NASA GRC Channel Emulator

User Manual

Compatibility Test Sets — Data Link Team

Tad Kollar — DB Consulting Group

SEPTEMBER 2016

Channel Emulator software release 1.8.6.5

Table of Contents

1 Introduction

1.1 Glossary of Terms and Acronyms

AOS – Advanced Orbiting Systems (AOS); when used in this document refers to the Space Data Link Protocol

Auxiliary input link – A managed connection from a source segment through which a target can receive a secondary data stream, i.e. that does not drive the primary traffic flow in the channel.

Auxiliary output link – A managed connection to target segment that a source segment sends secondary data stream to, i.e. it does not drive the primary traffic flow in the channel.

Buffer – A contiguous area of memory that may either be empty, contain a known number of octets of unknown content or, in the context of the CE, contain a recognized data structure with a network byte-order.

CCSDS - Consultative Committee for Space Data Systems

Channel – A container for modular communications segments.

Channel Emulator (CE) – The entire system that provides devices, channels, and administrative interfaces for performing network testing.

Device – A data source or sink that the CE can manage or access, but is not necessarily in direct control of its behavior.

DLL (Dynamic Link Library) – Also known as a shared library. A file containing symbols to be loaded while a program is starting or while executing instead of being included within the executable itself. All CE segments originate from DLLs.

GVCID - Global Virtual Channel Identifier; combination of MCID and VCID (AOS)

High watermark - The maximum octet count that a segment’s message queue can hold. When this limit is reached, new messages are disallowed and writers block until the queue size falls below the low watermark.

Low watermark – The size that a segment’s message queue must shrink to after hitting the high watermark before it will accept data again.

MCID – Master Channel Identifier; combination of TFVN and SCID

Primary input link – A managed connection from a source segment through which a target segment receives its primary data stream.

Primary output link – A managed connection to a target segment through which a source segment sends its primary data stream.

SCID – Spacecraft Identifier

Segment – An instance of a modular communications entity, managed by a channel in the CE, which handles traffic flow. There can be multiple instances of each type of segment.

TFVN – Transfer Frame Version Number

VCID – Virtual Channel Identifier

Wrapper – An object that manages an associated buffer and provides functionality such as pointers to fields in the buffer, memory access safeguards, checksum computation and validation, constants, and various other tasks.

1.2 References

[1] *AOS Space Data Link Protocol*. Recommendation for Space Data System Standards, CCSDS 732.0-B-2. Blue book. Issue 2. Washington, D.C.: CCSDS, July 2006.

[2] *Encapsulation Service.* Recommendation for Space Data System Standards, CCSDS 133.1-B-2. Blue Book. Issue 2. Washington, D.C.: CCSDS, October 2009.

[3] *Space Link Identifiers*. Recommendation for Space Data System Standards, CCSDS 135.0-B-4. Blue Book. Issue 4. Washington, D.C.: CCSDS, October 2009.

[4] *TC Space Data Link Protocol*. Recommendation for Space Data System Standards, CCSDS 232.0-B-2. Blue Book. Issue 2. Washington, D.C.: CCSDS, September 2010.

[5] *IEEE Standard for Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements, Part 3: CSMA/CD Access Method and Physical Layer Specifications*. IEEE Std 802.3-2008. Second printing. New York, NY: IEEE, 1 February 2010.

[6] *TM Synchronization and Channel Coding*. Recommendation for Space Data System Standards, CCSDS 131.0-B-2. Blue Book. Issue 2. Washington, D.C.: CCSDS, August 2011.

[7] *TM Space Data Link Protocol.* Recommendation for Space Data System Standards, CCSDS 132.0-B-1. Blue Book. Issue 1. Washington, D.C.: CCSDS, September 2003.

1.3 Overview

The Channel Emulator (CE) is a software-based network-testing tool. Its primary functions are providing framing services, acting as a flexible protocol gateway, and providing network emulation capabilities.

There is limited discussion of protocol specifications in this document. However, some knowledge is required of the operator to be able to configure most modular segments correctly. Section 1.2, References, is a list of the documents that contain detailed information on the standard network protocols supported by the Channel Emulator.

1.3.1 Features and Architecture

The software receives input from multiple types of sources and can send output to different ones. These sources and sinks are referred to as *devices* in CE nomenclature. Some or all of the incoming protocol data can be stripped, units can be resized, and new headers or trailers added. Network emulation such as delay, jitter, or errors can be inserted at any point in the stream. Metrics and partial or complete data units can be logged to a database (currently under development).

Pathways between devices are referred to as channels. Channels contain instances of modules called segments, each of which performs a small task on the data stream before sending it on.

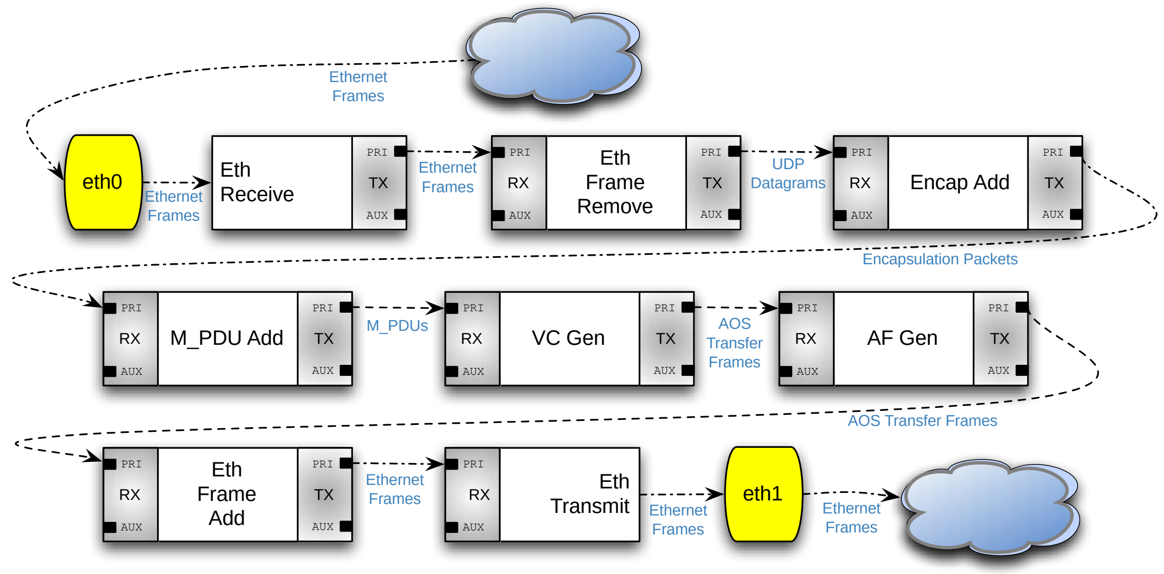

Figure 1.3-1: Channel Emulator Sample Scenario Architecture (AOS Virtual Channel Uplink)

Figure 1.3-1 graphically depicts a sample CE configuration in which UDP datagrams arrive encapsulated in Ethernet frames, are extracted, split, and put into AOS transfer frames, that are re-encapsulated in Ethernet frames and sent out again. The yellow boxes eth0 and eth1 are the input and output devices. The square white-and-gray boxes represent the modular segments, all of which are managed in a single channel.

The core functionality of the CE is managed by a multithreaded daemon. That daemon must run with superuser privileges to be able to modify the configuration of network interfaces and other devices. It can be configured either by a settings file, XML-RPC calls, or a mixture of both; when the daemon shuts down, it can save its complete state so that it can be easily restarted in the same configuration. The useful functionality of the CE is added by loading shared libraries that the modular communications segments are instantiated from. Each module runs in its own thread to reduce latency and maximize the capabilities of the hardware.

The CE has robust support of the AOS Space Data Link Protocol [1]. Features include virtual channel and master channel multiplexing and de-multiplexing; frame error control; multiplexing and bitstream PDU splitting and recombination; and insert zone and operational control field services.

1.3.2 History

The first instance of the Channel Emulator was developed in 2004 for the Protocol Research and Evaluation Environment (PREE). At that time it consisted of a Linux-based router that added delays and errors to packets it forwarded, as well as providing a gateway for the SCPS protocol. The configuration method was invoking custom shell scripts.

During the Space Communications Testbed (SCT) project in 2004-2005, it became known as the Link Emulator. The major changes were frame handling at the data link level, management capabilities for an entire testbed, and remote control via SOAP.

Following the end of SCT, the Link Emulator became known as the Channel Emulator and was used in the Evolutionary Prototyping and Development (EPaD) project in 2006. The testbed management software was discontinued and the CE once again only provided network emulation. However, since the EPaD project was located at MSFC and the CE was being developed at GRC, the system was put onto a Live CD so that the correct Linux kernel and packages would be present and in a known configuration. A web-based GUI was also added at this time.

In 2008, funding for the development of the CE was provided by the Ohio Aerospace Institute and then by the NASA Compatibility Test Sets (CTS) project. This is the current version of the CE under development and represents the largest effort to date. Figure 1.3-2 outlines major changes that have been incorporated.

Previous (PREE, SCT, EPaD)	Current (CTS)
All traffic handled by Linux kernel	Traffic captured and modified in user space
Functionality provided by queuing disciplines, bridges, and other kernel structures	Functionality provided by modular segments loaded from shared libraries
Little to no capability to convert from one protocol to another	Can easily accept one protocol, extract payload, send out as different protocol, or encapsulate one in another
Shell/Perl scripts employed to simplify interaction with kernel interfaces	Server side written in C++; clients can be written in various languages
PHP/JavaScript web page for GUI (EPaD)	HTML/JavaScript/XML-RPC web interface
Distributed on LiveCD (EPaD)	Distributed as installable Ubuntu and Red Hat/CentOS packages
Remotely configurable via SOAP (SCT)	Remotely configurable via XML-RPC
Not portable	Portable (theoretically)

Figure 1.3-2: Comparison of Previous CE Features to Current Version

2 Core Operations

2.1 Server

The core server functionality of the Channel Emulator is responsible for the initial configuration of the system; reacting to operator requests and internal events while the system is running; and smoothly shutting down when the system exits. It manages the list of channels, devices, loaded modules, segments, and settings. It uses a separate configuration file for settings not to be changed during a session or overwritten by remote user actions.

When new objects are created on the server remotely via XML-RPC calls, the safest method is to first create a section in the configuration file for that particular object using one or more setting.* methods.

Entity	Description
Behavioral Parameters	
`debugLevel = d;`	Set the verbosity level of system messages to *d*, where *d* is an integer from 0 to 8: 8 (Debug) Describe every call, event, and data unit that every module handles. Must be enabled at compile time 7 (Informative) Describes significant events in addition to notices, warnings, and errors. Fine for most usage 6 (Notices) Allow only notices (describe that something unusual has happened but wasn't necessarily bad) and higher 5 (Warnings) Allow only warnings (something bad happened but not fatal) and higher 4 (Errors) Display only errors (something bad happened that may have result in a loss of functionality) and higher 0-3 Effectively disables all output because messages of higher level than "error" are unused
`defaultDebug = true | false;`	*Only for builds with debug level 8 enabled.* If `true`, enable debugging by default for each new segment. This is useful when trying to obtain debugging messages from only a small set of segments
`foreground = true | false;`	If `true`, sends output to the terminal and prevents the process from daemonizing; `false` causes it to fork into the background and log to `logFile`
`reactorThreads = r;`	Use integer *r* threads to handle events
`saveCfgAtExit = true | false;`	If `true`, save the configuration file when the CE shuts down. Uses the same filename, appended with the time since the epoch in seconds
HTTP Server Parameters	
`listenAddress = "a.b.c.d";`	Let the web server listen on IPv4 address "*a.b.c.d*". Defaults to "127.0.0.1"
`listenPort = p;`	Let the web server listen on TCP port *p*. Defaults to `8080`
`xmlrpcPath = "p";`	HTTP POST requests to path "*p*" will be handled by the XML-RPC subsystem. Defaults to "/RPC2"
`webDocDir= "r";`	The web server finds files relative to path "*r*". Defaults to "/var/lib/ctsce/www"

Other File/Directory Parameters	
`cfgPath = "c";`	The full path to the CE config files. Defaults to `"/etc/ctsce"`
`cfgFile = "f";`	The filename of the initial device/channel configuration file within cfgPath. Defaults to `"devices-channels.cfg"`
`dllPath = "p";`	Set the location of the CE modules (modXYZ.so files) to path/file "*p*". Default is `"/usr/lib/ctsce-modules"` for Ubuntu and `"/usr/lib64/ctsce-modules"` for Red Hat/CentOS
`fileIODir = "d";`	The full path to the directory where data files are read from/written to. Defaults to `"/var/lib/ctsce/fileIO"`
`logFile = "l";`	Send output to path/file "*l*". Defaults to `"/var/log/ctsced.log"`
`protectedEthInterfaces = [ "ethX", "ethY", … ]`	An array of Ethernet interfaces that the CE will not allow to be modified

Table 2.1–1: Core Server Configuration File Settings

```
# Behavioral parameters
foreground = false;
debugLevel = 7;
defaultDebug = 7;
reactorThreads = 4;
saveCfgAtExit = false;

# HTTP server parameters
listenAddress = "127.0.0.1";
listenPort = 8080;
xmlrpcPath = "/RPC2";
webDocDir = "/var/lib/ctsce/www";

# Other file/directory parameters
cfgPath = "/etc/ctsce";
cfgFile = "devices-channels.cfg";
dllPath = "/usr/lib/ctsce-modules";
fileIODir = "/var/lib/ctsce/fileIO";
logFile = "/var/log/ctsced.log";
protectedEthInterfaces = [ "lo", "eth0" ];
```

Table 2.1–2: Core Server Sample Configuration File

Table 2.1–1 contains configuration file options for the core server. Additional modification of default startup behavior can be performed with command line options, found in Table 2.1–3.

Long Format	Short Format	Effect
`--config f`	`-c f`	Read configuration file `f` to get initial device/channel settings
`--config-path p`	`-C p`	Path to the directory containing device/channel configuration files
`--debug x`	`-d x`	Set the debug level to integer `x` (refer to Table 2.1–1 for complete explanation. Overrides the `debugLevel` configuration file setting
`--default-debug`	`-D x`	*Only for builds with debug level 8 enabled.* Disable/enable debugging messages by specifying `x` as 0 or 1. This is useful when trying to obtain debugging messages from only a small set of segments
`--fileio-path p`	`-F p`	The path to the directory containing data files
`--foreground`	`-f`	Send output to the terminal and prevent the process from daemonizing. Overrides the `foreground` configuration file setting
`--log p`	`-l p`	Full path to the log file
`--no-save`	`-n`	Do not write out the final configuration file when the system exits. This file contains all of the settings in the initial configuration file, plus all default settings, plus counters for several modules
`--protect ethX,ethY,…`	`-p ethX,…`	Mark the specified Ethernet interfaces as protected. The Channel Emulator will not modify them
`--srv-config p`	`-s p`	Full path to the core server config file
`--web-doc-path p`	`-w p`	Full path to the directory containing web documents

Table 2.1–3: Channel Emulator Server Command Line Options

2.2 Devices

A device is a data source, sink, or both. Examples include Ethernet interfaces, serial ports, and files. Although devices are usually connected to at least one channel, they are not part of the channel – they have a separate management interface and namespace.

Device names must be unique across the entire system. The names are not required to match those of the actual resources they govern, though it should be done when possible to avoid confusion.

Entity	**Container**	**Description**
`Devices: { … };`	Top level	Holds all device definitions
`device_name``: { … };`	`Devices`	A uniquely-named section containing settings for a single device
`devType = "Ethernet" | "File" | "Tcp4Server" | "Tcp4Client" | "Sink" | "Source";`	*`device_name`* subsection	Determines whether to configure this device as an Ethernet interface or file, the device types currently supported by the CE. Setting this correctly is critical
`autoLoad = true | false;`	*`device_name`* subsection	If `false`, the device is not loaded automatically by the server, although its settings are still defined. Defaults to `true`

Table 2.2–1: Device Configuration File Settings

```
Devices :
{
  dummy0 :
  {
    devType = "Ethernet";
    …
  };

  dummy1 :
  {
    devType = "Ethernet";
    …
  };
};
```

Table 2.2–2: Devices Configuration File Excerpt

Operation	**Effect**
`device.getList()`	Return an array of all devices currently loaded in the system
`device.getType(d)`	Return the type (e.g. `"File"` or `"Ethernet"`) of specified device `d`. **Note**: the string `"dev"` must be pre-pended to this value to make correct XML-RPC calls, e.g. **`dev`**`File.add("example")`. This is a change from CE version 1.0 where `"File.add"` was used
`device.exists(d)`	Returns whether a device named `d` exists in the system
`dev`*`Type`*`.add(d)`	Add a new instance of the `dev`*`Type`* module to manage a resource, and call it `d`
`dev`*`Type`*`.getReaderInfo(d)`	Return the channel and name (in an array) of device `d`'s reader segment if it exists
`dev`*`Type`*`.getWriterInfo(d)`	Return the channel and name (in an array) of device `d`'s writer segment if it exists
`dev`*`Type`*`.isInitialized(d)`	Return whether device `d` has been initialized – generally meaning that it has all the necessary configuration information and is managing a resource
`dev`*`Type`*`.setDebugging(d,b)`	If the CE has been compiled with debugging enabled, set debugging on device `d` to Boolean value `b`
`dev`*`Type`*`.getDebugging(d)`	If the CE has been compiled with debugging enabled, get the Boolean debugging setting for device `d`

Table 2.2–3: XML-RPC Directives for Common Device Operations

2.2.1 File Device (devFile)

A single file device can be either read or written to, but not both. A second file device can be configured to access the same file, but the results of doing so are untested.

Entity	Container	Description
`fileName = "f";`	*`device_name`* subsection	The path and filename, relative to the `IOpath`, of the file to be operated on
`isInput = true | false;`	*`device_name`* subsection	If `true`, the file is used for reading; if `false`, use it for writing to (the file will be truncated if it already exists)

Table 2.2–4: File Device Configuration File Options

Operation	Effect
`devFile.manage(d,f)`	Create a new FileDevice named `d` to manage file `f`
`devFile.unmanage(d)`	Remove the FileDevice named `d`
`devFile.openForReading(d)`	Open the file managed by device `d` in read-only mode
`devFile.openForWriting(d)`	Open the file managed by device `d` in write-only mode
`devFile.close(d)`	Close the file under management by device `d`
`devFile.isOpen(d)`	Return whether the file managed by device `d` is open
`devFile.setFilename(d,f)`	Reset the filename for device `d` to `f`
`devFile.getFilename(d)`	Return the filename for device `d`

Table 2.2–5: XML-RPC Directives for File Device Operations

2.2.2 Ethernet Device (devEthernet)

This Channel Emulator's Ethernet device management handles activating and deactivating the device, setting flags, and getting counters. When management of an interface is finished, the CE restores its flags to the state it found them in, but counters are not reset.

The name of the devEthernet instance in the configuration file will automatically be taken as the name of the Ethernet interface to manage, unless the `"ifaceName"` setting is used.

If the HTTP/XML-RPC interface is in use, the Ethernet interface that it is accessed through should **not** be managed by devEthernet; the same applies to any Ethernet interface in use by other operating system services. Doing so will very likely result in loss of ability to communicate with the host.

Entity	**Container**	**Description**
`ifaceName = "`*`i`*`";`	*`device_name`* subsection	If set, the string `"`*`i`*`"` will be used to refer to the interface at the OS level instead of the name of the CE device. This allows, for example, for the CE device to be named "eth_0_vlan_5" even though the Ethernet interface it manages is actually called "eth0.5" by the OS
`flagPromisc = true | false;`	*`device_name`* subsection	If `true`, put the interface in promiscuous mode. If `false`, don't change the promiscuous mode setting
`flagNoARP = true | false;`	*`device_name`* subsection	If `true`, disable ARP request processing on the interface. If `false`, don't change the ARP setting
`snapLen = `*`s`*`;`	*`device_name`* subsection	Read `s` octets from the interface (performed by a "receiving" segment)
`activateOnLoad = `*`a`*`;`	*`device_name`* subsection	If `a` is `true`, activate (bring up) the Ethernet interface as soon as management begins. If `false`, wait until instructed to do so

Table 2.2–6: Ethernet Device Configuration File Options

Operation	Effect
`devEthernet. setIfaceName(d,i)`	Set the OS-level name of the managed Ethernet interface to `i` for the devEthernet segment named `d`
`devEthernet. getIfaceName(d)`	Get the OS-level name of the managed Ethernet interface for the devEthernet segment named `d`
`devEthernet. activate(d)`	Direct the devEthernet segment named `d` to begin management of its associated Ethernet interface. Automatically puts the interface into the "UP" state
`devEthernet. deactivate(d)`	Direct the devEthernet segment named `d` to stop managing its associated Ethernet interface.
`devEthernet. isUp(d)`	Return whether the interface managed by devEthernet segment `d` is up
`devEthernet. setPromiscuous(d, p)`	Set promiscuous mode for the interface managed by devEthernet segment `d` to Boolean `p`
`devEthernet. getPromiscuous(d)`	Return whether the interface managed by devEthernet segment `d` is promiscuous
`devEthernet. setNoARP(d, a)`	Set of the ARP protocol on the interface managed by devEthernet segment `d` to Boolean value `a`
`devEthernet. getNoARP(d)`	Retrieve the state of ARP protocol processing on the interface managed by devEthernet segment `d`
`devEthernet. getCounters(d)`	Return an array of counters for the interface managed by devEthernet segment `d`
`devEthernet. setMTU(d,t)`	Set the MTU (Maximum Transfer Unit) for the interface managed by devEthernet segment `d` to `t` octets
`devEthernet. getMTU(d)`	Return for MTU for the interface managed by devEthernet segment `d`
`devEthernet. setSnapLength(d,s)`	Set the snap length (amount of data to collect with every read) for the interface managed by devEthernet segment `d` to `s` octets
`devEthernet. getSnapLength(d)`	Return for snap length for the interface managed by devEthernet segment `d`
`devEthernet. isProtected(e)`	Return whether Ethernet device `e` is protected (i.e. allowed to be altered) or not
`devEthernet. listAllInterfaces()`	Return an array of all unprotected Ethernet interfaces

	on the host

Table 2.2–7: XML-RPC Directives for Ethernet Device Operations

2.2.3 IPv4 TCP Client Device (devTcp4Client)

This device creates and maintains an IPv4 TCP connection with a remote server. If the remote IP address and port are known on startup, the device attempts to connect immediately. If the attempt is unsuccessful, it will continue trying every five seconds until it succeeds. Similarly, if the connection is closed, the device will try to reconnect every five seconds.

Entity	Container	Description
`address = "a.b.c.d";`	*device_name* subsection	Set the numerical IPv4 address of the remote server to string `"a.b.c.d"`, where each of `a`-`d` is a single octet
`port = p;`	*device_name* subsection	Set the TCP port of the remote server to integer `p` between 0 and 65535.

Table 2.2–8: IPv4 TCP Client Device Configuration File Options

Operation	Effect
`devTcp4Client.create(c,a,p)`	Create an IPv4 TCP client device named `c`, using optional remote IPv4 address `a` and optional TCP port `p`. If both a and p are specified, a connection will be attempted immediately
`devTcp4Client.remove(c)`	Remove the existing IPv4 TCP client device named `c`
`devTcp4Client.isConnected(c)`	Return true if the IPv4 TCP client device named `c` has an active connection, false otherwise
`devTcp4Client.setAddress(c,a)`	Set the remote IPv4 server address to string `a` for device `c`
`devTcp4Client.getAddress(c)`	Retrieve the remote IPv4 server address from device `c`
`devTcp4Client.setPort(c,p)`	Set the TCP port to integer `p` for device `c`
`devTcp4Client.getPort(c)`	Retrieve the TCP port that device `c` is using
`devTcp4Client.openConnection(c)`	Direct device c to open an IPv4 TCP connection to a remote server if it hasn't already

Table 2.2–9: XML-RPC Directives for IPv4 TCP Client Device Operations

2.2.4 IPv4 TCP Server Device (devTcp4Server)

The IPv4 TCP Server Device listens on a port for incoming connections. A module (i.e. modTcp4Receiver) must manage its listening and connected sockets, because a connection closed by a client can only be detected while attempting to receive input. Only one connection is accepted at a time although more may be queued.

Entity	Container	Description
`address = "a.b.c.d";`	*device_name* subsection	Set the numerical IPv4 address to listen on to string `"a.b.c.d"`, where each of `a-d` is a single octet
`port = p;`	*device_name* subsection	Set the TCP port to listen on to integer `p` between 0 and 65535.

Table 2.2–10: IPv4 TCP Server Device Configuration File Options

Operation	Effect
`devTcp4Server.create(c,a,p)`	Create an IPv4 TCP server device named `c`, using optional IPv4 address `a` and optional TCP port `p` to listen on
`devTcp4Server.remove(c)`	Remove the existing IPv4 TCP server device named `c`
`devTcp4Server.isServing(c)`	Return true if the device is listening for new connections, false otherwise
`devTcp4Server.isConnected(c)`	Return true if the IPv4 TCP server device named `c` has an active client connection, false otherwise
`devTcp4Server.setAddress(c,a)`	Set the IPv4 server address to listen on to string `a` for device `c`
`devTcp4Server.getAddress(c)`	Retrieve the IPv4 server address from device `c`
`devTcp4Server.setPort(c,p)`	Set the TCP port to integer `p` for device `c`
`devTcp4Server.getPort(c)`	Retrieve the TCP port that device `c` is using
`devTcp4Server.startListening(c)`	Direct device c to begin listening for new connections

Table 2.2–11: XML-RPC Directives for IPv4 TCP Server Device Operations

2.2.5 IPv4 UDP Device (devUdp4)

The IPv4 UDP Server Device simply opens and binds to an IPv4 UDP socket. One module, usually modUdp4Receiver, can receive from the socket and another, usually modUdp4Transmitter, can send through it.

Entity	Container	Description
`address = "a.b.c.d";`	*device_name* subsection	Set the numerical IPv4 address to listen on to string `"a.b.c.d"`, where each of `a-d` is a single octet
`port = p;`	*device_name* subsection	Set the UDP port to listen on to integer *p* between 0 and 65535.

Table 2.2–12: IPv4 UDP Device Configuration File Options

Operation	Effect
`devUdp4.create(c,a,p)`	Create an IPv4 UDP device named `c`, using optional IPv4 address `a` and optional UDP port `p` to listen on
`devUdp4.remove(c)`	Remove the existing IPv4 UDP device named `c`
`devUdp4.setAddress(c,a)`	Set the IPv4 UDP address to listen on to string `a` for device `c`
`devUdp4.getAddress(c)`	Retrieve the IPv4 UDP address from device `c`
`devUdp4.setPort(c,p)`	Set the UDP port to integer `p` for device `c`
`devUdp4.getPort(c)`	Retrieve the UDP port that device `c` is using
`devUdp4.bind(c)`	Direct UDP device `c` to (re)bind to the address and port that it's presently configured with. Useful when there has been a change to the address or port or if the initial bind failed during construction

Table 2.2–13: XML-RPC Directives for IPv4 UDP Device Operations

2.2.6 Data Source Device (devSource)

This device generates a continuous, repeating pattern and writes it to a pipe, which in Linux has a buffer of 64k by default. Another module, typically modFdReceiver, will read blocks from the buffer and encapsulate each block in a network data wrapper of any type.

Entity	Container	Description
`pattern = [ a, b, …, z];`	*device_name* subsection	Set the repeating pattern to the array specified by [a, b, …, z], where each specified element of the array is an integer value in the range 0 to 255 (or 0x00 to 0xff). If unspecified, the default pattern is: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
`patternBufferSize = p;`	*device_name* subsection	A buffer of length `p` octets will be created and filled with the current pattern. The default is 10,240

Table 2.2–14: Data Source Device Configuration File Options

Operation	Effect
`devSource.create(c)`	Create an Data Source device named `c`
`devSource.destroy(c)`	Remove the existing Data Source named `c`
`devSource.setPattern(c,a)`	Set the repeating pattern for Data Source `c` to the array specified by `a`
`devSource.getPattern(c)`	Retrieve an array containing the current from Data Source device `c`
`devSource.setPatternBufferSize(c,s)`	Set the pattern buffer to integer `s` for device `c`
`devSource.getPatternBufferSize(c)`	Retrieve the current pattern buffer size for device `c`

Table 2.2–15: XML-RPC Directives for Data Source Device Operations

2.2.7 Data Sink Device (devSink)

This device receives any type of data on a pipe descriptor and discards it. It doesn't have any unique device settings.

Operation	Effect
`devSink.create(c)`	Create an Data Sink device named `c`
`devSink.destroy(c)`	Remove the existing Data Sink named `c`

Table 2.2–16: XML-RPC Directives for Data Source Device Operations

2.3 Channels

A channel is a container for modular communications segments. All segments added to a channel must have unique identifiers within that channel. Two segments with the same name may technically be added to two different channels, but to avoid confusion it's recommended to maintain unique names throughout the system.

Entity	Container	Description
`Channels: { … };`	Top level	Holds all channel definitions
`*channel_name*: { … };`	`Channels`	A uniquely-named section containing segment subsections and related settings
`autoLoad = true | false;`	*`channel_name`* subsection	If `false`, the channel will not be automatically created when the server is started. Defaults to `true`
`*segment_name*: { … };`	*`channel_name`* subsection	A section, uniquely named within the channel, containing information about one segment.

Table 2.3–1: Channel Configuration File Settings

```
Channels: {
  Channel1: {
    Segment1: {
    …
    };
    Segment2: {
    …
    };
    …
  };
  Channel2: {
    …
  };
  Channel3: {
    …
  };
};
```

Figure 2.3-1: Channels Configuration File Excerpt

Operation	Effect
`channel.addChannel(c)`	Add a new channel named `c`
`channel.removeChannel(c)`	Remove the channel named `c`
`channel.activateChannel(c)`	Start traffic in all segments contained in channel `c`
`channel.pauseChannel(c)`	Stop traffic in all segments contained in channel `c`
`channel.listChannels()`	Return the names of all channels in an array
`channel.addSegment(c,d,s)`	Add a modular segment named `s` from DLL `d` (loading if necessary) to channel `c`
`channel.removeSegment(c,s)`	Remove a modular segment named `s` from channel `c`
`channel.listSegments(c)`	Return the names of all segments in channel `c` as an array
`channel.getSegmentType(c,s)`	Return the type of module from which segment `s` in channel `c` was instantiated
`channel.segmentExists(c,s)`	Return whether a segment with the specified name `s` exists in channel `c`

Table 2.3–2: XML-RPC Directives for Channel Operations

2.4 Modular Segments

Segments are the building blocks of a communications channel in the CE. Segments generally take some form of input, perform a series of related operations on it, and send output onward to the next segment or device.

None of the options in the configuration file are strictly required, depending on how a segment will be initiated. That is, a segment can be loaded and configured with XML-RPC calls without any configuration file options existing at all. However, for a segment to be loaded and activated on startup, several options are necessary. Table 2.4–1 contains settings common to all segments. Some of these may be overridden by specific implementations.

Entity	Container	Description
`deviceName = "d1";`	*segment_name* subsection	The segment will read or write directly from/to device `d1`, which must be defined in the `Devices` section. Only certain segments do this and are

		typically named “mod*Receiver” or “mod*Transmitter”
`dllName = “modXYZ”;`	*segment_name* subsection	Load shared library (DLL) `modXYZ`.so before performing other operations on this segment
`highWaterMark = h;`	*segment_name* subsection	Integer *h* is the maximum octet count that the segment’s message queue can hold. When this limit is reached, new messages are disallowed and writers block until the queue size falls below the `lowWaterMark`.
`lowWaterMark = l;`	*segment_name* subsection	If the message queue has hit the `highWaterMark`, wait until the size drops below *l* octets before accepting data again
`primaryOutput = [ “s”, “PrimaryInput” | “AuxInput” ];`	*segment_name* subsection	Primary output is sent to either the “PrimaryInput” or “AuxInput” of segment `s`. Segment `s` must already exist!
`auxOutput = [ “s”, “PrimaryInput” | “AuxInput” ];`	*segment_name* subsection	Auxiliary output is sent to either the “PrimaryInput” or “AuxInput” of segment `s`. Segment `s` must already exist! *Note: This setting is uncommon*
`autoLoad = true | false;`	*segment_name* subsection	If `false`, the segment will not automatically be loaded when the server is initiated. Defaults to `true`
`immediateStart = true | false;`	*segment_name* subsection	If `true`, the segment will begin processing as soon as it is configured. Otherwise, it will need to receive an XML-RPC directive to start it
`MTU = t;`	*segment_name* subsection	Transmit a maximum of *t* octets per data unit. Some segments will override with a derived value
`MRU = r;`	*segment_name* subsection	Receive a maximum of *r* octets per data unit. Some segments will override with a derived value
`sendIntervalUsec = i;`	*segment_name* subsection (periodic services only)	Wait integer *i* microseconds between data transmissions, whether incoming traffic has been received or not. The segment will need to generate idle units

		when there is no new data to send.
`debug = true | false;`	*segment_name* subsection (debugging builds only)	If `true`, and the CE was built with debugging enabled, and the global debug level is set to 8, debugging messages will be printed to the log. Defaults to the global defaultDebug setting

Table 2.4–1: Segment Configuration File Settings

```
Channels: {
  …
  Channel1: {

    Segment1: {
      dllName = “modEthTransmitter”;
      deviceName = “eth1”;
      highWaterMark = 16777216;
      lowWaterMark = 12582912;
      immediateStart = true;
      # Segment-specific settings follow
      …
    };

    Segment2 {
      dllName = “modEthReceiver”;
      deviceName = “eth2”;
      # Watermarks ignored because modEthReceiver has no queue;
      # reads from the device itself.
      primaryOutput = [ “Segment1”, “PrimaryInput” ];
      immediateStart = true;
      # Segment-specific settings follow
      …
    };
    …
  };
};
```

Figure 2.4-1: Segment Configuration File Excerpt

Table 2.4–2 contains XML-RPC calls common to all segments. Calls may be redefined in modules (such as `connectOutput` in de-multiplexing modules) and others may be ignored (e.g. `connectDevice` in modules that don't communicate directly with a device). The *`modName`* part of the call refers to the type of the module (e.g. modUDPAdd or modEthReceiver) and not the name of the specific segment.

Operation	**Effect**
`modName``.startup(c,s)`	Direct segment `s` in channel `c` to begin receiving, processing, and sending data
`modName``.pause(c,s)`	Direct segment `s` in channel `c` to stop receiving, processing, and sending data as soon as possible

modName.connectOutput(c,s1,s2)	In channel `c`, connect the primary output of segment `s1` to the primary input of segment `s2`
modName.connectDevice(c,s,d)	Direct segment `s` in channel `c` to read from or write to device `d`, which must already be defined.
modName.getMRU(c,s)	Get the maximum receive unit (MRU) for segment `s` in channel `c`
modName.setMRU(c,s,r)	Set the maximum receive unit (MRU) to `r` for segment `s` in channel `c`
modName.getMTU(c,s)	Get the maximum transmit unit (MTU) for segment `s` in channel `c`
modName.setMTU(c,s,t)	Set the maximum transmit unit (MTU) to `t` for segment `s` in channel `c`
modName.getLowWaterMark(c,s)	Query segment `s` in channel `c` for its low watermark
modName.setLowWaterMark(c,s,l)	Change the low watermark setting to `l` for segment `s` in channel `c`
modName.getHighWaterMark(c,s)	Query segment `s` in channel `c` for its high watermark
modName.setHighWaterMark(c,s,h)	Change the high watermark setting to `l` for segment `s` in channel `c`
modName.getReceivedUnitCount(c,s,i)	Query segment `s` in channel `c` for the number of units it has received on input rank `i` (optional parameter). Values allowed for `i` are either “Primary” or “Auxiliary”
modName.setReceivedUnitCount(c,s,l,i)	Change the number of received units on input rank `i` (optional parameter) to `l` for segment `s` in channel `c`. Values allowed for `i` are either “Primary” or “Auxiliary”
modName.getReceivedOctetCount(c,s,i)	Query segment `s` in channel `c` for the number of octets it has received on input rank `i` (optional parameter). Values allowed for `i` are either “Primary” or “Auxiliary”
modName.setReceivedOctetCount(c,s,h,i)	Change the number of received octets on input rank `i` (optional parameter) to `l` for segment `s` in channel `c`. Values allowed for `i` are either “Primary” or “Auxiliary”

`modName.getQueuedUnitCount(c,s,i)`	Query segment `s` in channel `c` for the number of units held in its message queue of optional type `i`. Values allowed for `i` are "Primary" or "Auxiliary" (defaults to "Primary")
`modName.getQueuedOctetCount(c,s,i)`	Query segment `s` in channel `c` for the number of octets waiting in its message queue of optional type `i`. Values allowed for `i` are "Primary" or "Auxiliary" (defaults to "Primary")
`modName.getInputLinks(c,s)`	Get the list of all segments producing input to segment `s` in channel `c`. Returns a structure with the name of each input segment paired with the input type (either `"Primary"` or `"Aux"`)
`modName.getOutputLinks(c,s)`	Get the list of all segments for which segment `s` in channel `c` is producing output. Returns a structure with the name of each output segment paired with the output type (either `"Primary"` or `"Aux"`)
`modName.getInfo(c,s)`	Get *all* information for segment `s` in channel `c`. Returned is a structure of structures called "names", "links", "counters", "settings", and "timestamps". Every module returns the same basic information (such as described in this section). Individual modules may add keys to each structure or entirely new structures if necessary to describe themselves
`modName.getIntervalUsec(c,s)`	*Periodic services only.* Retrieve the integral number of microseconds that the segment waits between sends.
`modName.setIntervalUsec(c,s,i)`	*Periodic services only.* Set the wait between data transmissions to integer *`i`* microseconds. The segment will generate idle units when there is no new data to send
`modName.getDebugging(c,s)`	*Debugging builds only.* Return whether segment `s` in channel `c` is enabled for printing debugging messages to the log
`modName.setDebugging(c,s,d)`	*Debugging builds only.* Enable/disable debugging messages for segment `s` in channel `c` by setting Boolean `d`

Table 2.4–2: Common XML-RPC Directives for Segment Operations

2.5 Configuration File / Settings Interface

Although most settings are automatically managed by modules or have specific XML-RPC calls to access them, it is occasionally necessary to read or write directly from or to the configuration file. Table 2.5–1 lists XML-RPC methods for performing these tasks.

Operation	**Effect**
`server.getConfigFile()`	Return the contents of the entire current configuration file as one string
`server.listConfigFiles()`	Return the list of existing configuration file names as an array of strings
`server.listDataFiles()`	Return the list of existing data file names as an array of strings
`server.loadConfigFile(f)`	Load the specified configuration file name `f`, destroying the current configuration
`server.saveConfigFile(f)`	Save the current configuration to the specified file name `f` in the configuration file folder
`setting.remove(p)`	Delete the setting at path `p` from the configuration file
`setting.exists(p)`	Return `true` if setting path `p` exists in the configuration file, `false` otherwise
`setting.getType(p)`	Return a string describing the type of setting at path `p`. Possible values are "`Int`", "`Int64`", "`Float`", "`String`", "`Boolean`", "`Array`", "`List`", or "`Group`"
`setting.setBool(p,b)`	Change the Boolean value at configuration file path `p` to the provided value `b`
`setting.getBool(p)`	Retrieve the Boolean value at configuration file path `p`
`setting.setString(p,s)`	Change the string value at configuration file path `p` to the provided value `s`
`setting.getString(p)`	Retrieve the string value at configuration file path `p`
`setting.setInt(p,i)`	Change the 32-bit integer value at

	configuration file path `p` to the provided value `i`
`setting.getInt(p)`	Retrieve the 32-bit integer value at configuration file path `p`
`setting.setInt64(p,i)`	Change the 64-bit integer value at configuration file path `p` to the provided value `i`
`setting.getInt64(p)`	Retrieve the 64-bit integer value at configuration file path `p`
`setting.setDouble(p,d)`	Change the floating-point value at configuration file path `p` to the provided value `d`
`setting.getDouble(p)`	Retrieve the floating-point value at configuration file path `p`
`setting.setArray(p,a)`	Change the array at configuration file path `p` to the provided value `a`
`setting.getArray(p)`	Retrieve the array at configuration file path `p`
`setting.setList(p,l)`	Change the list at configuration file path `p` to the provided value `l`
`setting.getList(p)`	Retrieve the list at configuration file path `p`
`setting.createGroup(p)`	Create a group setting in the configuration file at path `p`
`setting.getGroup(p)`	Retrieve a group setting as an XML-RPC struct from configuration file path `p`
`setting.setAutoType(p,s)`	Change the setting value at configuration file path `p` to value `s`, whose type is auto-selected based on its XML-RPC paramater type
`setting.getAutoType(p)`	Retrieve the value of the setting at configuration file path `p`, whose XML-RPC parameter type is determined by the type in the configuration file

Table 2.5–1: Configuration File Settings XML-RPC Directives

2.6 DLL Managment Interface

The vast majority of Channel Emulator functions are provided by selectively loaded modules, or dynamically linked libraries (DLLs). To manage the CE remotely from a cold start (i.e. no channels, segments, or devices pre-configured), most clients will need to query the CE for the list of available modules, their functions, and interfaces. Table 2.6–1 describes the XML-RPC methods that provide that functionality.

Operation	**Effect**
`dll.isLoaded(d)`	Returns `true` if module identified by string `d` is loaded, otherwise `false`
`dll.listLoaded()`	Returns an array containing loaded modules
`dll.load(d)`	Load the module identified by string `d`
`dll.unload(d)`	Unload the module identified by string `d`
`dll.isAvailable(d)`	Returns `true` if the module identified by string `d` is known and available, otherwise `false`
`dll.listAvailableDevices()`	Returns an array containing known, available device module identifiers
`dll.listAvailableMacros()`	Returns an array containing known, available macro module identifiers
`dll.listAvailableSegments()`	Returns an array containing known, available channel segment module identifiers
`dll.getDescription(d)`	Returns a string describing the available module identified by string `d`
`dll.getSettings(d)`	Returns an array of structures describing the settings of the module identified by string `d`
`dll.getPropertiesXML(d)`	Returns the raw XML properties file of the module identified by string `d`. This file contains a great deal of information about the module, including an overall description, dependencies, settings, acceptable values, and much more. The XML schema for the properties files is available from http://*your.ce.address.here:port*/module-properties.xsd
`dll.refreshAvailable()`	Rebuild the list of known modules by searching the path specified in configuration file setting `Server.dllPath`

Table 2.6–1: DLL Management XML-RPC Directives

3 CCSDS AOS Space Data Link Protocol-Related Modules

3.1 AOS Space Data Link Common Management Settings

3.1.1 AOS Physical Channel

These settings are common to all modules that are concerned with the AOS Transfer Frame structure as governed by the Physical Channel. These include any module with an "AF" (All Frames), "MC" (Master Channel), or "VC" (Virtual Channel) in its name; Multiplexing, Bitstream, and other non-frame services are not derived from the same base. Most settings appear in the `AOS_PhysicalChannel` subsection of the configuration file, which the aforementioned modules share for these options.

Entity	**Container**	**Description**
`frameSize = `*`n`*`;`	`AOS_PhysicalChannel` subsection	Set the exact size for all AOS transfer frames in the channel to *`n`* octets
`useHeaderErrorControl = true | false;`	`AOS_PhysicalChannel` subsection	If `true`, enable frame header error control using a Reed-Solomon (10,6) code
`useFrameErrorControl = true | false;`	`AOS_PhysicalChannel` subsection	If `true`, enable frame error control via a checksum
`insertZoneSize = `*`s`*`;`	`AOS_PhysicalChannel` subsection	Reserve *`s`* octets in each transfer frame for the Insert Zone. If the Insert Zone is not in use, this should be set to zero
`idlePattern = [ `*`h1, h2, h3, …`*` ];`	`AOS_PhysicalChannel` subsection	For services that are periodic, the array of integers *`h1, h2, h3, …`* is used as a pattern to fill the data field in idle units
`dropBadFrames = true | false;`	*segment_name* subsection	If `true` (default), discard malformed frames (or frames that fail ECC in All Frames Reception) in this segment. It is recommended to only use the `false` setting in specific situations such as with frame services, otherwise traffic loss/corruption may occur

Table 3.1–1: Common AOS Physical Channel Configuration File Settings

```
Channels:
{
  …
  chanAOS_test:
  {
    …
    AOS_PhysicalChannel:
    {
      frameSize = 250;
      useHeaderErrorControl = false;
      useFrameErrorControl = true;
      useInsertZone = false;
      insertZoneSize = 0;
      idlePattern = [ 0xA1, 0xB2, 0xC3 ];
    };

    afGen: {
      …

    };
    …
  };
};
```

Figure 3.1-1: Common AOS Physical Channel Settings - Configuration File Excerpt

Operation	Effect
`*modName*.setFrameSize(c,s,f)`	Set the exact AOS transfer frame size to `f` octets for segment `s` in channel `c`. All other segments in the channel using this setting will be instantly affected
`*modName*.getFrameSize(c,s)`	Retrieve the AOS transfer frame size in octets from segment `s` of channel `c`
`*modName*.setUseHeaderEC(c,s,h)`	Enable or disable frame header error control in segment `s` in channel `c` depending on the value of Boolean `h`. All other segments in the channel using this setting will be instantly affected
`*modName*.getUseHeaderEC(c,s)`	Determine whether frame header error control is enabled for segment `s` in channel `c`

`modName.` `setUseFrameEC(c,s,f)`	Enable or disable frame error control in segment `s` in channel `c` depending on the value of Boolean `f`. All other segments in the channel using this setting will be instantly affected
`modName.` `getUseFrameEC(c,s)`	Determine whether frame error control is enable for segment `s` in channel `c`
`modName.` `setInsertZoneSize(c,s,z)`	Set the exact Insert Zone size to `z` octets for segment `s` in channel `c`. All other segments in the channel using this setting will be instantly affected
`modName.` `getInsertZoneSize(c,s)`	Retrieve the Insert Zone size in octets from segment `s` in channel `c`
`modName.` `setValidFrameCount(c,s,f)`	Set the valid frame count to `f` for segment `s` in channel `c`. Probably only useful when initializing a segment. Corresponds to the Virtual Channel Frame Count in “VC” modules
`modName.` `getValidFrameCount(c,s)`	Return the valid frame count for segment `s` in channel `c`. Corresponds to the Virtual Channel Frame Count in “VC” modules
`modName.` `setBadFrameCRCCount(c,s,r)`	Set the bad frame checksum count to `r` for segment `s` in channel `c`. Probably only useful when initializing a segment
`modName.` `getBadFrameCRCCount(c,s)`	Return the bad frame checksum count for segment `s` in channel `c`
`modName.` `setBadHeaderCount(c,s,h)`	Set the bad header count to `h` for segment `s` in channel `c`. Probably only useful when initializing a segment

`modName.getBadHeaderCount(c,s)`	Return the bad header tally for segment `s` in channel `c`. Reasons for encountering a "bad" header include an uncorrectable bit error, a bad version number, or a structural problem.
`modName.setBadLengthCount(c,s,l)`	Set the bad length count to `h` for segment `s` in channel `c`. Possibly useful when initializing a segment
`modName.getBadLengthCount(c,s)`	Return the bad frame length tally for segment `s` in channel `c`. Indicated the total number of frames encountered that were not exactly `frameSize` in length
`modName.setIdlePattern(c,s,p)`	Set the idle pattern for segment `s` in channel `c` to integer array `p`. The idle pattern is used to fill the data field of idle frames. All other segments in the channel using this setting will be instantly affected
`modName.getIdlePattern(c,s)`	Returns an integer array containing the idle pattern for segment `s` in channel `c`
`modName.setDropBadFrames(c,s,d)`	Enable or disable the discarding of malformed frames in segment `s` in channel `c` depending on the value of Boolean `d`
`modName.getDropBadFrames (c,s)`	Determine whether malformed frames are being discarded by segment `s` in channel `c`

Table 3.1–2: Common XML-RPC Directives for AOS Physical Channel Operations

3.1.2 AOS Master Channel

These settings are common to all modules that are concerned with the AOS Transfer Frame structure as governed by a Master Channel. These include any module with an "MC" (Master Channel), or "VC" (Virtual Channel) in its name; Multiplexing, Bitstream, or other non-frame services are not derived from the same base.

Entity	**Container**	**Description**
`SCID = x;`	*segment_name* subsection	Set the Spacecraft Identifier for the Master Channel to `x`.

Table 3.1–3: Common AOS Master Channel Configuration File Settings

```
Channels: {
  …
  channel1: {
    …
    mcMux: {
      dllName = "modAOS_MC_Mux";
      primaryOutput = [ "afGen", "PrimaryInput" ];
      immediateStart = true;
      SCID = 0xAB;
    };
    …
  };
};
```

Figure 3.1-2: Common AOS Master Channel Settings - Configuration File Excerpt

Operation	Effect
modName. `setSCID(c,s,i)`	Set the Spacecraft Identifier to integer `i` for segment `s` in channel `c`
modName. `getSCID(c,s)`	Retrieve the Spacecraft Identifier as an integer from segment `s` in channel `c`
modName. `setBadMCIDCount(c,s,b)`	Set the bad Master Channel Spacecraft Identifier count to `b` for segment `s` in channel `c`. Possibly useful when initializing a segment
modName. `getBadMCIDCount(c,s)`	Return the bad Master Channel Spacecraft Identifier count for segment `s` in channel `c`. Indicated the total number of frames encountered that did not have a Spacecraft Identifier matching the assigned `SCID`

Table 3.1–4: Common XML-RPC Directives for AOS Master Channel Operations

3.1.3 AOS Virtual Channel

These settings are common to all modules that are concerned with the AOS Transfer Frame structure as governed by a Virtual Channel. These include any module with an "VC" (Virtual Channel) in its name; Multiplexing, Bitstream, or other non-frame services are not derived from the same base.

Entity	Container	Description
`useOperationalControl = true | false;`	*segment_name* subsection	If `true`, use the Operational Control Field in this Virtual Channel
`VCID = y;`	*segment_name* subsection	Set the Virtual Channel Identifier for this segment to an integer `y` between 0 and 63

`serviceType = "Access" | "Bitstream" | "Packet";`	*segment_name* subsection	Specify the type of packet service that this segment will provide
`useVCFrameCycle = true | false;`	*segment_name* subsection	If `true`, count the number of times that the Virtual Channel Frame Count has been reset to 0

Table 3.1–5: Common AOS Virtual Channel Configuration File Settings

```
Channels:
{
  …
  chanX:
  {
    …
    vcGen: {
      dllName = "modAOS_VC_Gen";
      highWaterMark = 16777216;
      lowWaterMark = 12582912;
      primaryOutput = [ "afGen", "PrimaryInput" ];
      immediateStart = true;
      SCID = 0xAB;
      VCID = 0x1;
      serviceType = "Bitstream";
      useOperationalControl = false;
      useVCFrameCycle = false;
    };
    …
  };
};
```

Figure 3.1-3: Common AOS Virtual Channel Settings - Configuration File Excerpt

Operation	**Effect**
modName.`setUseOperationalControlField(c,s,u)`	If `u` is `true`, enable the OCF for segment `s` in channel `c`
modName.`getUseOperationalControlField(c,s)`	Determine if the OCF is being used in segment `s` of channel `c`
modName.`setVCID(c,s,v)`	Set the Virtual Channel Identifier to `v`, an integer between 0 and 63, for segment `s` in channel `c`
modName.`getVCID(c,s)`	Get the integer value for the Virtual Channel Identifier of segment `s` in channel `c`
modName.`setUseVCFrameCountCycle(c,s,o)`	If `o` is `true`, count the number of times that the Virtual Channel Frame Count has been reset to 0 in segment `s` of channel `c`

modName. getUseVCFrameCountCycle(c,s)	Return whether Virtual Channel Frame Count Cycle counter is in use for segment `s` of channel `c`
modName. setFrameCountCycle(c,s,y)	Set the Virtual Channel Frame Count Cycle counter to integer `y` for segment `s` of channel `c`. Possibly useful during initialization
modName. getFrameCountCycle(c,s)	Retrieve the Virtual Channel Frame Count Cycle counter for segment `s` of channel `c`
modName. setServiceType(c,s,t)	Set the service type of segment `s` in channel `c` to `t`, where `t` is a string value equal to one of "`Bitstream`", "`Packet`", "`Access`", or "`Idle`"
modName. getServiceType(c,s)	Get the service type of segment `s` in channel `c` as a string value
modName. setBadGVCIDCount(c,s,b)	Set the tally of bad VC Identifiers encountered to integer `b`, for segment `s` in channel `c`. Possibly useful during initialization
modName. getBadGVCIDCount(c,s)	Return the tally of bad VC Identifiers encountered in segment `s` of channel `c`

Table 3.1–6: Common XML-RPC Directives for AOS Virtual Channel Operations

3.2 All Frames Generation (modAOS_AF_Gen)

The All Frames Generation function accepts AOS Transfer Frame wrappers on its primary input queue. The All Frames Generation function itself is *not* periodic and does not generate its own idle frames. The Reed-Solomon (10,6) algorithm generates the Frame Header Error Control code if enabled and inserts in the two octets reserved for it in the header.

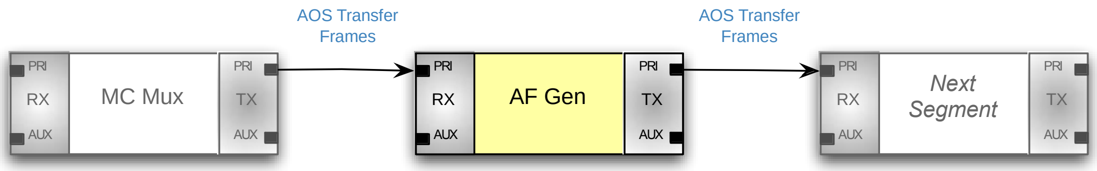

Figure 3.2-1: All Frames Generation Architecture Sample

Figure 3.2-1 depicts a sample layout of a configured All Frames Generation segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, "Modular Segments" and 3.1.1, "AOS Physical Channel". Figure 3.2-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanAOS_test:
  {
    …
    AOS_PhysicalChannel:
    {
      frameSize = 256;
      useHeaderErrorControl = false;
      useFrameErrorControl = true;
      useInsertZone = true;
      insertZoneSize = 10;
      idlePattern = [ 0xA1, 0xB2, 0xC3 ];
    };
    …
    mcMux: {
      dllName = "modAOS_MC_Mux";
      primaryOutput = [ "afGen", "PrimaryInput" ];
      …
    };

    afGen: {
      dllName = "modAOS_AF_Gen";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      immediateStart = true;
    };
    …
  };
};
```

Figure 3.2-2: All Frames Generation Configuration File Example

3.3 All Frames Reception (modAOS_AF_Rcv)

The All Frames Reception function accepts AOS Transfer Frame wrappers into its primary input queue. It performs error control decoding if enabled: if the code in Frame Error Control Field is invalid, the frame is dropped. If header error correction is enabled, the Reed-Solomon (10,6) algorithm performs correction on the header. If it is uncorrectable, the frame is dropped and the bad header counter incremented. If the frame is valid, it is passed on to the segment configured as the modAOS_AF_Rcv segment's primary output.

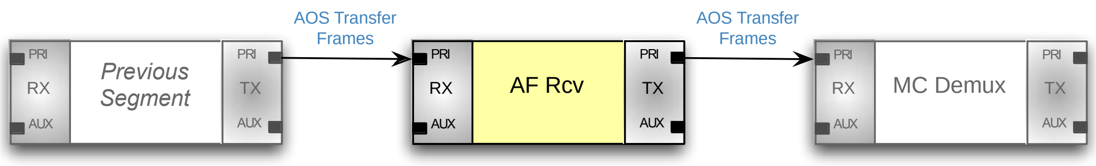

Figure 3.3-1: All Frames Reception Architecture Sample

Figure 3.3-1 depicts a sample layout of a configured All Frames Reception segment. This module derives all of its configuration file settings and XML-RPC directives from those described in sections 2.4, “Modular Segments” and 3.1.1, “AOS Physical Channel”. Figure 3.3-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanAOS_test:
  {
    …
    AOS_PhysicalChannel:
    {
      frameSize = 256;
      useHeaderErrorControl = false;
      useFrameErrorControl = true;
      useInsertZone = true;
      insertZoneSize = 10;
      idlePattern = [ 0xA1, 0xB2, 0xC3 ];
    };
    …
    afRcv: {
      dllName = "modAOS_AF_Rcv";
      primaryOutput = [ "mcDemux", "PrimaryInput" ];
      immediateStart = true;
    };
    …
  };
};
```

Figure 3.3-2: All Frames Reception Configuration File Example

3.4 Bitstream Forward Service (modAOS_B_PDU_Add)

This service accepts wrappers with untyped data, splits (or merges) their buffers into the length configured for the Bitstream PDU Data Field, and finally transmits Bitstream PDU wrappers on its primary output.

It can be configured as either asynchronous or periodic. If periodic, a Bitstream PDU will always be sent after a constant time interval. If too little (or no) data is available at transmission time, the Data Field will be filled with an idle pattern.

To determine the length of outgoing Bitstream PDUs, modAOS_B_PDU_Add first checks for a locally specified MTU (e.g. specified by the configuration file `MTU` setting in the segment’s section). If that value is zero, it queries its primary output target for an *MRU* value and uses it for the length. If still zero or too short, the service does not have enough information to create a new Bitstream PDU and must drop any data it receives.

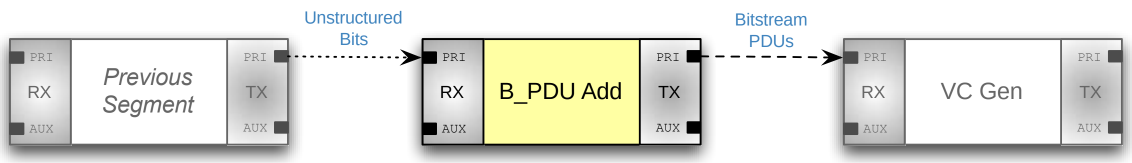

Figure 3.4-1: Bitstream Encoding Service Architecture Sample

Figure 3.4-1 shows a sample layout for an operating Bitstream Encoding segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments". Figure 3.4-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {
  …
  chanAOS1: {
    …
    vcGen: {
      dllName = "modAOS_VC_Gen";
      …
    };

    bpduAdd: {
      dllName = "modAOS_B_PDU_Add";
      primaryOutput = [ "vcGen", "PrimaryInput" ];
      immediateSend = true;
      immediateStart = true;
      MTU = 0; # Zero value means retrieve from vcGen
    };
  };
};
```

Figure 3.4-2: Bitstream Forward Service Configuration File Excerpt

Explanations of settings with behavior particular to this module are found in Table 3.4–1: Bitstream Forward Service Configuration File Settings and Table 3.4–2: XML-RPC Directives for Bitstream Forward Operations.

Entity	Container	Description
`immediateSend = true | false;`	*segment_name* subsection	If `true`, transmit each B_PDU as it is created (default). If `false`, wait until an incoming buffer has been completely encoded into multiple B_PDUs before sending.
`MTU = t;`	*segment_name* subsection	Set the size of each outgoing Bitstream PDU to `t` octets. This behavior is unchanged from that described in Table 2.4–1, however, if unspecified, it is automatically derived from the *MRU* of its primary output target. The latter is the preferred behavior in a complete AOS channel.

Table 3.4–1: Bitstream Forward Service Configuration File Settings

Operation	Effect
`modAOS_B_PDU_Add.setLength(c,s,x)`	Set the MTU (aka the Bitstream PDU length) to `x` octets for segment `s` in channel `c`
`modAOS_B_PDU_Add.getLength(c,s)`	Retrieve the MTU/Bitstream PDU length for segment `s` in channel `c`
`modAOS_B_PDU_Add.setImmediateSend(c,s,i)`	Set the immediate send flag to `i` for segment `s` in channel `c`. Refer to Table 3.4–1 for an explanation of the immediate send flag
`modAOS_B_PDU_Add.getImmediateSend(c,s)`	Get the immediate send flag status for segment `s` in channel `c`

Table 3.4–2: XML-RPC Directives for Bitstream Forward Operations

3.5 Bitstream Return Service (modAOS_B_PDU_Remove)

This relatively simple service accepts Bitstream PDU wrappers, extracts buffers from the Bitstream PDU Data Field, and wraps those as untyped network data and transmits them on its primary output. Idle fill is dropped.

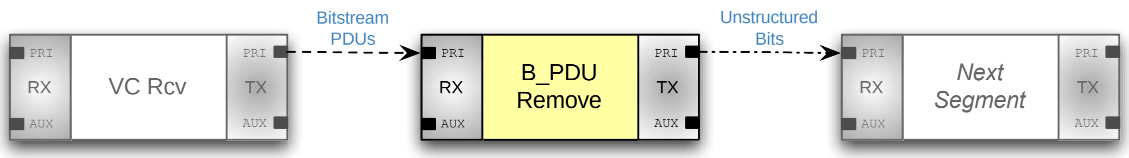

Figure 3.5-1: Bitstream Return Service Architecture Sample

Figure 3.5-1 depicts a sample layout of a configured Bitstream decoding segment. This module derives all of its configuration file settings and XML-RPC directives from those described in section 2.4, “Modular Segments”. Figure 3.5-2 contains configuration file excerpts that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chan_B_PDU_Test: {
    …
    bpduDel: {
      dllName = "modAOS_B_PDU_Remove";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      immediateStart = true;
    };

    vcRcv: {
      dllName = "modAOS_VC_Rcv";
      primaryOutput = [ "bpduDel", "PrimaryInput" ];
      …
    };
    …
  };
};
```

Figure 3.5-2: Bitstream Return Service Configuration File Excerpt

3.6 Master Channel De-multiplexing Function (modAOS_MC_Demux)

This module accepts AOS Transfer Frame wrappers on its primary input, separates them based on their Master Channel Identifier (MCID), and sends them out an associated primary output. There can be as many primary outputs as there are possible master channels in one physical channel, and each is associated with a single MCID.

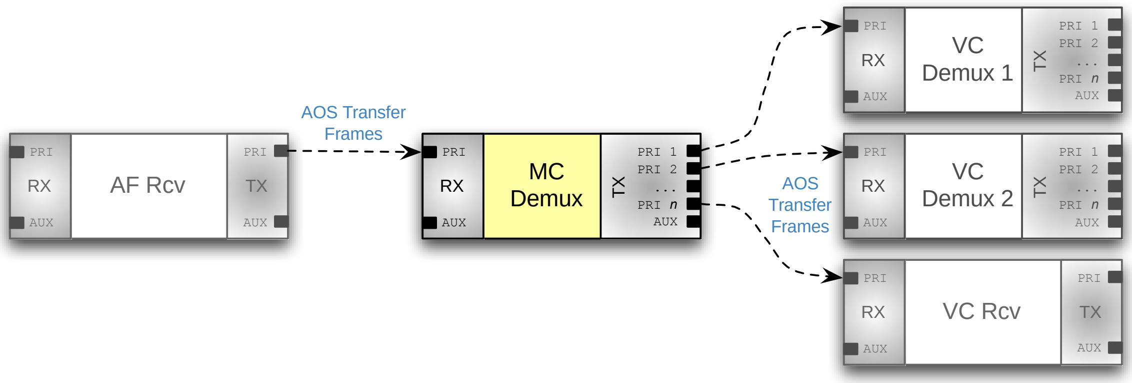

Figure 3.6-1: Master Channel De-multiplexing Sample Architecture

Figure 3.6-1 depicts a sample layout of a configured Master Channel De-multiplex segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments” and chapter

3.1.1, "AOS Physical Channel". The exception is the `primaryOutputs` setting. Similar to `primaryOutput`, found in most modules, `primaryOutputs` also specifies which segments the main flow of traffic is directed to, and on which input. However, rather than being a one-dimensional array, this setting is a list of lists. Each sub list contains the next segment name, its input type, and a third parameter – the Spacecraft Identifier (SCID) of that master channel (not the MCID, which is computed automatically from the SCID and TVFN).

Figure 3.6-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {

    AOS_PhysicalChannel:
    {
      frameSize = 250;
      useHeaderErrorControl = false;
      useFrameErrorControl = true;
      useInsertZone = false;
      insertZoneSize = 0;
      idlePattern = [ 0xA1, 0xB2, 0xC3 ];
    };
    …
    vcDemux1: { # segment name
      dllName = "modAOS_VC_Demux";
      primaryOutputs = ( ( "Next Segment E", "PrimaryInput", 0x1),
                         ( "Next Segment D", "PrimaryInput", 0x2),
                         …
                       );
      SCID = 0xA1;
      immediateStart = true;
      …
    };

    vcDemux2: { # segment name
      dllName = "modAOS_VC_Demux";
      primaryOutputs = ( ( "Next Segment C", "PrimaryInput", 0x1),
                         ( "Next Segment B", "PrimaryInput", 0x2),
                         …
                       );
      SCID = 0xB2;
      immediateStart = true;
      …
    };

    vcRcv: { # segment name
      dllName = "modAOS_VC_Rcv";
      primaryOutput = [ "Next Segment A", "PrimaryInput" ];
      SCID = 0xD4;
      VCID = 0x1;
      immediateStart = true;
      …
    };

    mcDemux: { # segment name
      dllName = "modAOS_MC_Demux";
      primaryOutputs = ( ( "vcDemux1", "PrimaryInput", 0xA1),
                         ( "vcDemux2", "PrimaryInput", 0xB2),
                         …,
                         ( "vcRcv", "PrimaryInput", 0xD4),
                       );
      immediateStart = true;
      …
    };

    afRcv: { # segment name
      dllName = "modAOS_AF_Rcv";
      primaryOutput = [ "mcDemux", "PrimaryInput" ];
      immediateStart = true;
      …
    };
    …
  };
};
```

Figure 3.6-2: Master Channel De-multiplexing Configuration File Excerpt

Entity	Container	Description
`primaryOutputs = ( ( "s", "PrimaryInput" | "AuxInput", i ), … );`	*segment_name* subsection	AOS Transfer Frames received with Spacecraft Identifier `i` are sent on to either the Primary or Auxiliary input of segment `s`. Segment `s` must already exist in the channel

Table 3.6–1: Master Channel De-multiplexing Configuration File Settings

Operation	Effect
`modAOS_MC_Demux.connectOutput(c,s1,d,s2,o,i)`	In channel `c`, connect an output of segment `s1` to an input of segment `s2`. If the output type of `s1`, specified by `o`, is "`Primary`", use the integer `d` to specify the Spacecraft Identifier for AOS Transfer Frames to be sent over that link; if `o` is "`Auxiliary`", `d` is ignored. The input type (either "`Primary`" or "`Auxiliary`") of `s2` is specified with parameter `i`

Table 3.6–2: XML-RPC Directives for Master Channel De-multiplexing Operations

3.7 Master Channel Frame Service (modAOS_MC_Frame)

This service accepts a generic network data wrapper whose buffer must contain a correctly formed AOS Transfer Frame. The segment re-wraps it as an AOS Transfer Frame. It's sent on via the primary output link to the next segment, typically either the Master Channel Multiplexing function or All Frames Generation function.

Incoming transfer frames may have different GVCIDs as long as their MCIDs are identical. Frames with a mismatched MCID (i.e. that does not match the one the segment is configured to accept) will be dropped.

The service can be configured as either asynchronous or periodic. If periodic, an AOS Transfer Frame will always be sent after a constant time interval. If no data is available at transmission time, an Idle Frame will be sent using the Spacecraft Identifier specified for the segment and Virtual Channel Identifier 63 (3Fh).

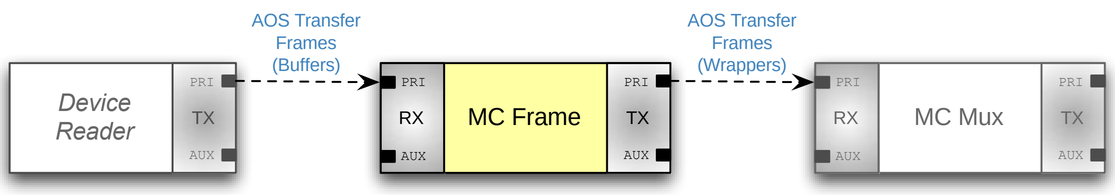

Figure 3.7-1: Master Channel Frame Service Architecture Sample

Figure 3.7-1 depicts a sample layout of a configured Master Channel Frame Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments", chapter 3.1.1, "AOS Physical Channel", and chapter 3.1.2, "AOS Master Channel". Figure 3.7-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    mcMux: { # segment name
      dllName = "modAOS_MC_Mux";
      immediateStart = true;
      …
    };

    mcFrame: { # segment name
      dllName = "modAOS_MC_Frame";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      SCID = 0x95;
      immediateStart = true;
    };

    deviceReader: { # segment name
      primaryOutput = [ "mcFrame", "PrimaryInput" ];
      deviceName = "sourceDevice";
      immediateStart = true;
      …
    };
    …
  };
};
```

Figure 3.7-2: Master Channel Frame Service Configuration File Excerpt

3.8 Master Channel Multiplexing Function (modAOS_MC_Mux)

This module has multiple primary input links from which it accepts AOS Transfer Frames with different Master Channel Identifiers (MCIDs). The frames are prioritized in the queue according to their MCID. By default, all have the same priority. If those are modified, frames with the highest-priority MCID are sent until there are none remaining; then, frames with the next highest priority are sent until there are none remaining, and so on. If high priority traffic does not abate, low priority traffic will never be sent.

This service is optionally periodic – it sends frames at a constant rate and will send Idle Frames when there is no incoming data. The SCID setting in the configuration file is used to specify which Master Channel Identifier the Idle Frames will have (and that is the only use of the SCID setting in this particular module). The Virtual Channel Identifier is always 63 (3Fh).

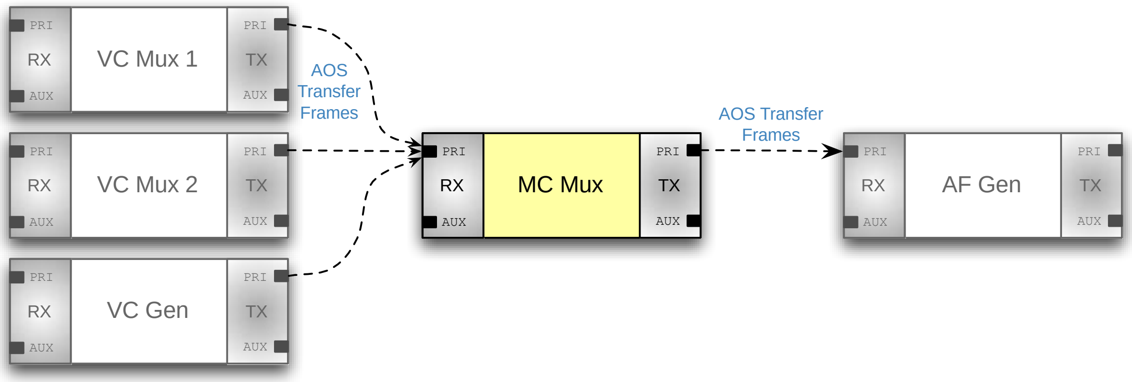

Figure 3.8-1: Master Channel Multiplexing Architecture Sample

Figure 3.8-1 depicts a sample layout of a configured Master Channel Multiplexing segment. This module derives most of its configuration file settings and XML-RPC directives from those described in section 2.4, “Modular Segments”, section 3.1.1, “AOS Physical Channel”, and section 3.1.2, “AOS Master Channel”. Additional settings are described later in this section. Figure 3.8-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    afGen: { # segment name
      dllName = "modAOS_AF_Gen";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      immediateStart = true;
      …
    };

    mcMux: { # segment name
      dllName = "modAOS_MC_Mux";
      primaryOutput = [ "afGen", "PrimaryInput" ];
      defaultPriority = 1000;
      channelID_Priorities = ( [ 0xA1, 1010 ], [ 0xB2, 900 ] );
      immediateStart = true;
    };

    vcMux1: { # segment name
      dllName = "modAOS_VC_Mux";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      immediateStart = true;
      …
    };

    vcMux2: { # segment name
      dllName = "modAOS_VC_Mux";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      immediateStart = true;
      …
    };

    vcGen: { # segment name
      dllName = "modAOS_VC_Gen";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      immediateStart = true;
      …
    };
    …
  };
};
```

Figure 3.8-2: Master Channel Multiplexing Function Configuration File Excerpt

Entity	Container	Description
`defaultPriority = `*`p`*`;`	*segment_name* subsection	Incoming AOS Transfer Frames are inserted into the message queue with integer priority `p` if they don't have their own priority specified
`channelID_Priorities = ( [ `*`m`*`, `*`p`*` ], … );`	*segment_name* subsection	For an incoming AOS Transfer Frame with SCID *`m`*, insert it into the message queue with priority *`p`*

Table 3.8–1: Master Channel Multiplexing Function Configuration File Settings

Operation	Effect
`modAOS_MC_Mux.setDefaultPriority(c,s,p)`	Set the default priority for incoming AOS Transfer Frames to integer `p` for segment `s` of channel `c`
`modAOS_MC_Mux.getDefaultPriority(c,s)`	Retrieve the default priority for incoming AOS Transfer Frames for segment `s` of channel `c`
`modAOS_MC_Mux.setPriority(c,s,m,p)`	Set the priority for incoming AOS Transfer Frames with Spacecraft ID `m` to `p`, for segment `s` of channel `c`
`modAOS_MC_Mux.getPriority(c,s,m)`	Get the priority for incoming AOS Transfer Frames with Spacecraft ID `m` in segment `s` of channel `c`

Table 3.8–2: XML-RPC Directives for Master Channel Multiplexing Function Operations

3.9 Packet Forward Service (modAOS_M_PDU_Add)

This service accepts variable-length Encapsulation Packet wrappers. A single large Encapsulation Packet may either be split across multiple M_PDUs, or, several small Encapsulation Packets may be inserted into a single M_PDU Packet Zone. If there is empty space remaining in the Packet Zone and another Packet is waiting in the queue, the front of that Packet will be added to the current M_PDU. The rest will be put into the next M_PDU. Finally, M_PDU wrappers are sent via the primary output to the next segment, typically the primary input of Virtual Channel Generation segment.

The size of the M_PDU (and from it, the size of the Packet Zone) is determined automatically by querying the MRU of the primary output's target segment if it already exists. Otherwise, the MTU setting is used for the M_PDU length. If neither is set, the incoming data must be dropped.

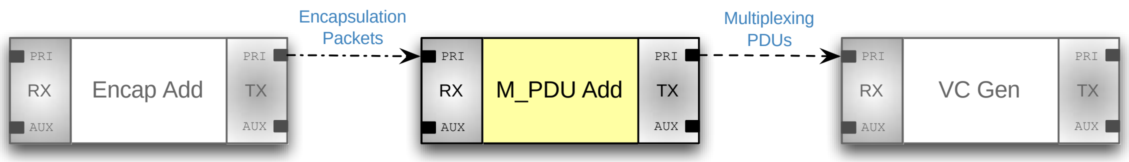

Figure 3.9-1: Packet Forward Service Architecture Sample

Figure 3.9-1 depicts a sample layout of a configured Multiplexing PDU Encoding Service segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments". Additional settings are described later in this section. Figure 3.9-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {
  chanTest: {
    …
    vcGen:
    {
      dllName = "modAOS_VC_Gen";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      …
    };

    mpduAdd:
    {
      dllName = "modAOS_M_PDU_Add";
      primaryOutput = [ "vcGen", "PrimaryInput" ];
      fillPattern = [ 0xE0 ];
      multiPacketZone = true;
    };

    encapAdd :
    {
      dllName = "modEncapPkt_Add";
      primaryOutput = [ "mpduAdd", "PrimaryInput" ];
    };
    …
  };
};
```

Figure 3.9-2: Packet Forward Service Configuration File Excerpt

Entity	Container	Description
`maxUsecsForNextPacket = s;`	*segment_name* subsection	Set the number of microseconds to wait for a new incoming packet before sending the current one partially empty (i.e. the remainder of the Packet Zone containing only fill packets instead of data)
`multiPacketZone = true | false;`	*segment_name* subsection	If `true`, one packet may end in the Packet Zone and one or more additional packets may be inserted afterward (assuming there is room) in the same unit. If `false`, new packets will only begin only at the first index in the Packet Zone
`fillPattern = [ o1, o2, ... ]`	*segment_name* subsection	Sets the Packet Zone fill pattern to the specified array

Table 3.9–1: Packet Forward Service Configuration File Settings

Operation	Effect
`modAOS_M_PDU_Add. getWaitForNextPacket(c,s)`	Get the number of microseconds that segment `s` in channel `c` will wait for a new incoming packet before sending the current one partially empty (i.e. the remainder of the Packet Zone containing only fill packets instead of data)
`modAOS_M_PDU_Add. setWaitForNextPacket(c,s,u)`	Set the number of microseconds to `u` that segment `s` in channel `c` will wait for a new incoming packet before sending the current one partially empty (i.e. the remainder of the Packet Zone containing only fill packets instead of data)
`modAOS_M_PDU_Add. getMultiPacketZone(c,s)`	Retrieve whether or not segment `s` in channel `c` allows more than one packet in the Packet Zone of MPDUs it creates.
`modAOS_M_PDU_Add. setMultiPacketZone(c,s,z)`	Modify whether or not segment `s` in channel `c` allows more than one packet in the Packet Zone of MPDUs by supplying Boolean value `z`.
`modAOS_M_PDU_Add. getFillPattern(c,s)`	Get an array containing the Packet Zone fill pattern used in segment `s` of channel `c`
`modAOS_M_PDU_Add. setFillPattern(c,s,p)`	Set the Packet Zone fill pattern to integer array `p` in segment `s` of channel `c`

Table 3.9–2: XML-RPC Directives for Multiplexing PDU Encoding Operations

3.10 Packet Return Service (modAOS_M_PDU_Remove)

This service accepts fixed-length Multiplexing Protocol Data Units (M_PDUs), with Packet Zones that contain variable-length Encapsulation Packets. A single M_PDU may carry a partial or whole Packet, or multiples, depending on the relative lengths; there may also be idle data if no incoming traffic was available to fill the remainder of the Packet Zone.

Encapsulation Packets are extracted from the Packet Zone buffers and reassembled in their own buffer, then wrapped and sent via the primary output to the next segment, which typically removes the Packet header.

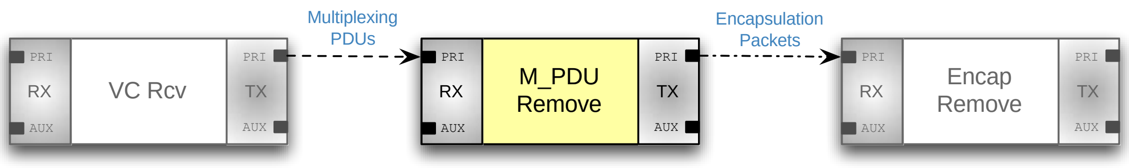

Figure 3.10-1: Packet Return Service Architecture Sample

Figure 3.10-1 depicts a sample layout of a configured Multiplexing PDU Decoding Service segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”. Additional settings are described later in this section. Figure 3.10-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    encapRemove :
    {
      dllName = "modEncapPkt_Remove";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      immediateStart = true;
    };

    mpduRemove:
    {
      dllName = "modAOS_M_PDU_Remove";
      primaryOutput = [ "encapRemove", "PrimaryInput" ];
      multiPacketZone = true;
      allowPacketsAfterFill = false;
    };

    vcRcv:
    {
      dllName = "modAOS_VC_Rcv";
      primaryOutput = [ "mpduRemove", "PrimaryInput" ];
      immediateStart = true;
      VCID = 0x2B;
      SCID = 0x35;
      …
    };
    …
  };
};
```

Figure 3.10-2: Packet Return Service Configuration File Excerpt

Explanations of settings with behavior particular to this module are found in Table 3.10–1: Packet Return Service Configuration File Settings and Table 3.10–2: XML-RPC Directives for Packet Return Service Operations.

Entity	Container	Description
`supportIPE = true | false;`	*segment_name* subsection	If `true` (default), expect the presence of the Internet Protocol Extension (IPE) shim, which may be 1 to 8 octets.
`multiPacketZone = true | false;`	*segment_name* subsection	If `true`, expect that one packet may end in the Packet Zone and one or more additional packets may be found afterward in the same unit. If `false`, new packets will only begin only at the first index in the Packet Zone
`allowPacketsAfterFill = true | false;`	*segment_name* subsection	If `false`, stop searching for new packets in the Packet Zone when a fill value of `0xe0` is encountered. If `true`, continue searching for new packets until a non-empty one is found or the end of the Packet Zone is reached.

Table 3.10–1: Packet Return Service Configuration File Settings

Operation	Effect
`modAOS_M_PDU_Remove.setSupportIPE(c,s,i)`	Enable or disable Internet Protocol Extension (IPE) support in segment `s` of channel `c`, depending on whether `e` is `true` or `false`, respectively. Defaults to `true`
`modAOS_M_PDU_Remove.getSupportIPE(c,s)`	Fetch the state of IPE support for segment `s` in channel `c`
`modAOS_M_PDU_Remove.getMultiPacketZone(c,s)`	Retrieve whether or not segment `s` in channel `c` can handle more than one packet in the Packet Zone of MPDUs it receives
`modAOS_M_PDU_Remove.setMultiPacketZone(c,s,z)`	Modify whether or not segment `s` in channel `c` will handle more than one packet in the Packet Zone of MPDUs by supplying Boolean value `z`
`modAOS_M_PDU_Remove.getAllowPacketsAfterFill(c,s)`	Get whether segment `s` in channel `c` will continue searching for packets in a single Packet Zone after it encounters a fill pattern of `0xe0`
`modAOS_M_PDU_Remove.setAllowPacketsAfterFill(c,s,f)`	Set to Boolean `f` whether segment `s` in channel `c` will continue searching for packets in a single Packet Zone after it encounters a fill pattern of `0xe0`

Table 3.10–2: XML-RPC Directives for Packet Return Service Operations

3.11 Virtual Channel De-multiplexing Function (modAOS_VC_Demux)

The Virtual Channel De-multiplexing module accepts AOS Transfer Frames that may have varying GVCIDs. It selects them based on their GVCID and sends them out one of several primary outputs to a segment that only accepts a single GVCID.

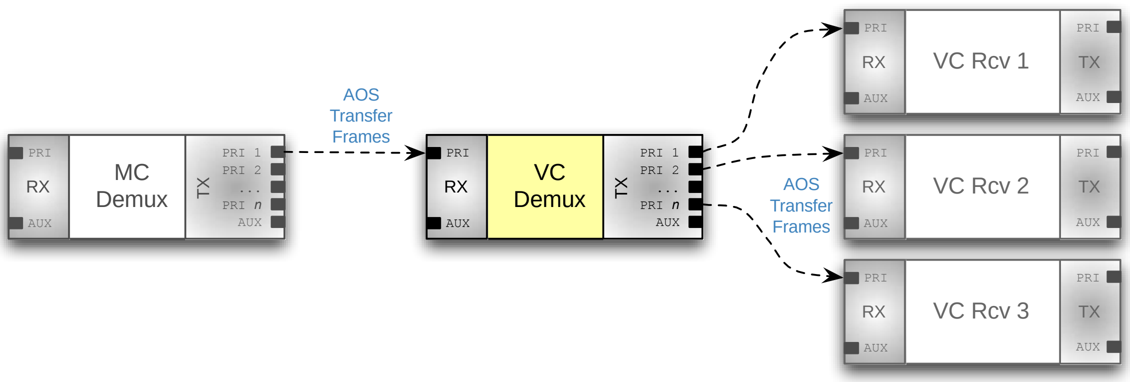

Figure 3.11-1: Virtual Channel De-multiplexing Function Architecture Sample

Figure 3.11-1 depicts a sample layout of a configured Virtual Channel De-multiplexing segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”, chapter 3.1.1, “AOS Physical Channel”, and chapter 3.1.2, “AOS Master Channel”. The exception is the primaryOutputs setting. Similar to primaryOutput, found in most modules, primaryOutputs also specifies which segments the main flow of traffic is directed to, and on which input. However, rather than being a one-dimensional array, this setting is a list of lists. Each sub list contains the next segment name, its input type, and a third parameter – the Virtual Channel Identifier (VCID) of that virtual channel (not the GVCID, which is computed automatically from the VCID, SCID, and TVFN).

Figure 3.11-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {

    AOS_PhysicalChannel:
    {
      frameSize = 256;
      useHeaderErrorControl = false;
      useFrameErrorControl = true;
      useInsertZone = false;
      insertZoneSize = 0;
      idlePattern = [ 0xAA, 0xBB, 0xCC ];
    };
    …
    vcRcv3: { # segment name
      dllName = "modAOS_VC_Rcv";
      primaryOutput = [ "Next Segment C", "PrimaryInput" ];
      SCID = 0xA1;
      VCID = 0x14;
      immediateStart = true;
      …
    };

    vcRcv2: { # segment name
      dllName = "modAOS_VC_Rcv";
      primaryOutput = [ "Next Segment B", "PrimaryInput" ];
      SCID = 0xA1;
      VCID = 0x12;
      immediateStart = true;
      …
    };

    vcRcv1: { # segment name
      dllName = "modAOS_VC_Rcv";
      primaryOutput = [ "Next Segment A", "PrimaryInput" ];
      SCID = 0xA1;
      VCID = 0x11;
      immediateStart = true;
      …
    };

    vcDemux: { # segment name
      dllName = "modAOS_VC_Demux";
      primaryOutputs = ( ( "vcRcv1", "PrimaryInput", 0x11),
                         ( "vcRcv2", "PrimaryInput", 0x12),
                         …,
                         ( "vcRcv3", "PrimaryInput", 0x14),
                       );
      SCID = 0xA1;
      immediateStart = true;
      …
    };

    mcDemux: { # segment name
      dllName = "modAOS_MC_Demux";
      primaryOutputs = ( ( "vcDemux", "PrimaryInput", 0xA1),
                         …
                       );
      immediateStart = true;
      …
    };
    …
  };
};
```

Figure 3.11-2: Virtual Channel De-multiplexing Function Configuration File Excerpt

Entity	Container	Description
`primaryOutputs = ( ( "s", "PrimaryInput" | "AuxInput", i ), … );`	*segment_name* subsection	AOS Transfer Frames received with Virtual Channel Identifier `i` are sent on to either the Primary or Auxiliary input of segment `s`. Segment `s` must already exist in the channel

Table 3.11–1: Master Channel De-multiplexing Configuration File Settings

Operation	Effect
`modAOS_VC_Demux.connectOutput(c,s1,d,s2,o,i)`	In channel `c`, connect an output of segment `s1` to an input of segment `s2`. If the output type of `s1`, specified by `o`, is "`Primary`", use the integer `d` to specify the Virtual Channel Identifier for AOS Transfer Frames to be sent over that link; if `o` is "`Auxiliary`", `d` is ignored. The input type (either "`Primary`" or "`Auxiliary`") of `s2` is specified with parameter `i`

Table 3.11–2: XML-RPC Directives for Master Channel De-multiplexing Operations

3.12 Virtual Channel Frame Service (modAOS_VC_Frame)

This service accepts a generic network data wrapper whose buffer must contain a correctly formed AOS Transfer Frame. The segment re-wraps it with an AOS Transfer Frame wrapper. It's sent on via the primary output link to the next segment, typically the Virtual Channel Multiplexing function.

Incoming transfer frames must have identical GVCIDs. Frames with a mismatched GVCID (i.e. that does not match the one the segment is configured to accept) will be dropped.

The service can be configured as either asynchronous or periodic. If periodic, an AOS Transfer Frame will always be sent after a constant time interval. If no data is available at transmission time, an Idle Frame will be generated. Note that the Idle Frame will have a VCID of 63 (3Fh) rather than the same VCID as incoming frames.

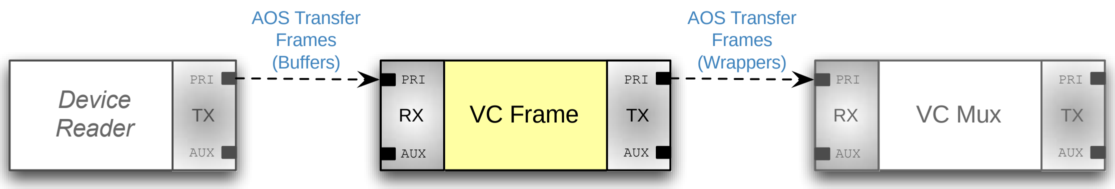

Figure 3.12-1: Virtual Channel Frame Service Architecture Sample

Figure 3.12-1 depicts a sample layout of a configured Virtual Channel Frame Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments" and chapters 3.1 through 3.1.3, "AOS Common Settings". Figure 3.12-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {

    AOS_PhysicalChannel:
    {
      frameSize = 256;
      useHeaderErrorControl = false;
      useFrameErrorControl = true;
      useInsertZone = false;
      insertZoneSize = 0;
      idlePattern = [ 0xAA, 0xCC, 0xEE ];
    };
    …
    vcMux: { # segment name
      dllName = "modAOS_VC_Mux";
      immediateStart = true;
      SCID = 0x95;
      …
    };

    vcFrame: { # segment name
      dllName = "modAOS_VC_Frame";
      primaryOutput = [ "vcMux", "PrimaryInput" ];
      SCID = 0x95;
      VCID = 0x22;
      sendIntervalUsec = 100;
      immediateStart = true;
    };

    deviceReader: { # segment name
      primaryOutput = [ "vcFrame", "PrimaryInput" ];
      deviceName = "Source Device";
      immediateStart = true;
      …
    };
    …
  };
};
```

Figure 3.12-2: Virtual Channel Frame Service Configuration File Excerpt

3.13 Virtual Channel Generation Function (modAOS_VC_Gen)

This module is responsible for encoding data into AOS Transfer Frames. It accepts units on its primary input with a specific buffer length, but otherwise the type is ignored (it's the responsibility of the operator to ensure that the correct type of data is being fed). It inserts these units into the Data Field of new frames, and sends them on via the segment's primary output link.

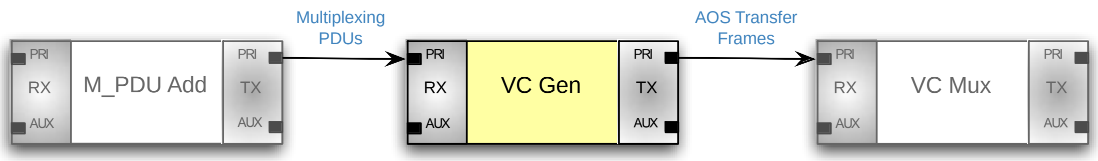

Figure 3.13-1: Virtual Channel Generation Function Sample Architecture

Figure 3.13-1 depicts a sample layout of a configured Virtual Channel Generation segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments" and chapter 3.1, "AOS Space Data Link Common Management Settings". Figure 3.13-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {

    AOS_PhysicalChannel:
    {
      frameSize = 256;
      useHeaderErrorControl = false;
      useFrameErrorControl = true;
      useInsertZone = false;
      insertZoneSize = 0;
      idlePattern = [ 0xAA, 0xCC, 0xEE ];
    };
    …

    mpduAdd:
    {
      dllName = "modAOS_M_PDU_Add";
      primaryOutput = [ "vcGen", "PrimaryInput" ];
      immediateStart = true;
    };

    vcGen: { # segment name
      dllName = "modAOS_VC_Gen";
      primaryOutput = [ "vcMux", "PrimaryInput" ];
      SCID = 0x95;
      VCID = 0x22;
      useOperationalControl = true;
      useVCFrameCycle = true;
      immediateStart = true;
    };

    vcMux: { # segment name
      dllName = "modAOS_VC_Mux";
      immediateStart = true;
      SCID = 0x95;
      …
    };
    …
  };
};
```

Figure 3.13-2: Virtual Channel Generation Function Configuration File Excerpt

3.14 Virtual Channel Multiplexing Function (modAOS_VC_Mux)

This module has multiple primary input links from which it accepts AOS Transfer Frames with different Global Virtual Channel Identifiers (GVCIDs). The frames are prioritized in the queue according to their GVCID. By default, all have the same priority. If those are modified, frames with the highest-priority GVCID are sent until there are none remaining; then, frames with the next highest priority are sent until there are none remaining, and so on. If high priority traffic does not abate, low priority traffic will never be sent.

This service is optionally periodic – it can send frames at a constant rate and send Idle Frames with VCID 63 (3Fh) when no incoming data is available.

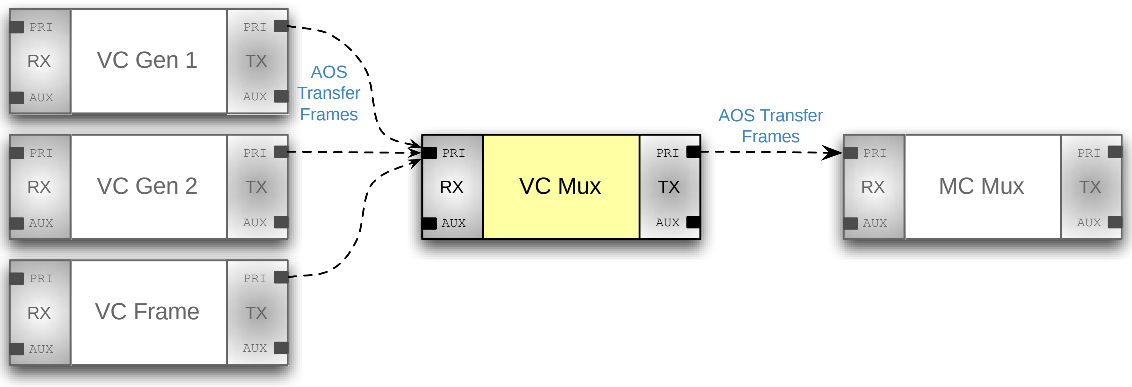

Figure 3.14-1: Virtual Channel Multiplexing Function Architecture Sample

Figure 3.14-1 depicts a sample layout of a configured Virtual Channel Multiplexing segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”, and chapter 3.1, “AOS Space Data Link Common Management Settings”. Additional settings are described later in this chapter. Figure 3.14-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {

    AOS_PhysicalChannel:
    {
      frameSize = 250;
      useHeaderErrorControl = false;
      useFrameErrorControl = true;
      useInsertZone = false;
      insertZoneSize = 0;
      idlePattern = [ 0xAA, 0xCC, 0xEE ];
    };
    …
    mcMux: { # segment name
      dllName = "modAOS_MC_Mux";
      primaryOutput = [ "afGen", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    vcMux: { # segment name
      dllName = "modAOS_VC_Mux";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      defaultPriority = 1000;
      channelID_Priorities = ( [ 0x2, 1010 ], [ 0x3, 900 ] );
      SCID = 0xAB;
      sendIntervalUsec = 100;
      …
      immediateStart = true;
    };

    vcGen2: { # segment name
      dllName = "modAOS_VC_Gen";
      primaryOutput = [ "vcMux", "PrimaryInput" ];
      SCID = 0xAB;
      VCID = 0x3;
      …
      immediateStart = true;
    };

    vcGen1: { # segment name
      dllName = "modAOS_VC_Gen";
      primaryOutput = [ "vcMux", "PrimaryInput" ];
      SCID = 0xAB;
      VCID = 0x2;
      …
      immediateStart = true;
    };

    vcFrame: { # segment name
      dllName = "modAOS_VC_Frame";
      primaryOutput = [ "vcMux", "PrimaryInput" ];
      SCID = 0xAB;
      VCID = 0x01;
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 3.14-2: Virtual Channel Multiplexing Function Configuration File Excerpt

Entity	Container	Description
`defaultPriority = `*`p`*`;`	*segment_name* subsection	Incoming AOS Transfer Frames are inserted into the message queue with integer priority `p` if they don't have their own priority specified
`channelID_Priorities = ( [ `*`v`*`, `*`p`*` ], … );`	*segment_name* subsection	For an incoming AOS Transfer Frame with VCID *`v`*, insert it into the message queue with priority *`p`*

Table 3.14–1: Virtual Channel Multiplexing Function Configuration File Settings

Operation	Effect
`modAOS_VC_Mux.setDefaultPriority(c,s,p)`	Set the default priority for incoming AOS Transfer Frames to integer `p` for segment `s` of channel `c`
`modAOS_VC_Mux.getDefaultPriority(c,s)`	Retrieve the default priority for incoming AOS Transfer Frames for segment `s` of channel `c`
`modAOS_VC_Mux.setPriority(c,s,m,p)`	Set the priority for incoming AOS Transfer Frames with VCID `m` to `p`, for segment `s` of channel `c`
`modAOS_VC_Mux.getPriority(c,s,m)`	Get the priority for incoming AOS Transfer Frames with VCID `m` in segment `s` of channel `c`

Table 3.14–2: XML-RPC Directives for Virtual Channel Multiplexing Function Operations

3.15 Virtual Channel Reception Function (modAOS_VC_Rcv)

This module accepts AOS Transfer Frames and extracts the payload from the Data Field. The segment must be explicitly configured to recognize the data structure in that field. That type can be one of either a Multiplexing PDU (M_PDU), Bitstream PDU (B_PDU), Virtual Channel Access PDU (VCA_PDU), or Idle data. The Data Field buffer is put into the appropriate type of wrapper and sent via the primary output.

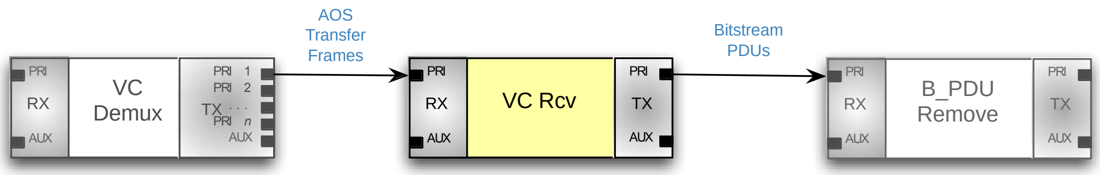

Figure 3.15-1: Virtual Channel Reception Function Architecture Sample

Figure 3.15-1 depicts a sample layout of a configured Virtual Channel Reception segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments” and chapter 3.1, “AOS Space Data Link Common Management Settings”. Figure 3.15-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {

    AOS_PhysicalChannel:
    {
      frameSize = 250;
      useHeaderErrorControl = false;
      useFrameErrorControl = true;
      useInsertZone = false;
      insertZoneSize = 0;
      idlePattern = [ 0xAA, 0xBB, 0xCC ];
    };
    …
    bpduRemove: {
      dllName = "modAOS_B_PDU_Remove";
      primaryOutput = [ "Next Segment A", "PrimaryInput" ];
      immediateStart = true;
    };

    vcRcv: { # segment name
      dllName = "modAOS_VC_Rcv";
      primaryOutput = [ "bpduRemove", "PrimaryInput" ];
      auxOutput = [ "clcwRcv", "PrimaryInput" ];
      SCID = 0xA1;
      VCID = 0x11;
      useOperationalControl = true;
      useVCFrameCycle = true;
      …
      immediateStart = true;
    };

    vcDemux: { # segment name
      dllName = "modAOS_VC_Demux";
      primaryOutputs = ( ( "vcRcv", "PrimaryInput", 0x11),
                           …
                         );
      SCID = 0xA1;
      …
      immediateStart = true;
      …
    };
    …
  };
```

Figure 3.15-2: Virtual Channel Reception Function Configuration File Excerpt

3.16 Operational Control Field Insertion Service (modAOS_OCF_Insert)

The OCF Insertion Service accepts AOS Transfer Frame wrappers on its primary input queue. On its auxiliary input queue it receives four-octet data units to insert into the Operational Control Field of the frames it receives. As frames arrive, the buffers of any units waiting in the auxiliary input queue are copied into the appropriate location of each frame, and then the altered frames are sent via the primary output queue. By placing this segment immediately after a Virtual Channel Generation function, it will provide a Virtual Channel Operational Control Field Service; by placing it after a Master Channel Generation function, it will provide a Master Channel Operational Control Field Service.

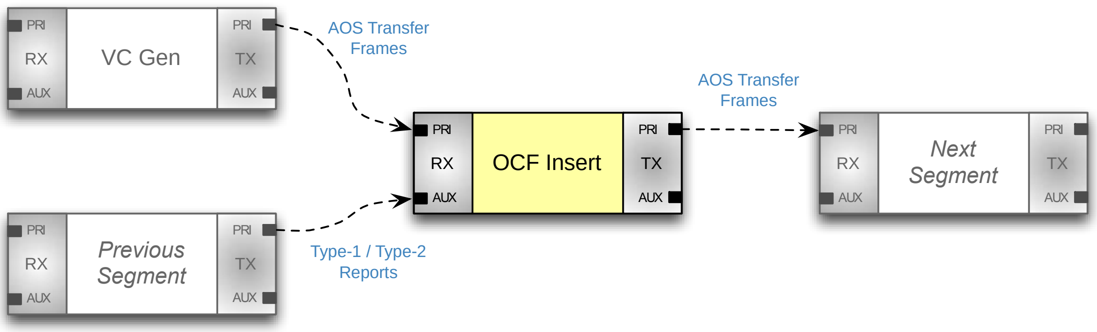

Figure 3.16-1: AOS Operational Control Field Insertion Service Architecture Sample

Figure 3.16-1 depicts a sample layout of a configured Operational Control Field Insertion Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments" and chapter 3.1, "AOS Space Data Link Common Management Settings". Figure 3.16-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanAOS_test:
  {
    …
    AOS_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    vcMux: {
      dllName = "modAOS_VC_Mux";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      …
    };

    ocfInsert: {
      dllName = "modAOS_OCF_Insert";
      primaryOutput = [ "vcMux", "PrimaryInput" ];
      useOperationalControl = true;
    };

    prvSeg: {
      primaryOutput = [ "ocfInsert", "AuxInput" ];
      …
    };

    vcGen: {
      dllName = "modAOS_VC_Gen";
      primaryOutput = [ "ocfInsert", "PrimaryInput" ];
      SCID = 5;
      VCID = 3;
```

```
        useOperationalControl = true;
        …
      };
      …
    };
  };
```

Figure 3.16-2: AOS Operational Control Field Insertion Service Configuration File Example

3.17 Operational Control Field Extraction Service (modAOS_OCF_Extract)

The OCF Extraction Service accepts AOS Transfer Frame wrappers on its primary input queue. If the Operational Control Field is in use, its OCF is wrapped separately and sent via the auxiliary output. The complete, unmodified frame is then send via the primary output.

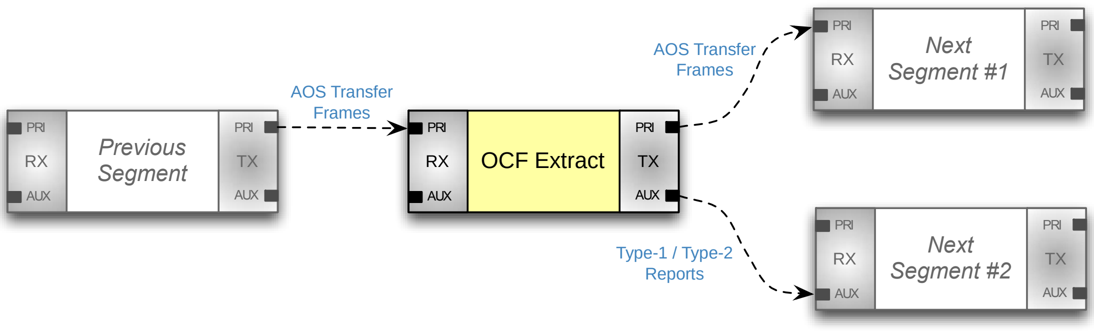

Figure 3.17-1: AOS Operational Control Field Extraction Service Architecture Sample

Figure 3.17-1 depicts a sample layout of a configured Operational Control Field Extraction Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments” and chapter 3.1, “AOS Space Data Link Common Management Settings”. Figure 3.17-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanAOS_test:
  {
    …
    AOS_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    ocfExtract: {
      dllName = "modAOS_OCF_Extract";
      primaryOutput = [ "Next Segment #1", "PrimaryInput" ];
      auxOutput = [ "Next Segment #2", "PrimaryInput" ];
      useOperationalControl = true;
    };

    prvSeg: {
      primaryOutput = [ "ocfExtract", "PrimaryInput" ];
      …
    };
    …
  };
};
```

Figure 3.17-2: AOS Operational Control Field Extraction Service Configuration File Example

3.18 IN_SDU Insertion Service (modAOS_IN_SDU_Insert)

The IN_SDU Insertion Service accepts AOS Transfer Frame wrappers on its primary input queue. On its auxiliary input queue it receives fixed length Insert Service Data Units (IN_SDU) to insert into the Insert Zone of the frames it receives. As frames arrive, the buffers of any units waiting in the auxiliary input queue are copied into the appropriate location of each frame, and then the altered frames are sent via the primary output queue. Typically, this segment is placed immediately before the All Frames Generation service so that the Frame Error Control Field can be properly encoded.

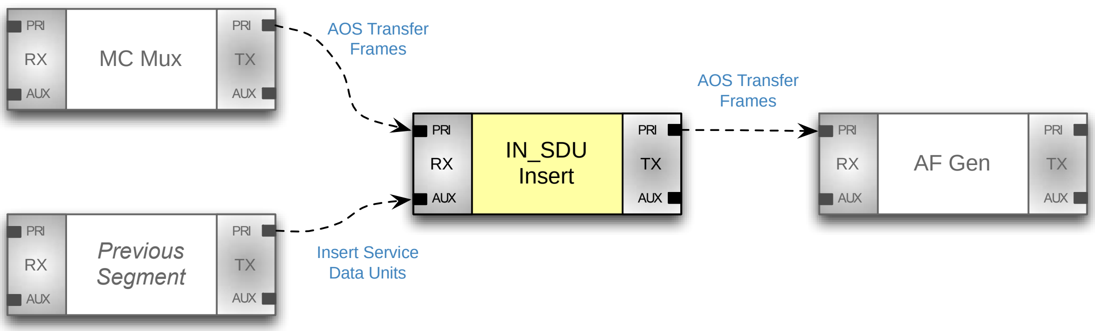

Figure 3.18-1: AOS IN_SDU Insertion Service Architecture Sample

Figure 3.18-1 depicts a sample layout of a configured Operational Control Field Insertion Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments” and chapter 3.1, “AOS Space Data Link Common Management Settings”. Figure 3.18-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanAOS_test:
  {
    …
    AOS_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    mcMux: {
      dllName = "modAOS_MC_Mux";
      primaryOutput = [ "inSduInsert", "PrimaryInput" ];
      …
    };

    prvSeg: {
      primaryOutput = [ "inSduInsert", "AuxInput" ];
      …
    };

    inSduInsert: {
      dllName = "modAOS_IN_SDU_Insert";
      primaryOutput = [ "afGen", "PrimaryInput" ];
    };


    afGen: {
      dllName = "modAOS_AF_Gen";
      …
    };
    …
  };
};
```

Figure 3.18-2: AOS IN_SDU Insertion Service Configuration File Example

3.19 IN_SDU Extraction Service (modAOS_IN_SDU_Extract)

The IN_SDU Extraction Service accepts AOS Transfer Frame wrappers on its primary input queue. If the Insert Zone is in use, the Insert Service Data Unit (IN_SDU) payload is wrapped separately and sent via the auxiliary output. The complete, unmodified frame is then send via the primary output.

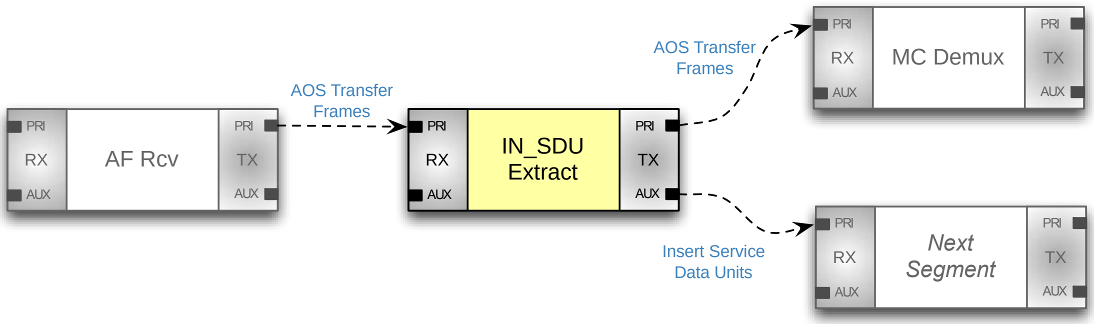

Figure 3.19-1: AOS IN_SDU Extraction Service Architecture Sample

Figure 3.19-1 depicts a sample layout of a configured IN_SDU Extraction Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments" and chapter 3.1, "AOS Space Data Link Common Management Settings". Figure 3.19-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanAOS_test:
  {
    …
    AOS_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };

    afRcv: {
      dllName = "modAOS_AF_Rcv";
      primaryOutput = [ "inSduExtract", "PrimaryInput" ];
    };

    inSduExtract: {
      dllName = "modAOS_IN_SDU_Extract";
      primaryOutput = [ "mcDemux", "PrimaryInput" ];
      auxOutput = [ "nextSegment", "PrimaryInput" ];
    };

    mcDemux: {
      dllName = "modAOS_MC_Demux";
      …
    };

    nextSegment: {
      …
    };
    …
  };
};
```

Figure 3.19-2: AOS IN_SDU Extraction Service Configuration File Example

3.20 Macros Interface (modAOS_Macros)

Assembling a complete AOS Physical Channel from scratch can be difficult without detailed awareness of the protocol. The AOS Macros module provides a pair of methods that create a template of either a forward or return channel, using only a small number of settings. Table 3.20–1 describes these functions.

Operation	Effect
`modAOS_Macros.newForwardChannel(c,s)`	In pre-existing CE channel `c`, create a new forward AOS Physical Channel, complete with settings, services, and functions specified in XML-RPC struct `s`. The struct may contain any or all of the following members:

`frameLength`	The exact size for all AOS Transfer Frames in the channel in whole octets.
`bitstreamSvcCount`	The number of virtual channels that will provide Bitstream Services.
`packetSvcCount`	The number of virtual channels that will provide Packet Services.
`vcAccessSvcCount`	The number of virtual channels that will provide VC Access Services.
`vcFrameSvcCount`	The number of virtual channels that will consist of VC Frame Services.
`mcFrameSvcCount`	The number of master channels that will consist of MC Frame Services.
`insertZoneLength`	Reserve the specified number of octets in each transfer frame for the Insert Zone. If the Insert Zone is not in use, this should always be set to zero.
`masterChannelCount`	Select the number of master channels to create. Multiplies the number of Packet, Bitstream, VC Access, and VC Frame Services.
`maxErrorsReedSolomon`	If 8 or 16, generate Reed-Solomon parity symbols and append them to each frame.
`useIdleVC`	If true, send Idle Frames with Virtual Channel Identifier value 63.
`useHeaderEC`	If true, enable frame header error control using a Reed-Solomon (10,6) code.
`useFrameEC`	If true, enable error control for the entire frame via a checksum.
`useASM`	If true, attach an sync marker to each frame.
`usePseudoRandomize`	If true, exclusive-OR the entire unit with a predetermined sequence
`minimize`	If true, create only enough channel segments to perform the requested functions. If false (default), leave some functions in (e.g. mux/demux) to accommodate additional services
`segNamePrefix`	A string to prepend to each segment name.
`segNameSuffix`	A string to append to each segment name.

`modAOS_Macros.newReturnChannel(c,s)`	In pre-existing CE channel `c`, create a new return AOS Physical Channel, complete with settings, services, and functions specified in XML-RPC struct `s`. The struct may contain any or all of the following members:

`frameLength`	The exact size for all AOS Transfer Frames in the channel in whole octets.
`bitstreamSvcCount`	The number of virtual channels that will provide Bitstream Services.
`packetSvcCount`	The number of virtual channels that will provide Packet Services.
`insertZoneLength`	Reserve the specified number of octets in each transfer frame for the Insert Zone. If the Insert Zone is not in use, this should always be set to zero.
`masterChannelCount`	Select the number of master channels to create. Multiplies the number of Packet, Bitstream, VC Access, and VC Frame Services.
`maxErrorsReedSolomon`	If 8 or 16, generate Reed-Solomon parity symbols and append them to each frame.
`useHeaderEC`	If true, enable frame header error control using a Reed-Solomon (10,6) code.
`useFrameEC`	If true, enable error control for the entire frame via a checksum.
`useASM`	If true, attach an sync marker to each frame.
`usePseudoRandomize`	If true, exclusive-OR the entire unit with a predetermined sequence
`minimize`	If true, create only enough channel segments to perform the requested functions. If false (default), leave some functions in (e.g. mux/demux) to accommodate additional services
`segNamePrefix`	A string to prepend to each segment name.
`segNameSuffix`	A string to append to each segment name.

Table 3.20–1: XML-RPC Directives for AOS Macros

4 CCSDS TM Space Data Link Protocol-Related Modules

4.1 TM Space Data Link Common Management Settings

4.1.1 TM Physical Channel

These settings are common to all modules that are concerned with the TM Transfer Frame structure as governed by the Physical Channel. These include any module with an “AF” (All Frames), “MC” (Master Channel), or “VC” (Virtual Channel) in its name. Most settings appear in the `TM_PhysicalChannel` subsection of the configuration file, which the aforementioned modules share for these options.

Entity	Container	Description
`frameSize = `*`n`*`;`	`TM_PhysicalChannel` subsection	Set the exact size for all TM transfer frames in the channel to *`n`* octets
`useFrameErrorControl = true | false;`	`TM_PhysicalChannel` subsection	If `true`, enable frame error control via a checksum
`idlePattern = [ `*`h1, h2, h3`*`, … ];`	`TM_PhysicalChannel` subsection	For services that are periodic, the array of integers *`h1, h2, h3`*, … is used as a pattern to fill the data field in idle units
`dropBadFrames = true | false;`	*segment_name* subsection	If `true` (default), discard malformed frames (or frames that fail ECC in All Frames Reception) in this segment. It is recommended to only use the `false` setting in specific situations such as with frame services, otherwise traffic loss/corruption may occur

Table 4.1–1: Common TM Physical Channel Configuration File Settings

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 250;
      useFrameErrorControl = true;
      idlePattern = [ 0xA1, 0xB2, 0xC3 ];
    };

    afGen: {
      …
    };
    …
  };
};
```

Figure 4.1-1: Common TM Physical Channel Settings - Configuration File Excerpt

Operation	**Effect**
`modName.setFrameSize(c,s,f)`	Set the exact TM transfer frame size to `f` octets for segment `s` in channel `c`. All other segments in the channel using this setting will be instantly affected
`modName.getFrameSize(c,s)`	Retrieve the TM transfer frame size in octets from segment `s` of channel `c`
`modName.setUseFrameEC(c,s,f)`	Enable or disable frame error control in segment `s` in channel `c` depending on the value of Boolean `f`. All other segments in the channel using this setting will be instantly affected
`modName.getUseFrameEC(c,s)`	Determine whether frame error control is enable for segment `s` in channel `c`
`modName.setValidFrameCount(c,s,f)`	Set the valid frame count to `f` for segment `s` in channel `c`. Probably only useful when initializing a segment
`modName.getValidFrameCount(c,s)`	Return the valid frame count for segment `s` in channel `c`
`modName.setBadFrameCRCCount(c,s,r)`	Set the bad frame checksum count to `r` for segment `s` in channel `c`. Probably only useful when initializing a segment
`modName.getBadFrameCRCCount(c,s)`	Return the bad frame checksum count for segment `s` in channel `c`

modName. setBadHeaderCount(c,s,h)	Set the bad header count to `h` for segment `s` in channel `c`. Probably only useful when initializing a segment
modName. getBadHeaderCount(c,s)	Return the bad header tally for segment `s` in channel `c`. Reasons for encountering a "bad" header include a bad version number or a structural problem.
modName. setBadLengthCount(c,s,l)	Set the bad length count to `h` for segment `s` in channel `c`. Possibly useful when initializing a segment
modName. getBadLengthCount(c,s)	Return the bad frame length tally for segment `s` in channel `c`. Indicated the total number of frames encountered that were not exactly `frameSize` in length
modName. setIdlePattern(c,s,p)	Set the idle pattern for segment `s` in channel `c` to integer array `p`. The idle pattern is used to fill the data field of idle frames. All other segments in the channel using this setting will be instantly affected
modName. getIdlePattern(c,s)	Returns an integer array containing the idle pattern for segment `s` in channel `c`
modName. setDropBadFrames(c,s,d)	Enable or disable the discarding of malformed frames in segment `s` in channel `c` depending on the value of Boolean `d`
modName. getDropBadFrames (c,s)	Determine whether malformed frames are being discarded by segment `s` in channel `c`

Table 4.1–2: Common XML-RPC Directives for TM Physical Channel Operations

4.1.2 TM Master Channel

These settings are common to all modules that are concerned with the TM Transfer Frame structure as governed by a Master Channel. These include any module with an "MC" (Master Channel), or "VC" (Virtual Channel) in its name.

Entity	Container	Description
`SCID = x;`	*segment_name* subsection	Set the Spacecraft Identifier for the Master Channel to *x*.
`useOperationalControl = true | false;`	*segment_name* subsection	If `true`, use the Operational Control Field in this Master or Virtual

		Channel
`useFSH = true | false;`	*segment_name* subsection	If true, use the Frame Secondary Header in this Master or Virtual Channel
`fshSize = `*`s`*`;`	*segment_name* subsection	If s is an integer other than 0, fix the size of the FSH to it. If s is zero than the FSH will be variable in length. Only used if `useFSH` is `true`
`syncFlag = true | false;`	*segment_name* subsection	If `true`, the synchronization flag will be set (or expected to be set in the case of receiving services) for all TM Transfer Frames. If false, the flag will be unset (or expected to be unset)

Table 4.1–3: Common TM Master Channel Configuration File Settings

```
Channels: {
  …
  channel1: {
    …
    mcMux: {
      dllName = "modTM_MC_Mux";
      primaryOutput = [ "afGen", "PrimaryInput" ];
      immediateStart = true;
      SCID = 0x02;
      useFSH = true;
      fshSize = 32;
      useOperationalControl = false;
      syncFlag = true;
    };
    …
  };
};
```

Figure 4.1-2: Common TM Master Channel Settings - Configuration File Excerpt

Operation	**Effect**
`modName``.setSCID(c,s,i)`	Set the Spacecraft Identifier to integer `i` for segment `s` in channel `c`
`modName``.getSCID(c,s)`	Retrieve the Spacecraft Identifier as an integer from segment `s` in channel `c`
`modName``.setOperationalControlFlag(c,s,u)`	If `u` is `true`, enable the OCF for segment `s` in channel `c`
`modName``.getOperationalControlFlag(c,s)`	Determine if the OCF is being used in segment `s` of channel `c`

`modName.setUseFSH(c,s,f)`	If `f` is `true`, enable the Frame Secondary Header for segment `s` in channel `c`
`modName.getUseFSH(c,s)`	Return the usage setting of the Frame Secondary Header for segment `s` in channel `c`
`modName.setFSHSize(c,s,z)`	Make the Frame Secondary Header have a size of `z` octets in segment `s` of channel `c`
`modName.getFSHSize(c,s)`	Return the length setting of the Frame Secondary Header for segment `s` in channel `c`
`modName.setSyncFlag(c,s,u)`	If `u` is `true`, set the synchronization flag for segment `s` in channel `c`
`modName.getSyncFlag(c,s)`	Determine if the synchronization flag is set in segment `s` of channel `c`
`modName.setBadMCIDCount(c,s,b)`	Set the bad Master Channel Spacecraft Identifier count to `b` for segment `s` in channel `c`. Possibly useful when initializing a segment
`modName.getBadMCIDCount(c,s)`	Return the bad Master Channel Spacecraft Identifier count for segment `s` in channel `c`. Indicated the total number of frames encountered that did not have a Spacecraft Identifier matching the assigned `SCID`

Table 4.1–4: Common XML-RPC Directives for TM Master Channel Operations

4.1.3 TM Virtual Channel

These settings are common to all modules that are concerned with the TM Transfer Frame structure as governed by a Virtual Channel. These include any module with an "VC" (Virtual Channel) in its name.

Entity	**Container**	**Description**
`VCID = y;`	*segment_name* subsection	Set the Virtual Channel Identifier for this segment to an integer *y*

Table 4.1–5: Common TM Virtual Channel Configuration File Settings

```
Channels:
{
  …
  chanX:
  {
    …
    vcGen: {
      dllName = "modAOS_VC_Gen";
      highWaterMark = 16777216;
      lowWaterMark = 12582912;
      primaryOutput = [ "afGen", "PrimaryInput" ];
      immediateStart = true;
      SCID = 0x2;
      VCID = 0x1;
      useOperationalControl = false;
      useFSH = false;
      fshSize = 0;
    };
    …
  };
};
```

Figure 4.1-3: Common TM Virtual Channel Settings - Configuration File Excerpt

Operation	**Effect**
`*modName*. setVCID(c,s,v)`	Set the Virtual Channel Identifier to `v`, an integer for segment `s` in channel `c`
`*modName*. getVCID(c,s)`	Get the integer value for the Virtual Channel Identifier of segment `s` in channel `c`
`*modName*. setBadVCIDCount(c,s,b)`	Set the tally of bad VC Identifiers encountered to integer `b`, for segment `s` in channel `c`. Possibly useful during initialization
`*modName*. getBadVCIDCount(c,s)`	Return the tally of bad VC Identifiers encountered in segment `s` of channel `c`

Table 4.1–6: Common XML-RPC Directives for TM Virtual Channel Operations

4.2 All Frames Generation (modTM_AF_Gen)

The All Frames Generation function accepts TM Transfer Frame wrappers on its primary input queue. The All Frames Generation function itself is *not* periodic and does not generate its own idle frames. Error code generation is performed on the Transfer Frame, if enabled.

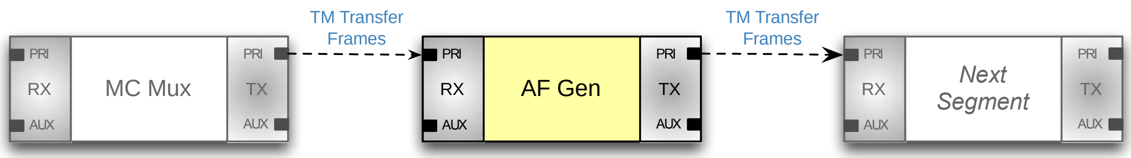

Figure 4.2-1: TM All Frames Generation Architecture Sample

Figure 4.2-1 depicts a sample layout of a configured All Frames Generation segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, "Modular Segments" and 4.1.1, "AOS Physical Channel". Figure 4.2-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    mcMux: {
      dllName = "modTM_MC_Mux";
      primaryOutput = [ "afGen", "PrimaryInput" ];
      …
    };


    afGen: {
      dllName = "modTM_AF_Gen";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      immediateStart = true;
    };
    …
  };
};
```

Figure 4.2-2: TM All Frames Generation Configuration File Example

4.3 All Frames Reception (modTM_AF_Rcv)

The All Frames Reception function accepts TM Transfer Frame wrappers into its primary input queue. It performs error control decoding if enabled: if the code in Frame Error Control Field is invalid, the frame is dropped. If the frame is valid, it is passed on to the segment configured as the modTM_AF_Rcv segment's primary output.

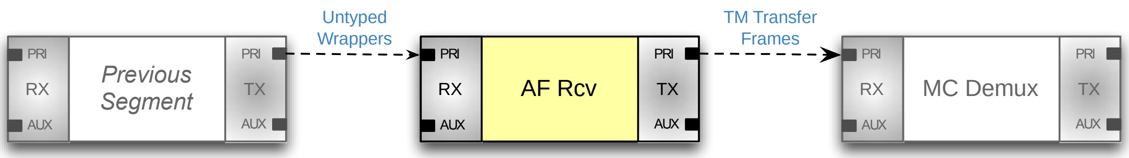

Figure 4.3-1: TM All Frames Reception Architecture Sample

Figure 4.3-1 depicts a sample layout of a configured All Frames Reception segment. This module derives all of its configuration file settings and XML-RPC directives from those described in sections 2.4, “Modular Segments” and 4.1.1, “AOS Physical Channel”. Figure 5.3-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 1024;
      useFrameErrorControl = true;
      idlePattern = [ 0xA1, 0xB2, 0xC3 ];
    };
    …
    afRcv: {
      dllName = "modTM_AF_Rcv";
      primaryOutput = [ "mcDemux", "PrimaryInput" ];
    };
    …
  };
};
```

Figure 4.3-2: TM All Frames Reception Configuration File Example

4.4 Master Channel Frame Service (modTM_MC_Frame)

The Master Channel Frame Service accepts TM Transfer Frame wrappers on its primary input queue. The length and MCID of the frame are tested, and if there is a mismatch, it is dropped. Otherwise the unmodified frame is forwarded to the next target.

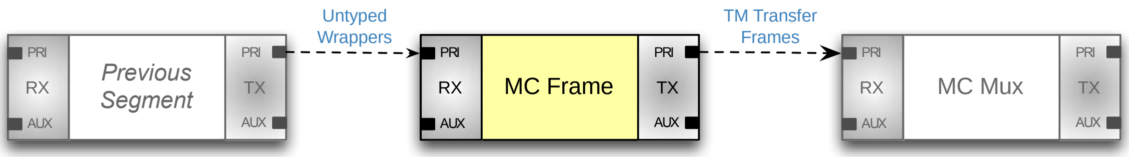

Figure 4.4-1: TM Master Channel Frame Service Architecture Sample

Figure 4.4-1 depicts a sample layout of a configured Master Channel Frame Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM

Physical Channel", and 4.1.2, "TM Master Channel". Figure 4.4-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    mcMux: {
      dllName = "modTM_MC_Mux";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      …
    };

    mcFrame: {
      dllName = "modTM_MC_Frame";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      SCID = 0x1;
      useFSH = true;
      fshSize = 24;
      useOperationalControl = false;
    };
    …
  };
};
```

Figure 4.4-2: TM Master Channel Frame Service Configuration File Example

4.5 Master Channel Generation Service (modTM_MC_Gen)

The Master Channel Generation Service accepts TM Transfer Frame wrappers on its primary input queue. The length and MCID of the frame are tested, and if there is a mismatch, it is dropped. Otherwise the Master Channel Frame Count is incremented (or reset to 0 if it is already 255) and the frame is forwarded to the next target.

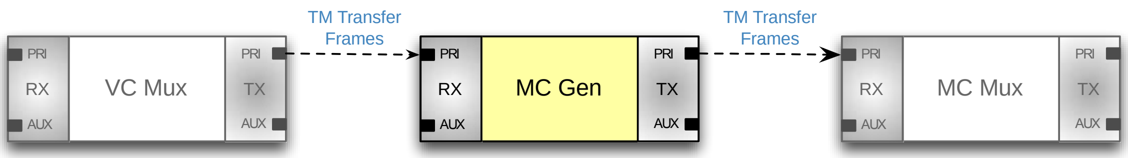

Figure 4.5-1: TM Master Channel Generation Service Architecture Sample

Figure 4.5-1 depicts a sample layout of a configured Master Channel Generation Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, "Modular Segments", 4.1.1, "TM Physical Channel", and 4.1.2, "TM Master Channel". Figure 4.5-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    mcMux: {
      dllName = "modTM_MC_Mux";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      …
    };

    mcGen: {
      dllName = "modTM_MC_Gen";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      SCID = 0x1;
      useFSH = true;
      fshSize = 24;
      useOperationalControl = false;
    };
    …
  };
};
```

Figure 4.5-2: TM Master Channel Generation Service Configuration File Example

4.6 Master Channel Reception Service (modTM_MC_Rcv)

The Master Channel Generation Service accepts TM Transfer Frame wrappers on its primary input queue. The length and MCID of the frame are tested, and if there is a mismatch, it is dropped. The Master Channel Frame Count of the incoming frame is compared to a counter, and a warning is generated if they are not identical. Idle Data is discarded.

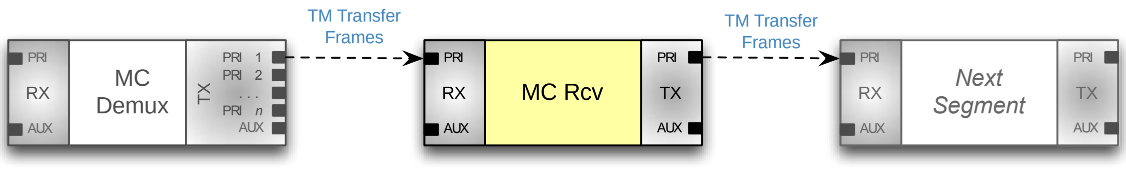

Figure 4.6-1: TM Master Channel Reception Service Architecture Sample

Figure 4.6-1 depicts a sample layout of a configured Master Channel Reception Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM Physical Channel”, and 4.1.2, “TM Master Channel”. Figure 4.6-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    mcRcv1: {
      dllName = "modTM_MC_Rcv";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      SCID = 0x1;
      useFSH = true;
      fshSize = 24;
      useOperationalControl = false;
    };

    mcDemux: {
      dllName = "modTM_MC_Demux";
      primaryOutputs = ( ( "mcRcv1", "PrimaryInput", 1 ), ... );
      …
    };
    …
  };
};
```

Figure 4.6-2: TM Master Channel Reception Service Configuration File Example

4.7 Master Channel Multiplexing Function (modTM_MC_Mux)

This module has multiple primary input links from which it accepts TM Transfer Frames with different Master Channel Identifiers (MCIDs). The frames are prioritized in the queue according to their MCID. By default, all have the same priority. If those are modified, frames with the highest-priority MCID are sent until there are none remaining; then, frames with the next highest priority are sent until there are none remaining, and so on. If high priority traffic does not abate, low priority traffic will never be sent.

This service is optionally periodic – it can be configured to send frames at a constant rate and will send Idle Frames when there is no incoming data. The VCID and SCID settings in the configuration file are used to specify which GVCID value the Idle Frames will have (and that is the only use of the SCID/VCID settings in this particular module).

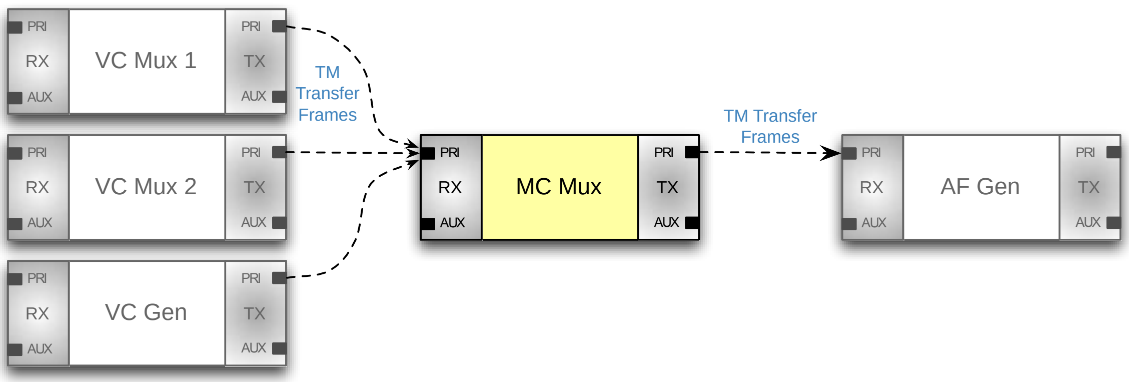

Figure 4.7-1: Master Channel Multiplexing Architecture Sample

Figure 4.7-1 depicts a sample layout of a configured Master Channel Multiplexing segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM Physical Channel”, 4.1.2, “TM Master Channel”, and 4.1.3, “TM Virtual Channel”. Additional settings are described later in this section. Figure 4.7-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    afGen: { # segment name
      dllName = "modTM_AF_Gen";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      …
    };

    mcMux: { # segment name
      dllName = "modTM_MC_Mux";
      primaryOutput = [ "afGen", "PrimaryInput" ];
      defaultPriority = 1000;
      channelID_Priorities = ( [ 0xA1, 1010 ], [ 0xB2, 900 ] );
      SCID = 10; # only used for Idle Frames
      VCID = 7;  # only used for Idle Frames
    };

    vcMux1: { # segment name
      dllName = "modTM_VC_Mux";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      SCID = 1;
      …
    };

    vcMux2: { # segment name
      dllName = "modTM_VC_Mux";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      SCID = 1;
    };

    vcGen: { # segment name
      dllName = "modTM_VC_Gen";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
```

```
        SCID = 1;
        VCID = 1;
      };
      …
    };
  };
```

Figure 4.7-2: Master Channel Multiplexing Function Configuration File Excerpt

Entity	Container	Description
`defaultPriority = p;`	*segment_name* subsection	Incoming TM Transfer Frames are inserted into the message queue with integer priority `p` if they don't have their own priority specified
`channelID_Priorities = ( [ m, p ], … );`	*segment_name* subsection	For an incoming TM Transfer Frame with SCID *m*, insert it into the message queue with priority *p*

Table 4.7–1: Master Channel Multiplexing Function Configuration File Settings

Operation	Effect
`modTM_MC_Mux.setDefaultPriority(c,s,p)`	Set the default priority for incoming TM Transfer Frames to integer `p` for segment `s` of channel `c`
`modTM_MC_Mux.getDefaultPriority(c,s)`	Retrieve the default priority for incoming TM Transfer Frames for segment `s` of channel `c`
`modTM_MC_Mux.setPriority(c,s,m,p)`	Set the priority for incoming TM Transfer Frames with Spacecraft ID `m` to `p`, for segment `s` of channel `c`
`modTM_MC_Mux.getPriority(c,s,m)`	Get the priority for incoming TM Transfer Frames with Spacecraft ID `m` in segment `s` of channel `c`

Table 4.7–2: XML-RPC Directives for Master Channel Multiplexing Function Operations

4.8 Master Channel De-multiplexing Function (modTM_MC_Demux)

This module accepts TM Transfer Frame wrappers on its primary input, separates them based on their Master Channel Identifier (MCID), and sends them out an associated primary output. There can be as many primary outputs as there are possible master channels in one physical channel, and each is associated with a single MCID.

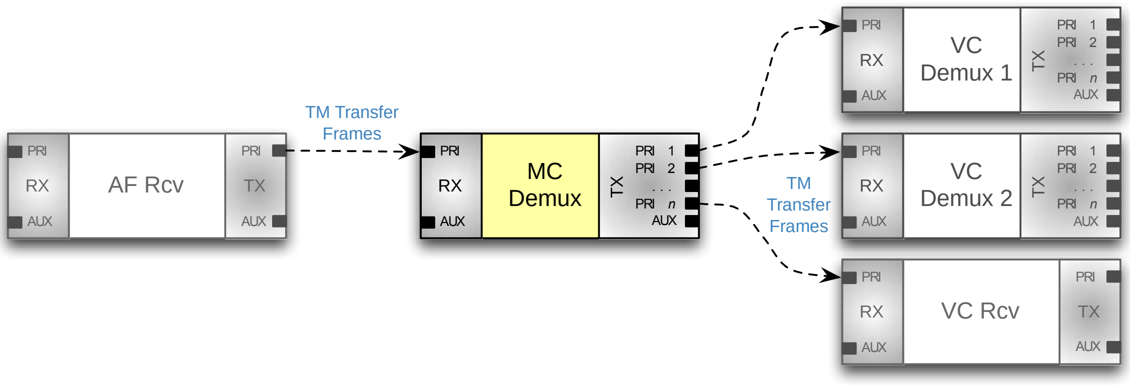

Figure 4.8-1: Master Channel De-multiplexing Sample Architecture

Figure 4.8-1 depicts a sample layout of a configured Master Channel De-multiplex segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapters 2.4, "Modular Segments", 4.1.1, "TM Physical Channel", 4.1.2, "TM Master Channel", and 4.1.3, "TM Virtual Channel". The exception is the `primaryOutputs` setting. Similar to `primaryOutput`, found in most modules, `primaryOutputs` also specifies which segments the main flow of traffic is directed to, and on which input. However, rather than being a one-dimensional array, this setting is a list of lists. Each sub list contains the next segment name, its input type, and a third parameter – the Spacecraft Identifier (SCID) of that master channel (not the MCID, which is computed automatically from the SCID and TVFN).

Figure 4.8-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {

    TM_PhysicalChannel:
    {
      frameSize = 256;
      useFrameErrorControl = true;
      idlePattern = [ 0xA1, 0xB2, 0xC3 ];
    };
    …
    vcDemux1: { # segment name
      dllName = "modTM_VC_Demux";
      primaryOutputs = ( ( "Next Segment E", "PrimaryInput", 0x1),
                         ( "Next Segment D", "PrimaryInput", 0x2),
                         …
                       );
      SCID = 0xA1;
      …
    };

    vcDemux2: { # segment name
      dllName = "modTM_VC_Demux";
      primaryOutputs = ( ( "Next Segment C", "PrimaryInput", 0x1),
                         ( "Next Segment B", "PrimaryInput", 0x2),
                         …
                       );
```

```
        SCID = 0xB2;
        …
      };

      vcRcv: { # segment name
        dllName = "modTM_VC_Rcv";
        primaryOutput = [ "Next Segment A", "PrimaryInput" ];
        SCID = 0xD4;
        VCID = 0x1;
        …
      };

      mcDemux: { # segment name
        dllName = "modTM_MC_Demux";
        primaryOutputs = ( ( "vcDemux1", "PrimaryInput", 0xA1),
                           ( "vcDemux2", "PrimaryInput", 0xB2),
                           …,
                           ( "vcRcv", "PrimaryInput", 0xD4),
                         );
        …
      };

      afRcv: { # segment name
        dllName = "modTM_AF_Rcv";
        primaryOutput = [ "mcDemux", "PrimaryInput" ];
        …
      };
      …
    };
};
```

Figure 4.8-2: Master Channel De-multiplexing Configuration File Excerpt

Entity	Container	Description
`primaryOutputs = ( ( "s", "PrimaryInput" | "AuxInput", i ), … );`	*segment_name* subsection	TM Transfer Frames received with Spacecraft Identifier `i` are sent on to either the Primary or Auxiliary input of segment `s`. Segment `s` must already exist in the channel

Table 4.8–1: Master Channel De-multiplexing Configuration File Settings

Operation	Effect
`modTM_MC_Demux.connectOutput(c,s1,d,s2,o,i)`	In channel `c`, connect an output of segment `s1` to an input of segment `s2`. If the output type of `s1`, specified by `o`, is "`Primary`", use the integer `d` to specify the Spacecraft Identifier for TM Transfer Frames to be sent over that link; if `o` is "`Auxiliary`", `d` is ignored. The input type (either "`Primary`" or "`Auxiliary`") of `s2` is specified with parameter `i`

Table 4.8–2: XML-RPC Directives for Master Channel De-multiplexing Operations

4.9 Virtual Channel Frame Service (modTM_VC_Frame)

The Virtual Channel Frame Service accepts TM Transfer Frame wrappers on its primary input queue. The length and GVCID of the frame are tested, and if there is a mismatch, it is dropped. Otherwise the unmodified frame is forwarded to the next target.

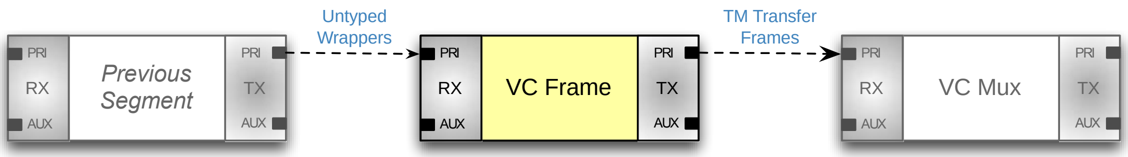

Figure 4.9-1: TM Virtual Channel Frame Service Architecture Sample

Figure 4.9-1 depicts a sample layout of a configured Virtual Channel Frame Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM Physical Channel”, 4.1.2, “TM Master Channel”, and 4.1.3, “TM Virtual Channel”. Figure 4.9-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    vcMux: {
      dllName = "modTM_VC_Mux";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      …
    };

    vcFrame: {
      dllName = "modTM_VC_Frame";
      primaryOutput = [ "vcMux", "PrimaryInput" ];
      SCID = 0x1;
      VCID = 0x2;
      useFSH = true;
      fshSize = 24;
      useOperationalControl = false;
    };
    …
  };
};
```

Figure 4.9-2: TM Virtual Channel Frame Service Configuration File Example

4.10 Virtual Channel Generation – Access Service (modTM_VC_Gen_Access)

The Virtual Channel Generation – Access Service accepts wrappers on its primary input queue containing a buffer filled with a VCA_SDU (Virtual Channel Access Service Data Unit). Each VCA_SDU must fit exactly into the configured size of the TM Transfer Frame Data Field. The synchronization flag of the frame is set before being sent via the primary output path.

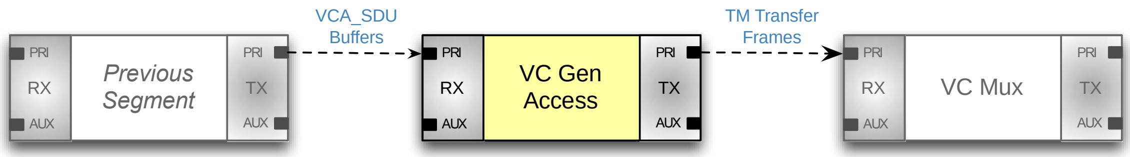

Figure 4.10-1: TM Virtual Channel Generation – Access Service Architecture Sample

Figure 4.10-1 depicts a sample layout of a configured Virtual Channel Generation – Access Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, "Modular Segments", 4.1.1, "TM Physical Channel", 4.1.2, "TM Master Channel", and 4.1.3, "TM Virtual Channel". Figure 4.10-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    vcMux: {
      dllName = "modTM_VC_Mux";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      …
    };

    vcaGen: {
      dllName = "modTM_VC_Gen_Access";
      primaryOutput = [ "vcMux", "PrimaryInput" ];
    };
    …
  };
};
```

Figure 4.10-2: TM Virtual Channel Generation – Access Service Configuration File Example

4.11 Virtual Channel Reception – Access Service (modTM_VC_Rcv_Access)

The Virtual Channel Reception – Access Service accepts TM Transfer Frame wrappers on its primary input queue. It extracts a VCA_SDU (Virtual Channel Access

Service Data Unit) from each frame that is exactly the size of the TM Transfer Frame Data Field, and sends it via its primary output queue. However, if there is a problem with the frame, such as a size mismatch or the synchronization flag being unset, the data is dropped.

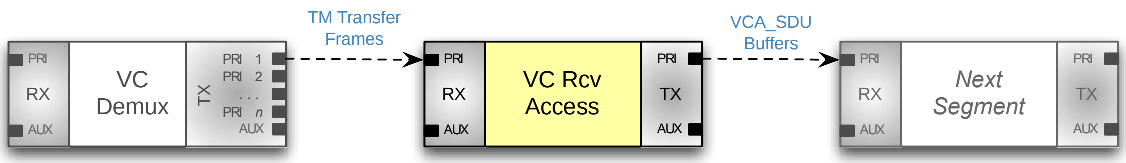

Figure 4.11-1: TM Virtual Channel Generation – Access Service Architecture Sample

Figure 4.11-1 depicts a sample layout of a configured Virtual Channel Reception – Access Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, "Modular Segments", 4.1.1, "TM Physical Channel", 4.1.2, "TM Master Channel", and 4.1.3, "TM Virtual Channel". Figure 4.11-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …

    vcaRcv: {
      dllName = "modTM_VC_Rcv_Access";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
    };

    vcDemux: {
      dllName = "modTM_VC_Demux";
      primaryOutput = [ "vcaRcv", "PrimaryInput" ];
      …
    };
    …
  };
};
```

Figure 4.11-2: TM Virtual Channel Reception – Access Service Configuration File Example

4.12 Virtual Channel Generation – Packet Service (modTM_VC_Gen_Packet)

This service accepts variable-length Encapsulation Packet or Space Packet wrappers and inserts them into TM Transfer Frames. A single large Packet may either be split across multiple frames, or, several small Packets may be inserted into a single Data Field. If there is empty space remaining in the Data Field and another Packet is waiting in the queue, the front of that Packet will be added to the Data Field of the current frame. The remainder will be put into the next frame. Finally, TM Transfer Frame wrappers are sent via the primary output to the next segment, for example the primary input of a Virtual Channel Multiplexing segment.

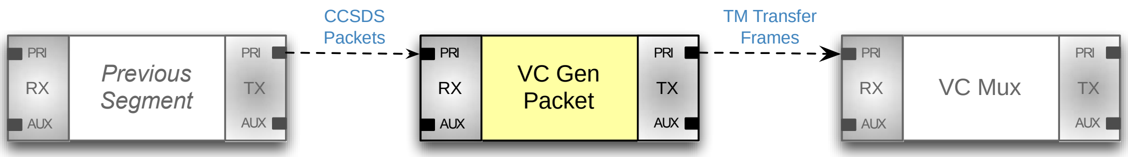

Figure 4.12-1: Virtual Channel Generation – Packet Service Architecture Sample

Figure 4.12-1 depicts a sample layout of a configured Virtual Channel Generation - Packet Service segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM Physical Channel”, 4.1.2, “TM Master Channel”, and 4.1.3, “TM Virtual Channel”. Additional settings are described later in this section. Figure 4.12-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    vcMux:
    {
      dllName = "modTM_VC_Mux";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      …
    };

    vcGenPkt:
    {
      dllName = "modTM_VC_Gen_Packet";
      primaryOutput = [ "vcMux", "PrimaryInput" ];
      fillPattern = [ 0xAA, 0xBB, 0xCC ];
      multiPacketDataField = true;
    };

    encapAdd :
    {
      dllName = "modEncapPkt_Add";
      primaryOutput = [ "vcGenPkt", "PrimaryInput" ];
    };
    …
  };
};
```

Figure 4.12-2: Virtual Channel Generation – Packet Service Configuration File Excerpt

Entity	Container	Description
`maxUsecsForNextPacket = s;`	*segment_name* subsection	Set the number of microseconds to wait for a new incoming packet before sending the current one partially empty (i.e. the remainder of the Data Field containing only an Idle Packet instead of data)
`multiPacketDataField= true | false;`	*segment_name* subsection	If `true`, one packet may end in the Data Field and one or more additional packets may be inserted afterward (assuming there is room) in the same unit. If `false`, new packets will only begin only at the first index in the Data Field
`fillPattern = [ o1, o2, ... ]`	*segment_name* subsection	Fill Idle Packets (to pad the Data Field) with the pattern in the specified array

Table 4.12–1: Virtual Channel Generation – Packet Service Configuration File Settings

Operation	Effect
`modTM_VC_Gen_Packet.getWaitForNextPacket(c,s)`	Get the number of microseconds that segment `s` in channel `c` will wait for a new incoming Packet before sending the current one partially empty (i.e. the end of the Data Field containing only an Idle Packet)
`modTM_VC_Gen_Packet.setWaitForNextPacket(c,s,u)`	Set the number of microseconds to `u` that segment `s` in channel `c` will wait for a new incoming Packet before sending the current one partially empty (i.e. the end of the Data Field containing only an Idle Packet instead of data)
`modTM_VC_Gen_Packet.getMultiDataField(c,s)`	Retrieve whether or not segment `s` in channel `c` allows more than one Packet in the Data Field of frames it creates.
`modTM_VC_Gen_Packet.setMultiDataField(c,s,z)`	Modify whether or not segment `s` in channel `c` allows more than one Packet in the Data Field by supplying Boolean value `z`.
`modTM_VC_Gen_Packet.getFillPattern(c,s)`	Get an array containing the Idle Packet fill pattern used in segment `s` of channel `c`
`modTM_VC_Gen_Packet.setFillPattern(c,s,p)`	Set the Idle Packet fill pattern to integer array `p` in segment `s` of channel `c`

Table 4.12–2: XML-RPC Directives for Virtual Channel Generation – Packet Service Operations

4.13 Virtual Channel Reception – Packet Service (modTM_VC_Rcv_Packet)

This service accepts fixed-length TM Transfer Frames, with Data Fields that contain variable-length Encapsulation Packets or Space Packets. A single frame may carry a partial or whole Packet, or multiples, depending on the relative lengths; there may also be an Idle Packet if no incoming traffic was available to fill the remainder of the Data Field.

Packets are extracted from the Data Field buffers and reassembled in their own buffer, then wrapped and sent via the primary output to the next segment, which typically removes the Packet header or processes it in some other fashion.

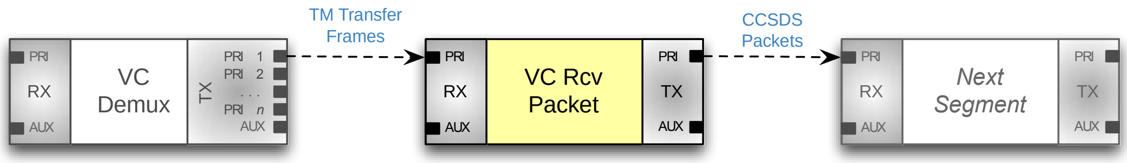

Figure 4.13-1: Virtual Channel Reception – Packet Service Architecture Sample

Figure 4.13-1 depicts a sample layout of a configured Virtual Channel Reception – Packet Service segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM Physical Channel”, 4.1.2, “TM Master Channel”, and 4.1.3, “TM Virtual Channel”.

Important to note is the `primaryOutputs` setting. Similar to `primaryOutput`, found in most modules, `primaryOutputs` also specifies which segments the main flow of traffic is directed to, and on which input. However, instead of a one-dimensional array, this setting is a list of lists. Each sub list contains the next segment name, its input type, and a third parameter – the Packet Version Number (PVN). This implementation of the Packet Service may receive Space Packets (PVN 0) or Encapsulation Packets (PVN 7); SCPS_NP Packets (PVN 1) are not supported at this time.

Additional settings are described later in this section.

Figure 4.13-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    encapRemove :
    {
      dllName = "modEncapPkt_Remove";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
    };

    vcRcvPkt:
    {
      dllName = "modTM_VC_Rcv_Packet";
      primaryOutputs = ( ( "encapRemove", "PrimaryInput", 7 ), … );
      multiDataField = true;
      VCID = 2;
      SCID = 3;
    };

    vcDemux:
    {
      dllName = "modTM_VC_Demux";
      primaryOutputs =  ( ("vcRcvPkt", "PrimaryInput", 2), ... );
      SCID = 3;
      …
    };
    …
  };
};
```

Figure 4.13-2: Virtual Channel Reception – Packet Service Configuration File Excerpt

Explanations of settings with behavior particular to this module are found in Table 4.13–1: Virtual Channel Reception – Packet Service Configuration File Settings and

Table 4.13–2: XML-RPC Directives for Virtual Channel Reception – Packet Service Operations.

Entity	**Container**	**Description**
`supportIPE = true | false;`	*segment_name* subsection	If `true` (default), expect the presence of the Internet Protocol Extension (IPE) shim for Encapsulation Packets, which may be 1 to 8 octets.
`multiDataField = true | false;`	*segment_name* subsection	If `true`, expect that one Packet may end in the Data Field and one or more additional packets may be found afterward in the same unit. If `false`, new packets will only begin only at the first index in the Data Field
`primaryOutputs = ( ( "s", "PrimaryInput" | "AuxInput", v ), … );`	*segment_name* subsection	Packets extracted with Packet Version Number (PVN) v are sent on to either the Primary or Auxiliary input of segment `s`. Segment `s` must already exist in the channel
`allowPacketsAfterFill = true | false;`	*segment_name* subsection	If `false`, stop searching for new packets in the Data Field when a fill value of `0xe0` is encountered. If `true`, continue searching for new packets until a non-empty one is found or the end of the Data Field is reached. *Note: This setting will have no effect if an Idle Packet was used to pad the Data Field, as modTM_VC_Gen_Packet does*

Table 4.13–1: Virtual Channel Reception – Packet Service Configuration File Settings

Operation	Effect
`modTM_VC_Rcv_Packet.setSupportIPE(c,s,i)`	Enable or disable Internet Protocol Extension (IPE) support for Encapsulation Packets in segment `s` of channel `c`, depending on whether `e` is `true` or `false`, respectively. Defaults to `true`
`modTM_VC_Rcv_Packet.getSupportIPE(c,s)`	Fetch the state of IPE support for Encapsulation Packets for segment `s` in channel `c`
`modTM_VC_Rcv_Packet.getMultiDataField(c,s)`	Retrieve whether or not segment `s` in channel `c` can handle more than one Packet in the Data Field of frames it receives
`modTM_VC_Rcv_Packet.setMultiDataField(c,s,z)`	Modify whether or not segment `s` in channel `c` will handle more than one Packet in the Data Field of TM Transfer Frames by supplying Boolean value `z`
`modTM_VC_Rcv_Packet.getAllowPacketsAfterFill(c,s)`	Get whether segment `s` in channel `c` will continue searching for Packets in a single Data Field after it encounters a fill pattern of `0xe0` (see note in Table 4.13–1)
`modTM_VC_Rcv_Packet.setAllowPacketsAfterFill(c,s,f)`	Set to Boolean `f` whether segment `s` in channel `c` will continue searching for Packets in a single Data Field after it encounters a fill pattern of `0xe0` (see note in Table 4.13–1)
`modTM_VC_Rcv_Packet.connectOutput(c,s1,v,s2,o,i)`	In channel `c`, connect an output of segment `s1` to an input of segment `s2`. If the output type of `s1`, specified by `o`, is "`Primary`", use the integer `v` to specify the Packet Version Number (PVN) of Packets to be sent over that link; if `o` is "`Auxiliary`", `d` is ignored. The input type (either "`Primary`" or "`Auxiliary`") of `s2` is specified with parameter `i`

Table 4.13–2: XML-RPC Directives for Virtual Channel Reception – Packet Service Operations

4.14 Virtual Channel Multiplexing Function (modTM_VC_Mux)

This module has multiple primary input links from which it accepts TM Transfer Frames with different Virtual Channel Identifiers (VCIDs). The frames are prioritized in the queue according to their VCID. By default, all have the same priority. If those are modified, frames with the highest-priority VCID are sent until there are none remaining; then, frames with the next highest priority are sent until there are none remaining, and so on. If high priority traffic does not abate, low priority traffic will never be sent.

This service is optionally periodic – it can be configured to send frames at a constant rate and will send Idle Frames when there is no incoming data. The VCID setting in the configuration file are used to specify which Virtual Channel Identifier the Idle Frames will have (and that is the only use of the VCID setting in this particular module).

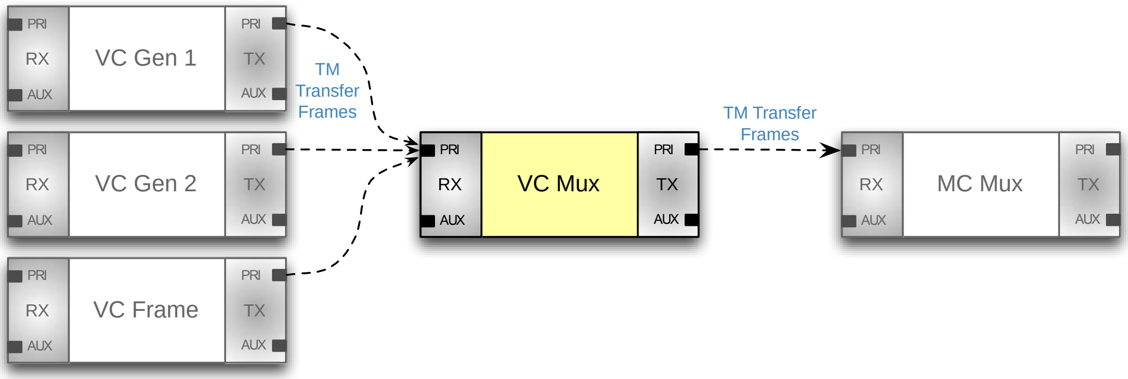

Figure 4.14-1: Virtual Channel Multiplexing Architecture Sample

Figure 4.14-1 depicts a sample layout of a configured Virtual Channel Multiplexing segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM Physical Channel”, 4.1.2, “TM Master Channel”, and 4.1.3, “TM Virtual Channel”. Additional settings are described later in this section. Figure 4.14-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    mcMux: { # segment name
      dllName = "modTM_MC_Mux";
      primaryOutput = [ "afGen", "PrimaryInput" ];
      …
    };

    vcMux1: { # segment name
      dllName = "modTM_VC_Mux";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      defaultPriority = 1000;
      channelID_Priorities = ( [ 1, 1010 ], [ 2, 900 ] );
      SCID = 1;
      VCID = 7;   # only used for Idle Frames
      …
    };

    vcGen1: { # segment name
      dllName = "modTM_VC_Gen";
      primaryOutput = [ "vcMux1", "PrimaryInput" ];
      SCID = 1;
      VCID = 1;
    };

    vcGen2: { # segment name
      dllName = "modTM_VC_Gen";
      primaryOutput = [ "vcMux1", "PrimaryInput" ];
      SCID = 1;
      VCID = 2;
    };
    …
  };
};
```

Figure 4.14-2: Virtual Channel Multiplexing Function Configuration File Excerpt

Entity	Container	Description
`defaultPriority = p;`	*segment_name* subsection	Incoming TM Transfer Frames are inserted into the message queue with integer priority `p` if they don't have their own priority specified
`channelID_Priorities = ( [ m, p ], … );`	*segment_name* subsection	For an incoming TM Transfer Frame with VCID *m*, insert it into the message queue with priority *p*

Table 4.14–1: Virtual Channel Multiplexing Function Configuration File Settings

Operation	Effect
`modTM_MC_Mux.setDefaultPriority(c,s,p)`	Set the default priority for incoming TM Transfer Frames to integer `p` for segment `s` of channel `c`
`modTM_MC_Mux.getDefaultPriority(c,s)`	Retrieve the default priority for incoming TM Transfer Frames for segment `s` of channel `c`
`modTM_MC_Mux.setPriority(c,s,v,p)`	Set the priority for incoming TM Transfer Frames with Virtual Channel Identifier `v` to `p`, for segment `s` of channel `c`
`modTM_MC_Mux.getPriority(c,s,v)`	Get the priority for incoming TM Transfer Frames with Virtual Channel Identifier `v` in segment `s` of channel `c`

Table 4.14–2: XML-RPC Directives for Virtual Channel Multiplexing Function Operations

4.15 Virtual Channel De-multiplexing Function (modTM_VC_Demux)

This module accepts TM Transfer Frame wrappers on its primary input, separates them based on their Virtual Channel Identifier (VCID), and sends them out an associated primary output. There can be as many primary outputs as there are possible virtual channels in one master channel, and each is associated with a single VCID.

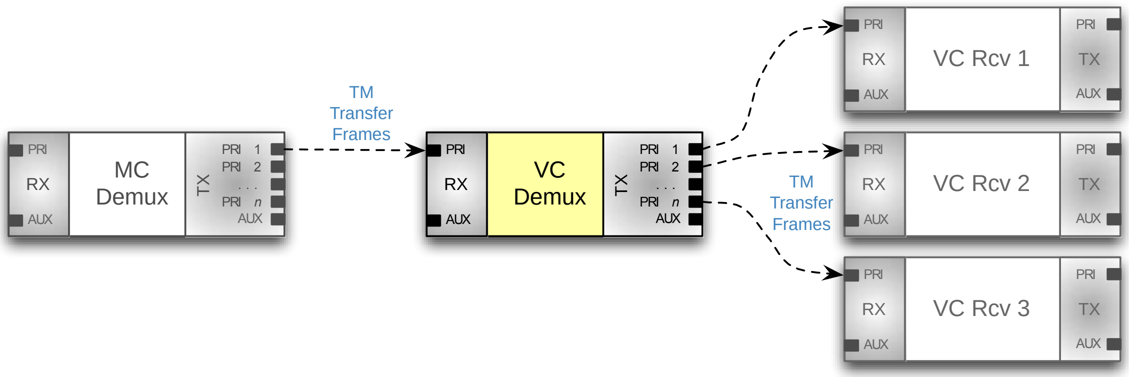

Figure 4.15-1: Virtual Channel De-multiplexing Sample Architecture

Figure 4.15-1 depicts a sample layout of a configured Virtual Channel De-multiplex segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM Physical Channel”, 4.1.2, “TM Master Channel”, and 4.1.3, “TM Virtual Channel”. The exception is the `primaryOutputs` setting. Similar to `primaryOutput`, found in most modules, `primaryOutputs` also specifies which segments the main flow of traffic is directed to, and on which input. However, rather than being a one-dimensional array, this setting is a list of lists. Each sub list contains the next segment name, its input type, and a third parameter – the Virtual Channel Identifier (VCID) of that virtual channel.

```
Channels: {

  chanTest: {

    TM_PhysicalChannel:
    {
      frameSize = 256;
      useFrameErrorControl = true;
      idlePattern = [ 0xAA, 0xBB, 0xCC ];
    };
    …
    vcRcv3: { # segment name
      dllName = "modTM_VC_Rcv";
      primaryOutputs =  ( ( "Next Segment C", "PrimaryInput", 7 ), … );
      SCID = 0xA1;
      VCID = 0x4;
      …
    };

    vcRcv2: { # segment name
      dllName = "modTM_VC_Rcv";
      primaryOutputs =  ( ( "Next Segment B", "PrimaryInput", 7 ), … );
      SCID = 0xA1;
      VCID = 0x2;
      …
    };

    vcRcv1: { # segment name
      dllName = "modTM_VC_Rcv";
      primaryOutputs =  ( ( "Next Segment A", "PrimaryInput", 7 ), … );
      SCID = 0xA1;
      VCID = 0x1;
      …
    };

    vcDemux: { # segment name
      dllName = "modTM_VC_Demux";
      primaryOutputs = ( ( "vcRcv1", "PrimaryInput", 0x1),
                         ( "vcRcv2", "PrimaryInput", 0x2),
                         …,
                         ( "vcRcv3", "PrimaryInput", 0x4),
                       );
      SCID = 0xA1;
      …
    };

    mcDemux: { # segment name
      dllName = "modTM_MC_Demux";
      primaryOutputs = ( ( "vcDemux", "PrimaryInput", 0xA1), … );
      …
    };
    …
  };
};
```

Figure 4.15-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {

    TM_PhysicalChannel:
    {
      frameSize = 256;
      useFrameErrorControl = true;
      idlePattern = [ 0xAA, 0xBB, 0xCC ];
    };
    …
    vcRcv3: { # segment name
      dllName = "modTM_VC_Rcv";
      primaryOutputs =  ( ( "Next Segment C", "PrimaryInput", 7 ), … );
      SCID = 0xA1;
      VCID = 0x4;
      …
    };

    vcRcv2: { # segment name
      dllName = "modTM_VC_Rcv";
      primaryOutputs =  ( ( "Next Segment B", "PrimaryInput", 7 ), … );
      SCID = 0xA1;
      VCID = 0x2;
      …
    };

    vcRcv1: { # segment name
      dllName = "modTM_VC_Rcv";
      primaryOutputs =  ( ( "Next Segment A", "PrimaryInput", 7 ), … );
      SCID = 0xA1;
      VCID = 0x1;
      …
    };
    vcDemux: { # segment name
      dllName = "modTM_VC_Demux";
      primaryOutputs = ( ( "vcRcv1", "PrimaryInput", 0x1),
                         ( "vcRcv2", "PrimaryInput", 0x2),
                         …,
                         ( "vcRcv3", "PrimaryInput", 0x4),
                       );
      SCID = 0xA1;
      …
    };

    mcDemux: { # segment name
      dllName = "modTM_MC_Demux";
      primaryOutputs = ( ( "vcDemux", "PrimaryInput", 0xA1), … );
      …
    };
    …
  };
};
```

Figure 4.15-2: Virtual Channel De-multiplexing Configuration File Excerpt

Entity	Container	Description
`primaryOutputs = ( ( "s", "PrimaryInput" | "AuxInput", i ), … );`	*segment_name* subsection	TM Transfer Frames received with Virtual Channel Identifier `i` are sent on to either the Primary or Auxiliary input of segment `s`. Segment `s` must already exist in the channel

Table 4.15–1: Virtual Channel De-multiplexing Configuration File Settings

Operation	Effect
`modTM_MC_Demux. connectOutput(c,s1,d,s2,o,i)`	In channel `c`, connect an output of segment `s1` to an input of segment `s2`. If the output type of `s1`, specified by `o`, is "`Primary`", use the integer `d` to specify the Virtual Channel Identifier for TM Transfer Frames to be sent over that link; if `o` is "`Auxiliary`", `d` is ignored. The input type (either "`Primary`" or "`Auxiliary`") of `s2` is specified with parameter `i`

Table 4.15–2: XML-RPC Directives for Virtual Channel De-multiplexing Operations

4.16 Frame Secondary Header Insertion Service (modTM_FSH_Insert)

The FSH Insertion Service accepts TM Transfer Frame wrappers on its primary input queue. On its auxiliary input queue it receives fixed-length data units to insert into the Secondary Header Field of the frames it receives. As frames arrive, the buffers of any units waiting in the auxiliary input queue are copied into the appropriate location of each frame, and then the altered frames are sent via the primary output queue. By placing this segment immediately after a Virtual Channel Generation function, it will provide a Virtual Channel Frame Secondary Header

Service; by placing it after a Master Channel Generation function, it will provide a Master Channel Frame Secondary Header Service.

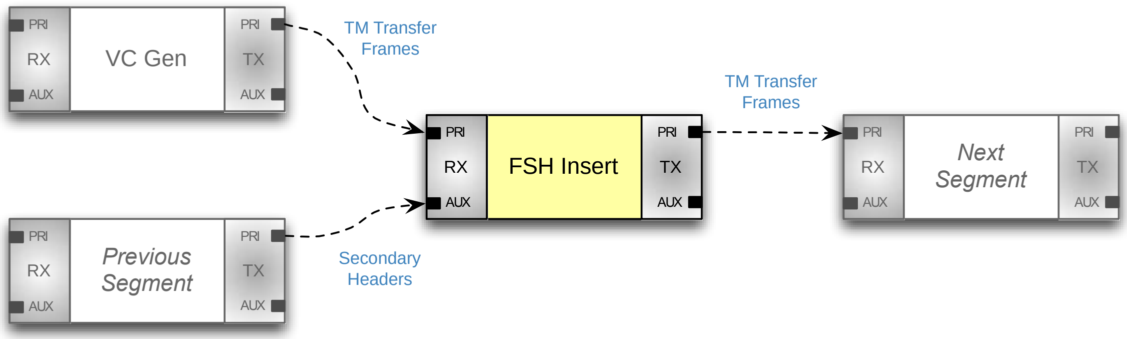

Figure 4.16-1: TM Frame Secondary Header Insertion Service Architecture Sample

Figure 4.16-1 depicts a sample layout of a configured Frame Secondary Header Insertion Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM Physical Channel”, 4.1.2, “TM Master Channel”, and 4.1.3, “TM Virtual Channel”. Figure 4.16-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    vcMux: {
      dllName = "modTM_VC_Mux";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      …
    };

    fshInsert: {
      dllName = "modTM_FSH_Insert";
      primaryOutput = [ "vcMux", "PrimaryInput" ];
      fshSize = 32;
      useFSH = true;
    };

    prvSeg: {
      primaryOutput = [ "fshInsert", "AuxInput" ];
      …
    };

    vcGen: {
      dllName = "modTM_VC_Gen";
      primaryOutput = [ "fshInsert", "PrimaryInput" ];
      SCID = 5;
      VCID = 3;
      fshSize = 32;
      useFSH = true;
      …
    };
    …
  };
};
```

Figure 4.16-2: TM Frame Secondary Header Insertion Service Configuration File Example

4.17 Frame Secondary Header Extraction Service (modTM_FSH_Extract)

The FSH Extraction Service accepts TM Transfer Frame wrappers on its primary input queue. If the Secondary Header Flag of a frame is true, its FSH is wrapped separately and sent via the auxiliary output. The complete, unmodified frame is then send via the primary output.

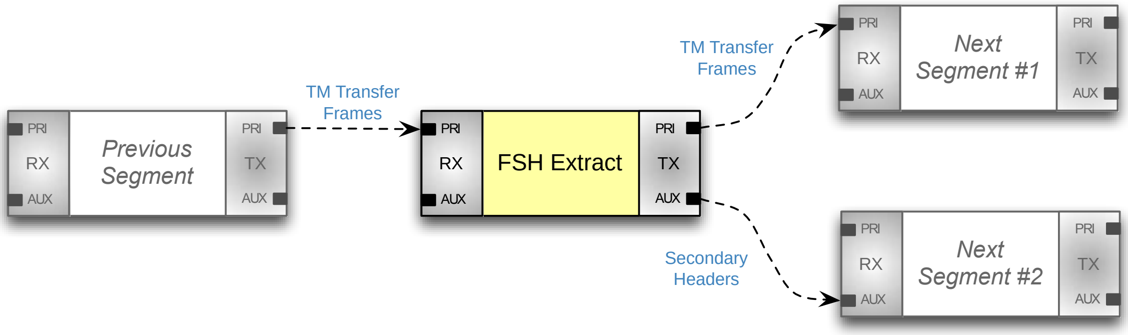

Figure 4.17-1: TM Frame Secondary Header Extraction Service Architecture Sample

Figure 4.17-1 depicts a sample layout of a configured Frame Secondary Header Extraction Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM Physical Channel”, 4.1.2, “TM Master Channel”, and 4.1.3, “TM Virtual Channel”. Figure 4.17-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    fshExtract: {
      dllName = "modTM_FSH_Extract";
      primaryOutput = [ "Next Segment #1", "PrimaryInput" ];
      auxOutput = [ "Next Segment #2", "PrimaryInput" ];
      fshSize = 32;
      useFSH = true;
    };

    prvSeg: {
      primaryOutput = [ "fshExtract", "PrimaryInput" ];
      …
    };
    …
  };
};
```

Figure 4.17-2: TM Frame Secondary Header Extraction Service Configuration File Example

4.18 Operational Control Field Insertion Service (modTM_OCF_Insert)

The OCF Insertion Service accepts TM Transfer Frame wrappers on its primary input queue. On its auxiliary input queue it receives four-octet data units to insert into the Operational Control Field of the frames it receives. As frames arrive, the buffers of any units waiting in the auxiliary input queue are copied into the appropriate location of each frame, and then the altered frames are sent via the primary output queue. By placing this segment immediately after a Virtual Channel Generation function, it will provide a Virtual Channel Operational Control Field Service; by placing it after a Master Channel Generation function, it will provide a Master Channel Operational Control Field Service.

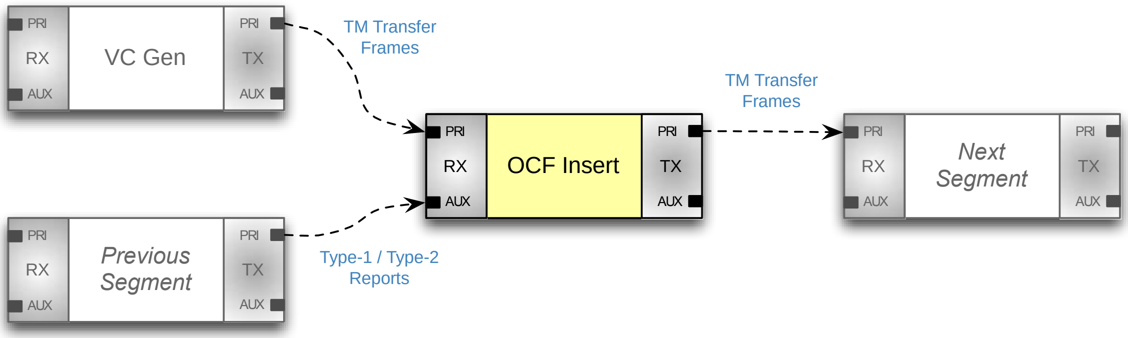

Figure 4.18-1: TM Operational Control Field Insertion Service Architecture Sample

Figure 4.16-1 depicts a sample layout of a configured Operational Control Field Insertion Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM Physical Channel”, 4.1.2, “TM Master Channel”, and 4.1.3, “TM Virtual Channel”. Figure 4.16-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    vcMux: {
      dllName = "modTM_VC_Mux";
      primaryOutput = [ "mcMux", "PrimaryInput" ];
      …
    };

    ocfInsert: {
      dllName = "modTM_OCF_Insert";
      primaryOutput = [ "vcMux", "PrimaryInput" ];
      useOperationalControl = true;
    };

    prvSeg: {
      primaryOutput = [ "ocfInsert", "AuxInput" ];
      …
    };

    vcGen: {
      dllName = "modTM_VC_Gen";
      primaryOutput = [ "ocfInsert", "PrimaryInput" ];
      SCID = 5;
      VCID = 3;
      useOperationalControl = true;
      …
    };
    …
  };
};
```

Figure 4.18-2: TM Operational Control Field Insertion Service Configuration File Example

4.19 Operational Control Field Extraction Service (modTM_OCF_Extract)

The OCF Extraction Service accepts TM Transfer Frame wrappers on its primary input queue. If the Operational Control Field Flag of a frame is true, its OCF is wrapped separately and sent via the auxiliary output. The complete, unmodified frame is then send via the primary output.

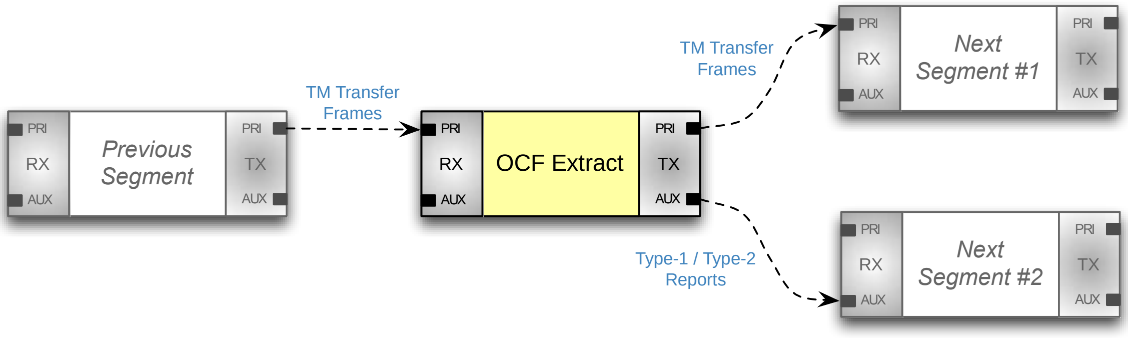

Figure 4.19-1: TM Operational Control Field Extraction Service Architecture Sample

Figure 4.19-1 depicts a sample layout of a configured Operational Control Field Extraction Service segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapters 2.4, “Modular Segments”, 4.1.1, “TM Physical Channel”, 4.1.2, “TM Master Channel”, and 4.1.3, “TM Virtual

Channel". Figure 4.19-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  …
  chanTM_test:
  {
    …
    TM_PhysicalChannel:
    {
      frameSize = 2048;
      useFrameErrorControl = true;
    };
    …
    ocfExtract: {
      dllName = "modTM_OCF_Extract";
      primaryOutput = [ "Next Segment #1", "PrimaryInput" ];
      auxOutput = [ "Next Segment #2", "PrimaryInput" ];
      useOperationalControl = true;
    };

    prvSeg: {
      primaryOutput = [ "ocfExtract", "PrimaryInput" ];
      …
    };
    …
  };
};
```

Figure 4.19-2: TM Operational Control Field Extraction Service Configuration File Example

4.20 Macros Interface (modTM_Macros)

Assembling a complete TM Physical Channel from scratch can be difficult without detailed awareness of the protocol. The TM Macros module provides a pair of methods that create a template of either a forward or return channel, using only a small number of settings. Table 4.20–1 describes these functions.

Operation	Effect
`modTM_Macros.` `newForwardChannel(c,s)`	In pre-existing CE channel `c`, create a new forward TM Physical Channel, complete with settings, services, and functions specified in XML-RPC struct `s`. The struct may contain any or all of the following members:

`frameLength`	The exact size for all TM Transfer Frames in the channel in whole octets.
`packetSvcCount`	The number of virtual channels that will provide Packet Services.
`vcAccessSvcCount`	The number of virtual channels that will provide VC Access Services.
`vcFrameSvcCount`	The number of virtual channels that will consist of VC Frame Services.
`mcFrameSvcCount`	The number of master channels that will consist of MC Frame Services.
`masterChannelCount`	Select the number of master channels to create. Multiplies the number of Packet, Bitstream, VC Access, and VC Frame Services.
`maxErrorsReedSolomon`	If 8 or 16, generate Reed-Solomon parity symbols and append them to each frame.
`useFrameEC`	If true, enable error control for the entire frame via a checksum.
`useASM`	If true, attach an sync marker to each frame.
`usePseudoRandomize`	If true, exclusive-OR the entire unit with a predetermined sequence
`minimize`	If true, create only enough channel segments to perform the requested functions. If false (default), leave some functions in (e.g. mux/demux) to accommodate additional services
`segNamePrefix`	A string to prepend to each segment name.
`segNameSuffix`	A string to append to each segment name.

`modTM_Macros.newReturnChannel(c,s)`	In pre-existing CE channel `c`, create a new return TM Physical Channel, complete with settings, services, and functions specified in XML-RPC struct `s`. The struct may contain any or all of the following members:

`frameLength`	The exact size for all AOS Transfer Frames in the channel in whole octets.
`packetSvcCount`	The number of virtual channels that will provide Packet Services.
`masterChannelCount`	Select the number of master channels to create. Multiplies the number of Packet, Bitstream, VC Access, and VC Frame Services.
`maxErrorsReedSolomon`	If 8 or 16, generate Reed-Solomon parity symbols and append them to each frame.
`useFrameEC`	If true, enable error control for the entire frame via a checksum.
`useASM`	If true, attach an sync marker to each frame.
`usePseudoRandomize`	If true, exclusive-OR the entire unit with a predetermined sequence
`minimize`	If true, create only enough channel segments to perform the requested functions. If false (default), leave some functions in (e.g. mux/demux) to accommodate additional services
`segNamePrefix`	A string to prepend to each segment name.
`segNameSuffix`	A string to append to each segment name.

Table 4.20–1: XML-RPC Directives for TM Macros

5 Ethernet-Related Modules

5.1 Ethernet Frame Encoding (modEthFrame_Add)

This module can take any type of data as input, encapsulate it in an IEEE 802.3 Ethernet frame, and send it via its primary output to the next segment. Source and destination MAC addresses must be manually configured.

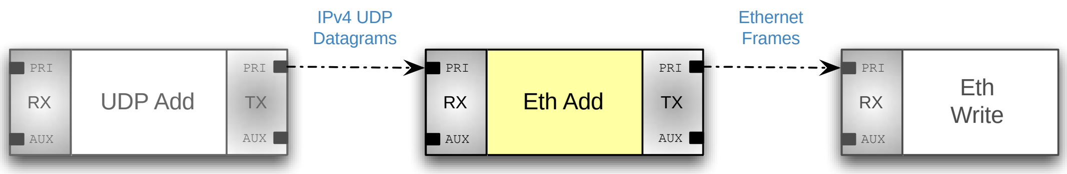

Figure 5.1-1: Ethernet Frame Encoding Architecture Sample

Figure 5.1-1 depicts a sample layout of a configured Ethernet Frame Encoding segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”, but requires several protocol-specific options described in tables below. Figure 5.1-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    datagramAdd: { # segment name
      dllName = "modUDP_Add”;
      primaryOutput = [ "ethAdd", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    ethAdd: { # segment name
      dllName = "modEthFrame_Add";
      primaryOutput = [ "ethWrite", "PrimaryInput" ];
      dstMAC = "00:30:48:57:6a:d5";
      srcMAC = "00:FA:D0:B1:71:35";
      immediateStart = true;
    };

    ethWrite: { # segment name
      dllName = "modEthTransmitter";
      deviceName = "Configured Ethernet Device";
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 5.1-2: Ethernet Frame Encoding Configuration File Excerpt

Entity	Container	Description
`dstMAC = "a:b:c:d:e:f";`	*segment_name* subsection	Set the destination MAC address to "*a:b:c:d:e:f*", where each octet `a-f` is in hexadecimal format
`srcMAC = "a:b:c:d:e:f";`	*segment_name* subsection	Set the source MAC address to "*a:b:c:d:e:f*", where each octet `a-f` is in hexadecimal format
`defaultPayloadType = t;`	*segment_name* subsection	Set the default payload type to integer t. This value is put into the Type/Length Field in the Ethernet header if the incoming wrapper type is unrecognized. Without this setting, the payload length is used in that field instead of the type.

Table 5.1–1: Ethernet Frame Encoding Configuration File Settings

Operation	Effect
`modEthFrame_Add. getDstMAC(c,s)`	Retrieve the current destination MAC address from segment `s` in channel `c` as a string in the form of "`a:b:c:d:e:f`", with each octet `a-f` in hexadecimal format
`modEthFrame_Add. setDstMAC(c,s,m)`	Set the destination MAC address for segment `s` in channel `c` to `m`, which is a string in the form of "`a:b:c:d:e:f`", with each octet `a-f` in hexadecimal format
`modEthFrame_Add. getSrcMAC(c,s)`	Retrieve the current source MAC address from segment `s` in channel `c` as a string in the form of "`a:b:c:d:e:f`", with each octet `a-f` in hexadecimal format
`modEthFrame_Add. setSrcMAC(c,s,m)`	Set the source MAC address for segment `s` in channel `c` to `m`, which is a string in the form of "`a:b:c:d:e:f`", with each octet `a-f` in hexadecimal format
`modEthFrame_Add. getDefaultPayloadType(c,s)`	Get the default payload type as an integer for segment `s` in channel `c` (see Table 5.1–1)
`modEthFrame_Add. setDefaultPayloadType(c,s,t)`	Set the default payload type to integer `t` for segment `s` in channel `c` (see Table 5.1–1)

Table 5.1–2: XML-RPC Directives for Ethernet Frame Encoding Operations

5.2 Ethernet Frame Reception (modEthReceiver)

The Ethernet Frame Reception function reads from a packet socket managed by a CE Ethernet Device. When an IEEE 802.3 Ethernet frame buffer is received, it is wrapped, time stamped, and then sent to the target via the segment’s primary output link. Little checking on the validity of the structure in the buffer is performed; it is assumed that it has already been validated at lower levels in the operating system. The target is typically a Ethernet Frame Decoding segment.

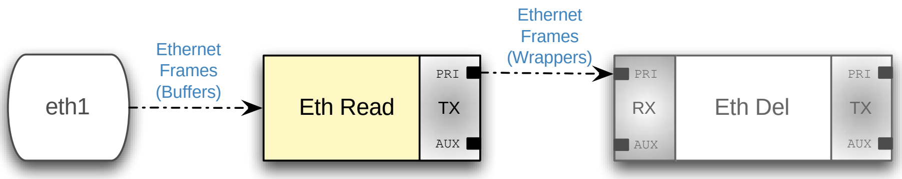

Figure 5.2-1: Ethernet Frame Reception Sample Architecture

Figure 5.2-1 depicts a sample layout of a configured Ethernet Frame Reception segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”. Figure 5.2-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
…
Devices :
{

  EthernetDevice_Common :
  {
    protectedDevices = [ "eth0" ];
    ignoredDevices = [ "lo" ];
  };

  eth1 :
  {
    devType = "Ethernet";
    snapLen = 1520;
  };
  …
};

Channels: {

  chanTest: {
    …
    ethDel: { # segment name
      dllName = "modEthFrame_Remove”;
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    ethRead: { # segment name
      dllName = "modEthReceiver";
      deviceName = "eth1";
      primaryOutput = [ "ethDel", "PrimaryInput" ];
      immediateStart = true;
      };
    };
  };
```

Figure 5.2-2: Ethernet Frame Reception Configuration File Excerpt

5.3 Ethernet Frame Decoding (modEthFrame_Remove)

This module accepts IEEE 802.3 Ethernet frames on its primary input. It removes the frame header and wraps the payload in the data field. If the payload if of a recognized and supported type, it is wrapped with an object with specific support for that type; otherwise, it is treated as untyped. The wrapper is then sent via the primary output link to the target.

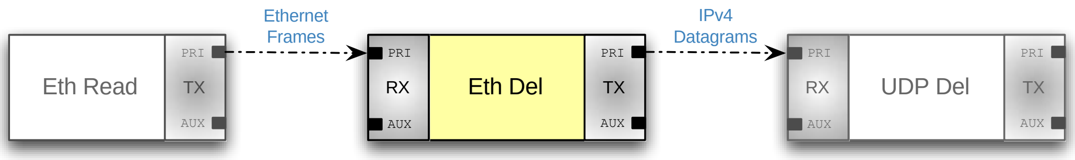

Figure 5.3-1: Ethernet Frame Decoding Architecture Sample

Figure 5.3-1 depicts a sample layout of a configured Ethernet Frame Reception segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”. Figure 5.3-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
…
Channels: {

  chanTest: {
    …
    ethRead: { # segment name
      dllName = "modEthReceiver";
      deviceName = "Configured Ethernet Device";
      primaryOutput = [ "ethDel", "PrimaryInput" ];
      immediateStart = true;
    };

    ethDel: { # segment name
      dllName = "modEthFrame_Remove”;
      primaryOutput = [ "udpDel", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    udpDel: { # segment name
      dllName = "modUDP_Remove”;
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 5.3-2: Ethernet Frame Decoding Configuration File Excerpt

5.4 Ethernet Frame Transmission (modEthTransmitter)

The Ethernet Frame Transmission function accepts an IEEE 802.3 Ethernet frame wrapper on its primary input, typically from an Ethernet Frame Encoding segment. It writes the buffer to a packet socket associated with a CE Ethernet Device.

This module has a few settings that manage the link detection capabilities of the Ethernet Device. These are occasionally necessary because a CE channel can come online so quickly that a newly activated hardware device may not be ready to send. If data were written to it anyway, it would just be lost, which is not an acceptable situation for “unreliable” protocols where a connection is expected nevertheless. The link detection settings can be used to block transmission until a carrier is found or a timeout occurs.

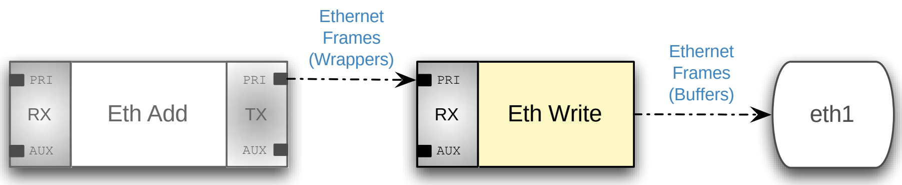

Figure 5.4-1: Ethernet Frame Transmission Architecture Sample

Figure 5.4-1 depicts a sample layout of a configured Ethernet Frame Reception segment. Except for the options relating to link checking (detailed at the end of this section), this module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”. Figure 5.4-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
…
Devices :
{

  EthernetDevice_Common :
  {
    protectedDevices = [ "eth0" ];
    ignoredDevices = [ "lo" ];
  };

  eth1 :
  {
    devType = "Ethernet";
  };
  …
};

Channels: {

  chanTest: {
    …
    ethAdd: { # segment name
      dllName = "modEthFrame_Add”;
      primaryOutput = [ "ethWrite", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    ethWrite: { # segment name
      dllName = "modEthTransmitter";
      deviceName = "eth1";
      linkCheckFreq = "Once";
      linkCheckMaxIterations = 100;
      linkCheckSleepMsec = 100;
      immediateStart = true;
    };
  };
};
```

Figure 5.4-2: Ethernet Frame Transmission Configuration File Excerpt

Entity	Container	Description
`linkCheckFreq = “Never” | “Once” | “Always”;`	*segment_name* subsection	How often to check for a carrier. `Never` Do not check `Once` Check before the segment enters its active loop, but not again `Always` Check before each frame is sent
`linkCheckMaxIterations = `*`m`*`;`	*segment_name* subsection	When testing for a carrier, check integer *`m`* times before failing
`linkCheckSleepMsec = `*`s`*`;`	*segment_name* subsection	When testing for a carrier, wait *`s`* milliseconds between checks

Table 5.4–1: Ethernet Frame Transmission Configuration File Settings

Operation	Effect
`modEthTransmitter.getLinkCheckFrequency(c,s)`	Retrieve a string from segment `s` in channel `c` describing how often a carrier is tested for
`modEthTransmitter.setLinkCheckFrequency(c,s,f)`	Test for a carrier with frequency `f` in segment `s` of channel `c`. String `f` may be “Never”, “Once”, or “Always” (see Table 5.4–1)
`modEthTransmitter.getLinkCheckMaxIterations(c,s)`	Query segment `s` in channel `c` for the number of times it checks for a carrier before failing, when testing is enabled
`modEthTransmitter.setLinkCheckMaxIterations(c,s,m)`	Set to integer `m` the number of times segment `s` in channel `c` will check for a link

	before failing, when testing is enabled
`modEthTransmitter.getLinkCheckSleepMSec(c,s)`	Query segment `s` in channel `c` for the number of milliseconds between carrier checks, when testing is enabled
`modEthTransmitter.setLinkCheckSleepMSec(c,s,l)`	Set to integer `l` the number of milliseconds segment `s` in channel `c` will wait between link checks, when testing is enabled

Table 5.4–2: XML-RPC Directives for Ethernet Frame Transmission Operations

6 File-Related Modules

6.1 File Descriptor Reader (modFdReceiver)

This module reads input from a file descriptor managed by a CE File Device (or other Device that provides a file descriptor for I/O). It refers to its MRU setting to determine how much to read at once. If the device is a file, it may be reread and resent a specified number of times. Each buffer is wrapped as untyped data and sent via the primary output link to the next segment.

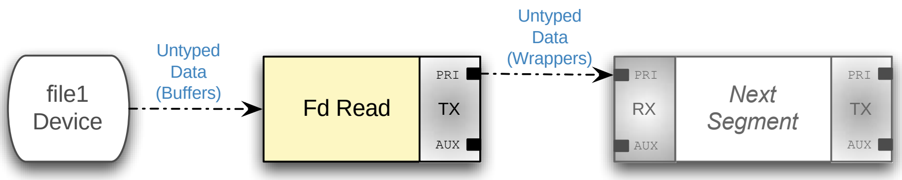

Figure 6.1-1: File Descriptor Reader Architecture Sample

Figure 6.1-1 depicts a sample layout of a configured File Descriptor Reader segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments"; module-specific options are described at the end of this section. Figure 6.1-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
…
Devices :
{

  FileDevice_Common :
  {
    IOpath = "/var/CE";
  };

  file1 :
  {
    fileName = "test.tgz";
    devType = "File";
    isInput = true;
  };
  …
};

Channels: {

  chanTest: {
    …
    fileRead: { # segment name
      dllName = "modFdReceiver";
      deviceName = "file1";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      immediateStart = true;
    };
  };
};
```

Figure 6.1-2: File Descriptor Reader Configuration File Excerpt

Entity	Container	Description
`repeatMax = m;`	*segment_name* subsection	Re-read and re-send the file *m* times. If m is zero, the file is sent only once; if m is one, it is sent twice; etc.

Table 6.1–1: File Descriptor Reader Configuration File Settings

Operation	Effect
`modFdReceiver.getMaxRead(c,s)`	Get the number of octets that will be read from the file with each iteration, for segment `s` of channel `c`
`modFdReceiver.setMaxRead(c,s,m)`	Set the number of octets to read from the file with each iteration, for segment `s` of channel `c`
`modFdReceiver.getRepeatCount(c,s)`	Get the number of times that the complete file has already been sent for segment `s` of channel `c`
`modFdReceiver.getRepeatMax(c,s)`	Get the number of times that the file will be re-read/re-sent for segment `s` of channel `c`
`modFdReceiver.setRepeatMax(c,s,m)`	Set the number of times that the file will be re-read/re-sent to `m` for segment `s` of channel `c`

Table 6.1–2: XML-RPC Directives for File Descriptor Reader Operations

6.2 File Descriptor Writer (modFdTransmitter)

This module receives any type of network data on its primary input and writes it to the file descriptor managed by a CE File Device configured for output.

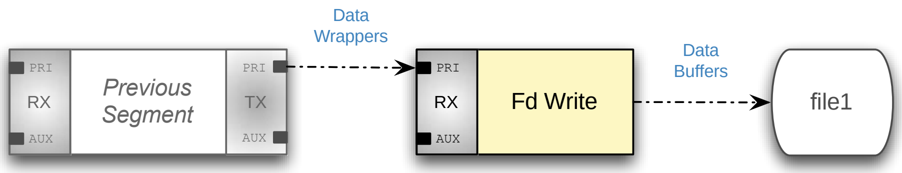

Figure 6.2-1: File Descriptor Writer Architecture Sample

Figure 6.2-1 depicts a sample layout of a configured File Descriptor Reader segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments. Figure 6.2-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
…
Devices :
{

  FileDevice_Common :
  {
    IOpath = "/var/CE";
  };

  file1 :
  {
    fileName = "test.bin";
    devType = "File";
    isInput = false;
  };
  …
};

Channels: {

  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "fileWrite", "PrimaryInput" ];
      …
      immediateStart = "true";
    };

    fileWrite: { # segment name
      dllName = "modFdTransmitter";
      deviceName = "file1";
      …
      immediateStart = true;
    };
  };
};
```

Figure 6.2-2: File Descriptor Reader Configuration File Excerpt

7 Internet Protocol Modules

7.1 IPv4 TCP Receiver (modTcp4Receiver)

This module receives raw data from either a TCP Server or TCP Client device; the packet headers are not retained. It then wraps it as untyped network data and sends it via its primary output link. If the associated device is a Server, it will also handle accepting new connections.

Only one connection is allowed at a time (although more clients may be waiting to have their connection accepted). This is because for two-way channels, there is not necessarily any address information that can be used to properly de-multiplex outgoing data to multiple clients.

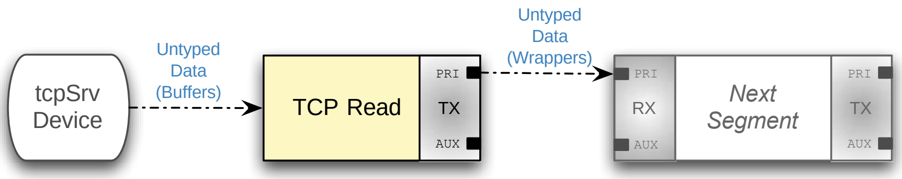

Figure 7.1-1: IPv4 TCP Receiver Architecture Sample

Figure 7.1-1 depicts a sample layout of a configured IPv4 TCP Receiver segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”; module-specific options are described at the end of this section. Figure 7.1-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
…
Devices :
{

  tcpSrv :
  {
    devType = "Tcp4Server";
    address = "127.0.0.1";
    port = 6750;
  };
  …
};

Channels: {

  chanTest: {
    …
    tcpRead: { # segment name
      dllName = "modTcp4Receiver";
      deviceName = "tcpSrv";
      MRU = 1024;
      receiveMax = true;
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      immediateStart = true;
    };
  };
};
```

Figure 7.1-2: IPv4 TCP Receiver Configuration File Excerpt

Entity	Container	Description
`receiveMax = true|false;`	*segment_name* subsection	If `true`, the segment will wait to receive its full MRU before continuing; if `false`, it will continue once there is nothing else to read

Table 7.1–1: IPv4 TCP Receiver Configuration File Settings

Operation	Effect
`modTcp4Receiver.getBufferSize(c,s)`	Get the size of the buffer used to receive data from segment `s` in channel `c`
`modTcp4Receiver.setBufferSize(c,s,b)`	Set the size of the buffer used to receive data from segment `s` in channel `c` to integer `b`
`modTcp4Receiver.closeConnection(c,s)`	If the TCP device associated with segment `s` in channel `c` has an active TCP connection, close it
`modTcp4Receiver.getReceiveMax(c,s)`	Returns whether segment `s` of channel `c` will wait to receive its full MRU before continuing
`modTcp4Receiver.setReceiveMax(c,s,r)`	If `r` is `true`, segment `s` of channel `c` will wait to receive its full MRU before continuing; if `r` is `false` it will continue when there is nothing else to read

Table 7.1–2: XML-RPC Directives for IPv4 TCP Receiver Operations

7.2 IPv4 TCP Transmitter (modTcp4Transmitter)

This module can receive any type of wrapper and write its buffer to a socket maintained by an IPv4 TCP Client or Server device, after which it will be encoded as the payload of an IPv4 TCP packet. Data will only be transmitted its associated TCP device has an active connection; otherwise, data will remain in the message queue. If it fills up, it will cause the segment on the primary input link to block.

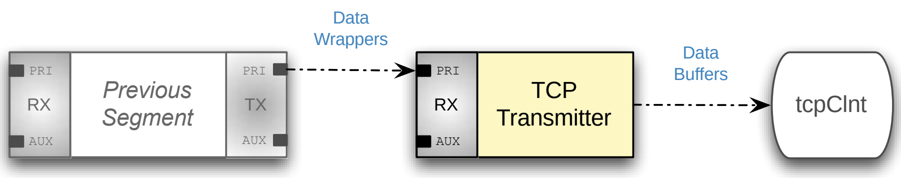

Figure 7.2-1: IPv4 TCP Transmitter Architecture Sample

Figure 7.2-1 depicts a sample layout of a configured IPv4 TCP Transmitter segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments"; module-specific options are described at the end of this section. Figure 7.2-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
…
Devices :
{

  tcpClnt :
  {
    devType = "Tcp4Client";
    address = "127.0.0.1";
    port = 5001;
  };
  …
};

Channels: {

  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "tcpTransmitter", "PrimaryInput" ];
      …
      immediateStart = "true";
    };

    tcpTransmitter: { # segment name
      dllName = "modTcp4Transmitter";
      deviceName = "tcpClnt";
      …
      immediateStart = true;
    };
  };
};
```

Figure 7.2-2: IPv4 TCP Transmitter Configuration File Excerpt

Operation	**Effect**
`modTcp4Transmitter.closeConnection(c,s)`	If the TCP device associated with segment `s` in channel `c` has an active TCP connection, close it

Table 7.2–1: XML-RPC Directives for IPv4 TCP Transmitter Operations

7.3 IPv4 UDP Datagram Encoding (modUDP_Add)

This module accepts any type of wrapped data on its primary input, adds an IPv4 UDP Datagram header to it, and then sends it via its primary output link. It must be manually configured with the source/destination IP addresses and ports. The source address and port may well be fictitious. However, some care should still be used in selecting them if there is a real receiver so a firewall or other safeguards on the remote system do not reject them. The UDP checksum can be added to each packet, if desired.

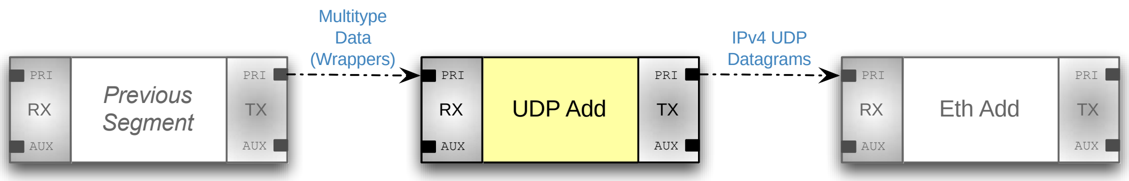

Figure 7.3-1: IPv4 UDP Datagram Encoding Architecture Sample

Figure 7.3-1 depicts a sample layout of a configured IPv4 UDP Datagram Encoding segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments"; module-specific options are described at the end of this section. Figure 7.3-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "udpAdd", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    udpAdd: { # segment name
      dllName = "modUDP_Add”;
      primaryOutput = [ "ethAdd", "PrimaryInput" ];
      srcAddr = "10.10.1.1";
      srcPort = 12345;
      dstAddr = "10.10.1.2";
      dstPort = 54321;
      useUDPCRC = true;
      …
      immediateStart = true;
    };

    ethAdd: { # segment name
      dllName = "modEthFrame_Add";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      dstMAC = "00:30:48:57:6a:d5";
      srcMAC = "00:FA:D0:B1:71:35";
      immediateStart = true;
    };
    …
  };
};
```

Figure 7.3-2: IPv4 UDP Datagram Encoding Configuration File Excerpt

Entity	Container	Description
`srcAddr = "a.b.c.d";`	*segment_name* subsection	Set the source IPv4 address on each outgoing datagram to `a.b.c.d`, where each octet `a`-`d` is in decimal notation.
`srcPort = p;`	*segment_name* subsection	Set the source port on each outgoing datagram to integer `p`.
`dstAddr = "a.b.c.d";`	*segment_name* subsection	Set the destination IPv4 address on each outgoing datagram to `a.b.c.d`, where each octet `a`-`d` is in decimal notation.
`dstPort = p;`	*segment_name* subsection	Set the destination port on each outgoing datagram to integer `p`.
`useUDPCRC = true | false;`	*segment_name* subsection	Select whether the UDP checksum will be calculated and inserted into each datagram.

Table 7.3–1: IPv4 UDP Datagram Encoding Configuration File Settings

Operation	Effect
`modUDP_Add.getSrcAddr(c,s)`	Retrieve the source IPv4 address as a string from segment `s` in channel `c`
`modUDP_Add.setSrcAddr(c,s,a)`	Set the source IPv4 address to `a` for segment `s` in channel `c`. The format of `a` is a string in the format "a.b.c.d" where each octet a-d is in decimal notation
`modUDP_Add.getSrcPort(c,s)`	Get the source port as an integer for segment `s` in channel `c`
`modUDP_Add.setSrcPort(c,s,p)`	Set the source port to integer `p` for segment s in channel `c`
`modUDP_Add.getDstAddr(c,s)`	Retrieve the destination IPv4 address as a string from segment `s` in channel `c`
`modUDP_Add.setDstAddr(c,s,a)`	Set the destination IPv4 address to `a` for segment `s` in channel `c`. The format of `a` is a string in the format "a.b.c.d" where each octet a-d is in decimal notation
`modUDP_Add.getDstPort(c,s)`	Get the destination port as an integer for segment `s` in channel `c`
`modUDP_Add.`	Set the destination port to integer `p` for segment s

`setDstPort(c,s,p)`	in channel `c`
`modUDP_Add.getUDPCRC(c,s)`	Retrieve whether segment `s` in channel `c` is setting the UDP checksum filed in outgoing datagrams
`modUDP_Add.setUDPCRC(c,s,u)`	Enable or disable use of the UDP checksum field for segment `s` in channel `c` by setting Boolean `u`

Table 7.3–2: XML-RPC Directives for IPv4 UDP Datagram Encoding Operations

7.4 IPv4 UDP Datagram Decoding (modUDP_Remove)

This module accepts IPv4 UDP datagram wrappers on its primary input. If CRC checking is enabled, both the IP and UDP checksums are tested; if there is a mismatch, the datagram is dropped. If the checksums are valid, the payload is extracted, wrapped as untyped data, and sent on via the primary output link.

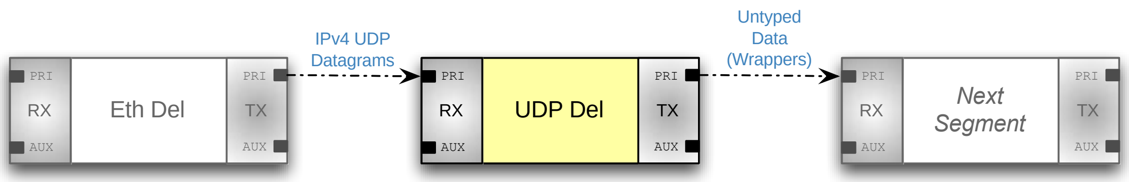

Figure 7.4-1: IPv4 UDP Datagram Decoding Architecture Sample

Figure 7.4-1 depicts a sample layout of a configured IPv4 UDP Datagram Decoding segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”; module-specific options are described at the end of this section. Figure 7.4-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    ethDel: { # segment name
      dllName = "modEthFrame_Remove”;
      primaryOutput = [ "udpDel", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    udpDel: { # segment name
      dllName = "modUDP_Remove";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      useUDPCRC = true;
      …
      immediateStart = true;
    };

    nextSegment: {
      …
      immediateStart = true;
    };
```

```
      …
   };
};
```

Figure 7.4-2: IPv4 UDP Datagram Decoding Configuration File Excerpt

Entity	Container	Description
`useUDPCRC = true | false;`	*segment_name* subsection	Select whether CRC testing will be performed on incoming datagrams

Table 7.4–1: IPv4 UDP Datagram Decoding Configuration File Settings

Operation	Effect
`modUDP_Remove.getCRCCheck(c,s)`	Determine whether segment `s` in channel `c` is testing the checksums of incoming datagrams
`modUDP_Remove.setCRCCheck(c,s,k)`	Enable or disable checksum testing for segment `s` in channel `c` by setting Boolean `k`
`modUDP_Remove.getBadIPCRCCount(c,s)`	Get the tally of incorrect IPv4 checksums from segment `s` in channel `c`
`modUDP_Remove.setBadIPCRCCount(c,s,b)`	Set the tally of incorrect IPv4 checksums to `b` for segment `s` in channel `c`
`modUDP_Remove.getBadUDPCRCCount(c,s)`	Get the tally of incorrect UDP checksums from segment `s` in channel `c`
`modUDP_Remove.setBadUDPCRCCount(c,s,b)`	Set the tally of incorrect UDP checksums to `b` for segment `s` in channel `c`

Table 7.4–2: XML-RPC Directives for IPv4 UDP Datagram Decoding Operations

7.5 IPv4 UDP Receiver (modUdp4Receiver)

This module receives raw data from a UDP device; the packet headers are not retained. It then wraps it as untyped network data and sends it via its primary output link.

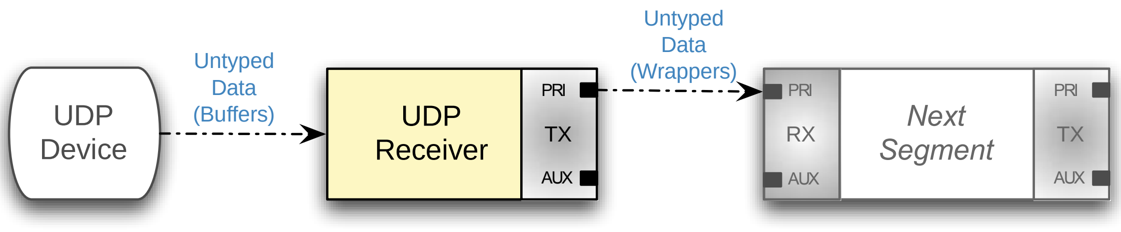

Figure 7.5-1: IPv4 UDP Receiver Architecture Sample

Figure 7.5-1 depicts a sample layout of a configured IPv4 UDP Receiver segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”; module-specific options are described at the end of this section. Figure 7.5-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
…
Devices :
{

  devUDP :
  {
    devType = "Udp4";
    address = "127.0.0.1";
    port = 6750;
  };
  …
};

Channels: {

  chanTest: {
    …
    tcpReceiver: { # segment name
      dllName = "modUdp4Receiver";
      deviceName = "devUDP";
      MRU = 1024;
      receiveMax = true;
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      immediateStart = true;
    };
  };
};
```

Figure 7.5-2: IPv4 UDP Receiver Configuration File Excerpt

Entity	**Container**	**Description**
`receiveMax = true|false;`	*segment_name* subsection	If `true`, the segment will wait to receive its full MRU before continuing; if `false`, it will continue once there is nothing else to read

Table 7.5–1: IPv4 UDP Receiver Configuration File Settings

Operation	Effect
`modUdp4Receiver.getBufferSize(c,s)`	Get the size of the buffer used to receive data from segment `s` in channel `c`
`modUdp4Receiver.setBufferSize(c,s,b)`	Set the size of the buffer used to receive data from segment `s` in channel `c` to integer `b`
`modUdp4Receiver.getReceiveMax(c,s)`	Returns whether segment `s` of channel `c` will wait to receive its full MRU before continuing
`modUdp4Receiver.setReceiveMax(c,s,r)`	If `r` is `true`, segment `s` of channel `c` will wait to receive its full MRU before continuing; if `r` is `false` it will continue when there is nothing else to read

Table 7.5–2: XML-RPC Directives for IPv4 UDP Receiver Operations

7.6 IPv4 UDP Transmitter (modUdp4Transmitter)

This module can receive any type of wrapper and write its buffer to a socket maintained by an IPv4 UDP device, after which it will be encoded as the payload of an IPv4 UDP datagram.

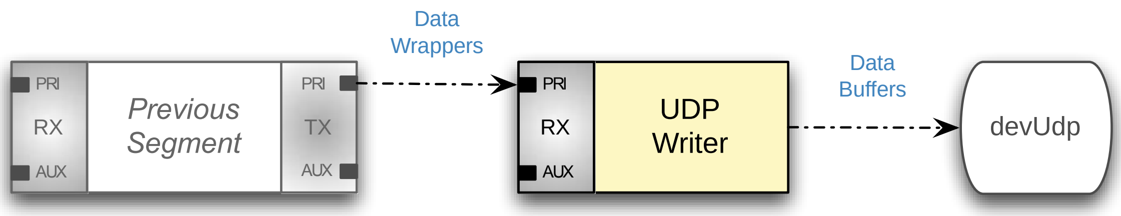

Figure 7.6-1: IPv4 UDP Transmitter Architecture Sample

Figure 7.6-1 depicts a sample layout of a configured IPv4 UDP Transmitter segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”; module-specific options are described at the end of this section. Figure 7.6-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
…
Devices :
{

  devUdp :
  {
    devType = "Udp4";
    address = "127.0.0.1";
    port = 5001;
  };
  …
};

Channels: {
  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "udpWriter", "PrimaryInput" ];
      …
      immediateStart = "true";
    };

    udpWriter: { # segment name
      dllName = "modUdp4Transmitter";
      deviceName = "devUdp";
      …
      immediateStart = true;
    };
  };
};
```

Figure 7.6-2: IPv4 UDP Transmitter Configuration File Excerpt

Entity	**Container**	**Description**
`destAddress = a.b.c.d;`	*segment_name* subsection	Set to `a.b.c.d` the IPv4 address to send datagrams to. No checking is performed as to whether any system is actually receiving at that address
`destPort = p;`	*segment_name* subsection	Set to `p` the UDP port to send datagrams to

Table 7.6–1: IPv4 UDP Transmitter Configuration File Settings

Operation	**Effect**
`modUdp4Transmitter.getDstAddr(c,s)`	Retrieve the IPv4 address to which segment `s` of channel `c` is sending datagrams
`modUdp4Transmitter.setDstAddr(c,s,a)`	Set the IPv4 address to `a` to which segment `s` of channel `c` sends datagrams
`modUdp4Transmitter.getDstPort(c,s)`	Retrieve the UDP port to which segment `s` of channel `c` is sending datagrams
`modUdp4Transmitter.setDstPort(c,s,p)`	Set the UDP port to `p` to which segment `s` of channel `c` sends datagrams

Table 7.6–2: XML-RPC Directives for IPv4 UDP Transmitter Operations

8 Network Emulation-Related Modules

8.1 Bit Error Emulation (modEmulateBitErrors)

This module accepts wrapped data of any type on its primary input. For each received data unit, it performs a random check with a configurable probability to determine whether an error will be introduced. If so, a bit is flipped at random within the unprotected section of the buffer (the area after `protectedHeaderBits` and before `protectedTrailerBits`). The module may be configured to allow more than one error per unit, so multiple random checks are performed up to the maximum configured, causing zero to `maxErrorsPerUnit` flipped bits per unit. The data wrapper is unchanged, and is sent via the output primary link to the target segment.

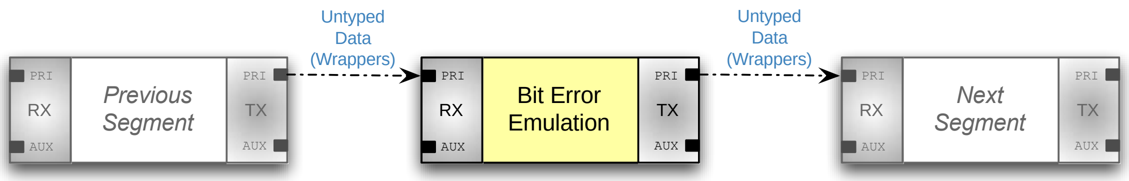

Figure 8.1-1: Bit Error Emulation Architecture Sample

Figure 8.1-1 depicts a sample layout of a configured Bit Error Emulation segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments"; module-specific options are described at the end of this section. Figure 8.1-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "bitErrorEmu", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    bitErrorEmu: { # segment name
      dllName = "modEmulateBitErrors";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      errorProbability = 0.0001;
      maxErrorsPerUnit = 1;
      protectedHeaderBits = 48;
      protectedTrailerBits = 16;
      …
      immediateStart = true;
    };

    nextSegment: {
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 8.1-2: Bit Error Emulation Configuration File Excerpt

Entity	Container	Description
`errorProbability = p;`	*segment_name* subsection	Set the chance that an error will occur in any one check to `p`, where `p` is a double-precision floating-point value between 0.0 and 1.0
`maxErrorsPerUnit = m;`	*segment_name* subsection	Set the number of checks that will be performed on every data unit to integer `m`
`timeLine = ( (t1, e1, m1), (t2, e2, m2), ... );`	*segment_name* subsection	Allow different BER/max error pairs to go into effect at different times. This is a list of time/BER/max error groupings. Time is a floating-point number representing seconds relative to the start of the emulation; BER is a floating point number between 0.0 and 1.0; and max errors per unit is an integer greater than or equal to 1. They must appear in the list in chronological order; out-of-order pairs will be rejected
`protectedHeaderBits = h;`	*segment_name* subsection	Prevent modification of the first `h` bits in each data unit
`protectedTrailerBits = t;`	*segment_name* subsection	Prevent modification of the last `t` bits in each data unit

Table 8.1–1: Bit Error Emulation Configuration File Settings

Operation	Effect
`modEmulateBitErrors.getErrorProbability(c,s,t)`	Retrieve the current bit error probability used in segment `s` of channel `c` at optional time `t` (seconds relative to the start of the emulation as a floating point number)
`modEmulateBitErrors.setErrorProbability(c,s,p,m,t)`	For segment `s` in channel `c`, set the bit error probability to `p` (a double-precision floating-point value between 0.0 and 1.0); the max errors per unit to integer `m` greater than 1; to go into effect at optional time `t` (seconds relative to the start of the simulation in floating point format)
`modEmulateBitErrors.getMaxErrorsPerUnit(c,s,t)`	Retrieve the maximum allowed errors per unit for segment `s` of channel `c` at optional time `t` (seconds relative to the start of the emulation as a floating point number)
`modEmulateBitErrors.getUnitsWithErrors(c,s)`	Retrieve the tally of units that have at least one error from segment `s` in channel `c`
`modEmulateBitErrors.setUnitsWithErrors(c,s,e)`	Set the tally of units that have at least one error to `e` for segment `s` in channel `c`
`modEmulateBitErrors.getTotalErrors(c,s)`	Retrieve the tally of all errors introduced, including multiple errors per unit, from segment `s` in channel `c`
`modEmulateBitErrors.setTotalErrors(c,s,t)`	Set the tally of all errors introduced, including multiple per unit, to `t` for segment `s` in channel `c`
`modEmulateBitErrors.getProtectedHeaderBits(c,s)`	Retrieve the current number of protected leading bits from segment `s` in channel `c`
`modEmulateBitErrors.setProtectedHeaderBits(c,s,h)`	Set the number of protected leading bits for segment `s` in channel `c` to `h`
`modEmulateBitErrors.getProtectedTrailerBits(c,s)`	Retrieve the current number of protected trailing bits from segment `s` in channel `c`
`modEmulateBitErrors.gstProtectedTrailerBits(c,s,t)`	Set the number of protected trailing bits for segment `s` in channel `c` to `h`
`modEmulateBitErrors.clearTimeLine(c,s)`	Erase all time/BER/max error sets for segment `s` in channel `c` and use the simple errorProbability / maxErrorsPerUnit values instead

Table 8.1–2: XML-RPC Directives for Bit Error Emulation Operations

8.2 Delay Emulation (modEmulateDelay)

This module accepts wrapped data of any type on its primary input. It holds the data unit for a specified amount of time, and then releases it via it primary output link to the target. The delay time can be variable if desired, resulting in an emulation of jitter, and as a result the order of the data units may optionally be modified from the order they were received in.

The delay time may either be calculated from the timestamp on the wrapper or the time that the module began processing the unit. The timestamp on the wrapper is generated when the wrapper is created, and calculating the delay from it may help to smooth out fluctuations in processing time.

The expected throughput may also be provided as a setting, and from it the module will calculate the delay-bandwidth product and resize its message queue accordingly. However, if the low or high water marks have been set manually, their values will not be changed.

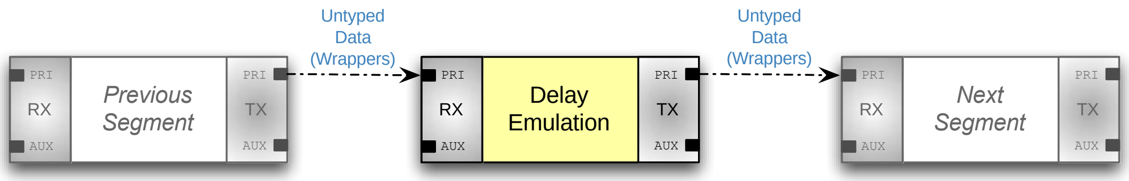

Figure 8.2-1: Delay Emulation Architecture Sample

Figure 8.2-1 depicts a sample layout of a configured Delay Emulation segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments"; module-specific options are described at the end of this section. Figure 8.2-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "delayEmu", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    delayEmu: { # segment name
      dllName = "modEmulateDelay";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      delaySeconds = 1.2;
      jitterSeconds = 0.0;
      allowJitterReorder = false;
      useTimeStamp = false;
      expectedKbits = 100000;
      …
      immediateStart = true;
    };

    nextSegment: {
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 8.2-2: Delay Emulation Configuration File Excerpt

Entity	Container	Description
`delaySeconds = s;`	*segment_name* subsection	Hold each unit for a base of `s` seconds, where `s` is a double-precision floating-point value
`jitterSeconds = j;`	*segment_name* subsection	Add a random value from 0.0 to `j` seconds onto the value of `delaySeconds` before sending the data out, where `j` is a double-precision floating-point value
`timeLine = ( (t1, d1, j1), (t2, d2, j2), ... );`	*segment_name* subsection	Allow different delay/jitter pairs to go into effect at different times. This is a list of time/delay/jitter groupings. Time is a floating-point number representing seconds relative to the start of the emulation; delay and jitter are also floating-point numbers representing seconds. They must appear in the list in chronological order; out-of-order pairs will be rejected
`allowJitterReorder = true | false;`	*segment_name* subsection	If `true`, variations in timing due to a `jitterSeconds` value greater than 0.0 can result in the reordering of data units
`expectedKbits = k;`	*segment_name* subsection	Integer `k` is used to calculate the delay-bandwidth product and set the length of the message queue to accommodate it
`useTimeStamp = true | false;`	*segment_name* subsection	If `true`, the delay is calculated from the timestamp of the wrapper; if `false`, it is calculated from the current time

Table 8.2–1: Delay Emulation Configuration File Settings

Operation	Effect
`modEmulateDelay.getDelaySeconds(c,s,t)`	Retrieve the minimum number of seconds (possibly fractional) that segment `s` in channel `c` will hold each unit before sending it at optional time `t` (seconds relative to the start of the emulation as a floating point number)
`modEmulateDelay.getJitterSeconds(c,s,t)`	Get the number of seconds (possibly fractional) that represents the maximum random amount of time added to the base delay of each data unit processed by segment `s` of channel `c` at optional time `t` (seconds relative to the start of the emulation as a floating point number)
`modEmulateDelay.setDelayAndJitter(c,s,d,j,t)`	Set delay to `d` seconds and jitter (the maximum random amount of time added to the base delay of each data unit) to `j` seconds processed by segment `s` of channel `c` at optional time `t` (seconds relative to the start of the emulation as a floating point number)
`modEmulateDelay.clearTimeLine(c,s)`	Erase all time/delay/jitter sets for segment `s` in channel `c` and use the simple delaySeconds / jitterSeconds values instead
`modEmulateDelay.getAllowJitterReorder(c,s)`	Determine whether segment `s` of channel `c` is allowing data units to be reordered due to jitter
`modEmulateDelay.setAllowJitterReorder(c,s,r)`	Use Boolean `r` to enable or disable the reordering of data units due to jitter in segment `s` of channel `c`
`modEmulateDelay.getExpectedKbits(c,s)`	Retrieve the expected maximum throughput in kilobits of segment `s` in channel `c`
`modEmulateDelay.setExpectedKbits(c,s,k)`	Set the expected maximum throughput in kilobits to integer `k` for segment `s` in channel `c`
`modEmulateDelay.getUseTimeStamp(c,s)`	Determine whether segment `s` in channel `c` is using the wrapper timestamp to compute the delay
`modEmulateDelay.setUseTimeStamp(c,s,t)`	Allow or disallow, according to Boolean `t`, segment `s` of channel `c` to use the wrapper timestamp to compute the delay

Table 8.2–2: XML-RPC Directives for Delay Emulation Operations

8.3 Rate Emulation (modEmulateRate)

This module accepts wrapped data of any type on its primary input. It relays the data immediately, but calculates the amount of time it would have taken to send it at the configured rate, and sleeps for that long before processing the next unit. The precision of the delay calculation can be improved by providing the tested throughput of the channel *without* this segment, so that value can be subtracted from the wait time. The higher the data rate, the more imprecise (usually slower) the rate becomes if the throughput value is not provided.

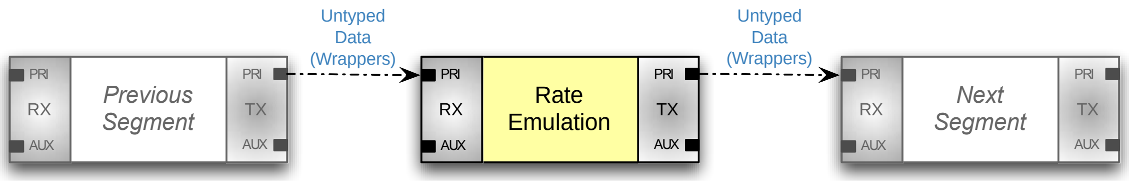

Figure 8.3-1: Rate Emulation Architecture Sample

Figure 8.3-1 depicts a sample layout of a configured Rate Emulation segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments"; module-specific options are described at the end of this section. Figure 8.3-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "rateEmu", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    rateEmu: { # segment name
      dllName = "modEmulateRate";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      rateLimit = 10000; # In kbit/s
      unlimitedThroughput = 135000; # In kbit/s
      timeLine = ( ( 120.01, 1000), ( 240.05, 2000 ), (360.07, 1000) );
      …
      immediateStart = true;
    };

    nextSegment: {
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 8.3-2: Rate Emulation Configuration File Excerpt

Entity	Container	Description

`rateLimit = r;`	*segment_name* subsection	Limit the transmission speed of the channel to `r` kilobits per second
`unlimitedThroughput = t;`	*segment_name* subsection	Set the manually tested peak throughput of the channel *without* this segment functioning to `t` kilobits per second. Improves the precision of the rate calculation
`timeLine = ( (t1, r1), (t2, r2), ... );`	*segment_name* subsection	Allow different rates to go into effect at different times. This is a list of time/rate pairs. Time is a floating-point number representing seconds relative to the start of the emulation; rate is in kilobits per second. They must appear in the list in chronological order; out-of-order pairs will be rejected

Table 8.3–1: Rate Emulation Configuration File Settings

Operation	Effect
`modEmulateRate.getRateLimit(c,s,t)`	Retrieve the transmission speed limit for segment `s` in channel `c` at optional time `t` (seconds relative to the start of the emulation as a floating point number)
`modEmulateRate.setRateLimit(c,s,r,t)`	Limit the transmission speed to `r` kilobits per second for segment `s` in channel `c` at optional time `t` (seconds relative to the start of the emulation as a floating point number)
`modEmulateRate.clearTimeLine(c,s)`	Erase all time/rate pairs for segment `s` in channel `c` and use the simple rateLimit integer instead (which may be set with the setRateLimit method or using rateLimit in the configuration file)
`modEmulateRate.getUnlimitedThroughput(c,s)`	Retrieve the value of the manually tested peak throughput used in calculating the rate for segment `s` in channel `c`
`modEmulateRate.setUnlimitedThroughput(c,s,t)`	Set the manually tested peak throughput of the channel *without* this segment functioning to `t` kilobits per second for segment `s` in channel `c`. Improves the precision of the rate calculation

Table 8.3–2: XML-RPC Directives for Rate Emulation Operations

8.4 Phase Ambiguity Emulation (modEmulatePhaseAmbiguity)

This module accepts wrapped data of any type on its primary input. It left shifts and/or inverts the bits of every octet in the data block. If there is a carry left over, it will put it at the beginning of the next incoming block; if there is no block waiting in the queue it will append the carry to the current block.

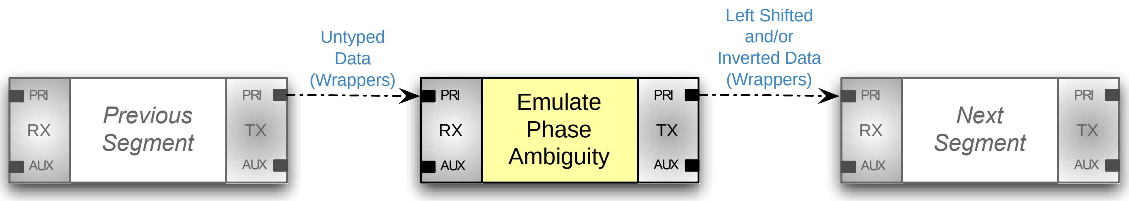

Figure 8.4-1: Phase Ambiguity Emulation Architecture Sample

Figure 8.4-1 depicts a sample layout of a configured Phase Ambiguity Emulation segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”; module-specific options are described at the end of this section. Figure 8.4-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "shifter", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    shifter: { # segment name
      dllName = "modEmulatePhaseAmbiguity";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      leftShiftBits = 3; # Range 0-7
      invert = true; # Flip all bits

      # Wait up to 1500 microseconds for a new data block before
      # sending the carry at the end of the current one:
      maxUsecsForNewData = 1500;
      …
      immediateStart = true;
    };

    nextSegment: {
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 8.4-2: Phase Ambiguity Emulation Configuration File Excerpt

Entity	Container	Description

`leftShiftBits = b;`	*segment_name* subsection	Left shift all data in each block by integer `b`, in the range 0-7
`invert = true|false;`	*segment_name* subsection	Whether to flip all the bits in the block or not
`maxUsecsForNewData = s;`	*segment_name* subsection	After processing a block, wait up to `s` microseconds for another block to appear in the queue. If nothing arrives, append the carry to the current block and send it

Table 8.4–1: Phase Ambiguity Emulation Configuration File Settings

Operation	**Effect**
`modEmulatePhaseAmbiguity.getLeftShiftBits(c,s)`	Retrieve the amount of left shift for segment `s` in channel `c`
`modEmulatePhaseAmbiguity.setLeftShiftBits(c,s,b)`	Set the number of left shift bits to integer `b` (range 0-7) for segment `s` in channel `c`
`modEmulatePhaseAmbiguity.getInversion(c,s)`	Retrieve the inversion value for segment `s` in channel `c`
`modEmulatePhaseAmbiguity.setInversion (c,s,i)`	Set the inversion value to Boolean `i` for segment `s` in channel `c`
`modEmulatePhaseAmbiguity.getMaxUsecsForNewData(c,s)`	Retrieve the maximum number of microseconds to wait for new data for segment `s` in channel `c`
`modEmulatePhaseAmbiguity.setMaxUsecsForNewData (c,s,u)`	Set the maximum number of microseconds to wait for new data to integer `u` for segment `s` in channel `c`

Table 8.4–2: XML-RPC Directives for Phase Ambiguity Emulation Operations

8.5 Phase Ambiguity Resolution (modResolvePhaseAmbiguity)

This module accepts wrapped data of any type on its primary input. It left shifts and/or inverts the bits of every octet in the data block. The value of the shift and inversion can either be set statically or dynamically. During dynamic detection, the segment searches for shifted/inverted ASMs; the contents of the ASM can be configured to any set of octets. If there is a carry left over, it will put it at the beginning of the next incoming block; if there is no block waiting in the queue it will drop the carry.

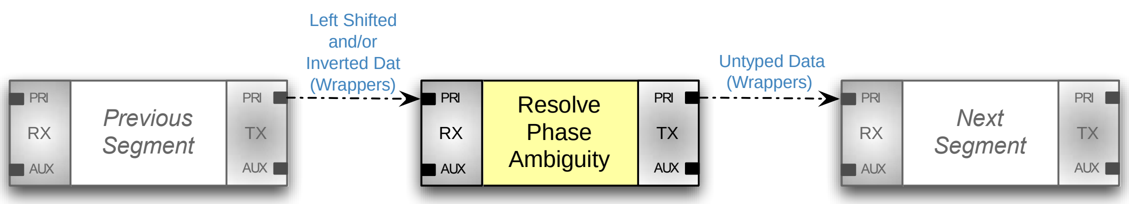

Figure 8.5-1: Phase Ambiguity Resolution Architecture Sample

Figure 8.5-1 depicts a sample layout of a configured Phase Ambiguity Resolution segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments"; module-specific options are described at the end of this section. Figure 8.5-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {
  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "aligner", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    aligner: { # segment name
      dllName = "modResolvePhaseAmbiguity";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      leftShiftBits = 3; # The left shift of incoming data. Range 0-7
      invert = true; # Flip all bits
      detectShift = false; # If true, leftShiftBits and invert are ignored

      # Used only if detectShift is true:
      markerPattern = [ 0x1A, 0xCF, 0xFC, 0x1D ];

      # Wait up to 1500 microseconds for a new data block before
      # dropping the carry:
      maxUsecsForNewData = 1500;

      # Allow 1 bit error when comparing shifted traffic to the shifted
      # ASM. Should be kept low or detection will fail.
      allowedMarkerBitErrors = 1;
      …
      immediateStart = true;
    };

    nextSegment: {
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 8.5-2: Phase Ambiguity Resolution Configuration File Excerpt

Entity	Container	Description

`leftShiftBits = b;`	*segment_name* subsection	Represents the left shift that is expected in incoming data, so **right shift** all data in each block by integer *b*, in the range 0-7
`invert = true|false;`	*segment_name* subsection	Represents the expected inversion in incoming data, so if **`true`**, flip all the bits in the block
`detectShift = true|false;`	*segment_name* subsection	If `true`, ignore the values of `leftShiftBits` and `invert`, and look for modified ASMs to determine the values automatically
`markerPattern = [ a, b, c, … ];`	*segment_name* subsection	Set the sync marker pattern to an array of integers
`allowedMarkerBitErrors = e;`	*segment_name* subsection	Set to *e* the number of bit errors that can be encountered in an ASM to still be considered valid
`maxUsecsForNewData = s;`	*segment_name* subsection	After processing a block, wait up to *s* microseconds for another block to appear in the queue. If nothing arrives, drop the carry

Table 8.5–1: Phase Ambiguity Resolution Configuration File Settings

Operation	Effect
`modResolvePhaseAmbiguity.getLeftShiftBits(c,s)`	Retrieve the amount of left shift for segment `s` in channel `c`
`modResolvePhaseAmbiguity.setLeftShiftBits(c,s,b)`	Set the number of left shift bits to integer *b* (range 0-7) for segment `s` in channel `c`
`modResolvePhaseAmbiguity.getInversion(c,s)`	Retrieve the inversion value for segment `s` in channel `c`
`modResolvePhaseAmbiguity.setInversion (c,s,i)`	Set the inversion value to Boolean *i* for segment `s` in channel `c`
`modResolvePhaseAmbiguity.getMaxUsecsForNewData(c,s)`	Retrieve the maximum number of microseconds to wait for new data for segment `s` in channel `c`
`modResolvePhaseAmbiguity.getDetectShift(c,s)`	Retrieve whether segment `s` in channel `c` is automatically detecting the left shift/inversion or not

`modResolvePhaseAmbiguity.setDetectShift(c,s,d)`	Set whether segment `s` in channel `c` is automatically detecting the left shift/inversion or not by providing Boolean `d`
`modResolvePhaseAmbiguity.getASM(c,s)`	Retrieve an array of bytes that represents the current sync marker used in segment `s` of channel `c`
`modResolvePhaseAmbiguity.setASM(c,s)`	Set the sync marker pattern in segment `s` of channel `c` to `m`, an array of bytes. Ignored if the segment is not auto-detecting the shift/inversion
`modResolvePhaseAmbiguity.getAllowedMarkerBitErrors(c,s)`	Get the number of allowed bit errors per ASM in segment `s` of channel `c`
`modResolvePhaseAmbiguity.setAllowedMarkerBitErrors(c,s,e)`	Set the number of allowed bit errors per ASM in segment `s` of channel `c` to integer `e`
`modResolvePhaseAmbiguity.setMaxUsecsForNewData (c,s,u)`	Set the maximum number of microseconds to wait for new data to integer *u* for segment `s` in channel `c`

Table 8.5–2: XML-RPC Directives for Phase Ambiguity Resolution Operations

9 Assorted CCSDS-Specific Modules

9.1 ASM Attachment (modASM_Add)

The ASM attachment function accepts any type of network data of unspecified length and prepends a sync marker of the configured type. The modified data unit is sent out via the primary output link.

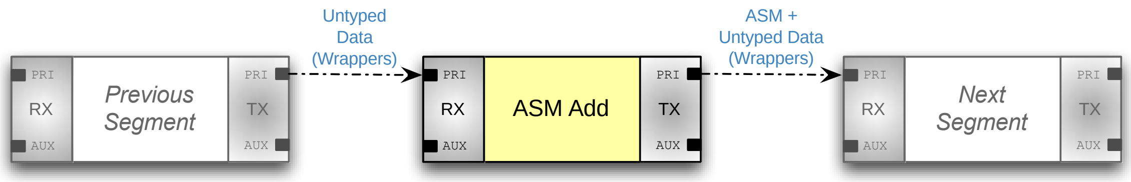

Figure 9.1-1: ASM Attachment Architecture Sample

Figure 9.1-1 depicts a sample layout of a configured ASM Attachment segment. This module derives all of its configuration file settings and XML-RPC directives from those described in section 2.4, "Modular Segments". Figure 9.1-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "asmAdd", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    asmAdd: {
      dllName = "modASM_Add";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      markerPattern = [ 0x1A, 0xCF, 0xFC, 0x1D ];
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 9.1-2: ASM Attachment Configuration File Example

Entity	Container	Description
`markerPattern = [ a, b, c, … ];`	*segment_name* subsection	Set the sync marker pattern to attach to an array of integers

Table 9.1–1: ASM Attachment Configuration File Settings

Operation	Effect
`modASM_Add.getASM(c,s)`	Retrieve an array of bytes that represents the current sync marker used in segment `s` of channel `c`
`modASM_Add.setASM(c,s,m)`	Set the sync marker pattern in segment `s` of channel `c` to `m`, an array of bytes
`modASM_Add.getASMCount(c,s)`	Get the grand total of all ASMs located for segment `s` in channel `c` as a 64-bit integer (I8)
`modASM_Add.setASMCount(c,s,a)`	Set the grand total of all ASMs located for segment `s` in channel `c` to 64-bit integer (I8) `a`
`modASM_Add.getASMValidCount(c,s)`	Get the tally of all ASMs found in their expected position for segment `s` in channel `c` as a 64-bit integer (I8)
`modASM_Add.setASMValidCount(c,s,v)`	Set the tally of all ASMs found in their expected position for segment `s` in channel `c` to 64-bit integer (I8) `v`
`modASM_Add.getASMMissedCount(c,s)`	Get the tally of all ASMs not found in their expected position for segment `s` in channel `c` as a 64-bit integer (I8)
`modASM_Add.setASMMissedCount(c,s,m)`	Set the tally of all ASMs not found in their expected position for segment `s` in channel `c` to 64-bit integer (I8) `m`
`modASM_Add.getASMDiscoveredCount(c,s)`	Get the tally of all ASMs found only after searching for segment `s` in channel `c` as a 64-bit integer (I8)
`modASM_Add.setASMDiscoveredCount(c,s,d)`	Set the tally of all ASMs found only after searching for segment `s` in channel `c` to 64-bit integer (I8) `d`

Table 9.1–2: XML-RPC Directives for ASM Attachment Operations

9.2 ASM Detachment (modASM_Remove)

The ASM detachment function accepts any type of network data units of uniform length. It must be configured with two things: the type of sync marker to look for and the expected length of each data unit. If an ASM is found where expected and there is enough data in the buffer, a new unit of the correct length is sent via the primary output link. If the ASM is missing, the check occurs again at the next octet and so on until the marker is found or the buffer is too small to hold one.

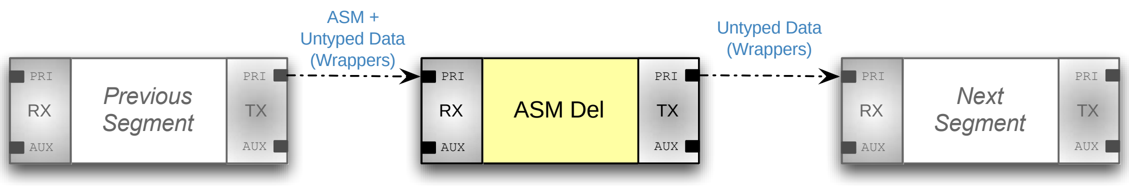

Figure 9.2-1: ASM Detachment Architecture Sample

Figure 9.2-1 depicts a sample layout of a configured ASM Attachment segment. This module derives all of its configuration file settings and XML-RPC directives from those described in section 2.4, “Modular Segments”. Figure 3.5-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "asmDel", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    asmDel: {
      dllName = "modASM_Remove";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      markerPattern = [ 0x1A, 0xCF, 0xFC, 0x1D ];
      expectedUnitLength = 250;
      allowedMakerBitErrors = 5;
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 9.2-2: ASM Detachment Configuration File Example

Entity	Container	Description
`markerPattern = [ a, b, c, … ];`	*segment_name* subsection	Set the sync marker pattern to an array of integers
`expectedUnitLength = l;`	*segment_name* subsection	Set the expected length of all units to *l* (does not include the ASM)
`allowedMarkerBitErrors = e;`	*segment_name* subsection	Set to *e* the number of bit errors that can be encountered in an ASM to still be considered valid

Table 9.2–1: ASM Detachment Configuration File Settings

Operation	Effect
`modASM_Remove.getASM(c,s)`	Retrieve an array of bytes that represents the current sync marker used in segment `s` of channel `c`
`modASM_Remove.setASM(c,s,m)`	Set the sync marker pattern in segment `s` of channel `c` to `m`, an array of bytes
`modASM_Remove.getExpectedUnitLength(c,s)`	Get the expected length of all units in segment `s` of channel `c`
`modASM_Remove.setExpectedUnitLength(c,s,u)`	Set the expected length of all units to integer `u` in segment `s` of channel `c` to `m`, an array of bytes
`modASM_Remove.getAllowedMarkerBitErrors(c,s)`	Get the number of allowed bit errors per ASM in segment `s` of channel `c`
`modASM_Remove.setAllowedMarkerBitErrors(c,s,e)`	Set the number of allowed bit errors per ASM in segment `s` of channel `c` to integer `e`

Table 9.2–2: XML-RPC Directives for ASM Detachment Operations

9.3 Reed-Solomon Encoding (modRSEncode)

This module accepts 8-bit aligned network data of unspecified length (up to a configured maximum). It generates interleaved parity-check symbols that are appended to the received data unit. The method for encoding is described in [7]. It allows for either 8 or 16 errors per code word (called *E* in [7] and maxErrorsPerCodeWord by modRSEncode) and interleaving depths of 1 (i.e. no interleaving), 2, 3, 4, 5, and 8 (called *I* in [7] and interleavingDepth by modRSEncode). However, modRSEncode does not place restrictions on interleaving depth. The modified data unit is sent out via the primary output link.

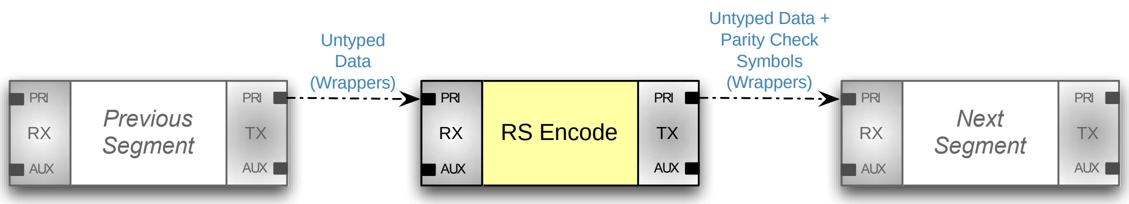

Figure 9.3-1: Reed-Solomon Encoder Architecture Sample

Figure 9.3-1 depicts a sample layout of a configured Reed-Solomon encoder segment. This module derives several of its configuration file settings and XML-RPC directives from those described in section 2.4, “Modular Segments”.

The module encodes the entire received unit, rather than truncating it and/or waiting for additional data. If the unit cannot fit in the message space provided by the configuration, it is dropped.

A code word is a portion of the received unit (the "message") with parity symbols attached; for 8-bit RS encoding code word length is always 255. To select the length of the largest unit than can fit into multiple interleaved code words, first find the message length for one code word as 255 – 2**E* (e.g. for *E* = 16, the message length per code word is 223). Next, multiply this by a value for *I* large enough to contain any incoming unit (e.g. for 512-octet frames, with *E* = 16, select *I* = 3, so that units of up to 669 octets may be encoded). The size of the encoded unit will be its original length plus 2**E***I*.

Figure 9.3-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "asmAdd", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    RSEncode: {
      dllName = "modRSEncode";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      maxErrorsPerCodeword = 16;
      interleavingDepth = 3;
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 9.3-2: Reed-Solomon Encoder Configuration File Example

Entity	Container	Description
`interleavingDepth = i;`	*segment_name* subsection	The number of codewords to use to generate parity check symbols
`maxErrorsPerCodeword = e;`	*segment_name* subsection	The maximum number of errors that can be corrected per codeword; *e* must be either 8 or 16

Table 9.3–1: Reed-Solomon Encoder Configuration File Settings

Operation	Effect
`modRSEncode.getMaxErrorsPerCodeWord(c,s)`	Retrieve an integer representing the maximum errors allowed per code word for segment `s` of channel `c`
`modRSEncode.setMaxErrorsPerCodeWord(c,s,e)`	Set the maximum errors allowed per code word for segment `s` of channel `c` to integer `e`. Call `modRSEncode.rebuildEncoder(c,s)` afterwards
`modRSEncode.getInterleavingDepth(c,s)`	Retrieve an integer representing the interleaving depth for segment `s` of channel `c`
`modRSEncode.setInterleavingDepth(c,s,i)`	Set the interleaving depth for segment `s` of channel `c` to integer `i`. Call `modRSEncode.rebuildEncoder(c,s)` afterwards
`modRSEncode.rebuildEncoder(c,s)`	If maxErrorsPerCodeWord or interleavingDepth have been modified after segment `s` of channel `c` has been initialized, this function must be called for the changes to be effective

Table 9.3–2: XML-RPC Directives for Reed-Solomon Encoder Operations

9.4 Reed-Solomon Decoding (modRSDecode)

This module accepts 8-bit aligned network data of unspecified length (up to a configured maximum). It uses parity-check symbols that are appended to the received data unit to correct errors in the message. The method for decoding is described in [7]. It allows for either 8 or 16 errors per code word (called *E* in [7] and maxErrorsPerCodeWord by modRSDecode) and interleaving depths of 1 (i.e. no interleaving), 2, 3, 4, 5, and 8 (called *I* in [7] and interleavingDepth by modRSDecode). However, modRSDecode does not place restrictions on interleaving depth. A new data unit, consisting of an untyped wrapper around the corrected message and minus the parity checks symbols is sent via the primary output link.

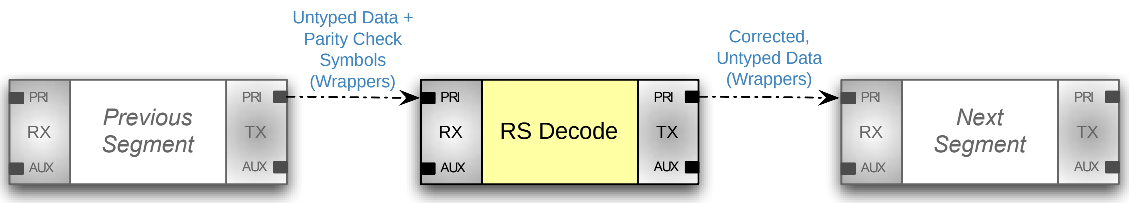

Figure 9.4-1: Reed-Solomon Decoder Architecture Sample

Figure 9.4-1 depicts a sample layout of a configured Reed-Solomon encoder segment. This module derives several of its configuration file settings and XML-RPC directives from those described in section 2.4, “Modular Segments”. Figure 9.4-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels:
{
  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "asmAdd", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    RSEncode: {
      dllName = "modRSEncode";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      maxErrorsPerCodeword = 16;
      interleavingDepth = 3;
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 9.4-2: Reed-Solomon Decoder Configuration File Example

Entity	**Container**	**Description**
`interleavingDepth = i;`	*segment_name* subsection	The number of codewords to correct
`maxErrorsPerCodeword = e;`	*segment_name* subsection	The maximum number of errors that can be corrected per codeword; *e* must be either 8 or 16

Table 9.4–1: Reed-Solomon Decoder Configuration File Settings

Operation	Effect
`modRSDecode.getMaxErrorsPerCodeWord(c,s)`	Retrieve an integer representing the maximum errors allowed per code word for segment `s` of channel `c`
`modRSDecode.setMaxErrorsPerCodeWord(c,s,e)`	Set the maximum errors allowed per code word for segment `s` of channel `c` to integer `e`. Call `modRSDecode.rebuildDecoder(c,s)` afterwards
`modRSDecode.getInterleavingDepth(c,s)`	Retrieve an integer representing the interleaving depth for segment `s` of channel `c`
`modRSDecode.setInterleavingDepth(c,s,i)`	Set the interleaving depth for segment `s` of channel `c` to integer `i`. Call `modRSDecode.rebuildDecoder(c,s)` afterwards
`modRSDecode.getCorrectedErrorCount(c,s)`	Get the tally of corrected errors as a 64-bit (I8) integer from segment `s` in channel `c`
`modRSDecode.setCorrectedErrorCount(c,s,e)`	Set the tally of corrected errors for segment `s` in channel `c` to 64-bit (I8) integer `e`. This is mostly useful for resetting it to zero
`modRSDecode.rebuildDecoder(c,s)`	If maxErrorsPerCodeWord or interleavingDepth have been modified after segment `s` of channel `c` has been initialized, this function must be called for the changes to be effective

Table 9.4–2: XML-RPC Directives for Reed-Solomon Decoder Operations

9.5 Encapsulation Packet Encoding Service (modEncapPkt_Add)

This service accepts a wrapper type limited to those specified in the CCSDS Space Link Identifiers recommended standard (currently only IPv4 datagram wrappers are auto-detected). It incorporates its buffer into an Encapsulation Packet wrapper, by default including an Internet Protocol Extension header (IPE), and sends that out via the primary output link. Usually the target is a Multiplexing PDU encoding segment.

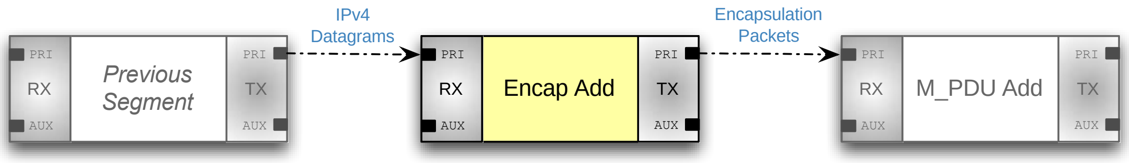

Figure 9.5-1: Encapsulation Packet Encoding Service Architecture Sample

Figure 9.5-1 depicts a sample layout of a configured Encapsulation Packet encoding segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”. Figure 9.5-2 contains configuration file excerpts that would be used to generate architecture similar to the diagram above.

```
Channels: {
  chanTest: {
    …
    encapAdd :
    {
      dllName = "modEncapPkt_Add";
      primaryOutput = [ "mpduAdd", "PrimaryInput" ];
      immediateStart = true;
    };

    mpduAdd:
    {
      dllName = "modAOS_M_PDU_Add";
      immediateStart = true;
      …
    };
    …
  };
};
```

Figure 9.5-2: Encapsulation Packet Encoding Service Configuration File Excerpt

Explanations of settings with behavior particular to this module are found in Table 9.5–1: Encapsulation Packet Encoding Service Configuration File Settings and Table 9.5–2: XML-RPC Directives for Encapsulation Packet Service Encoding Operations. Although there are a large number of customizable settings, if the segment is receiving supported wrapper types (i.e. currently only IPv4 datagrams) and using the Internet Protocol Extension, default values can be used for all of them.

Entity	Container	Description
`supportIPE = true | false;`	*segment_name* subsection	If `true` (default), use the Internet Protocol Extension (IPE) shim to identify the contents of the data field. If `false`, the protocol ID field or mission-customized methods are used.
`IPE = `*`i`*`L;`	*segment_name* subsection	Override IPE auto-detection and set the IPE to 64-bit value *`i`* in each packet. The "`L`" (representing a long integer) must appear in the configuration file immediately following *`i`* or an error will result.
`protocol = `*`p`*`;`	*segment_name* subsection	Set the value of the Protocol ID field to *`p`*, an integer between 0 and 7.
`lengthOfLength = `*`l`*`;`	*segment_name* subsection	Set the value of the Length of Length field to *`l`*, an integer between 0 and 3. Defaults to 2.
`userDefinedField = `*`u`*`;`	*segment_name* subsection	Set the value of the User Defined Field to *`u`*, an integer between 0 and 15. Defaults to 0.
`protocolIDExtension = `*`e`*`;`	*segment_name* subsection	Set the value of the Protocol ID Extension field to *`e`*, an integer between 0 and 15. Defaults to 0.
`CCSDSDefinedField = `*`c`*`;`	*segment_name* subsection	Set the value of the CCSDS Defined Field to *`c`*, an integer between 0 and 65,535. Note: This field is supposed to be reserved for future use and should be set to "all zeroes", which is the default.

Table 9.5–1: Encapsulation Packet Encoding Service Configuration File Settings

Operation	Effect
`modEncapPkt_Add.setSupportIPE(c,s,e)`	Enable or disable Internet Protocol Extension (IPE) support in segment `s` of channel `c`, depending on whether `e` is true or false, respectively. Defaults to true.
`modEncapPkt_Add.getSupportIPE(c,s)`	Fetch the state of IPE support for segment `s` in channel `c`.
`modEncapPkt_Add.setIPE(c,s,i)`	For segment `s` in channel `c` in which the IPE is supported, override IPE auto-selection and set the value to 64-bit (I8 in XLM-RPC terms) `i`.
`modEncapPkt_Add.getIPE(c,s)`	Fetch the 64-bit (I8) value of the overridden IPE from segment `s` in channel `c`. If this value is not in use by a segment that supports IPE, it will return an even integer (2) because all valid IPE values must be odd.
`modEncapPkt_Add.unsetIPE(c,s)`	Cause segment `s` of channel `c` to stop using a custom IPE value.
`modEncapPkt_Add.setProtocol(c,s,p)`	For segment `s` in channel `c`, override protocol ID auto-selection and use a custom integer `i` between 0 and 7.
`modEncapPkt_Add.getProtocol(c,s)`	Fetch the Protocol ID integer from segment `s` in channel `c`.
`modEncapPkt_Add.unsetProtocol(c,s)`	Cause segment `s` of channel `c` to stop using a custom Protocol ID value.
`modEncapPkt_Add.setLengthOfLength(c,s,l)`	Set the Length of Length field for segment `s` in channel `c` to `l`, an integer between 0 and 3. Defaults to 2.
`modEncapPkt_Add.getLengthOfLength(c,s)`	Get the value of the Length of Length field for segment `s` in channel `c`.
`modEncapPkt_Add.setUserDefinedField(c,s,u)`	Set the User Defined Field for segment `s` in channel `c` to `u`, an integer between 0 and 15. Defaults to 0.
`modEncapPkt_Add.getUserDefinedField(c,s)`	Get the value of the User Defined Field for segment `s` in channel `c`.
`modEncapPkt_Add.setProtocolIDExtension(c,s,x)`	Set the Protocol ID Extension field for segment `s` in channel `c` to `x`, an integer between 0 and 15. Defaults to 0.

`modEncapPkt_Add.getProtocolIDExtension(c,s)`	Get the value of the Protocol ID Extension field for segment `s` in channel `c`.
`modEncapPkt_Add.setCCSDSDefinedField(c,s,d)`	Set the CCSDS Defined Field for segment `s` in channel `c` to `d`, an integer between 0 and 65,535. Defaults to 0.
`modEncapPkt_Add.getCCSDSDefinedField(c,s)`	Get the value of the CCSDS Defined Field for segment `s` in channel `c`.

Table 9.5–2: XML-RPC Directives for Encapsulation Packet Service Encoding Operations

9.6 Encapsulation Packet Decoding Service (modEncapPkt_Remove)

This service accepts an Encapsulation Packet wrapper, extracts the data field buffer, wraps it as the correct type (currently only IPv4 datagram wrappers are auto-detected), and sends it on via the primary output link. The Internet Protocol Extension header (IPE) is used by default to determine the type of the buffer payload.

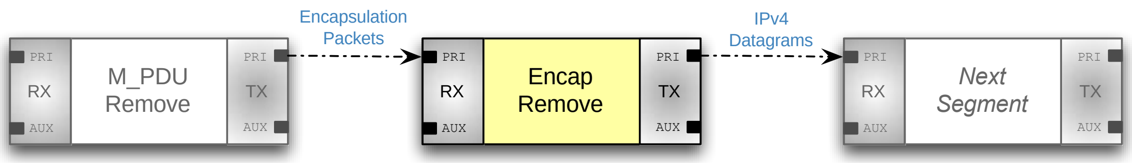

Figure 9.6-1: Encapsulation Packet Decoding Service Architecture Sample

Figure 9.6-1 depicts a sample layout of a configured Encapsulation Packet decoding segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”. Figure 9.6-2 contains configuration file excerpts that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    encapDel: { # segment name
      dllName = "modEncapPkt_Remove";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
      immediateStart = true;
    };

    mpduDel: { # segment name
      dllName = "modAOS_M_PDU_Remove";
      primaryOutput = [ "encapDel", "PrimaryInput" ];
      immediateStart = true;
    };
    …
  };
};
```

Figure 9.6-2: Encapsulation Packet Decoding Service Configuration File Excerpt

Explanations of settings with behavior particular to this module are found in Table 9.6–1: Encapsulation Packet Decoding Service Configuration File Settings and Table 9.6–2: XML-RPC Directives for Encapsulation Packet Decoding Service Operations.

Entity	**Container**	**Description**
`supportIPE = true | false;`	*segment_name* subsection	If `true` (default), use the Internet Protocol Extension (IPE) shim to identify the contents of the data field. If `false`, the protocol ID field or mission-customized methods are used.

Table 9.6–1: Encapsulation Packet Decoding Service Configuration File Settings

Operation	**Effect**
`modEncapPkt_Remove.setSupportIPE(c,s,i)`	Enable or disable Internet Protocol Extension (IPE) support in segment `s` of channel `c`, depending on whether `e` is true or false, respectively. Defaults to true.
`modEncapPkt_Remove.getSupportIPE(c,s)`	Fetch the state of IPE support for segment `s` in channel `c`.

Table 9.6–2: XML-RPC Directives for Encapsulation Packet Decoding Service Operations

9.7 TM Pseudo-Randomization (modPseudoRandomize)

This service accepts any type of wrapper on its primary input, then exclusive-ORs its buffer with the pre-calculated bit sequence defined in [6]. If the buffer has previously had this operation applied to it, the result will be the original message. The wrapper with the altered buffer is sent via the primary output.

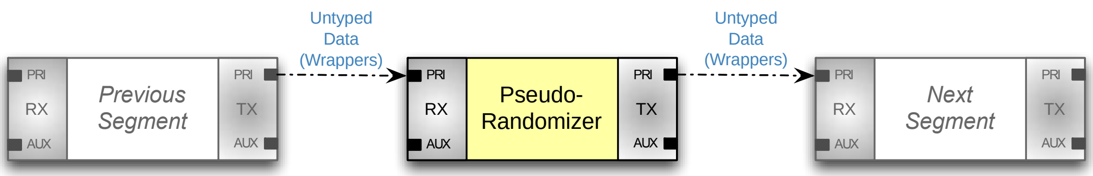

Figure 9.7-1: TM Pseudo-Randomization Architecture Sample

Figure 9.7-1 depicts a sample layout of a configured TM Pseudo-Randomizer segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments". Figure 9.7-2 contains configuration file excerpts that would be used to generate architecture similar to the diagram above.

```
Channels: {

  chanTest: {
    …
    pseudoRandom: { # segment name
      dllName = "modPseudoRandomize";
      primaryOutput = [ "Next Segment", "PrimaryInput" ];
    };
      …
  };
};
```

Figure 9.7-2: TM Pseudo-Randomization Configuration File Excerpt

10 Utility Modules

10.1 Extractor (modExtractor)

This module accepts wrapped data of any type on its primary input. A specified number of octets are removed from the header and the trailer of each received unit, and it is then sent via the output primary link to the target segment. If the original unit is too short it is sent unchanged.

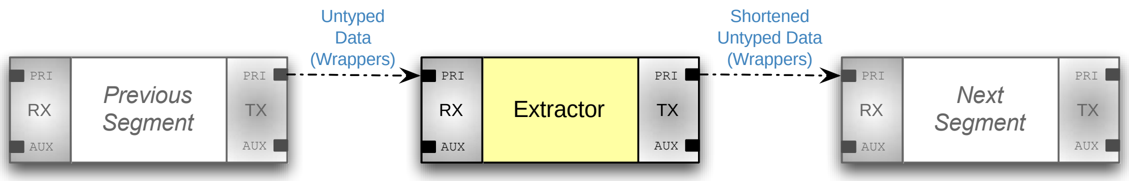

Figure 10.1-1: Extractor Architecture Sample

Figure 10.1-1 depicts a sample layout of a configured Extractor segment. This module derives most of its configuration file settings and XML-RPC directives from those described in chapter 2.4, “Modular Segments”; module-specific options are described at the end of this section. Figure 10.1-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {
  chanTest: {
    …
    previousSegment: {
      primaryOutput = [ "extractor", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    extractor: { # segment name
      dllName = "modExtractor";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      headerLength = 8;
      trailerLength = 2;
      …
      immediateStart = true;
    };

    nextSegment: {
      …
      immediateStart = true;
    };
    …
  };
};
```

Figure 10.1-2: Extractor Configuration File Excerpt

Entity	**Container**	**Description**
`headerLength = h;`	*segment_name* subsection	Remove `h` octets from the start of the received units
`trailerLength = t;`	*segment_name* subsection	Remove `t` octets from the end of the received units

Table 10.1–1: Extractor Configuration File Settings

Operation	Effect
`modExctractor.getHeaderLength(c,s)`	Retrieve the number of octets removed from the start of units received by segment `s` of channel `c`
`modExctractor.setHeaderLength(c,s,h)`	Set to `h` the number of octets removed from the start of units received by segment `s` of channel `c`
`modExctractor.getTrailerLength(c,s)`	Retrieve the number of octets removed from the end of units received by segment `s` of channel `c`
`modExctractor.setTrailerLength(c,s,t)`	Set to `t` the number of octets removed from the end of units received by segment `s` of channel `c`
`modExctractor.getHeaderOctetCount(c,s)`	Retrieve the tally of octets removed from all headers from segment `s` of channel `c`
`modExctractor.setHeaderOctetCount(c,s,h)`	Set to `h` tally of octets removed from all headers from segment `s` of channel `c`
`modExctractor.getTrailerOctetCount(c,s)`	Retrieve the tally of octets removed from all trailers from segment `s` of channel `c`
`modExctractor.setTrailerOctetCount(c,s,t)`	Set to `t` tally of octets removed from all trailers from segment `s` of channel `c`
`modExctractor.getStubCount(c,s)`	Get the number of units that were too short to be extracted in segment `s` of channel `c`
`modExctractor.setStubCount(c,s,m)`	Set to `m` the number of units that were too short to be extracted in segment `s` of channel `c`

Table 10.1–2: XML-RPC Directives for Extractor Operations

10.2 Splitter (modSplitter)

This module accepts wrapped data of any type on its primary input. A shallow copy of each unit is created; the original is sent on via the primary output. The copy is sent out via the auxiliary output. The copy retains only a generic wrapper of the buffer, so receiving segments expecting a specific wrapper type (for example, an AOS Transfer Frame) will report an error. One intended use of this module is logging.

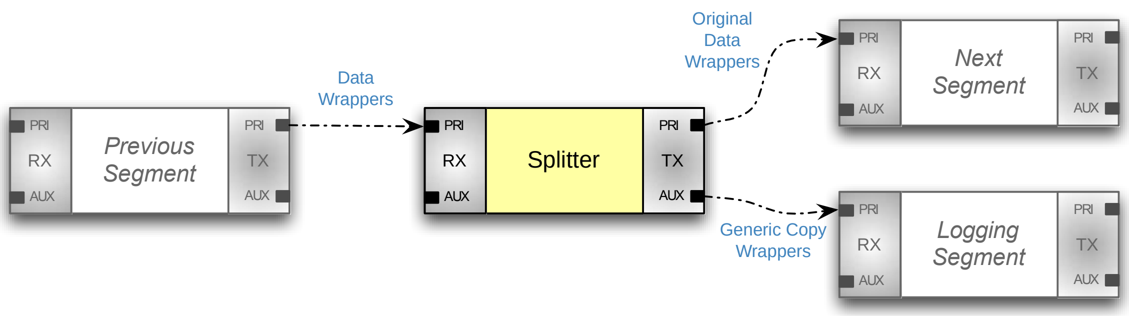

Figure 10.2-1: Splitter Architecture Sample

Figure 10.2-1 depicts a sample layout of a configured Splitter segment. This module derives all of its configuration file settings and XML-RPC directives from those described in chapter 2.4, "Modular Segments"; module-specific options are described at the end of this section. Figure 10.2-2 contains a configuration file excerpt that would be used to generate architecture similar to the diagram above.

```
Channels: {
  chanTest: {
    previousSegment: {
      primaryOutput = [ "splitter", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    splitter: { # segment name
      dllName = "modSplitter";
      primaryOutput = [ "nextSegment", "PrimaryInput" ];
      auxOutput = [ "loggingSegment", "PrimaryInput" ];
      …
      immediateStart = true;
    };

    loggingSegment: {
      …
      immediateStart = true;
    };

    nextSegment: {
      …
      immediateStart = true;
    };

    …
  };
};
```

Figure 10.2-2: Extractor Configuration File Excerpt

11 Appendix A – Configuration File Grammar

The Channel Emulator configuration files are parsed by the "libconfig" C/C++ library, found at http://www.hyperrealm.com/libconfig/. The remainder of Appendix A – Configuration File is a copy of chapter 5 of the libconfig HTML manual, titled "Configuration File Grammar":

Below is the BNF grammar for configuration files. Comments are not part of the grammar, and hence are not included here.

```
configuration = setting-list | empty
setting-list = setting | setting-list setting
setting = name (":" | "=") value ";"
value = scalar-value | array | list | group
value-list = value | value-list "," value
scalar-value = boolean | integer | integer64 | hex | hex64 | float
| string
scalar-value-list = scalar-value | scalar-value-list "," scalar-value
array = "[" (scalar-value-list | empty) "]"
list = "(" (value-list | empty) ")"
group = "{" (setting-list | empty) "}"
empty =
```

Terminals are defined below as regular expressions:

```
boolean        ([Tt][Rr][Uu][Ee])|([Ff][Aa][Ll][Ss][Ee])
string         \"([^\"\\]|\\.)*\"
name           [A-Za-z\*][-A-Za-z0-9 \*]*
integer        [-+]?[0-9]+
integer64      [-+]?[0-9]+L(L)?
hex            0[Xx][0-9A-Fa-f]+
hex64          0[Xx][0-9A-Fa-f]+L(L)?
float          ([-+]?([0-9]*)?\.[0-9]*([eE][-+]?[0-9]+)?)|([-+]([0-
               9]+)(\.[0-9]*)?[eE][-+]?[0-9]+)
```