchronous, isochronous and synchronous), different user-data formatting protocols (e.g., bitstreams, octet blocks, and packets), and different grades of reliability (error-control) are provided.

1.1. User Applications

The AOS recommendation supports single space vehicles or constellations of space vehicles which may simultaneously execute a wide-spectrum of applications in near-earth, geostationary or deep-space orbit. Application areas are categorized as either observational science, experimental science or core operations (operation of the space vehicle itself).

1.1.1. Observational Science

Observational science is typically performed from unpressurized platforms in orbit around the earth, or some other planetary body. Typical lifetime of such investigations is in the order of years. The user equipment is fairly stable in terms of location and functionality, and transmission data-rates are usually high. For such applications, the protocol must be optimized to reduce on-board processing and communication bandwidth, and a large degree of flexibility is unnecessary. The CCSDS *Path Service* is particularly suited for the data handling needs of the observational user.

1.1.2. Experimental Science

Experimental science, such as materials processing or the effects of space on human physiology, is conducted primarily in pressurized space vehicles and may require a high degree of flight crew interaction. Such experiments usually have a very limited lifetime, and require a wide variety of interaction between human operators (on both the ground and in space) and experimental apparatus. Hence, source-destination data-communication pairs may be only temporarily associated with any one experiment and these associations typically exist for relatively short sessions. Much of the information generated is processed on-board, and the volume of data transmitted to and from the ground is low. The experimental user requires protocols with routing flexibility, and a rich repertoire of upper-layer data-handling services. The CCSDS *Internet Service* is designed to meet the needs of the experimental user.

1.1.3. Core Operations

The core infrastructure operates and maintains the space vehicle systems that support the payload users. Core user requirements share attributes common to both observational and experimental applications. In addition, since the safety of the space mission (and of human lives) is involved, reliability is a strong concern for the transmission of core data. Core users are likely to use both the Internet and Path service.

1.2. The Space Networking Environment

The space data-communication environment poses unique problems, not found in conventional terrestrial networks. These include very large propagation delays (on the order of seconds or more), low signal-to-noise ratios, high Doppler shifts (due to vehicle motion), and short space-ground contact periods. Because of the intermittent nature of the space/ground link, data must be stored on-board when the link is down, and replayed later, when the link is re-established. Costly tracking facilities, and constraints on on-board power, weight, volume and the costs involved, all suggest a data-handling service that is robust, and optimized for efficiency and low utilization of on-board resources. Further, the data-handling system must provide for the removal of space-transmission related artifacts prior to delivery to the end users.

1.3. Production Data Processing

The removal of space-transmission related artifacts is known as *Production Data Processing*.

Spacecraft in near-earth orbit often have visibility of their ground station for only a few minutes per orbit. Deep-space missions have longer contact periods, but they may occur only once a day, or even once a week. Typically, only manned spacecraft are provided with virtually constant ground contact (subject to interruption due to unavoidable coverage gaps). To provide complete and continuous data sets to mission users, data is stored on board for retransmission during the next contact period, and to protect against loss, some of the data generated during the period of contact is stored as well, leading to overlap between real-time and stored data. If data storage is to tape, rewinding the tape may not be viable (since it reduces its life), and so replayed data is often transmitted in reverse.

Fairly comprehensive processing is needed to remove transmission induced artifacts like data-overlap and reversal. Some spacecraft may also have multiple links to ground, resulting in contemporaneous data being collected separately. Users may also require merging of such data sets. The functions included in production data processing are (i) reversal of on-board recorded data, (ii) removal of overlaps between real-time and stored data, (iii) removal of duplicate data sets, (iv) restoration of the sequence of data, and (v) generation of data-quality information.

The CCSDS recommendation for AOSs recognizes the need for such specialized production data processing, and incorporates features to facilitate their implementation. Because such processing is required only prior to delivery to the end-user, and because significant resources may be required to perform it, production data processing is always done on a ground station.

1.4. Scope of the CCSDS Recommendation for AOS

This recommendation defines a conceptual model for a CCSDS Principal Network (CPN). A CPN serves as (or is embedded within) the project data handling networks which provide end-to-end data flow for space mission users. A CPN consists of an *Onboard Network* in an orbiting segment connected via a CCSDS *Space Link Subnetwork* (SLS) to either a *Ground Network* or to an Onboard Network in another orbiting segment. (See Fig. 1.) Only a limited portion of the Ground Network (called the CCSDS Ground Network) and the Onboard Network (called the CCSDS Onboard Network) are within

the scope of this recommendation. Even though the recommendation uses the notion of a CCSDS Principal Network, it is silent on several issues that must be resolved before the CPN is completely defined. This includes extending the AOS protocols beyond the subnetworks, and complete end-to-end specification of the Path service (which is an end-to-end service provided under the recommendation). Additionally, the recommendation does not address real-world issues like data-storage (during link-down periods) and playback (when the link is resumed), and store-and-forward situations.

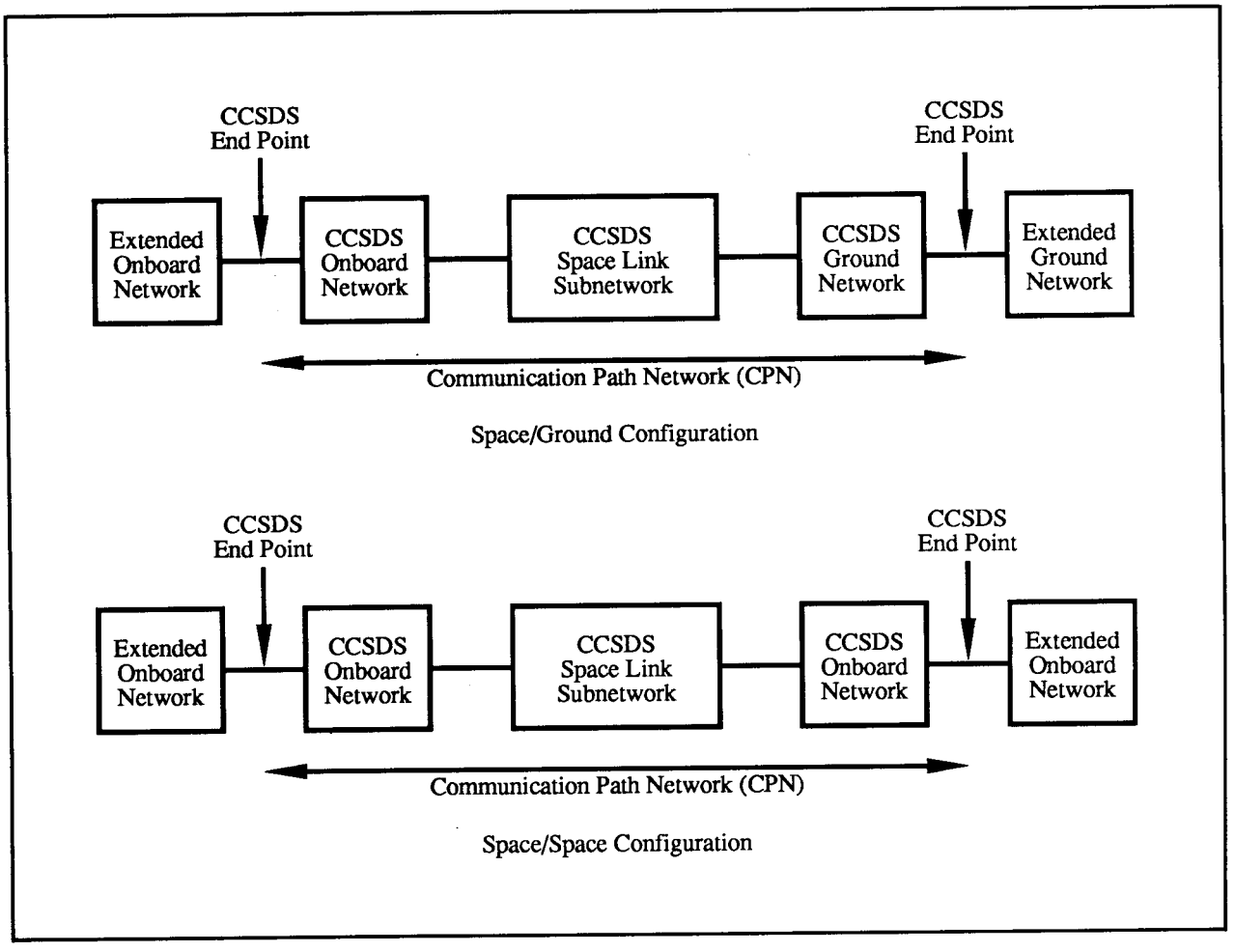


Fig. 1. CCSDS Principal Network Elements

The conceptual model of a typical CPN is shown in Fig. 2. Various kinds of services offered by SLS and the on-board and ground networks under the CCSDS AOS recommendation are indicated. While the figure presents a data-flow scenario between space and ground, the recommendation, due to its symmetric nature, does not differentiate conceptually between a Space Network and a Ground Network. The configuration for space-to-space and space-to-ground networks are therefore, conceptually identical. (Because of various physical and cost limitations on space platforms, the resources available to a ground station may be potentially much greater than that available to a space station. The recommendation is designed with efficiency, and ease of implementation on limited-resource space platforms

in mind. The additional processing power available to ground networks may be used to perform value-added processing prior to delivery of the data to the end-user, and to implement other aspects of the data-communication system, that are associated with, but not part of the recommendation.)

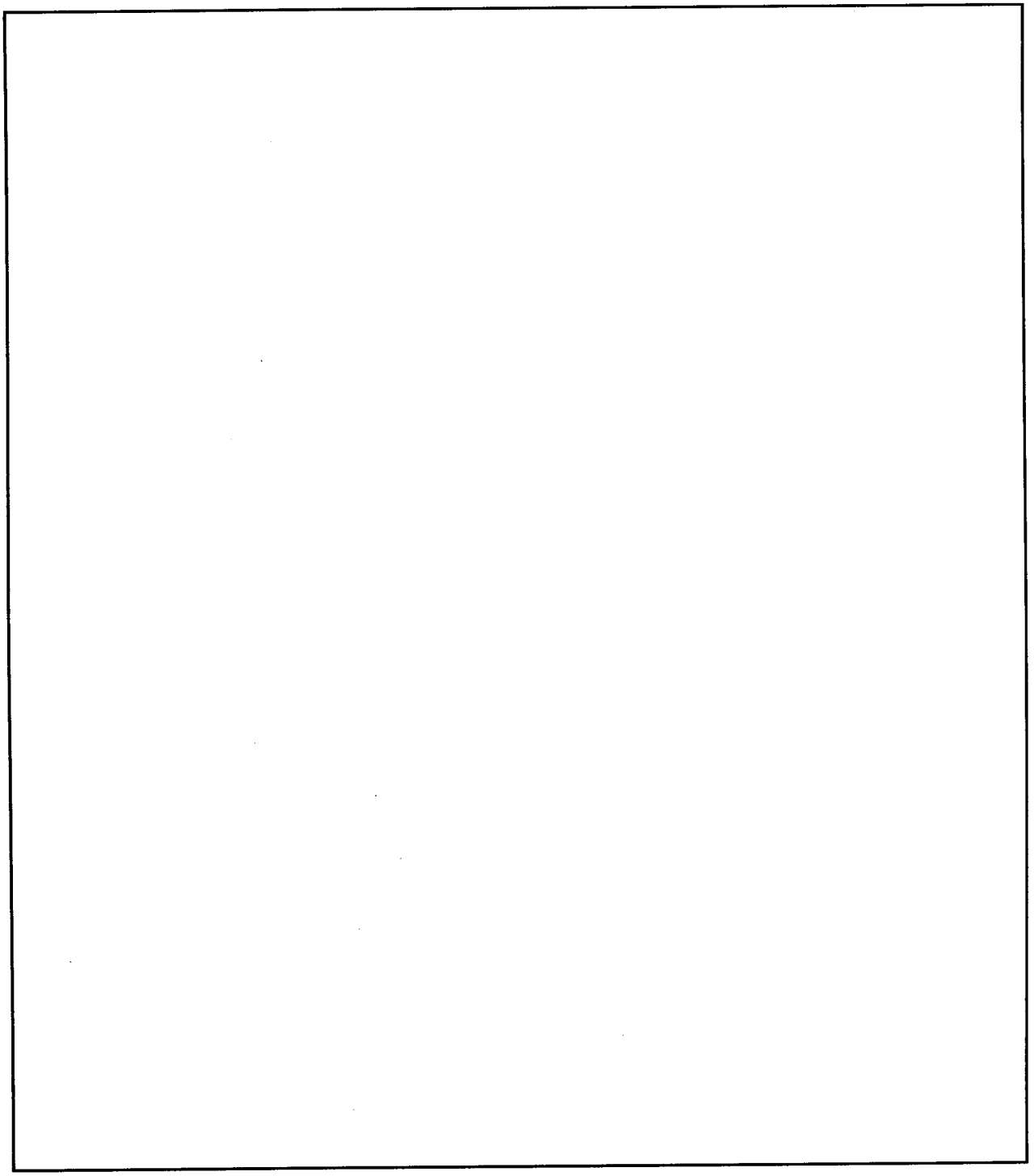


Fig. 2. CCSDS Principal Network Service Model

2. Overview of AOS Services and Protocols

The CCSDS recommendation for Advanced Orbiting Systems provides for eight separate services within a CPN. Two of these (*Internet Service* and *Path Service*) are end-to-end services operating across the entire CPN. The other six (*Encapsulation Service, Multiplexing Service, Bitstream Service, Virtual Channel Access Service, Insert Service* and *Virtual Channel Data Unit Service*) are provided by the SLS.

In keeping with the layered design principle of other major protocols, the CCSDS AOS recommendation provides at the bottom level, the *Physical Layer.* A SLS has available to it, a single physical space channel (either a space-ground link or a space-space link). To allow the transmission of multiple higher-level data-stream over the space link, the SLS uses the concept of Virtual Channels (*Virtual Layer*). A single physical space channel is divided (by time-division multiplexing) into several separate logical data channels, each known as a *Virtual Channel* (VC). (See Fig. 3.)

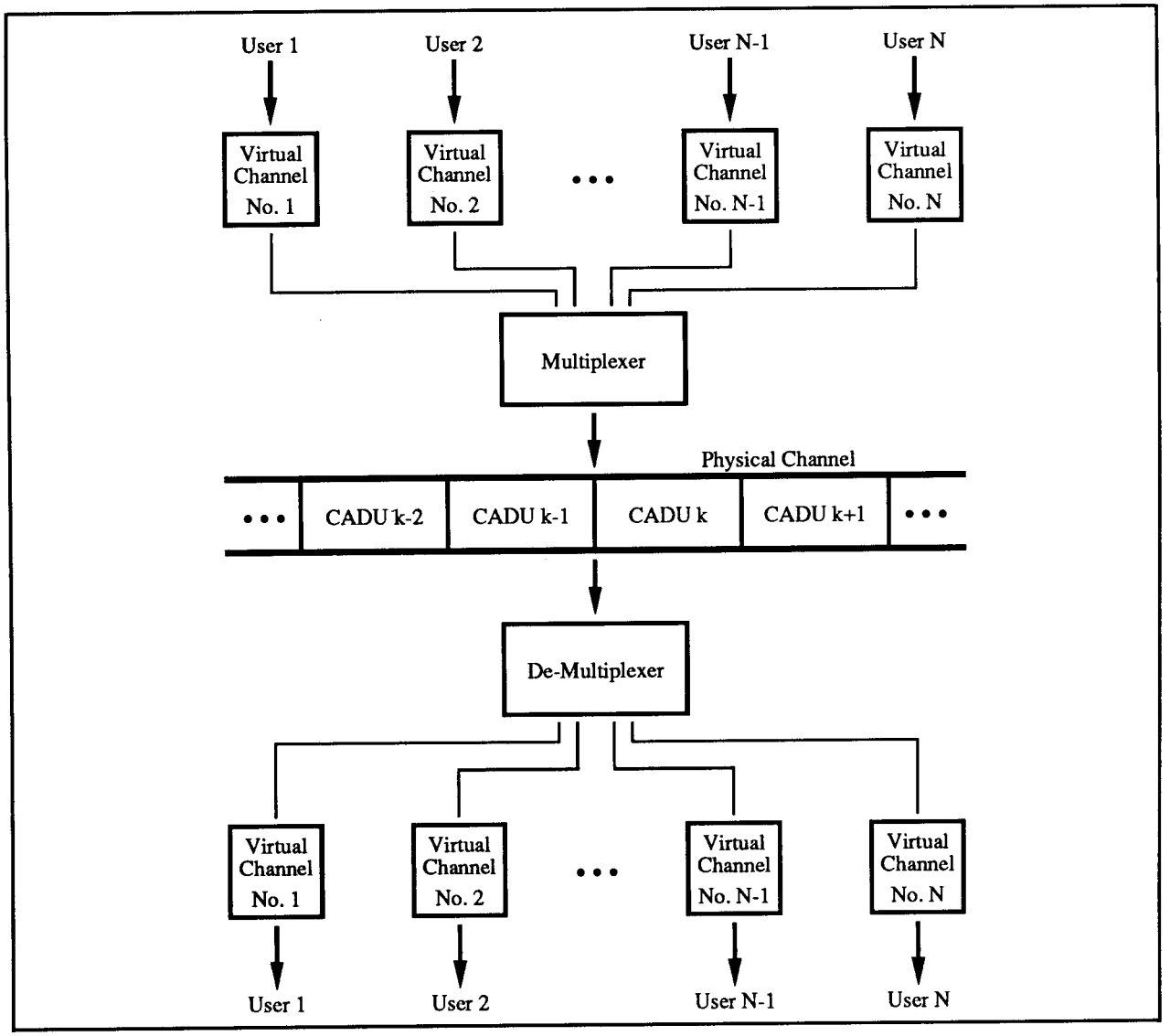


Fig. 3. Virtual Channels

Data flow across each Virtual Channel is *sequence preserving*, and *isochronous*. However, the services using a single Virtual Channel may operate either in *asynchronous* or isochronous mode. Because of the nature of various services (and the applications that use them), it may not be wise to dedicate a single virtual channel to a single data source (e.g., a low-volume Internet Service user may generate data only once a while.) The recommendation allows for multiplexing of multiple data-streams onto a single VC. On a conceptually higher level than the virtual layer, the recommendation also defines a *Path Layer*, which provides support for the end-to-end Path and Internet Service, and through a process called encapsulation, provides support for the integration of commercial protocols with the CCSDS AOS recommendation.

2.1. User Services

The CCSDS AOS recommendation provides for two end-to-end services: Path Service and Internet Service, both of which operate across the CPN. Since both Internet and Path Services rely on underlying sub-networks, which may not be sequence preserving, end-to-end data transfer using these services is defined to be non sequence-preserving. The Space Link Subnetwork, however, does preserve sequence.

2.1.1. Path Service

The Path Service is implemented using a special-purpose protocol designed by the CCSDS, optimized to handle telemetry data (e.g., measurement data from payload instruments), which is characterized by moderate to very high data rates, large volumes of structured, delimited data units between fairly static source and destination associations. The Path Service provides high processing speed and efficiency at the cost of flexibility.

To support the Path Service, *Logical Data Paths* (LDPs) which identify the fixed route between the source/destination pair are preconfigured by management. Each LDP is uniquely identified by a *Path Identifier*. Data is relayed across the CPN by tagging each packet with the thin Path identifier, rather than extensive global source-destination routing information. Routing decisions are made by using the Path ID as an index in routing tables supplied by management, giving the next point in the data flow.

The protocol data unit of the Path Service is called a *CCSDS Path Protocol Data Unit (CP_PDU)* and follows the format of the *Version-1 CCSDS Packet*, which will be described later.

The Path Service offers two service options: (i) a *Packet* service transfers CP_PDUs preformated by the user, intact across the CPN, and (ii) an *Octet String* service transfers delimited strings of user octets (8-bit words) across the CPN, by building them into CP_PDUs on the user's behalf.

The Path Service is available only in asynchronous mode.

2.1.2. Internet Service

The CCSDS Internet Service complements the Path Service by providing a large degree of flexibility in support of interactive applications at the cost of speed and efficiency. The CCSDS has chosen the commercially supported ISO 8473 connectionless network protocol for use within the Internet Service,

allowing space missions to exploit the rich upper-layer service structure of the OSI. This protocol features full addressing of the source and destination Service Access Points (SAPs), and the possibility of partial or full source routing, at the expense of a larger and more complex communications header than is provided within the Path Service.

This service is expected to be used for intermittently transferring, at low data-rates, low-to-moderate volumes of structured, delimited data from a single source to a single destination. This service could be used to support, for example, real-time interactive command and control operations, file transfer and interactive operations like electronic mail and remote-terminal access.

Within the SLS, the ISO 8473 packet is encapsulated into a CP_PDU using the *Encapsulation Service* (to be explained later) provided by the SLS.

The Internet Service is available only in asynchronous mode.

The SLS, which forms the core of the CPN supports bi-directional transmission of Path and Internet Service data units. It also provides six other services which do not extend further through the CPN.

Transmission of data across space-space and space-ground links poses problems unique to this environment. The CCSDS has designed customized protocols that make efficient use of the channel, and at the same time, make channel characteristics invisible to the higher layers.

The CCSDS Data-Link Layer (Physical Layer) uses fixed-length frames of data, with boundaries delimited by a 32-bit pseudo-noise encoded synchronization marker. A Virtual Channel identifier (VCID) is inserted into each frame header, allowing the implementation of a number of VCs on the same physical link (Virtual Layer). The basic protocol data unit of the Virtual Layer is known as a *Virtual Channel Data Unit* (VCDU).

To clean up the noise introduced by the low-SNR Physical Layer, two optional error-control schemes may be implemented: a bit-oriented convolutional code to encode the entire data stream, and/or a block-oriented Reed-Solomon code. The Reed-Solomon Code may be applied to selective Virtual Channels; a Reed-Solomon encoded VCDU is known as a *Coded VCDU* (CVCDU), and supports almost error-free transmission across the space-link.

Each spacecraft in the domain of the CCSDS AOS recommendation is assigned a unique *Spacecraft Identifier* (SCID). (In some cases, a more advanced space platform may be assigned more than one SCIDs.) The SCID is inserted into the frame header of each frame in the stream of VCDUs/CVCDUs generated by the spacecraft.

2.1.3. Virtual Channel Data Unit (VCDU) Service

In some AOS configurations, a spacecraft generating its own stream of VCDUs/CVCDUs (identified with its own SCID) may wish to accept a stream of VCDUs/CVCDUs that has been generated with another spacecraft (with a different SCID). The two streams may then be merged for transmission across a single space-link.

The VCDU Service allows such independently created VCDUs/CVCDUs from a guest spacecraft to be inserted frame by frame into the data stream of the host spacecraft without any protocol checking.

This service is available only to "trusted" guest users, who are certified during the design process to ensure that the independently created data units do not cause protocol violations.

2.1.4. Virtual Channel Access (VCA) Service

The VCA Service provides a facility whereby a project may transfer private service data units (which have size exactly equal to the fixed-length data-field of one dedicated VCDU/CVCDU), and of unknown content, across the SLS. The VCA Service is likely to be used to transmit high-rate video, telemetry information, or a privately encrypted data block.

2.1.5. Bitstream Service

The Bitstream Service allows a string of bits, whose internal structure and boundary is unknown to the data transmission system, to be transmitted across the space link. The Bitstream Service breaks down the stream of bits from each SLS user into blocks called *Bitstream Protocol Data Units* (B_PDUs) which are sized to exactly load the fixed-length data-field of one dedicated VCDU. (Fill data may be added and removed transparently as necessary to match the bitstream to the fixed length data field.) Bitstreams from different users may not be multiplexed onto the same virtual channel.

The Bitstream Service was originally intended primarily to support the bit-oriented replay of on-board tape recorders. The transfer is sequence preserving and may be either asynchronous or isochronous. (Isochronous transfer is provided with a specified maximum delay and a specified maximum jitter at the service interface. High rate video data may use this service in the isochronous mode.)

The format of the B_PDU is as shown in Fig. 4.

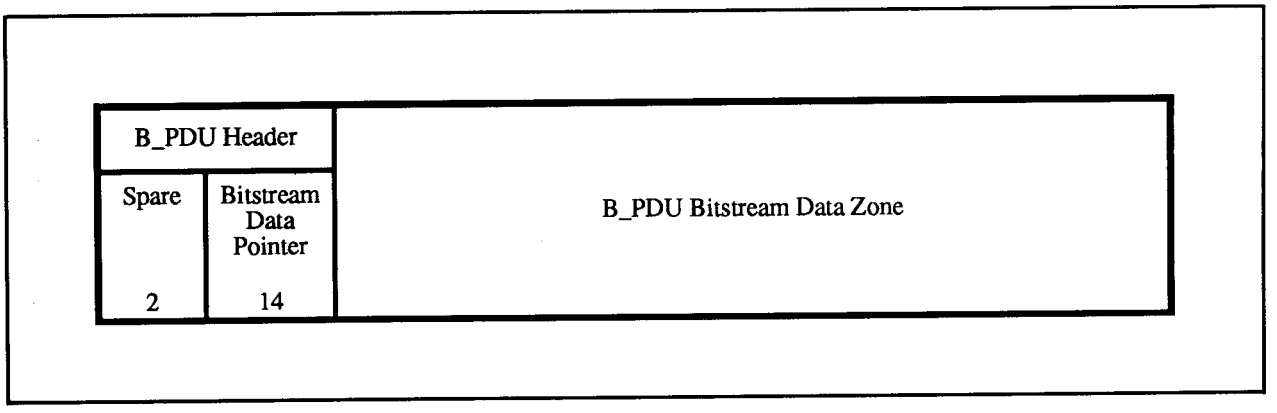


Fig. 4. Bitstream Protocol Data Unit (B_PDU)

SPARE (Bits 0–1) Currently undefined by CCSDS, Shall be set to "00".

Bitstream Data Pointer (Bits 2–15) When incremented by one, gives the location of the last valid user data in the Bitstream Data Zone. Project defined fill follows this position. If there is no fill data, this is set to "all ones". If there is no user data, this is set to "all ones minus one".

2.1.6. Insert Service

The Insert Service allows small fixed-length, octet-aligned service data units to be transferred isochronously across the SLS in a mode which efficiently utilizes the channel at relatively low data rates. When activated, the Insert Service is implemented by establishing a small *Insert Zone* in every frame that is transmitted across the particular Physical Channel, and by placing the Insert Service Data Units in this zone. (See Fig. 6–8.) When the Insert Service is used, the Insert Zone must be present in all Virtual Channels that share the same Physical channel, and its length is established by management. Since the size of the insert zone in each frame, and the size of each frame are fixed, a regular sampling interval is provided, and the Insert Service is therefore isochronous. VCA Service and Insert Service are mutually exclusive over a SLS: A SLS supporting Insert Service will not allow VCA Service users.

The most likely use of the Insert service is to support digitized audio over low-rate space links.

2.1.7. Encapsulation Service

The Encapsulation Service supports end-to-end services by allowing variable-length, octet aligned "foreign" service data units, that are not formatted as CP_PDUs to be transferred transparently through the space link. The CCSDS Path Packet Service, which uses CP_PDUs is directly compatible with the Multiplexing Service (to be explained later), and therefore bypasses the Encapsulation Function. The CCSDS Internet Service however uses the ISO 8473 packet format and uses this Service. The Encapsulation Service simply takes any delimited service data unit (in a non Version-1 CCSDS Packet format) (e.g., the ISO 8473 packet) and encapsulates it within a special Version-1 CCSDS "carrier" Packet called the *Encapsulation Protocol Data Unit* (E_PDU), which is compatible with the Multiplexing Service. At the other end of the SLS, the carrier Packet is stripped off, and the original service data unit continues its path through the CPN. The Encapsulation Service therefore provides the flexibility to support different OSI network-layer protocols across the CPN.

The format of the E_PDU is as shown in Fig. 5. This is identical to the structure of the CCSDS Version-1 Packet.

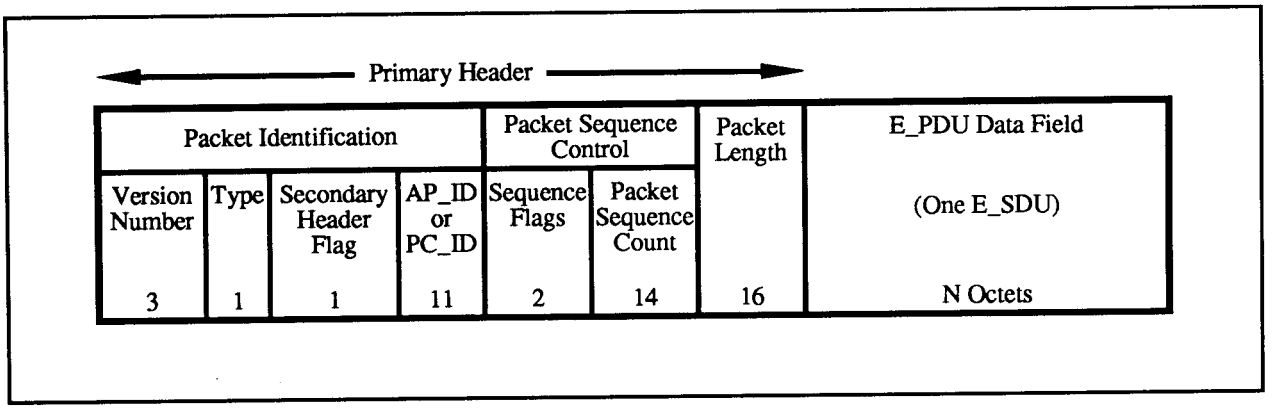


Fig. 5. Encapsulation Protocol Data Unit (E_PDU)

Version Number (Bits 0–2) Set to "000" indicating a Version-1 CCSDS Packet.

Type (Bit 3) Unused. May be set to "0" or "1".

Secondary Header Flag (Bit 4) Set to "0", since there is no secondary header in the E_PDU.

Application Process ID (AP_ID) or Packet Channel ID (PC_ID) (Bits 5–15) The assignment of this field is as shown in Table. 1.

PC_ID (11 Bits) (Hexadecimal Equivalent)	Utilization
7FF	Reserved by CCSDS to identify a "Fill Packet".
7FE	Reserved by CCSDS to Identify a Flow of Encapsulated ISO 8473 Packets.
7F0–7FD	Reserved by CCSDS for possible Future Use.
000–7EF	Available for User Domain Assignment by Project Organizations.

Table. 1. Packet Channel/Application Protocol Identifier Allocations

Sequence Flags (Bits 16–17) Set to "11".

Packet Sequence Count (Bits 18–31) This field contains a straight sequential count (modulo 16384) which numbers each E_PDU generated on each reserved Packet Channel (identified by the PC_ID). The count shall be incremented independently for each Packet Channel. If the Packet Channel contains fill data (PC_ID = "7FF" (hexadecimal)), the count shall be permanently set to the value "all zeros".

Packet Length (Bits 32–47) This contains the binary number corresponding to $N-1$, where N is the length of the E_PDU data field in octets.

E_PDU Data Field (Bits 48 onwards) This contains one E_SDU. The length of this field must be an integer number of octets.

2.1.8. Multiplexing Service

To efficiently utilize the space channel, variable-length service data units must be packet together so that they fully occupy the fixed-length data-zone of the CVCDU. Incoming CCSDS Packets are simply concatenated back-to-back, until they fill the data-zone of the CVCDU. The header of the Multiplexing Protocol Data Unit (M_PDU) (the data unit constructed by the Multiplexing Service) contains a pointer to the boundary between the first Packet pair, and individual Packet length fields then delimit the other boundaries. Multiplexing service can therefore accept both Encapsulation Service data units (encapsulated Internet Service packets) and Path Service data units and concatenate them within one Virtual Channel if desired. Several "Packet Channels" (each locally identified with an *Application Process ID* (APID)) may thus concurrently share a VC using the Multiplexing Service. (Path Service users with

a higher data-rate requirement may use dedicated VCs.) The demultiplexing and delivery of CCSDS Packets is achieved by examining the APID and *Packet Length* fields in the Packet headers. Since these fields are not protected against errors, demultiplexing and delivery can be performed reliably only if the the M_PDUs are placed within Reed-Solomon protected CVCDUs.

The format of the M_PDU is as shown in Fig. 6.

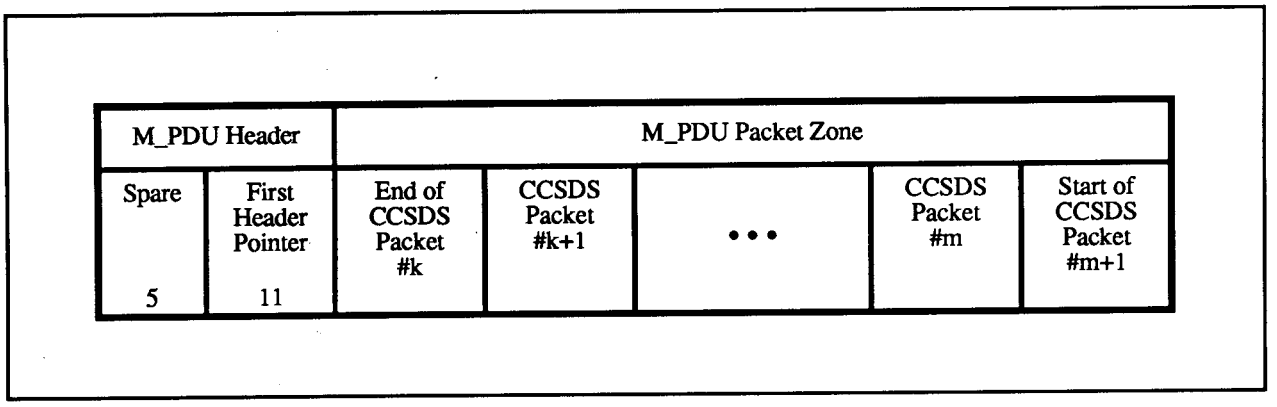


Fig. 6. Multiplexing Protocol Data Unit (M_PDU)

Spare (Bits 0–4) Currently undefined by CCSDS. Shall be set to "00000".

First Header Pointer (Bits 5–15) When incremented by one, gives the location in octets of the first CCSDS Packet header in the M_PDU Packet Zone. If the Packet Zone contains fill data, it is set to "all ones minus 1", and if the Packet zone does not contain the start of any header (if a packet being multiplexed is too long), it is set to "all ones".

M_PDU Packet Zone (Bits 16 onwards) Contains a variable number of (variable length) CCSDS Version-1 Packets (E_PDUs and CP_PDUs). The first and last packet of the M_PDU may not be complete, since the first one may be a continuation of a packet begun in the previous M_PDU, and the last one may continue into the next M_PDU.

2.2. Grades of Service

The various types of data transmitted over the space link may not have the same requirements for data quality and reliability. For instance, asynchronous packetized data transmission requires virtually error-free service, whereas raw video-data can tolerate fairly high error-rate without noticeable degradation. The CCSDS therefore provides three different *Grades of Service*. These Grades of Service are only defined across a space link, and do not extend automatically to end-to-end service. The error-control is provided using a combination of error detection, error correction and retransmission. Each VC supports a single Grade of Service.

2.2.1. Grade 3 Service

Grade 3 Service provides the lowest quality of service. Data transmitted using Grade 3 may be incomplete (due to lost packets) and there is a moderate probability of transmission induced errors being present. Further, data packet sequence may not be preserved.

The raw VCDU (without Reed-Solomon encoding) supports Grade 3 service. The error-rate therefore is that of the underlying Physical Layer (which may be encoded with a convolutional code). To protect critical VCDU Header routing information, a special header error-control code is provided. To detect errors in other fields of the VCDU, a CRC error control code (which covers the entire VCDU) is provided.

Grade 3 is not suitable for asynchronous packetized data transfer because of inadequate protection of control information in the packet headers. The structure of a Grade 3 frame is as shown in Fig. 7.

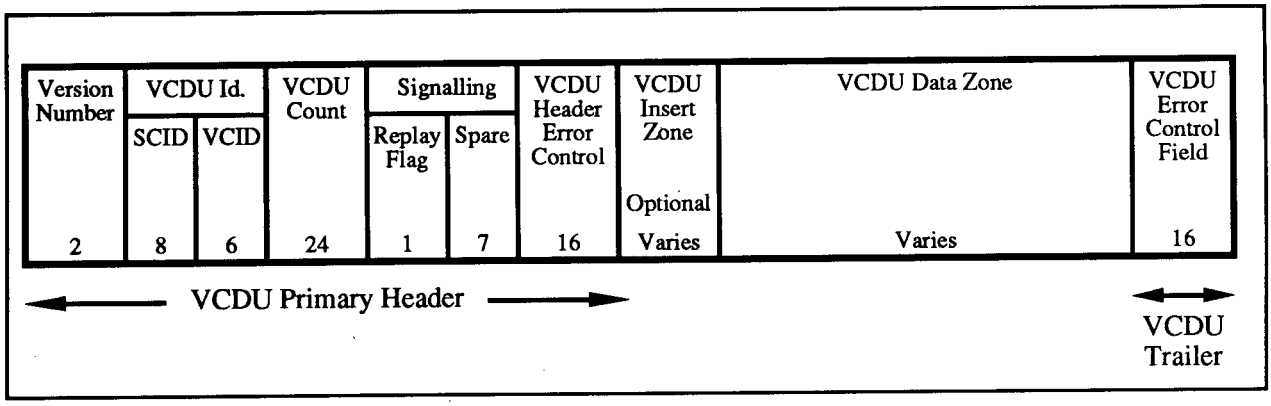


Fig. 7. Grade 3 CVCDU

2.2.2. Grade 2 Service

Grade 2 Service provides a much higher quality of service than Grade 3. Data transmitted using Grade 2 may be incomplete (due to lost packets), but data sequencing is preserved and there is a very low probability of induced transmission errors being present. The structure of a Grade 2 frame is as shown in Fig. 8.

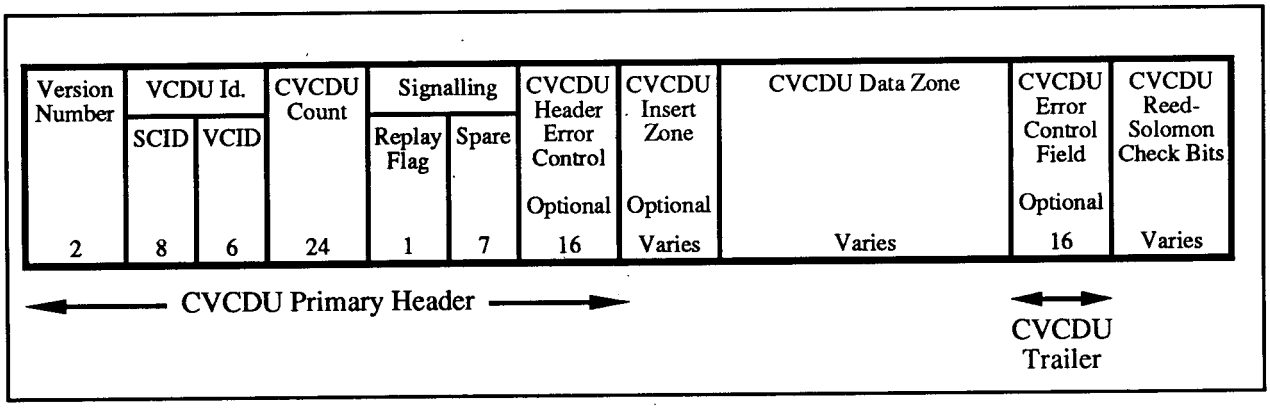


Fig. 8. Grade 2 CVCDU

The Grade 2 Service is implemented by using a powerful Reed-Solomon code. Because the overhead induced by this code is small, and is fully compensated by the huge coding gain, many missions implement the Grade 2 Service option only.

2.2.3. Grade 1 Service

Data transmitted using the Grade 1 Service option enjoys the highest reliability: Packets are delivered in sequence, without duplication or deletion, and with a very low probability of induced transmission errors.

Grade 1 Service is implemented using a Reed-Solomon forward-error correction scheme to correct most transmission errors, and an Automatic Repeat Queuing (ARQ) retransmission scheme (Go Back n) to retransmit those packets that are found to contain uncorrectable errors. A *Space Link ARQ Procedure* (SLAP) has been developed, which uses two paired VCs operating in opposite directions to implement bi-directional Grade 1 Service. The structure of a Grade 1 frame is as shown in Fig. 9.

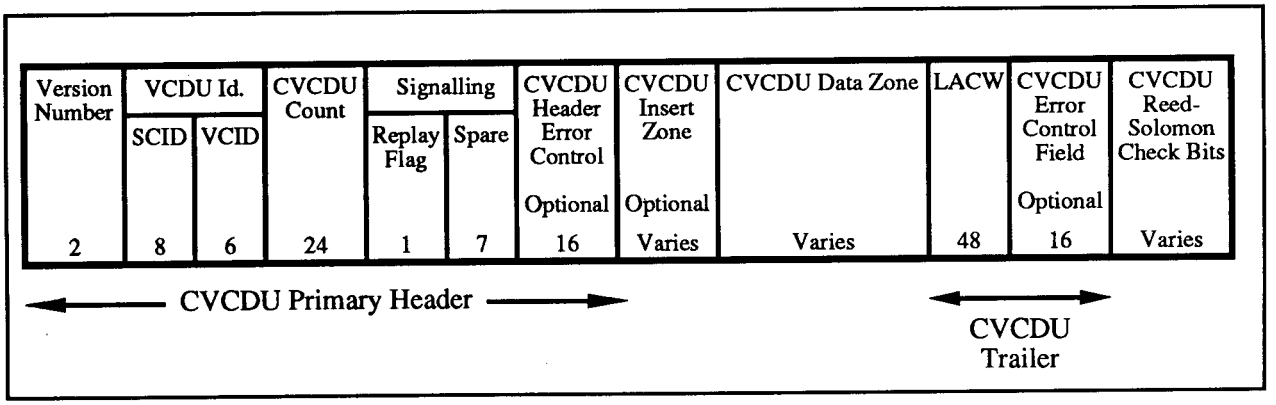


Fig. 9. Grade 1 CVCDU

Note that an implementation is not required to support all three Grades of Service. The Grades of Service are not signaled by protocols, but are rather set up by management.

Grade of Service	Retransmission Protocol	Reed-Solomon Encoding	Header Error Control	CRC Error Control
Grade 1	M	M	O/M	O
Grade 2	N	M	O/M	O
Grade 3	N	N	M	M

Table. 2. Features of SLS Grades of Service
N – Not Applicable, M – Mandatory, O – Optional. O/M – this feature is usually optional but may be mandatory in some cases.

	1		2		3		Grade of Service
	I	A	I	A	I	A	Transfer Type
SLS Services							
Multiplexing		×		×			
Encapsulation		×		×			
Bitstream		×	×	×	×	×	
Virtual Channel Access		×	×	×	×	×	
Virtual Channel Data Unit			×	×	×	×	
Insert			×		×		

Table. 3. Allowable Transmission Types and Grades of Service
×: allowed; I: isochronous mode; A: asynchronous mode.

2.3. Physical Layer

The Physical Layer supports the transmission of a sequence of bits across a single space-link. The Physical Layer encompasses all those components of the CCSDS AOS recommendation that are defined on a bit-level. These include frame synchronization, the use of NRZ-M/NRZ-L notations, convolutional encoding and pseudo-randomization. The Physical Layer accepts a sequence of frames (VCDUs/CVCDUs) from the Virtual Layer, and converts it into a data stream. In doing so, a synchronization marker called an *Attached Synchronization Marker* (ASM) is attached to the beginning of each frame (and the frame is pseudo-randomized if required), converting it into a *Channel Access Data Unit* (CADU) and the sequence of CADUs are concatenated to produce a bitstream. The frame (or its pseudo-randomized form) occupies the data zone of the CADU called the Channel Access Slot. (See Fig. 10.) Most of the bit-oriented Physical Layer functions operate on this bitstream.

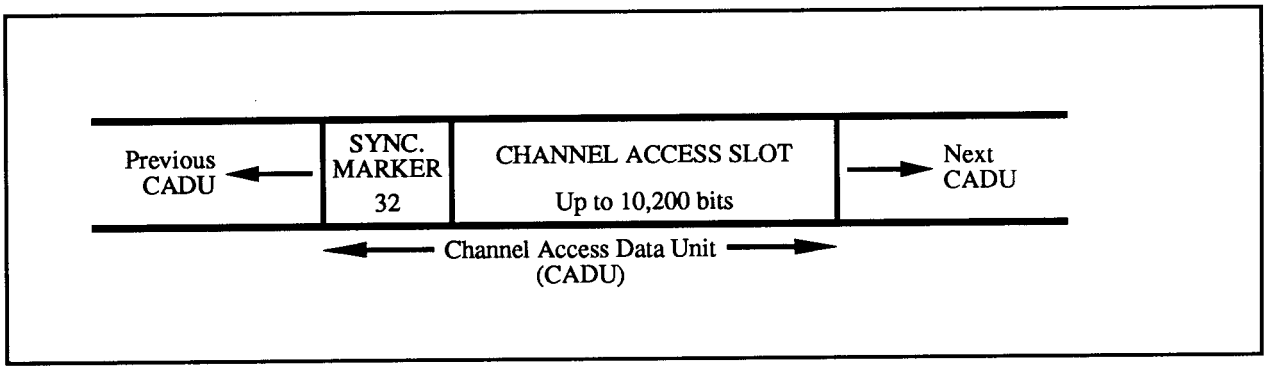


Fig. 10. Structure of the CADU

A functional overview of the physical layer, showing the various options available for control of the transmitted bit-stream is shown in Fig. 11.

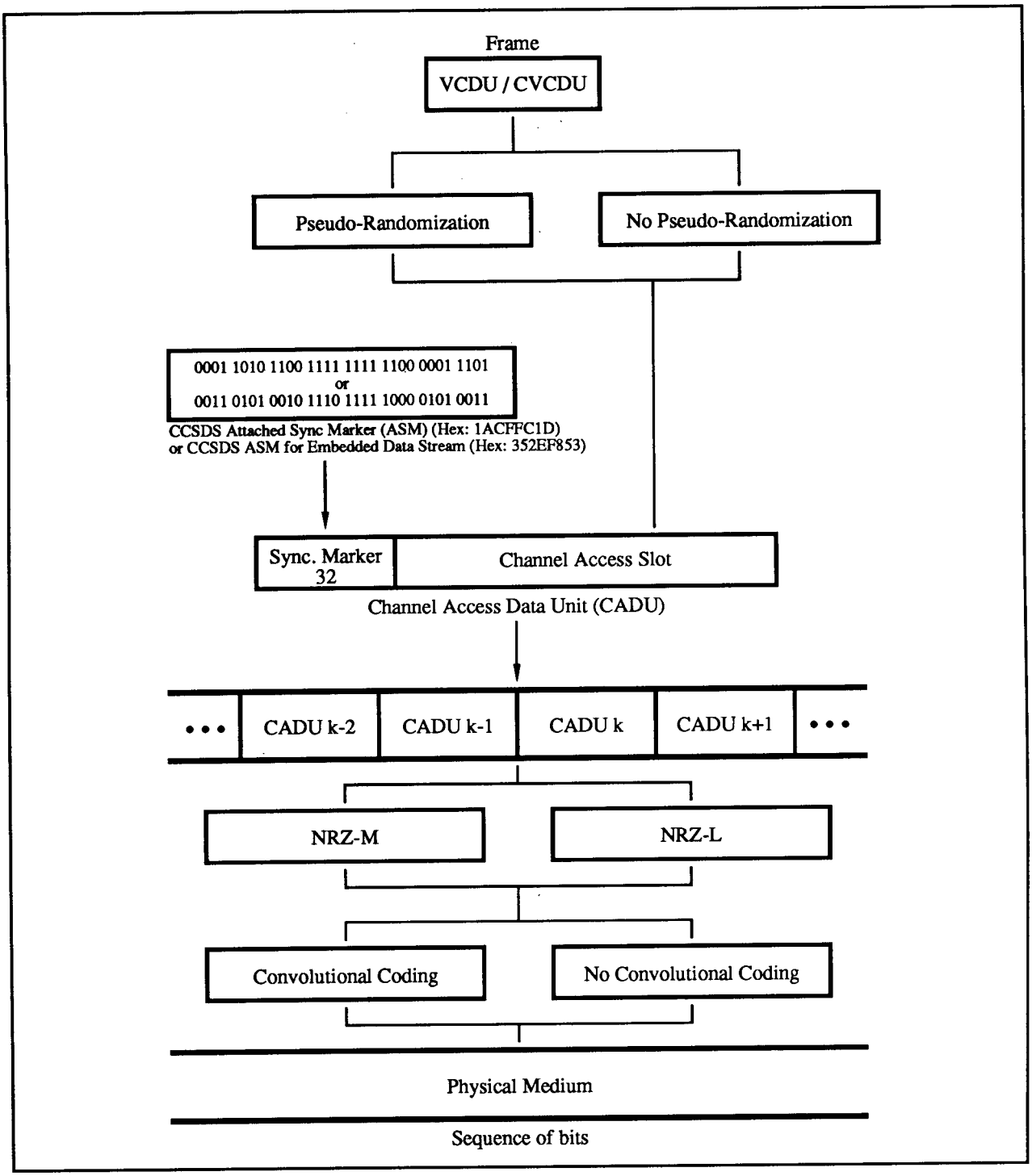


Fig. 11. Functional Overview of Physical Layer

2.3.1. Pseudo-Randomization

In order to maintain bit synchronization on an asynchronous link, the receiver requires that the incoming signal have a minimum density of bit transitions (so that the clock may be extracted). If a

sufficient bit-transition density is not assured by other means (e.g., choice of modulation scheme or use of the convolutional code) then a pseudo-randomizer is required. Its use is optional otherwise.

The method for ensuring adequate bit-transitions is to XOR the data sequence with a pre-determined binary pseudo-random sequence. The sequence to be used is specified by the polynomial

$$h(x) = x^8 + x^7 + x^5 + x^3 + 1.$$

The sequence generator is kept initialized to an all-ones state during the Synchronization Marker period. The first 40 bits of the sequence are as follows

1111 1111 0100 1000 0000 1110 1100 0000 1001 1010 ...

The left-most bit above is XORed with the first bit of the VCDU. The logic diagram of the pseudo-randomizer is shown in Fig. 12.

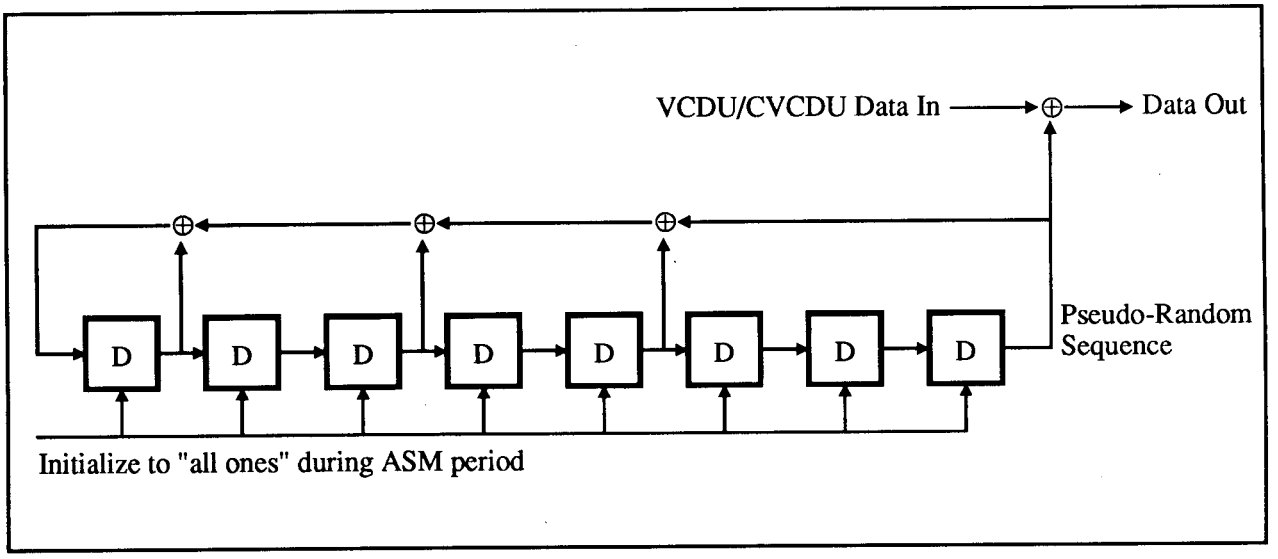


Fig. 12. Pseudo-Randomizer Logic Diagram

2.3.2. Frame Synchronization

Frame synchronization is the process of detecting the boundaries between frames. It is necessary for proper decoding of the Reed-Solomon code block, synchronization of the pseudo-randomizer and assists in the synchronization of the Viterbi decoder.

Synchronization is achieved by using a fixed block-size for the frame (i.e., all VCDUs/CVCDUs transmitted over a link have to have the same length) and by inserting a CCSDS Attached Sync Marker (ASM) at the head of each frame. Synchronization is acquired at the receiving end by recognizing the specific bit pattern of the ASM in the received data stream. Synchronization is then customarily confirmed by making further checks. The ASM used shall be the 32-bit marker

0001 1010 1100 1111 1111 1100 0001 1101

corresponding to the hexadecimal number 1ACFFC1D. The first bit transmitted shall be the leftmost bit, and the last bit transmitted shall be the rightmost bit.

A different ASM may be required when another data stream (e.g., a stream of transfer frames played back from the tape recorder played in the forward direction) is embedded within the data field of the transfer frames of the main stream appearing on the link. The ASM for the embedded data stream (to differentiate from the main data stream) shall consist of a 32-bit marker

0011 0101 0010 1110 1111 1000 0101 0011

corresponding to the hexadecimal number 352EF853.

2.3.3. NRZ-M and NRZ-L Options

Two conventions are provided to convert the binary data into a modulating waveform. In the *NRZ-L* notation, a "1" represents one level (usually high), and a "0" represents the other level (usually low). In the *NRZ-M* notation, a "1" represents a change in level, and a "0" represents no change in level. The NRZ-M notation has the added advantage of being immune to phase-reversal (i.e., complementation of all bits). See Table. 4.

Last Output Bit	Current Input Bit	NRZ-M Output	NRZ-L Output
0	0	0	0
0	1	1	1
1	0	0	1
1	1	1	0

Table. 4. NRZ-M/NRZ-L Encoding

Decoding is done according to Table. 5 below.

Last Received Bit	Current Received Bit	NRZ-M Output	NRZ-L Output
0	0	0	0
0	1	1	1
1	0	0	1
1	1	1	0

Table. 5. NRZ-M/NRZ-L Decoding

2.3.4. Convolutional Encoding/Decoding

In case the coding gain provided by the other coding methods is not enough, an option is provided to encode the data using a rate 1/2 convolutional code in Fig. 13. The bit in position 1 is the first bit to be transmitted, the bit in position 2, the second. The symbol corresponding to the second bit is inverted to provide adequate transition density (otherwise, a long sequence of zeros at the input of the

convolutional encoder would result in a corresponding sequence of zeros at the output). The decoder is a maximum-likelihood (Viterbi) decoder with at least 3-bit quantization for soft decisions. When the convolutional decoder makes a decision error, this results in a burst of errors. The Reed-Solomon code (especially with an interleaving depth greater than one) provides good protection against burst errors.

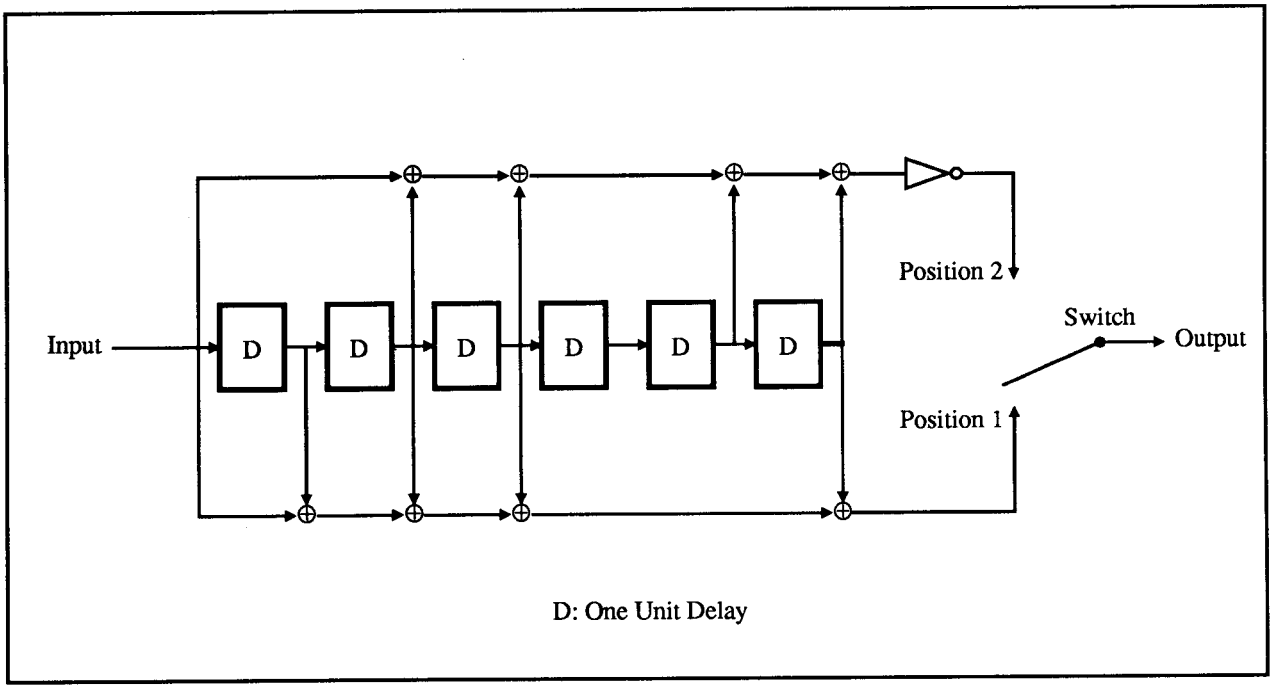


Fig. 13. Rate 1/2 Convolutional Encoder

2.4. Virtual Layer

In this section, we give a more detailed description of the Virtual Layer—which corresponds to the implementation of Virtual Channels. The Virtual Layer provides for the establishment of a number (a management parameter) of Virtual Channels over the same physical channel. Each Virtual Channel operates independently and provides for the transmission of a stream of VCDUs/CVCDUs. A functional block-diagram of the Virtual Layer is presented in Fig. 14.

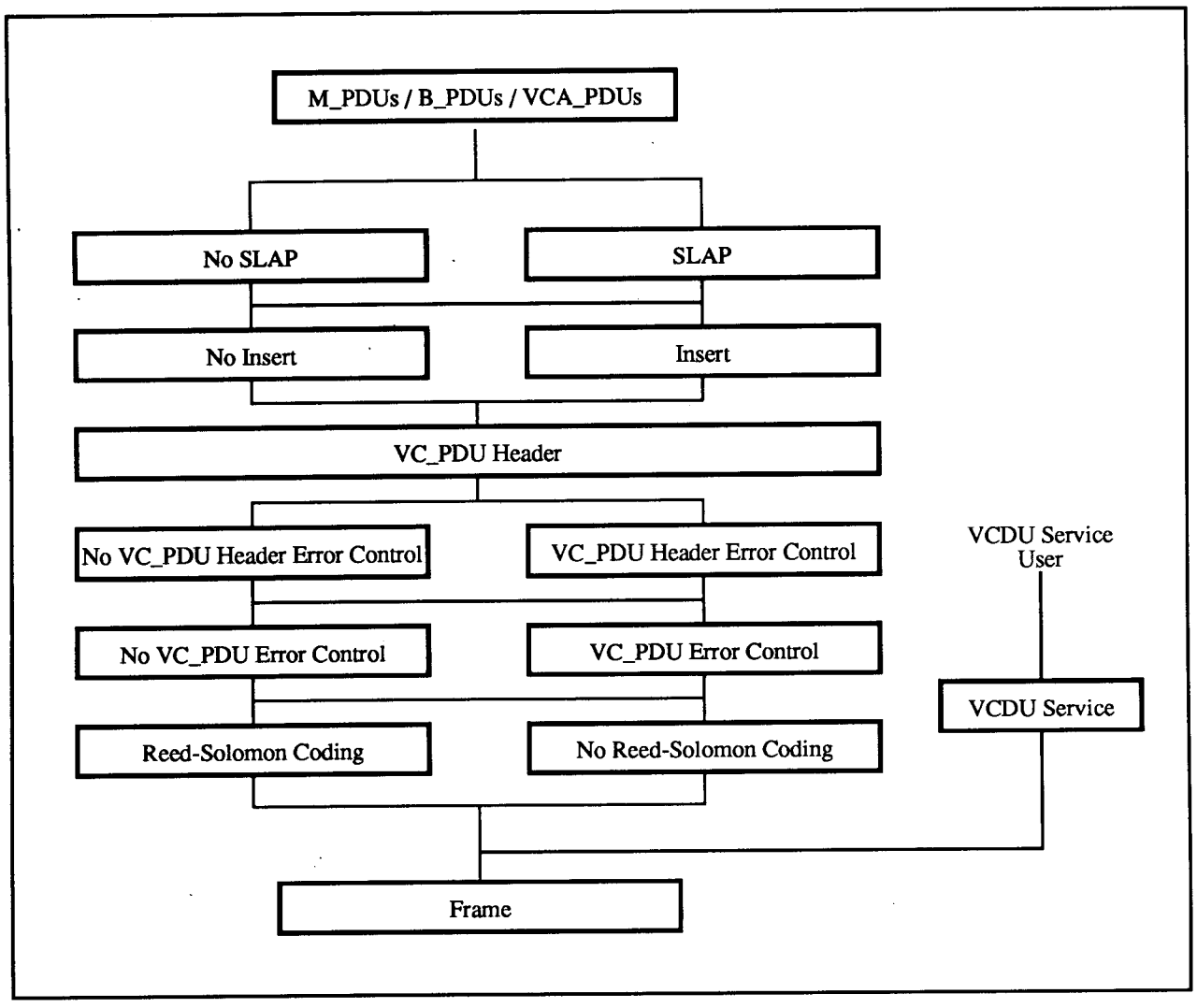
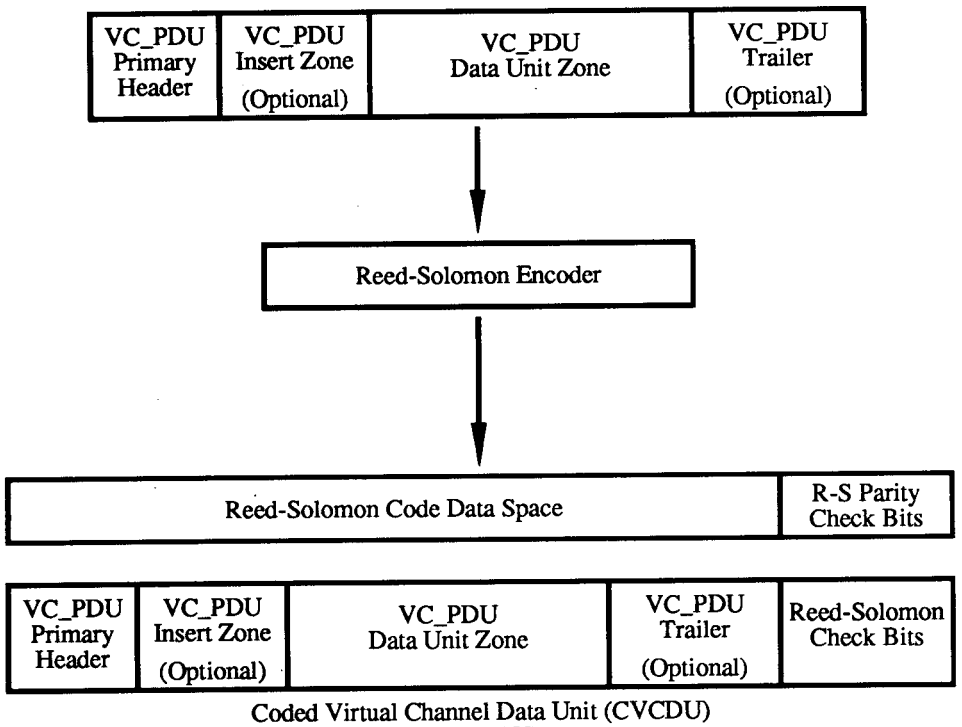


Fig. 14. Functional Overview of the Virtual Layer

2.4.1. Reed-Solomon Error Control

One of the functions of the Virtual Layer is the construction of frames. Depending on the Grade of Service implemented, the frame may be a VCDU (uncoded) or a CVCDU (coded). For a Grade 1 or Grade 2 Service option, Reed-Solomon error-control coding is implemented. A functional block-diagram description of the procedure is presented in Fig. 15. The Reed-Solomon Coding Function performs the action of construction of a CVCDU at the transmitting end, and error correction/detection of the received CVCDU at the receiving end. If the error was uncorrectable, a flag is set to inform higher level functions about this.



Coded Virtual Channel Data Unit (CVCDU)
(VC_PDU)

(2040, 1784) code used with
Interleave depth up to 5

Interleave Depth	VC_PDU Length	32-Bit Compatible VC_PDU Length
1	255 octets 2040 bits	252 octets 2016 bits
2	510 octets 4080 bits	508 octets 4064 bits
3	765 octets 6120 bits	756 octets 6048 bits
4	1020 octets 8160 bits	1020 octets 8160 bits
5	1275 octets 10200 bits	1260 octets 10080 bits

Table of Standard VC_PDU Lengths
(For any link transmitting CVCDUs)

Fig. 15. Reed-Solomon Encoding

2.4.2. VCDU Error Control

For those frames that are not protected with a Reed-Solomon code (Grade 3) it is mandatory to implement VCDU Error Control Coding, which provides excellent error-detection for VCDUs. (For CVCDUs, the Reed-Solomon Encoder provides excellent error-detection capabilities, even when it cannot correct the error, and VCDU Error Control Coding is therefore not required.) The VCDU Error Control field contains a 16-bit cyclic redundancy code (CRC) which provides the capability to detect transmission

errors. Since the Header Error Control Code (discussed later) independently protects key elements of the VCDU Primary Header, the main use of the CRC is to detect errors occuring elsewhere in the VCDU structure.

The CRC is characterized by the generator polynomial

$$g(x) = x^{16} + x^{12} + x^5 + 1.$$

Both the encoder and the decoder are initialized to "all ones" at the start of each VCDU. Parity generation is performed over the entire VCDU (excluding the last 16-bit VCDU Error Control field), and the generated parity symbols are inserted into the final 16-bit VCDU Error Control field. The logical block-diagram for the CRC is as shown in Fig. 16.

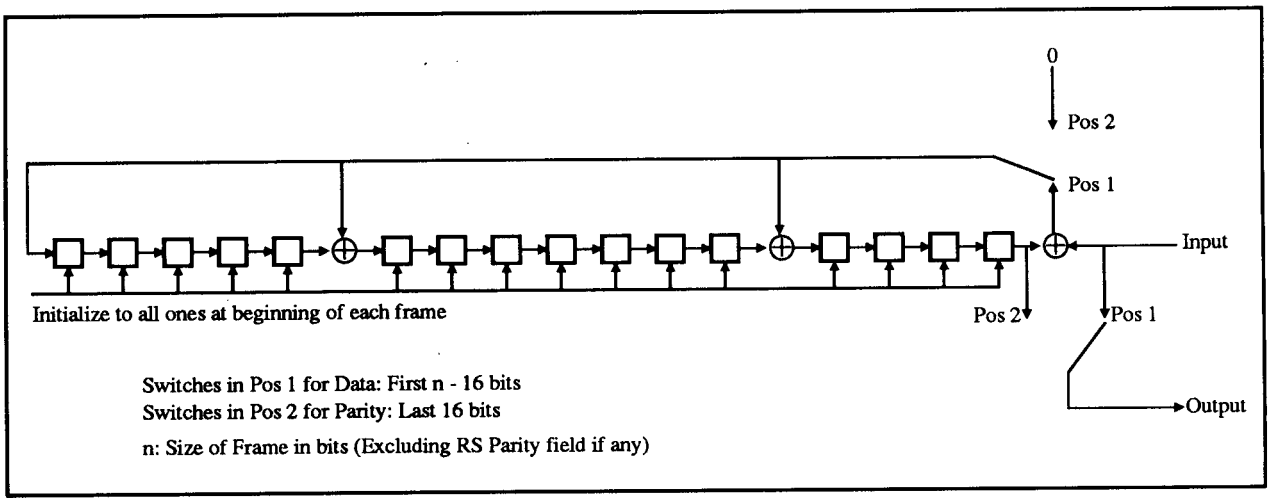


Fig. 16. A Shift-Register Implementation of VCDU Error Control Coding

2.4.3. VCDU Header Error Control

Some of the critical fields in the Primary header of the frame may be protected by a Header Error Control Code, whose check bits are contained within the 16-bit VCDU Header Error Control Field. This error-protection is optional for Grade 1 and Grade 2 Services (CVCDUs) because of the error-protection provided by the Reed-Solomon Code, and is mandatory for Grade 3 Service. The primary aim is to provide error-protection to critical fields of the frame, corruption of which may result in protocol errors like delivery of frames to the wrong destination, etc..

The code used is shortened Reed-Solomon (10,6) code over GF(16), i.e., 4 bits per symbol. This code is capable of correcting up to two symbol errors within the R-S codeword. The field GF(16) is generated using the irreducible polynomial

$$F(x) = x^4 + x^2 + 1$$

over GF(2). The generator polynomial for the code is

$$g(x) = (x + \alpha^6)(x + \alpha^7)(x + \alpha^8)(x + \alpha^9)$$
$$= x^4 + \alpha^3 x^3 + \alpha x^2 + \alpha^3 x + 1$$

over GF(16) where the mapping from group symbols to 4-tuples is:

$$0 = 0000, \alpha^0 = 0001, \alpha^1 = 0010, \alpha^2 = 0100,$$
$$\alpha^3 = 1000, \alpha^4 = 0011, \alpha^5 = 0110, \alpha^6 = 1100,$$
$$\alpha^7 = 1011, \alpha^8 = 0101, \alpha^9 = 1010, \alpha^{10} = 0111,$$
$$\alpha^{11} = 1110, \alpha^{12} = 1111, \alpha^{13} = 1100, \alpha^{14} = 1011.$$

Within a R-S Symbol, the transmission is from left to right. The Primary Header bit-position to R-S symbol mapping is:

$$0,1,2,3 \mapsto 0 \quad 4,5,6,7 \mapsto 1$$
$$8,9,10,11 \mapsto 2 \quad 12,13,14,15 \mapsto 3$$
$$40,41,42,43 \mapsto 4 \quad 44,45,46,47 \mapsto 5$$
$$48,49,50,51 \mapsto 6 \quad 52,53,54,55 \mapsto 7$$
$$56,57,58,59 \mapsto 8 \quad 60,61,62,63 \mapsto 9$$

2.5. Path Layer

The Path Layer provides the interface between the user-services (except VCA Service and VCDU Service) and the Virtual Layer. It accepts user-data in various formats (depending on the kind of service being used) and converts it into CP_PDUs. This conversion may be done by the user himself (as in Path Packet Service) or using the Encapsulation Service on the user's behalf. The CP_PDU is the functional data unit of the Path Layer, and has the same format as the E_PDU shown in Fig. ??. The Multiplexing Service provides an interface between the Path Layer and the Virtual Layer by converting the sequence of variable-length Version-1 CCSDS Packets into a sequence of fixed-length blocks (called M_PDUs), each sized to exactly fill the data zone of a VCDU. The functional over-view of Path Layer is given in Fig. 17.

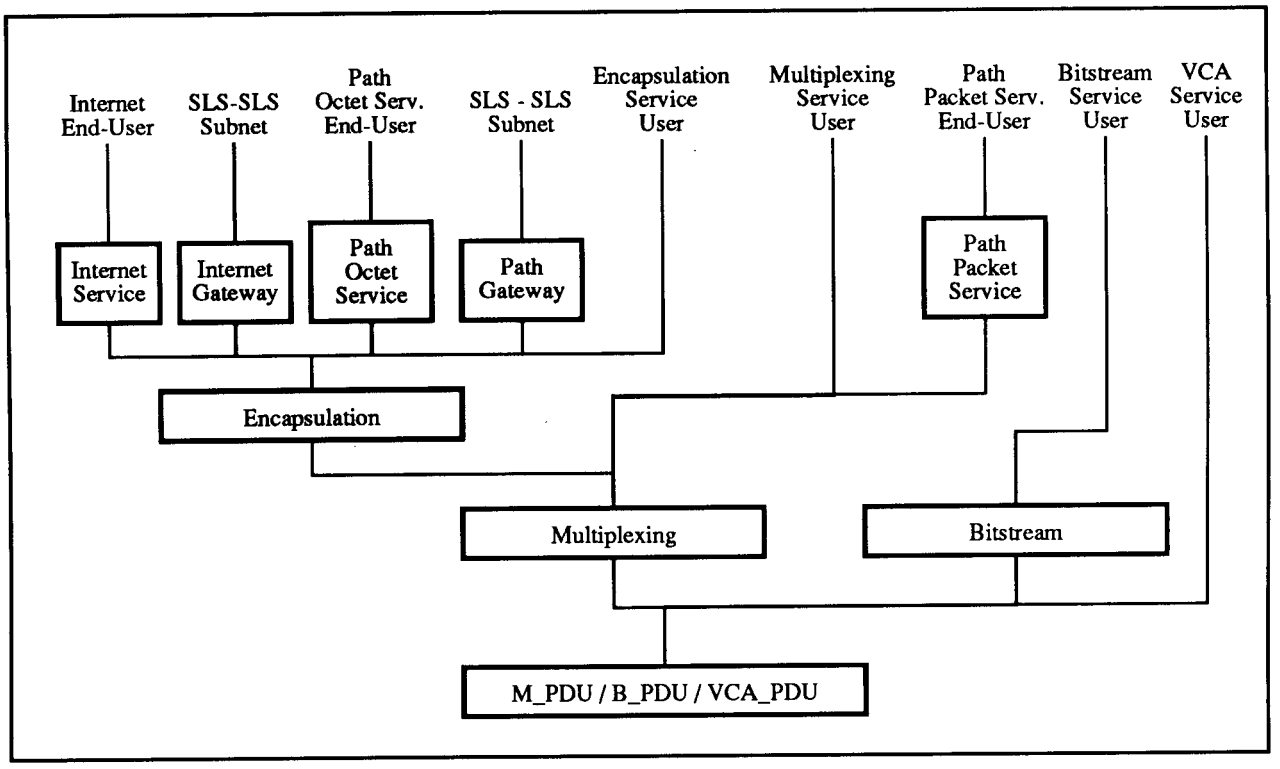


Fig. 17. Functional Overview of the Path Layer

All fuctions related to the path layer have been discussed in previous sections.

This completes our overview of the CCSDS-701.0-B-2 Recommendation for Advanced Orbiting Systems. In the next section, we discuss issues related to the design of a data generator and channel simulator for the above protocol.

3. AOS Data Generator/Simulator Project Design

In this section, we describe a design for the implementation of a simulator for the transmitter and channel sections of a SLS. The aim of this software is to provide a means for the user to generate a data-stream that resembles the data stream received at the receiver section of the SLS after being formatted using the CCSDS AOS recommendation at the transmitter section, and being transmitted over the space channel (possibly with transmission induced artifacts like channel noise). This shall in turn allow a user—who is interested in building a receiver section for an AOS system conforming to the CCSDS 701.0-B-2 recommendation—to test his software or hardware implementations at each stage, and to allow diagnostic testing of the receiver under various protocol errors caused by the occurrence of uncorrected channel errors in critical header sections of the data units. The software provides a means for the user to easily specify the contents of any field or sub-field of any packet or frame in the data stream, and also to specify the error in any location. An error model may also be used to inject errors.

3.1. Object-Oriented Programming

A goal of this project is to provide a structured implementation of the complex CCSDS AOS protocols described in the previous section. Upon careful analysis of the structure of the protocol and the data types, it was agreed that an object-oriented approach is best suited to this task. The protocol itself may be viewed as a collection of data types (e.g., the Version-1 CCSDS Packet, M_PDUs, etc..) and functions (e.g., the Encapsulation Function, the Multiplexing Function, procedures for error-encoding etc..). The data types exhibit a strong similarity between themselves—each packetized data type consists of a storage space for bits, and field information. On the other hand, the manner in which various fields are processed is very specific to each data type. This is an ideal application of object-oriented programming: those aspects of the data types that are common to all, are encapsulated within a common implementation, and then the specific data types are created from this "parent class" through the process of inheritance. Similarly, the various functional blocks of the protocol may be viewed as operations on these abstract data types, and this allows a cleaner implementation.

In addition to the advantages of object-oriented programming as applied to this project, the other advantages of this technique are all still valid: a structured programming style with increased flexibility, readability and reusability of code, all at no cost to efficiency, through the use of a standardized language that supports object-oriented programming: C++.

3.2. Object-Oriented Packetized Data Representation

At the heart of the AOS Data Generator/Simulator is a C++ class, called **`packet`**, which may be used to represent any packetized, formatted binary data unit.

A structured data unit can be described in terms of a sequence of bit-positions—which may be further grouped into higher level structures called fields—and the binary data stored at each of those bit positions. A field may contain within it, a number of smaller fields (called sub-fields). Further, it should be possible to access the data (both read and write) at the bit level, word (or octet) level, and the field level. For the representation of packetized data, we decided to create the **`packet`** class, which fulfills the above requirements, with the emphasis being flexibility, rather than efficiency. (See Fig. 18 below.)

It was decided to represent the field-structure of the packet as an ordered set of triples: the field-name, the starting bit-position and the length of the field (which is itself a class, called **`field`**). The ordering of the fields within the packet is user-specified. (Any gain in efficiency due to a ordering of the fields sorted in some fashion will be expected to be small, since there are only a handful of named fields in any realistic packetized data structure. At the same time, preserving the order in which the user specifies the fields may be of some convenience to the user.)

To store the binary data, and to provide both bit-level and octet-level access to it (field-level access to the data can be defined in terms of these), another class **`bin_array`** is created. This class simply allocates a contiguous storage space (consisting of a sequence of machine words) of adequate size, and provides functions to access this space in a structured manner.

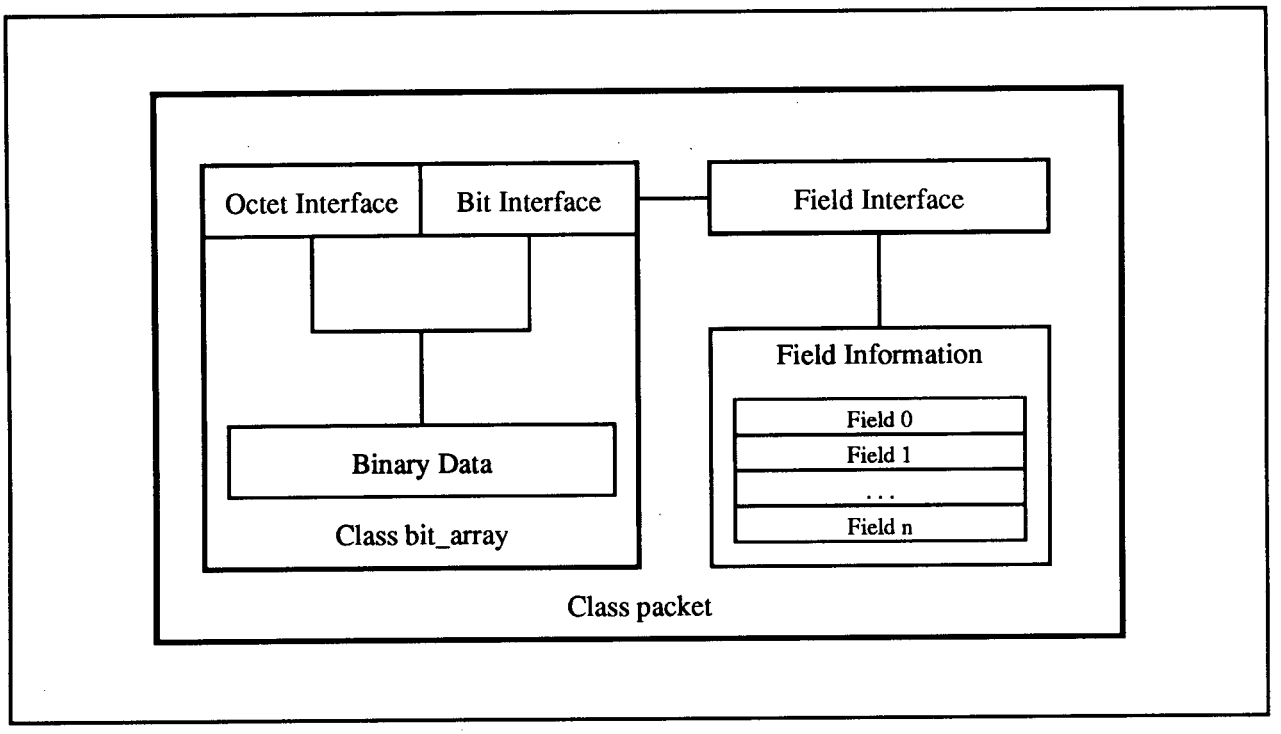


Fig. 18. The Structure of the Packet Class

3.3. Program Structure

The CCSDS 701.0-B-2 recommendation for AOSs is meant to be an efficient protocol for space data systems. To increase the efficiency of the protocol, and at the same time, to allow flexibility in implementation, many optional features of the protocol are designated as "management functions" and "management selected options", which are to be decided by the parties involved in the project. The manner of selection and configuration of these features in not a part of the protocol, but the behaviour of the protocol is dependent on the specific choices made. Once decided, these options are fixed for the duration of the life of the protocol (i.e., if a change is made, the system may need to be reinitialized, and restarted). We have similarly divided the implementaton of the protocol into two sections: the Management Function, and the Data-Generation functions, each with a separate user-interface to the software. These interfaces are separately termed the Management Interface, and the User Interface. The overall structure of the program is as shown below.

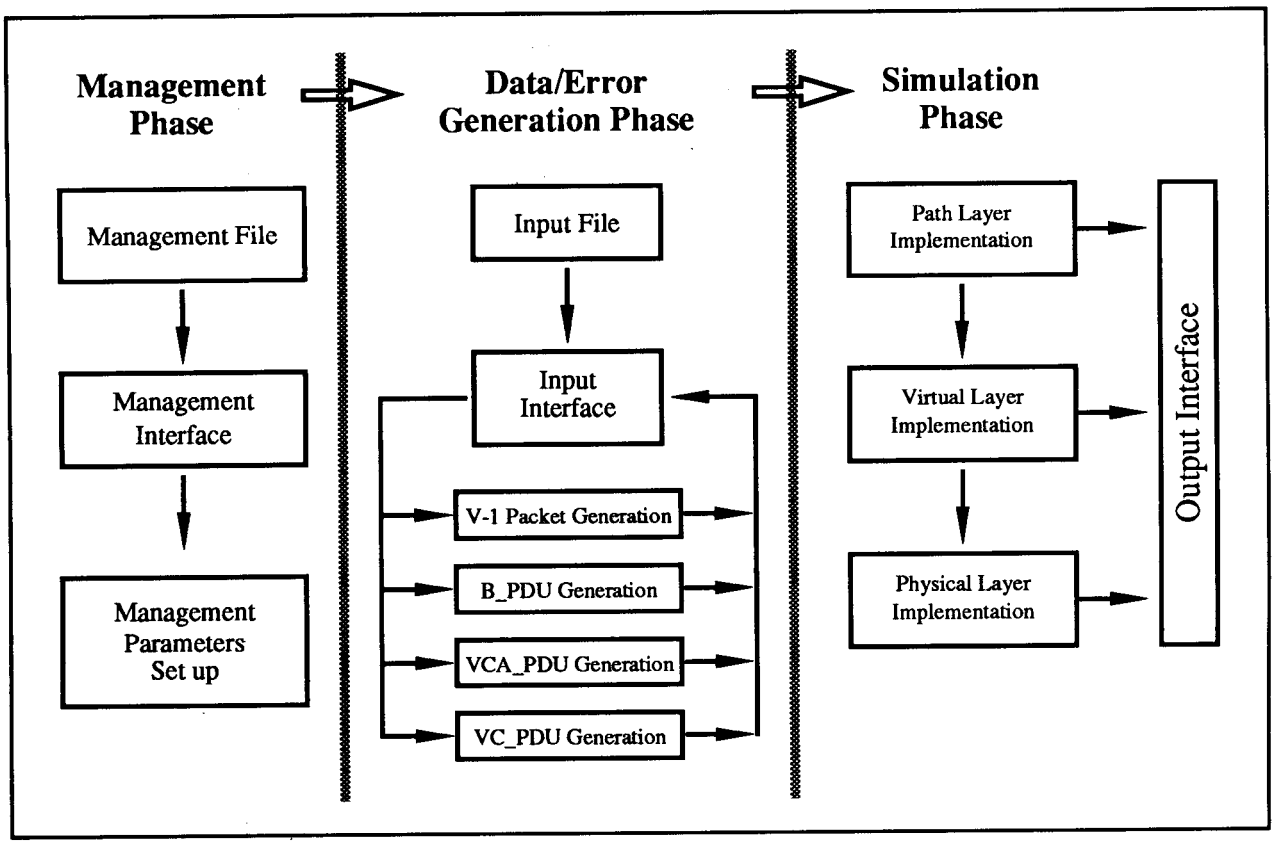


Fig. 18. Logical Structure of the Data Generator/Simulator

3.3.1. Management Functions

The user performs management functions by inserting commands in the management file, which comprises the management interface to the program. The structure of the management file is described below. For a detailed account of the management interface, see [5].

management_file

→ **NUM_CHANNELS** = decimal_number ;
channel_list
end_of_file

The first command in the management file is a declaration of the number of virtual channels that are supported on the SLS. The number of virtual channels must be a positive number *num_vc* (> 0). and the virtual channels are numbered $0, 1, \ldots, num\_vc - 1$. After that must follow a list of exactly *num_vc* channel configuration commands, one for each channel.

channel_list

→ channel channel ... channel (repeated *num_vc* times.)

Each channel configuration command corresponds to the set up of the management parameters related to one virtual channel. There is no restriction on the order in which the channels are configured. Each channel specification is as follows:

channel

→ **CHANNEL** decimal_number
BEGIN
VC_PDU_format
VCDU_ID_specification
END

The first line of a channel specification specifies the number of the virtual channel being specified. We call this number *vc*. The number *vc* must be in the range $0 <= vc <= num\_vc - 1$. The specification of the channel number *vc* is enclosed within the **BEGIN** ... **END** block. First after the **BEGIN** block is the VC_PDU format specification. It corresponds to the management functions that specify the format of the VC_PDU (both the Reed-Solomon encoded form: CVCDU, and the non-Reed-Solomon encoded form: VCDU) that are transported over the virtual channel. Each virtual channel supports only a single format for the VC_PDU. A virtual channel is uniquely identified by the VCDU_ID field of the VC_PDU. (All VC_PDUs transmitted over the same virtual channel must have the same VCDU_ID field.) The VCDU_ID_specification is the assignment of the VCDU_ID field. Both of these are described in more detail below.

VC_PDU Format Specification

The VC_PDU specification is enclosed within a **BEGIN** ... **END** block. Its syntax is as follows:

VC_PDU_format

→ **VC_PDU**
BEGIN
THIRTY_TWO_BIT query ;
SIZE_OPTION decimal_number ;
REED_SOLOMON_ENCODING query ;
HEADER_ERROR_CTRL query ;
CRC_ERROR query ;
OPERATIONAL_CTRL query ;
INSERT_ZONE decimal_number ;
END

where query matches either yes (**YES, Y, yes, y**) or no (**NO, N, no, n**).

THIRTY_TWO_BIT specifies whether 32-bit compatibility is required or not.

SIZE_OPTION specifies the size option chosen. When Reed-Solomon encoding is enabled, the following five size options are available:

Size Option	Size (Octets)	32-bit Compatible Size (Octets)
1	255	252
2	510	508
3	765	756
4	1020	1020
5	1275	1260

When Reed-Solomon encoding is disabled, the size is an integral number of octets, with a minimum value of 124, and a maximum value of 1275. In this case, the size option is equal to the size of the packet in octets.

REED_SOLOMON_ENCODING specifies whether Reed-Solomon encoding is required or not. (If yes, then the virtual channel supports Grade-1 or Grade-2 service, and if no, then the channel supports Grade-3 service.)

HEADER_ERROR_CTRL specifies whether header-error control is implemented. This is mandatory if **REED_SOLOMON_ENCODING** is set to no.

CRC_ERROR specifies whether crc-check bits are to be inserted in the trailer. This is mandatory if **REED_SOLOMON_ENCODING** is set to no.

OPERATIONAL_CTRL specifies whether the operational control field is present.

INSERT_ZONE specifies the size of the insert zone in octets. If this number is zero, there is no Insert zone in the VC_PDU, otherwise an insert zone is created.

VCDU_ID Specification

The VCDU_ID is the identifier for a particular virtual channel, and uniquely identifies the virtual channel on which, the VC_PDU is transmitted. The value of the VCDU_ID field set here shall be the default value of the VCDU_ID field of each data VC_PDU transmitted on this channel. The syntax for specifying the VCDU_ID field (or equivalently, the SCID and VCID fields) is as follows:

VCDU_ID_specification

→ **VCDU_ID** = bit_sequence;

| [**SCID** = bit_sequence ; **VCID** = bit_sequence ;]

The assignment of the VCDU_ID field, or equivalently, the assignment of the SCID and VCID fields obeys the general syntax for field-assignments discussed in detail in the users manual.

3.3.2. Data-Generation Functions

We described in detail, in the previous section, the various aspects of the CCSDS 701.0-B-2 Recommendation for Advanced Orbiting Systems. An overview of the protocol showing the various modules and data-paths involved is shown in Fig. 19.

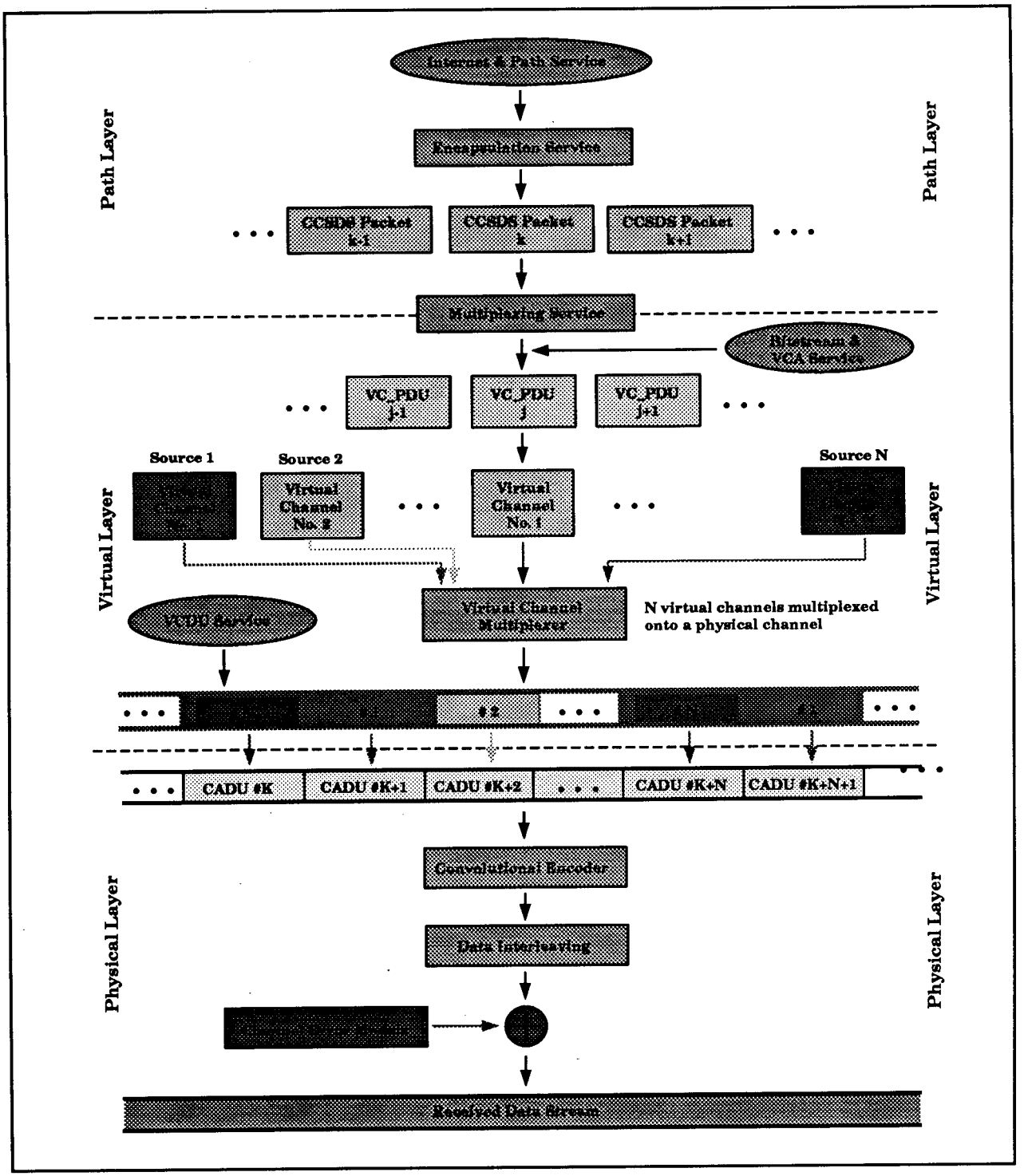


Fig. 19. Overview of the CCSDS 701.0-B-2 recommendation.

The user performs data-generation functions by inserting commands in the data file, which comprises the Data Generation Interface to the program. Because of the complex nature of the protocol, the description of the data-generation and error-injection functions is also quite cumbersome. The structure of the data file is as follows:

data_file

→ packet*
transmission_sequence*
output_format
end_of_file

The data file is divided into three parts. The first part consists of construction of data units through user specifications. The second part is the construction of transmission sequences, based on the the packets constructed in the first part. The third part is the specification of the output format. The packets section of the data file consists of a sequence of packet specifications. Each packet specification is the construction of a single data unit.

The user can declare whether this is to be used as a data packet or an error packet. (Data packets will get some of their fields initialized in accordance with the format for the CCSDS recommendation. Error packets will start with all zero's and the user will need to specify the locations of all the errors by filling in 1s in the appropriate places.)

packet

→ string = [**DATA** | **ERROR**] data_unit

where the syntax for the type, header and field_list is dependent on the kind of data-unit under construction. There are five kinds of data units accepted by the program. They are: CP_PDU (CCSDS Version-1 Packet), M_PDU, B_PDU, VCA_PDU and VC_PDU.

data_unit

→ cp_pdu
| m_pdu
| b_pdu
| vca_pdu
| vc_pdu

The details of the constructions of various data unit are described below. The syntax for the construction of each data unit is enclosed within a **BEGIN** ... **END** block.

Construction of CP_PDUs

The construction of a CP_PDU is as follows:

cp_pdu

→ **CP_PDU**
BEGIN
cp_pdu_construction_header
cp_pdu_field_assignment*
octet_assignment*
bit_assignment*
END

The keyword **CP_PDU** signifies that the following data-unit specification is for the construction of a CP_PDU. The construction is enclosed within a **BEGIN** ... **END** block. The first part of the construction is the header, which is as follows:

cp_pdu_construction_header
    → copy_cp_pdu
    | new_cp_pdu

A CP_PDU can be constructed in two ways: either an already constructed CP_PDU can be modified, or a construction can be done from scratch. The first option is useful if a CP_PDU differs only slightly from another one already constructed. The copy_cp_pdu construction header corresponds to this approach, and has the following syntax:

copy_cp_pdu
    → **COPY_FROM** string ;

where string is the name of the already constructed CP_PDU.

The new_cp_pdu construction header corresponds to the construction of a packet from scratch. When a CP_PDU is to be constructed from scratch, the user needs to specify (i) whether a secondary header is present or not (and if present, then the size of the secondary header), and (ii) the size of the data-field of the CP_PDU. The syntax of the new_cp_pdu construction header option allows this:

new_cp_pdu
    → sec_header
      set_size

sec_header
    → [**SEC_HEADER** decimal_number ; | **nothing**]

set_size
    → **DATA_FIELD_SIZE** = decimal_number ;
    | **PACKET_SIZE** = decimal_number ;

When a secondary header is to be added, the line
SEC_HEADER decimal_number ;

is present, and the decimal_number specifies the size of the secondary header in octets. If not, then the secondary header is not present. If the packet being constructed is a data packet (indicated by the presence of the **DATA** keyword immediately preceding the packet construction), the **PACKET_SEC_HDR_FLAG** field of the CP_PDU is set to 1 (indicating the presence of a user-defined secondary header). Next, the size of the packet is specified by specifying the size of the data-field using either the format:
DATA_FIELD_SIZE = decimal_number;

in which the total size of the packet is calculated automatically, or by directly specifying the size of the packet using the format:
PACKET_SIZE = decimal_number;

in which case, the size of the data field is adjusted accordingly. In this case, the packet size must be large enough to accommodate the header (primary and secondary header if any) and a single octet of data. The decimal_number specifies the size in octets.

If this is a new construction, and the CP_PDU being constructed is a data packet (indicated by the presence of the keyword **DATA** immediately preceding the data-unit construction), the **PACKET_LENGTH** field of the CP_PDU is set to the correct value. (If it is an error CP_PDU, the complete packet is set to all zeros (no errors anywhere).)

The assignment of the fields in the CP_PDU follow the syntax:

cp_pdu_field_assignment

→ **PRIMARY_HEADER** = bit_sequence;
| **PACKET_ID** = bit_sequence;
| **VERSION_NO** = bit_sequence;
| **PACKET_TYPE** = bit_sequence;
| **PACKET_SEC_HDR_FLAG** = bit_sequence;
| **PACKET_APID** = bit_sequence;
| **PACKET_SEQ_CTRL** = bit_sequence;
| **PACKET_SEQ_FLAGS** = bit_sequence;
| **PACKET_SEQ_COUNT** = bit_sequence;
| **PACKET_LENGTH** = bit_sequence;
| **PACKET_SEC_HEADER** = bit_sequence;
| **DATA_FIELD** = bit_sequence;

where each option is of the form:

field_name = bit_sequence;

and where bit_sequence is either a hexadecimal_number or a binary_number (from the lexical analyzer):

bit_sequence

→ [hexadecimal_number | binary_number]

The field **PACKET_SEC_HEADER** is only allowed if there exists a user-defined secondary header.

The bit_sequence is a sequence of zero or more bits. (If it is a binary_number, then the bit_sequence is directly equal to the binary_number sequence, after stripping off the leading #. If on the other hand, this is a hexadecimal_number, then it is converted into the equivalent binary_number by substituting each hexadecimal symbol with its equivalent 4-bit representation, and stripping off the leading $.) Each field specification of the form:

field_name = bit_sequence;

corresponds to the following assignment: The first bit of the bit_sequence is assigned to the first bit of the field, the second bit to the second bit, and so on, until either (i) all the bits in the field have been specified, or (ii) all the bits in the bit_sequence have been consumed, or (iii) both. If the bit_sequence is shorter than the length of the field, the remaining bits of the field remain unchanged; if longer, then the extra bits in the bit_sequence are discarded.

After the field assignments, follow a sequence of octet assignments. This allows the user to directly specify the contents of specific octets in the packet. The assignment of octets follows the syntax:

octet_assignment

→ **OCTET** (decimal_number) = bit_sequence ;

Consider an assignment of the form:

OCTET (n) = bit_sequence;

In this case, the assignment is done as follows: The first bit in the bit_sequence is assigned to bit 0 in the octet n, the second bit to bit 2 ...the 8th bit to bit 7, the 9th bit to bit 0 of octet n+1, ..., until either (i) all the bits in the bit_sequence have been used, or (ii) all the bits in the packet starting from bit 0 of octet n have been assigned, or (iii) both. The octets in the packet are numbered 0,1,.... After the octet assignments, come the bit assignments. This allows the user to directly specify the contents of specific bits. The assignment of bits follows the syntax:

bit_assignment

→ **BIT** (decimal_number) = bit_sequence ;

Consider an assignment of the form:

BIT (n) = bit_sequence;

In this case, the assignment is done as follows: The first bit in the bit_sequence is assigned to bit n, the second bit to bit n+1, ..., until either (i) all the bits in the bit_sequence have been used, or (ii) all the bits in the packet starting from bit n have been assigned, or (iii) both.

The bits in the packet are numbered 0,1,....

Construction of M_PDUs

The construction of a M_PDU is as follows:

m_pdu

→ **M_PDU**
BEGIN
m_pdu_construction_header
m_pdu_field_assignment
octet_assignment
bit_assignment
END

As before, the M_PDU may be constructed either by starting from a previously constructed M_PDU, or from scratch. The syntax for this is as follows:

m_pdu_construction_header

→ **COPY_FROM** string ;
| **CHANNEL** = decimal_number ;

where, for a construction from another M_PDU, string is the name of the already constructed M_PDU being copied from. For a new construction, management needs to set up the exact format of the VC_PDU (which decides the format for the M_PDU), which has already been done during the management interface, for each virtual channel. The line

CHANNEL = decimal_number ;

specifies which channel is to be used to derive the format for the M_PDU.

The assignment of fields in the M_PDU follows the syntax:

m_pdu_field_assignment

→ **HEADER** = bit_sequence ;

| **M_PDU_PACKET_ZONE** = bit_sequence ;

| **SPARE** = bit_sequence ;

| **FIRST_HDR_PTR** = bit_sequence ;

The behavior of field, octet and bit assignment functions is the same as for CP_PDU construction.

Construction of B_PDUs

The construction of a B_PDU is as follows:

b_pdu

→ **B_PDU**
BEGIN
b_pdu_construction_header
b_pdu_field_assignment
octet_assignment
bit_assignment
END

As before,

b_pdu_construction_header

→ **COPY_FROM** string ;

| **CHANNEL** = decimal_number ;

For a new construction, the line

CHANNEL = decimal_number ;

specifies which channel is to be used to derive the format for the B_PDU.

The assignment of fields in the B_PDU follows the syntax:

b_pdu_field_assignment

→ **HEADER** = bit_sequence ;

| **SPARE** = bit_sequence ;

| **BITSTREAM_DATA_PTR** = bit_sequence ;

| **DATA_ZONE** = bit_sequence ;

The behavior of the field, octet and bit assignments is the same as for CP_PDU construction.

Construction of VCA_PDUs

The construction of a VCA_PDU is as follows:

vca_pdu

→ **VCA_PDU**
BEGIN

vca_pdu_construction_header
octet_list
bit_list
END

As before,
vca_pdu_construction_header
→ **COPY_FROM** string ;
| **CHANNEL** = decimal_number ;

For a new construction, the line
CHANNEL = decimal_number ;

specifies which channel is to be used to derive the format for the VCA_PDU. No field_assignment functions are available for the VCA_PDU (because there are no fields in the VCA_PDU). Octet-assignment and bit-assignments follow the same syntax, and have the same behavior as for CP_PDU construction.

Construction of VC_PDUs

The construction of a VC_PDU is as follows:
vc_pdu
→ **VC_PDU**
BEGIN
vc_pdu_construction_header
vc_pdu_field_assignment
octet_assignment
bit_assignment
END

As before,
vc_pdu_construction_header
→ **COPY_FROM** string ;
| **CHANNEL** = decimal_number ;

For a new construction, the line
CHANNEL = decimal_number ;

specifies which channel is to be used to derive the format for the VC_PDU. The assignment of fields in the VC_PDU follows the syntax:
vc_pdu_field_assignment
→ **PRIMARY_HEADER** = bit_sequence ;
| **VERSION_NO** = bit_sequence ;
| **VCDU_ID** = bit_sequence ;
| **SCID** = bit_sequence ;
| **VCID** = bit_sequence ;
| **VCDU_COUNTER** = bit_sequence ;

| **SIGNALLING_FIELD** = bit_sequence ;
| **REPLAY_FLAG** = bit_sequence ;
| **SPARE** = bit_sequence ;
| **HDR_ERR_CTRL** = bit_sequence ;
| **INSERT_ZONE** = bit_sequence ;
| **DATA_ZONE** = bit_sequence ;
| **TRAILER** = bit_sequence ;
| **ERR_CTRL** = bit_sequence ;
| **OPER_CTRL** = bit_sequence ;
| **RS_PARITY** = bit_sequence ;

Depending on the VC_PDU format specified through management functions, some of these field assignments may be invalid. The behavior of field, octet and bit assignment are the same as for CP_PDU construction.

Multiplexing CP_PDUs

After a number of CP_PDUs have been constructed, we can use the multiplexing function to convert them into a sequence of M_PDUs. Each such sequence of CP_PDUs represents data generated by a single path-layer source. The syntax for the construction of a multiplexed sequence allows for the setting of the APID, which will then over-ride the default APID field assignment of all the data packets in the multiplexed sequence. Controls are provided to set this value, and to activate and deactivate this automatic assignment.

It also allows for the automatic sequencing of CP_PDUs. A sequence of CP_PDUs multiplexed into a M_PDU sequence have their PACKET SEQUENCE COUNT field increment by 1 (modulo 16384) for each new CP_PDU. The syntax for the construction of the multiplexed sequence also allows for the setting of the counter field, and its automatic increment with each CP_PDU being multiplexed. This value will then over-ride the PACKET_SEQ_COUNT field of all the data CP_PDUs in the multiplexed sequence. Controls are provided to set this value, to have it increment automatically (or to remain constant) and to deactivate its automatic assignment.

multiplexed_sequence

→ string = **MULTIPLEX**
CHANNEL = decimal_number
BEGIN
multiplex_one*
END

where string is the name of the resulting multiplexed sequence. This will produce two multiplexed sequences, one for the data packets, and one for the error packets. The line

CHANNEL = decimal_number

specifies which virtual channel is used to obtain the format for the M_PDU.

multiplex_one

→ set_mux_controls*

data_and_error_cp_pdu

data_and_error_cp_pdu

→ [**REPEAT** decimal_number | **nothing**]

→ **DATA** = string ;

| **DATA** = string , **ERROR** = string ;

Here,

DATA = string

specifies that the data CP_PDU is the one whose name is given by string. If the line

ERROR = string

is present, then the corresponding CP_PDU is used to inject errors. If absent, then there is assumed to be no error in the corresponding packet. The

REPEAT decimal_number

option may be used to repeat the following data-and-error combination a number of times specified by the decimal_number. **Note that the data and error packet must have the same size.**

set_mux_controls

→ **APID** = bit_sequence ;

| [**ACTIVATE_APID** ; | **DEACTIVATE_APID** ;]

| **COUNT** = bit_sequence ;

| [**ACTIVATE_COUNT** ; | **DEACTIVATE_COUNT** ;]

Each multiplexing sequence starts with the following initial values: the APID and COUNT assignment are deactivated; and the sequence APID and COUNT are set to 0. After the insertion of each CP_PDU, the COUNT is incremented. If the most recent APID control was a ACTIVATE_APID, the current value of the APID is assigned to the APID field of every data packet, otherwise the APID field of the data packet is unchanged. If the most recent COUNT control was a ACTIVATE_COUNT, the current value of COUNT is assigned to the PACKET_SEQ_COUNT field of the data packet, otherwise, the PACKET_SEQ_COUNT field of the data packet remains unchanged. Note that COUNT keeps incrementing, even when deactivated.

Thsi concludes the description of the data-generation and error-injection interfaces to the project.

4. Conclusions and Future Work

In this paper, we gave an overview of the CCSDS Recommendation 701.0-B-2 for Advanced Orbiting Systems. Next, we described the design of an object-oriented data-generation and error-injection program that implements the transmitter sections of this protocol. A discussion of the user interfaces to the software is included. The previous section describes the implementation of only the virtual layer and the path layer. The physical layer and the management functions associated with it have not been implemented yet, and are part of the next phase of this project.

Bibliography

[1] "Advanced Orbiting Systems, Networks and Data Links: Architectural Specification" Recommendation CCSDS 701.0-B-2, Blue Book, Issue 2 (Washington D.C.: CCSDS, November 1992 or later issue).

[2] "Advanced Orbiting Systems, Networks and Data Links: Summary of Concept, Rationale, and Performance" CCSDS 700.0-G-3, Green Book, Issue 3 (Washington D.C.: CCSDS, November 1992 or later issue).

[3] "Telemetry Channel Coding" Recommendation CCSDS 101.0-B-3, Blue Book, Issue 3 (Washington D.C.: CCSDS, May 1992 or later issue).

[4] "Telemetry: Summary of Concept and Rationale" CCSDS 100.0-G-1, Green Book, Issue 1 (Washington D.C.: CCSDS, December 1987 or later issue).

[5] "Advanced Orbiting Systems, Networks and Data Links: AOS Data Generator User's Manual", Version 2, December 1993.