DEVICE DRIVER DESIGN FOR A SINGLE AND QUAD T1/E1 TRANSCEIVER

By

Randeep S. Gakhal
Software Design Engineer,
PMC-Sierra, Inc.

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF APPLIED SCIENCE
in the School of Engineering Science

SIMON FRASER UNIVERSITY
March 17, 2001

APPROVAL

Name: Randeep S. Gakhal

Degree: Bachelor of Applied Science

Title of Thesis: Device Driver Design for a Single and Quad T1/E1 Transceiver

_____

Dr. John Jones
Director
School of Engineering Science, SFU

Examining Committee:

Chair and Academic Supervisor: _____

Dr. Ljiljana Trajkovic
Associate Professor
School of Engineering Science, SFU

Technical Supervisor: _____

Chris Topp
Manager, Software Development
PMC-Sierra, Inc.

Committee Member: _____

Kurt Fennig
Leader, Software Development
PMC-Sierra, Inc.

Date Approved: _____

ABSTRACT

This thesis project develops a device driver for a single and quad channel T1 and E1 transceiver integrated circuit device. Our device driver provides a software building block for an application developer creating T1 or E1 networking equipment to rapidly integrate the T1 and E1 transceivers into their system.

The design of our device driver ensures that the feature requirements of application developers are met and that the device driver is highly portable between different real-time operating systems and computer architectures. In development of the device driver for the single and quad T1 and E1 transceiver devices, we applied an in-depth understanding of T1 and E1 networking concepts, the device hardware, and formal software development methodologies.

Within the device driver architecture, we develop a series of software blocks that encapsulate the fundamental operations of the device, including a generic interrupt servicing architecture. By using a structured software architecture that maps the device features into carefully designed software blocks and by isolating code that is platform dependent, we meet the requirements of functionality, device abstraction, and portability.

ACKNOWLEDGEMENTS

I would like to thank my technical supervisor, Chris Topp, for providing me with the technical guidance and support required for me to handle an endeavour of this calibre. Thank you for your encouragement and patience without which this project would not have been a success.

I would also like to thank my academic supervisor, Dr. Ljiljana Trajkovic, for her feedback on my work. Your critique was invaluable in writing this thesis.

Finally, I would like to thank the members of the Software Product Development team at PMC-Sierra for being wonderful colleagues, mentors, and friends.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

T1 networks can be found everywhere carrying data and voice within the access network where a Wide Area Network (WAN) meets a Local Area Network (LAN). T1 streams are multiplexed into greater bandwidth aggregate data streams transported on high-speed optical transport networks. Despite being a technology developed in the 1960's, T1 is ubiquitous today and it is the basic building block in data and voice transport standards.

The telecommunications semiconductor industry today is dominated by highly specialized and feature rich devices that are dedicated to a single type of application. The main objective of this project is to develop a software device driver for both a single and four channel T1/E1 transceiver device. The device driver provides a software application developer with an interface that simplifies usage of the device, allowing the developer to take advantage of the many features of the device and to expedite the software application development process.

The thesis is organized as follows: we begin with a brief introduction to T1 and E1 networking concepts in Section 2 followed by an overview of the hardware for which we are developing the device driver in Section 3. In Section 4, we provide an overview of the device driver requirements and an introduction to the design and architecture of the device driver. Section 4 to Section 9 then proceed with a detailed discussion of the various device driver application programming interface components. Finally, we conclude by describing the test plan employed to validate the device driver and the test results in Section 10.

2 T1 and E1 Networking Concepts

In this section we provide a summary of basic T1 and E1 networking concepts that are required to understand the device driver. These concepts are used throughout the thesis to describe the development and testing of the device driver software.

2.1 Digital Signal Level 0 (DS0)

Digital signal level 0 (DS0) is the basic element in T1 and E1 networks and dates back to the 1960's when it was designed to carry voice signals. The highest concentration of spectral power in a human voice signal is in the frequencies up to 1000 Hz with the essential bandwidth of the signal in the neighbourhood of 3000 Hz [10]. In the DS0 format, the analog voice signal is sampled at a frequency of 8000 Hz. The basic process of deriving a DS0 pulse code modulation (PCM) signal is shown in Figure 1.

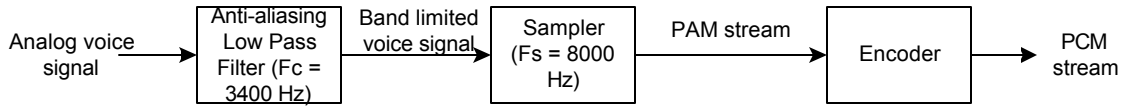

Figure 1 – Generating a DS0 PCM signal.

The voice signal first passes through an anti-aliasing filter that bandlimits the signal to 3.4 kHz. The bandlimited signal is then sampled at a rate of 8000 Hz, a frequency larger than the Nyquist rate of 6800 Hz.

The output of the sampler is a stream of pulse amplitude modulated (PAM) signals spaced at 125 μs intervals. Each of these pulses has a width of 2 μs. The encoder quantizes each PAM signal into eight bit digital samples, resulting in a 64 kbps (8 bits x 8000 Hz) signal. This digital bit stream is the DS0 signal.

2.2 Digital Signal Level 1 (DS1)

T1 networks carry a multiplexed sequence of DS0s, delimited by framing information. Digital signal level 1 (DS1) refers to the multiplexing and framing formats while T1 refers to the carrier, specifying the line coding and electrical transmission characteristics.

2.2.1 Framing Formats

The basic frame unit in a DS1 data stream is the D1 frame. A D1 frame is 193 bits in length and consists of one framing bit followed by 24 byte interleaved DS0 channel samples. The D1 frame is transmitted at a rate of 8000 Hz for a data rate of 1.544 Mbps. The D1 frame format is shown in Figure 2.

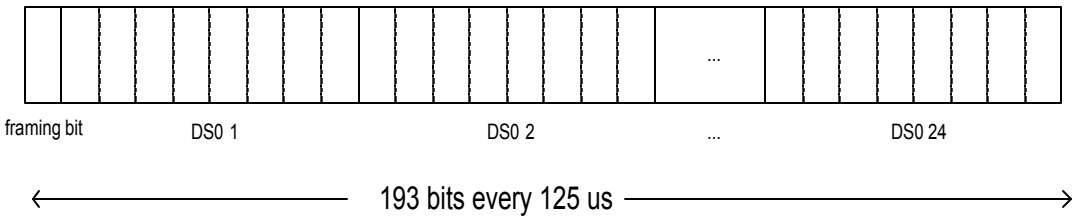

Figure 2 – Basic DS1 D1 frame.

Twelve D1 frames are grouped together to form a superframe (SF) or a D4 frame. The sequence of twelve framing bits in a superframe form the frame alignment signal (FAS), a predefined bit sequence that allows the framer to align to the superframe. The superframe FAS is the bit string "100011011100." The superframe format carries a user payload at a rate of 1536 kbps (24 x 64 kbps) and adds 8 kbps in framing overhead.

The extended superframe (ESF) format was proposed by AT&T in 1985 to provide a more robust framing format that could better support the complexities of T1 data communications equipment. The ESF format consists of 24 D1 frames, providing a total

of 24 framing bits. Assignment of the 24 framing bits in an ESF frame is shown below in Table 1.

Table 1 – T1 ESF framing bit assignment.

Frame Number	Framing Bit Assignment		
	FAS	FDL	CRC
1		M1	
2			C1
3		M2	
4	F1 = 0		
5		M3	
6			C2
7		M4	
8	F2 = 0		
9		M5	
10			C3
11		M6	
12	F3 = 1		
13		M7	
14			C4
15		M8	
16	F4 = 0		
17		M9	
18			C5
19		M10	
20	F5 = 1		
21		M11	
22			C6
23		M12	
24	F6 = 1		

In contrast to the D4 superframe format, all of the framing bits are not dedicated to producing the frame alignment signal. The frame alignment signal is 2 kbps signal consisting of only 6 of the 24 framing bits denoted F1-F6 in Table 1. ESF provides a 6-bit CRC check (C1-C6 in Table 1) over each ESF frame. The values of the CRC bits for any ESF frame contain the checksum for the previous ESF frame. The remaining 12 of the 24 framing bits (M1-M6 in Table 1) are reserved for a facilities data link (FDL)

channel that provides a communication mechanism for equipment at each end of a T1 link without interrupting the user data transmitted in the 24 DS0 channels.

2.2.2 Robbed-bit Signaling

Each user DS0 voice channel requires additional signaling information to communicate control and state information associated with the DS0 channel. T1 implements signaling by "robbing" bits from the user payload. In a superframe or extended superframe, the least significant bit in every 6^{th} DS0 PCM sample is reserved to convey signaling information. Thus, each superframe has two signaling bits associated with each of the 24 DS0s while each extended superframe has four signaling bits associated with each of the 24 DS0s; SF signaling captures four states per DS0 while ESF signaling captures 16 states per DS0.

Robbed-bit signaling is used to transmit information such as whether or not a telephone call is active or if the line is idle. Robbed-bit signaling is inserted for voice channels at the expense of a slight reduction in the user 64 kbps payload to 62.7 kbps.

2.2.3 Unchannelized T1

T1 was initially designed to carry multiplexed DS0 voice traffic in the DS1 format. However, for transmission of data, the 64 kbps DS0s can be amalgamated into a single 1.536 Mbps stream with no signaling information. Unchannelized T1 is used to transmit data for T1 subscriber lines where the content is not voice signals but a 1.536 Mbps user payload data channel.

2.2.4 Alarms

The alarm indication signal (AIS), is the most severe of the T1 alarms and is used to indicate a complete loss of the received signal. AIS is an unframed all 1's signal that overwrites all data. AIS may be generated, for instance, when a T1 regenerating device loses the incoming signal on one end and thus cannot regenerate it on the other end. The regenerator would, in this case, transmit a stream of 1's in the downstream direction until it detects a valid signal on the failed line. This scenario is depicted below in Figure 3.

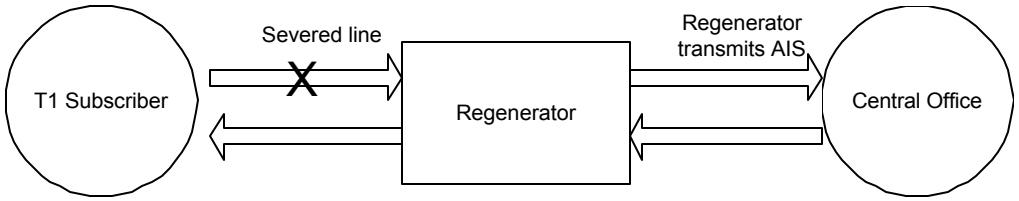

Figure 3 – Generating an AIS alarm.

The AIS alarm is a downstream alarm, implying that it is generated in the direction of the failed data stream. The T1 yellow alarm is an upstream alarm, generated back to a node from which the data stream has failed. The yellow alarm is shown below in Figure 4. The regenerator sends a yellow alarm back to the T1 subscriber indicating that a valid signal is not being received.

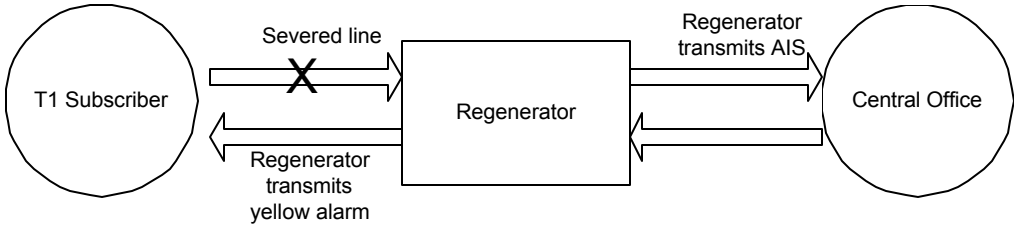

Figure 4 – Generating a yellow alarm.

For SF data formats, the yellow alarm is transmitted by setting bit 2 of all 24 DS0s to 1. Yellow alarms are transmitted using a special bit oriented code (see Section 2.2.5) over the facilities data link in ESF data streams.

2.2.5 ESF FDL Communication

The primary purpose of the ESF facilities data link is to perform operation, administration, and maintenance (OAM) of the network without disturbing the bandwidth of T1 line customers. The T1 FDL communication protocol is defined in [3].

The FDL transmits two types of data: bit oriented codes (BOCs) and link access procedure, D-channel (LAPD) frames. Bit oriented codes are 6-bit messages that are sent in the 16-bit container "0xxxxxx0 11111111" where the x's indicate the 6-bit BOC message. The 64 possible BOC messages are defined in the governing standard.

T1 ESF frame formats allow a node to transmit a performance report automatically to the remote end every second. These automatic performance reports are transmitted using LAPD, a variant of the High-Level Data Link Control (HDLC) protocol. The automatic performance report messages are sent every second on the FDL to provide statistics to the remote end about error conditions including line code violations, CRC errors, and framing errors.

2.3 E1

E1 is the European standard for transmission of DS0 signals, defined by the ITU-T in [7]. The basic E1 frame consists of 32 DS0s resulting in a frame length of 256 bits. The frames are transmitted at a rate of 8000 Hz, the PCM sample rate, and thus the data rate is

2.048 Mbps. No additional framing overhead is inserted because the first DS0 is used to transmit framing information.

2.3.1 Framing Formats

E1 frames are classified according to whether or not the framing byte, the first DS0 in each 256 bit E1 frame, is carrying the frame alignment signal. These frame types are frame alignment signal (FAS) frames, and non-frame alignment signal (NFAS) frames and are illustrated in Figure 5. An E1 framer searches for the FAS pattern to align to the framing pattern in the E1 bit stream.

	DS0 Bits							
	1	2	3	4	5	6	7	8
FAS	Internatnl Bit 1	0	0	1	1	0	1	1
NFAS	Internatnl Bit 0	1	A	National Bit 4	National Bit 5	National Bit 6	National Bit 7	National Bit 8

Figure 5 – DS0 0 bit allocation in E1 FAS and NFAS frames.

Seven of the eight bits of the first DS0 in a FAS frame transmit the FAS signal. The remaining bit is an international bit. There are a total of two international bits including the one in the NFAS frames and there is no specified usage for these bits at the current time other than for CRC-4 multiframe generation (Section 2.3.1.1).

The A bit in NFAS frames is called the remote alarm indication bit. The A bit is explained below in Section 2.3.4.

There are five national bits in NFAS frames. The national bits are spare bits whose use is flexible. The usage of these bits can be assigned on a national basis, allowing a country to define how these bits are used. A common use is for synchronization status messages.

2.3.1.1 The CRC-4 Multiframe

The CRC-4 multiframe consists of two sub-multiframes, each of which contains eight E1 basic frames of 32 DS0s. The CRC-4 multiframe format thus consists of sixteen consecutive E1 basic frames, alternating between FAS and NFAS frames. The CRC-4 framing information is transmitted using the two international bits shown in Figure 5.

A cyclic redundancy check is performed over each sub-multiframe and the four CRC bits are inserted into the international bits in the FAS signal in the next CRC-4 multiframe. The CRC-4 sub-multiframe structure is shown below in Figure 6.

Sub-Multiframe Number	Frame Number	DS0, 1 Bit 1
1	0 (FAS)	CRC bit 1
	1 (NFAS)	0
	2 (FAS)	CRC bit 2
	3 (NFAS)	0
	4 (FAS)	CRC bit 3
	5 (NFAS)	1
	6 (FAS)	CRC bit 4
	7 (NFAS)	0
2	8 (FAS)	CRC bit 1
	9 (NFAS)	1
	10 (FAS)	CRC bit 2
	11 (NFAS)	1
	12 (FAS)	CRC bit 3
	13 (NFAS)	CRC error
	14 (FAS)	CRC bit 4
	15 (NFAS)	CRC error

Figure 6 – CRC-4 multiframe structure and usage of the first bit in DS0 1.

The DS0 bits labeled "CRC bit" in the figure are used to carry the CRC value for the corresponding sub-multiframe in the previous CRC-4 multiframe. The CRC error bits are described below in Section 2.3.4.

2.3.2 Signaling

Signaling for DS0s in E1 is transmitted in what is called the signaling multiframe. Signaling information is inserted into DS0 16, resulting in only 30 of the 32 timeslots being used to carry user data.

The signaling multiframe uses the bits of DS0 16 over 16 consecutive E1 basic frame units as shown below in Figure 7. A signaling multiframe does not have to be aligned to the CRC-4 multiframe information inserted into timeslot 0.

Associated with each of the DS0 channels in the E1 stream are four signaling bits labeled "abcd". An E1 framer will search for the "0000" nibble pattern in frame 0 of the signaling multiframe because this pattern is not a valid E1 signaling state.

The "x" bits are called E1 extra bits and their usage is application specific. The "y" bit is an alarm bit described in Section 2.3.4.

Frame Number	DS0 16 Byte	
0	0000xyxx	
1	abcd DS0 2	abcd DS0 18
2	abcd DS0 3	abcd DS0 19
3	abcd DS0 4	abcd DS0 20
4	abcd DS0 5	abcd DS0 21
5	abcd DS0 6	abcd DS0 22
6	abcd DS0 7	abcd DS0 23
7	abcd DS0 8	abcd DS0 24
8	abcd DS0 9	abcd DS0 25
9	abcd DS0 10	abcd DS0 26
10	abcd DS0 11	abcd DS0 27
11	abcd DS0 12	abcd DS0 28
12	abcd DS0 13	abcd DS0 29
13	abcd DS0 14	abcd DS0 30
14	abcd DS0 15	abcd DS0 31
15	abcd DS0 17	abcd DS0 32

Figure 7 – Signaling multiframe information inserted into DS0 16.

2.3.3 Unchannelized E1

Analogous to T1, E1 supports an unchannelized payload where a single channel operating at 1.984 Mbps is possible. Signaling multiframes are not used because there are no channels for which signaling is to be transmitted. However, DS0 0 is used for framing.

2.3.4 Alarms

As in T1, the alarm indication signal (AIS) is represented by an unframed all 1's signal. T1 yellow alarms are called remote alarm indication (RAI) alarms in E1 and are transmitted as the A bit in DS0 0 of a CRC-4 multiframe as shown in Figure 5. The RAI is the E1 equivalent of yellow alarms in T1.

When using CRC-4 multiframe framing, a receiving station may find that a CRC error has occurred. The station can relay this information back to the terminal generating the traffic by using the far end block error (FEBE) alarm or CRC error alarm. The FEBE alarm is indicated by setting the CRC error bits in DS0 0 of a CRC-4 multiframe shown in Figure 6.

Another alarm is the DS0 16 alarm indication signal. The interpretation of this alarm is application dependent but is intended to represent an alarm that does not interrupt traffic and is shown as the "y" bit in Figure 7.

3 Device Driver Overview

The goal of the device driver is to allow a user to program the T1/E1 transceiver into a particular mode rapidly and in a manner that is logical and simple. In the design of the device driver, we focused on developing the application programming interface (API) so that it mapped logically into the different functional blocks within the device. We also desired reasonable abstraction of the many registers located on the device. This section describes the basic components of the device driver.

3.1 The T1/E1 Transceiver

Before proceeding into a detailed discussion of the development of the device driver for the single and quad T1/E1 transceiver, we must first provide some background about the COMET (Combined E1/T1 Transceiver) and COMET-Quad devices.

The COMET family of devices is a PMC-Sierra product that consists of an analog line interface unit combined with a T1 and E1 framer on the same device. Figure 8 is a block diagram containing the basic hardware components within the T1/E1 transceiver.

The COMET-Quad consists of four T1/E1 channels and its structure is identical to that of the COMET with a COMET representing one of the four T1/E1 channels. Each of the four T1/E1 channels within a COMET-Quad is referred to as a quadrant. The device driver supports both COMET and COMET-Quad devices.

The major difference between the COMET and COMET-Quad devices is that the COMET-Quad allows a user to multiplex the four T1 or E1 quadrants onto the backplane into a single 8.192 Mbps data channel. This multiplexed bus architecture is called the

High Density Multi-Vendor Integration Protocol (H-MVIP) standard and is defined by the Global Organization for Multi-Vendor Integration Protocol (GO-MVIP) in [6].

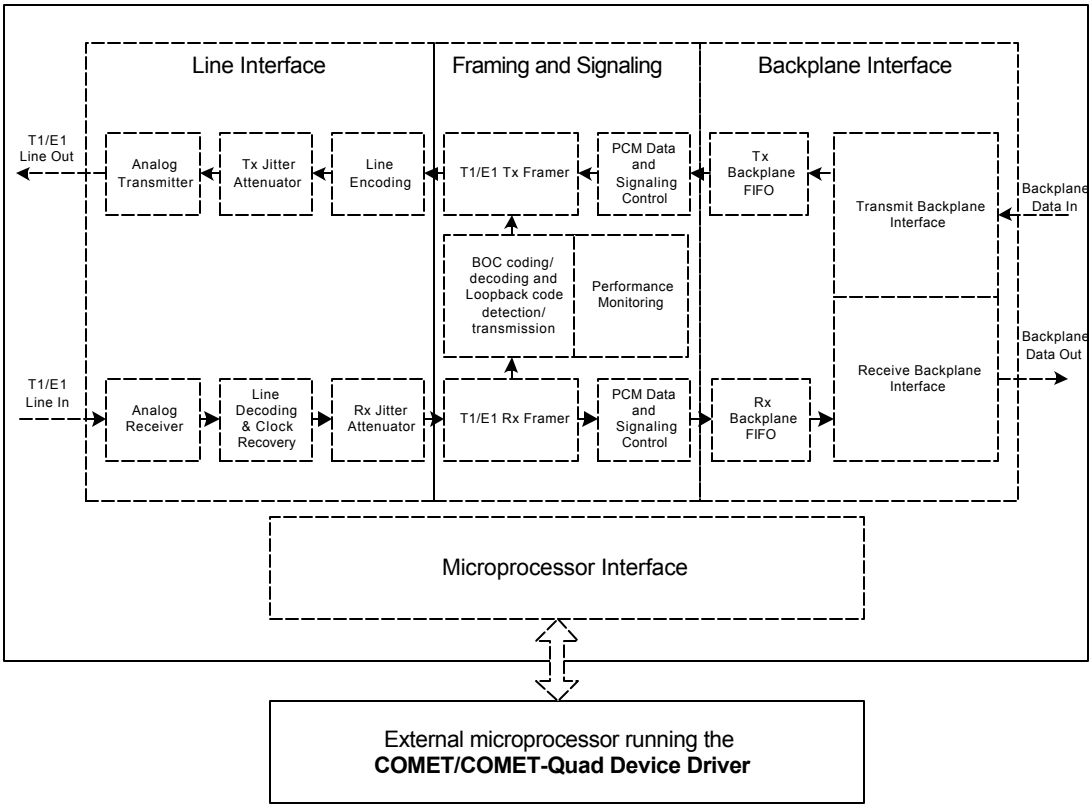

Figure 8 – Block diagram of COMET device and a single COMET-Quad quadrant.

Each of the device blocks shown in Figure 8 require configuration upon power-up of the device and this is the primary function of the device driver. There are a total of 256 registers for the COMET and a COMET-Quad device has four times this or 1024 registers.

3.2 Defining the API

While developing the device driver, we focused on defining the application programming interface. The application programming interface is the set of functions provided by the

device driver that are used by the application to control a COMET or COMET-Quad device. The API functions can be divided into the following categories:

- Configuration – These functions will normally only be used by the application upon power up or reset to bring the COMET or COMET-Quad into the required operational mode.

- Operation – These are functions that perform routine operations on the device that will be called regularly after the device is initialized and T1/E1 is traffic passing through the device. These functions include alarm insertion, device performance statistics retrieval, bit oriented code (BOC) reception and transmission, and inband loopback code reception and transmission.

- Interrupt Servicing – This group of device driver functions includes the interrupt service routine (ISR) provided by the device driver and the mechanism used to pass interrupt event information back to the application. This subset of the application programming interface also provides an interrupt polling mechanism and functions that enable and disable individual interrupts.

- Diagnostics – Diagnostic API functions are used when testing the device register integrity or when invoking loopbacks. These functions are intended to be used when the device is not in its normal mode of operation: taking an analog T1/E1 line input and presenting PCM data onto the backplane while generating an analog T1/E1 output from PCM data received from the backplane.

3.3 Device Driver Architecture

We based the basic device driver architecture on the PMC-Sierra, Inc. device driver model used across PMC-Sierra, Inc4. device drivers to promote consistency. This section describes this architecture, the parts of the device driver application programming interface defined particularly for this device driver, and the parts adopted from the PMC-Sierra device driver standard.

The PMC-Sierra device driver architecture divides the application programming interface into a common section and a device specific section. The common part of the device driver architecture consists of functions that will be provided by all device drivers. These functions provide general functionality such as loading the device driver and registering a device with the driver.

The device specific part of the application programming interface provides all of the functions that support the COMET and COMET-Quad devices. These functions consist of the four API sections described in Section 3.2.

The basic software architecture for the COMET and COMET-Quad device driver is shown in the following figure:

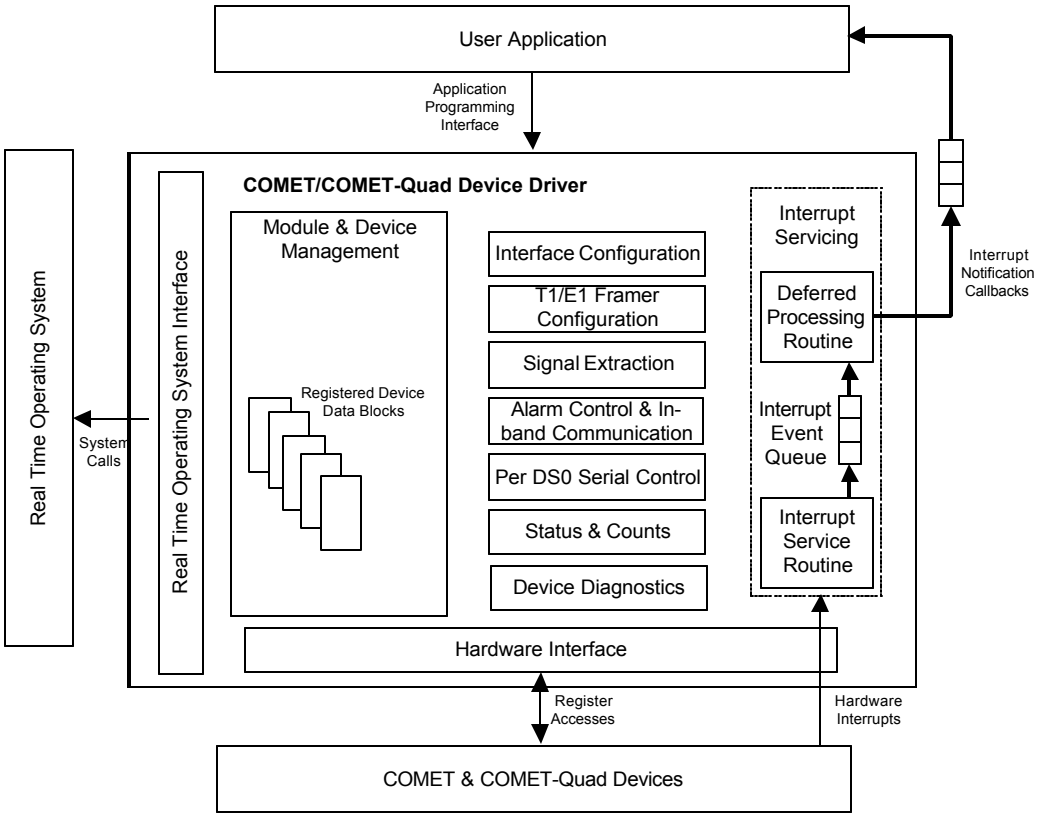

Figure 9 – Device driver software architecture.

The device driver is shown in Figure 9. Also shown are the operating system, the COMET and COMET-Quad devices being managed by the device driver, and the user application. We designed the device driver to use a modular approach with API functions categorized within the sub-blocks shown in the device driver block. We adapted the Module and Device Management API and the Interrupt Servicing architecture sections adapted from the proprietary PMC-Sierra device driver development architecture used across all PMC-Sierra device drivers. For these two sections, we defined the basic format of the API to conform to the PMC-Sierra propriety standard to promote consistency among PMC-Sierra drivers, however, our implementation of these two sections was specific for the COMET and COMET-Quad devices.

The group of blocks between the module and device management block and the interrupt servicing block is specific to the COMET and COMET-Quad devices and forms the core of the COMET and COMET-Quad device driver. This set of the API provides configuration, operation, and diagnostics functions for the device.

Interaction between the device driver and the operating system and the device driver and the actual COMET and COMET-Quad hardware occurs through a separate interface. The real time operating system (RTOS) interface and the hardware interface provide a layer of abstraction between the RTOS and the hardware platform respectively. We introduced this indirection to allow for ease of portability of the device driver to different hardware platforms. With this level of indirection, a user must only rewrite these two interfaces when porting the device driver to a operating system and hardware platform. Both hardware interface and the real time operating system interface are standard PMC-Sierra device driver modules that we adapted into the COMET and COMET-Quad device driver.

3.4 Development Environment

We implemented the COMET and COMET-Quad Device Driver in the "C" programming language using the Windriver VxWorks operating system. The target hardware was an Intel Pentium based single board computer in a compact peripheral component interconnect (PCI) chassis. A diagram of the target setup used is shown in Figure 10.

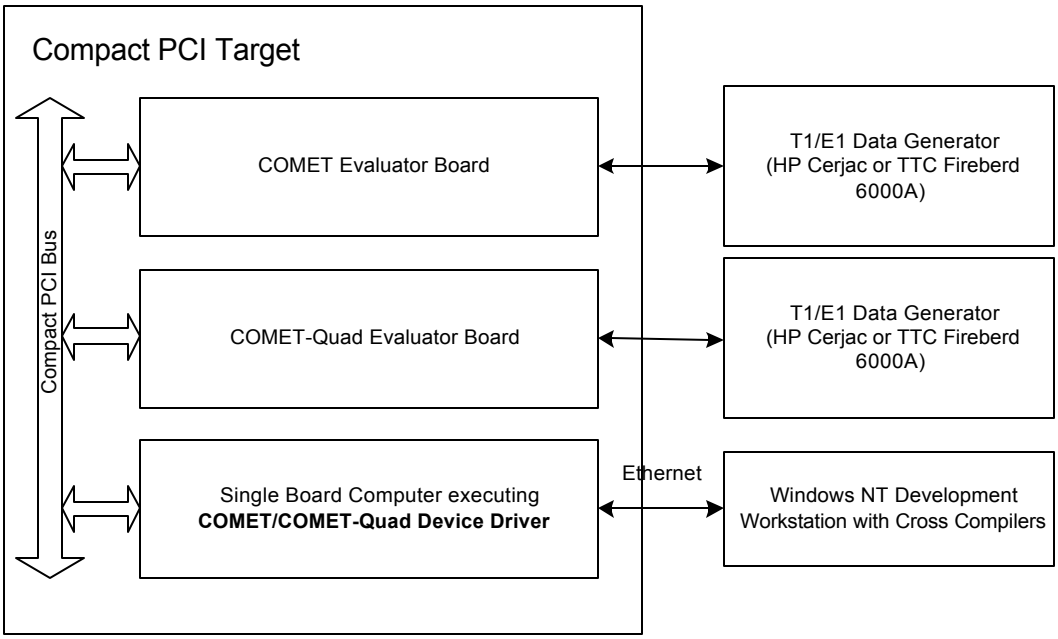

Figure 10 – COMET/COMET-Quad Device Driver test setup.

The development PC was running Windows NT and contained the Windriver VxWorks development facilities and GNU "C" cross compilers. The code was downloaded and executed on the target single board computer over an Ethernet interface. We used two PMC-Sierra proprietary evaluator boards that contained a COMET and COMET-Quad device, respectively, on the compact PCI bus. The single board computer then communicated with the evaluator boards over the compact PCI bus.

To test the device driver, we used the TTC Fireberd 6000A and the Hewlett Packard Cerjac T1 and E1 data generators. We used the TTC Fireberd tester when a test required simple T1 and E1 frame generation. For tests that required detailed access to the Tl or E1 framing and signaling bits, we used the Hewlett Packard Cerjac tester.

4 Module Management and Device Management Functions

The Module and Device Management section of the device driver provides the basic device driver management functions. We use the term "module" to refer to the device driver and the term "device" to refer to one of the COMET or COMET-Quad integrated circuit devices registered with the device driver.

The Module and Device Management device driver block is shown below in Figure 11.

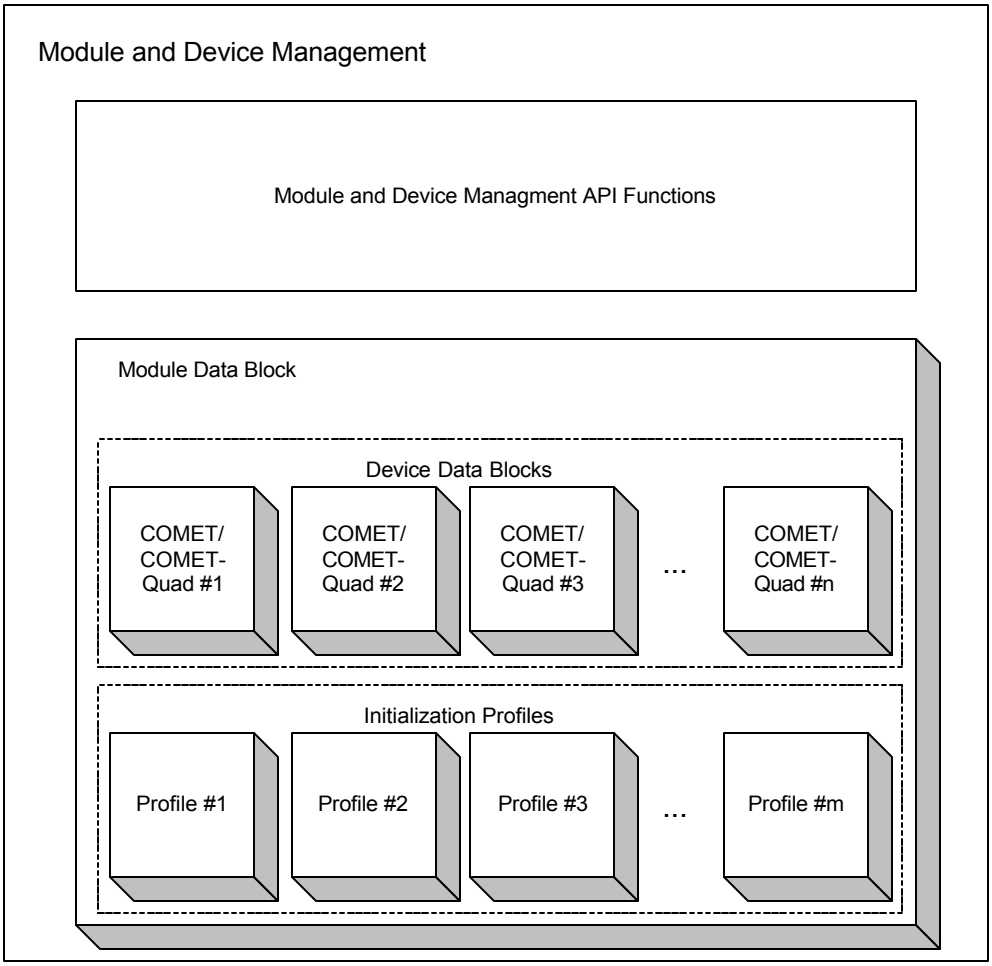

Figure 11 – Module and Device Management Architecture.

The module management API initializes the driver by allocating memory and securing RTOS resources that are needed by the driver. The device management functions are responsible for registering a device with the driver, providing basic read/write routines, initializing a device into a specific user defined configuration, as well as activating the device to enable interrupt servicing.

The device driver module is represented by a data structure called the module data block (MDB). The module data block contains the user configuration of the device driver as well as the user configuration of the devices registered with the device driver. The user configuration for each specific device is encapsulated in the device data block (DDB). The module data block contains a number of device data blocks corresponding to the maximum number of COMET or COMET-Quad devices that the device driver can support.

The module data block also contains initialization profiles. Initialization profiles are explained in Section 5. Initialization profiles are data structures that are stored within the driver and represent a user-defined configuration for a device. Once the user has added an initialization profile to the driver, they can initialize a device to a given configuration by specifying the initialization profile number.

4.1 Module and Device Data Blocks

Within the Module and Device Management section of the device driver reside the module data block and the device data blocks. We defined these data structures to encapsulate the device driver and the registered devices respectively.

4.1.1 The Module Data Block

The MDB is the top-level structure for the device driver containing all data structures that are maintained by the device driver. It contains configuration data about the device driver and contains the device data blocks and initialization profiles. The module data block structure members are described below in Table 2.

Table 2 – The module data block structure (`sCMQ_MDB`).

Field Name	Field Type	Field Description
`valid`	`int`	Flag to indicate whether or not the MDB has been initialized.
`stateModule`	`enum eCMQ_MOD_STATE`	Module state enumerated type. One of the following values: `CMQ_MOD_START`, `CMQ_MOD_IDLE` or `CMQ_MOD_READY` (Refer to Section 4.2.1)
`maxDevs`	`int`	Maximum number of COMET and COMET-Quad devices supported by the device driver. Corresponds to the number of device data blocks allocated in the driver.
`numDevs`	`int`	Number of devices currently registered.
`maxInitProfs`	`int`	Maximum number of initialization profiles
`pddb`	`sCMQ_DDB*`	Array of device data blocks whose size is set by the user upon driver initialization.
`pinitProfs`	`sCMQ_DIV*`	Array of initialization profiles whose size is set by the user upon driver initialization.

The must user must specify values for the maximum number of devices and initialization profiles supported by the device driver upon initialized. These values then reflect the number of device data blocks and initialization profiles allocated in the MDB respectively. We use the `numDevs` member of the structure to maintain the number of members of the device data block array, `pddb`, which are in use.

The flag `stateModule` provides us with a means to track the current state of the device driver module. The module state reflects whether or not the device driver has been initialized and is described in Section 4.2.1.

4.1.2 The Device Data Block

For each COMET or COMET-Quad device registered with the device driver, there is an associated device data block structure that encapsulates the configuration information about that particular device. The device data block structure is shown in Table 3.

Table 3 – The device data block structure (`sCMQ_DDB`).

Field Name	Field Type	Field Description
`valid`	`int`	Flag to indicate whether or not this DDB has been initialized.
`stateDevice`	`enum eCMQ_DEV_STATE`	Device State enumerated type. One of the following values: `CMQ_START`, `CMQ_PRESENT`, `CMQ_ACTIVE` or `CMQ_INACTIVE` (Refer to Section 4.3.1)
`baseAddr`	`unsigned char *`	Base address of the device. This address is used to perform memory mapped I/O to read and write to the device registers. The device has a data width of eight bits.

Field Name	Field Type	Field Description
cometqFlag	int	Used to specify whether the device is a COMET (0) or a COMET-Quad (1) device
modeE1	int	Indicates the global operational mode of the device. A one indicates E1 mode and a value of zero indicates T1 mode
pollISR	int	Flag that indicates the mode of interrupt servicing for the device. A 0 indicates ISR mode and a 1 indicates polling mode.
mask	struct sCMQ_ISR_MASK	Interrupt enable mask (Refer to Section 8)
profileNum	int	Initialization profile number associated with this device(Refer to Section 8)
usrCtxt	void *	User context pointer that is set to whatever the user specifies when the device is registered with the driver. This parameter is returned to the application as a parameter when the device driver invokes a callback.
cbackFramer	Function Pointer	Callback function pointer for framing interrupt events (Refer to Section 8.5)
cbackIntf	Function Pointer	Callback function pointer for line side interface events(Refer to Section 8.5)
cbackAlarmInBand	Function Pointer	Callback function pointer for inband code and bit oriented code events(Refer to Section 8.5)
cbackSigInsExt	Function Pointer	Callback function pointer for signaling insertion and extraction events(Refer to Section 8.5)
cbackPMon	Function Pointer	Callback function pointer for performance monitoring events(Refer to Section 8.5)

Field Name	Field Type	Field Description
`cbackSerialCtl`	`Function Pointer`	Callback function pointer for serial control events (Refer to Section 8.5)

The device data block contains a state member, as does the module data block. We defined the state variable, `stateDevice`, to maintain whether or not the device had been initialized and whether or not interrupt processing had been activated for the device. Device states are described in Section 4.3.1.

The COMET and COMET-Quad device registers were accessed using memory mapped I/O. The user must provide us with the base memory address of device register bank base when the user registers the device with the device driver. We then store the base address within the device data block in the member `baseAddr`. Given that the width of the registers on a COMET or COMET-Quad device is 8-bits wide, we define `baseAddr` to be a pointer to an eight bit integer value (`unsigned char*`).

4.2 Module Management

The module management API functions allow the user application to open and close the device driver. These API functions can initialize the driver, allocate memory required by the device driver, and allocate the required operating system resources such as creating message queues and installing the interrupt handler.

4.2.1 Module States

We defined a finite state machine to represent the current state of the device driver module. The module management API functions transition the device driver between the

various module states. In this manner, we are able to ensure that the user does not use any of the device API functions if the driver has not yet allocated all required memory and operating system resources. The module states used are: `CMQ_MOD_START`, `CMQ_MOD_IDLE`, and `CMQ_MOD_READY`.

4.2.1.1 The Start State (CMQ_MOD_START)

The `CMQ_MOD_START` state was used to indicate that the device driver was not yet initialized. In this state, we disallow the user from calling any one of the device API functions. This state is the default startup state for the device driver.

4.2.1.2 The Idle State (CMQ_MOD_IDLE)

We defined this state to represent when all of the required memory for the module data block, the device data blocks, and the initialization profiles has been allocated. In this state, however, operating system resources have not yet been allocated.

4.2.1.3 The Ready State (CMQ_MOD_READY)

The state `CMQ_MOD_READY` represents the condition where all required memory has been allocated and all required real time operating system resources have been acquired. Once the module is in this state, we are in the normal operational state of the driver and may proceed to use the device driver.

4.2.2 Module Management API Functions

The module management API functions are the means for the user to open, start, stop and close the device driver module. These functions transition the device driver from the default `CMQ_MOD_IDLE` state to the `CMQ_MOD_READY` state.

A state machine showing the device driver module states and the transition between them for each of the module management API calls is shown in Figure 12.

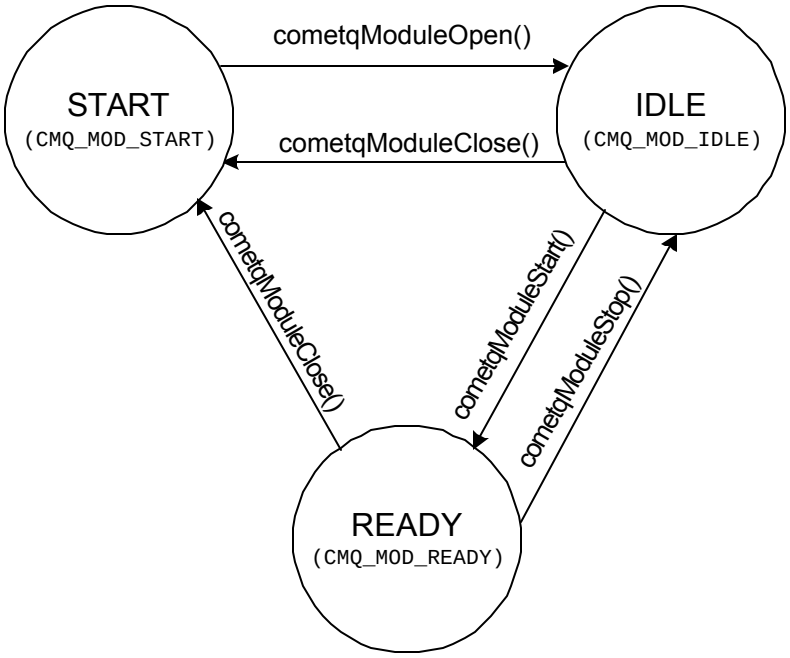

Figure 12 – Finite state machine of the device driver

4.2.2.1 Opening the Device Driver: cometqModuleOpen

This function performs initialization of the device driver. Once invoked, we see a transition in the module state from `CMQ_MOD_START` to `CMQ_MOD_IDLE` and memory for the module data block, the device data block, and the initialization profiles is allocated. The

user must pass this function a structure called the module initialization vector (MIV). The MIV structure is shown below in Table 4.

Table 4 – The module initialization vector structure (type: `sCMQ_MIV`).

Field Name	Field Type	Field Description
`maxDevs`	`unsigned int`	Maximum number of devices to be allowed by the driver
`maxInitProfs`	`unsigned int`	Maximum number of initialization profiles to be allowed by the driver

The field `maxDevs` allows the user to specify the number of devices that we are to support in the device driver. The user can thus control the amount of memory to be allocated for device data blocks. Similarly, the `maxInitProfs` member of the MIV specifies the number of initialization profiles we are to allocate space for within the driver.

Using the module initialization vector, the user has complete control of the amount of memory the device driver takes up in the system. Because of the nature of the COMET devices, it is possible that a T1/E1 line card application may require only four device data blocks while a voice over ATM application may be using 32 COMET devices and hence 32 device data blocks.

Because all memory for the device driver is allocated when the device driver is loaded, there is no subsequent dynamic memory allocation, preventing any performance degradation that can occur when memory is allocated during operation. Also, the device

driver memory will be contiguously allocated in memory, preventing fragmentation of the memory space within the system.

4.2.2.2 Starting the Driver Module: cometqModuleStart

We define this function to complete initialization of the device driver by acquiring all real time operating system resources. For the COMET and COMET-Quad device driver, the only real time operating system resources required are those used by the interrupt servicing architecture: message queues, message buffers, the ISR handler and the deferred processing task. The deferred processing task is part of the interrupt servicing architecture of the device driver and is spawned in this function call. After this stage of driver initialization is complete, the user can begin adding devices and using the device driver.

4.2.2.3 Stopping the Device Driver: cometqModuleStop

The `cometqModuleStop` function reverses the actions of `cometqModuleStart`. This API function disconnects the RTOS resources from the device driver by removing the registered message queues, freeing the allocated message buffers, removing the interrupt handler from the interrupt vector, and killing the deferred processing routine task. We remove all registered devices from the driver when a user invokes this function.

4.2.2.4 Closing the Device Driver: cometqModuleClose

`cometqModuleClose` removes the module data block and its contents, the device data blocks and the initialization profiles, from memory. We provide the user with a

mechanism to remove the device driver in its entirety from the system, releasing the memory allocated to it.

4.3 Device Management

The device management API functions allow the application to control a device by toggling it through the various device states. These functions allow the user to register a device with the device driver, initialize a device in a specific configuration, and enable a device for interrupt processing. We associate each device that a user registers with the driver with a device data block within the device driver module.

4.3.1 Device States

Within each device data block, we use the device state flag to maintain the current state of the device. We are then able to track whether a device has been initialized and whether it has been activated for interrupt processing. In this manner, we are able to limit operational API to work on a device only after it has been initialized and limit interrupt processing to a device that has been enabled to process interrupts. The possible device states are: `CMQ_START`, `CMQ_PRESENT`, `CMQ_INACTIVE`, and `CMQ_ACTIVE`.

4.3.1.1 The Start State (CMQ_ START)

We defined `CMQ_START` to identify unused device data blocks. We initialize the array of device data blocks in the module data block to `CMQ_START` when we initialize the device driver.

4.3.1.2 The Device Present State (CMQ_PRESENT)

This state indicates that we successfully detected a COMET or COMET-Quad device in the system and have associated it with the device data block. The device has not yet been initialized in this case and the device registers are in their default reset state. When a device is in this state, we allow the user to perform only basic register read and write functions until the device has been initialized.

4.3.1.3 The Device Inactive State (CMQ_INACTIVE)

The state `CMQ_INACTIVE` represents the condition where a COMET or COMET-Quad has been successfully initialized and the user can now perform any additional configuration required. In this state, we can transmit and receive T1 or E1 traffic and we can use the device driver operational API. However, when the device is inactive, we disable interrupt processing by both interrupt service routine and polling.

4.3.1.4 The Device Active State (CMQ_ACTIVE)

This state indicates that a device is configured and is now activated for interrupt processing. When in this state, we allow the user to specify whether or not they would like to poll the device for interrupt events or to process interrupt events using the interrupt service routine.

4.3.2 The Device Handle

We defined the device handle to allow a user to identify a device after it has been registered with the device driver. The device handle type is called `sCMQ_HNDL` and is

nothing more than a pointer to the device data block associated with the device when it was added to the driver. The definition for the device handle type is as follows:

Table 5 – Device handle definition (sCMQ_HNDL).

Definition:	`typedef sCMQ_DDB* sCMQ_HNDL`

By defining the type for the device handle in this manner, we are able to shield the user from knowledge that the device handle is in fact a pointer to one of the driver's internal structures.

Upon allocation of a device data block to the device that is being added by the user, we set the device handle by the device add function, `cometqAdd`, described below in Section 4.3.3.1. Subsequent calls to the device driver API that operate on a COMET or COMET-Quad device require this device handle so we are able to identify the device the user would like us to operate on. Functions that perform general device driver related operations such as module management and initialization profile management do not require a device handle as they do not operate on a device.

4.3.3 Device Management API Functions

The device management API functions transition the device between the four states as shown in Figure 13. The device state machine is only relevant when the module is in the `CMQ_MOD_READY` state. Until then, no API functions other than the module management functions can be used.

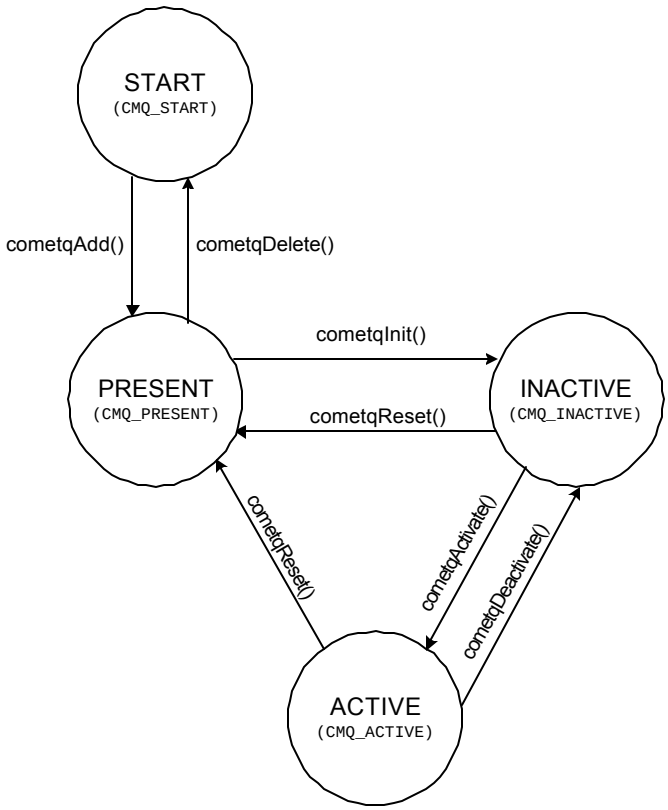

Figure 13 – Device state machine.

4.3.3.1 Adding a Device: cometqAdd

This function allows a user to register a COMET or COMET-Quad device with the driver. We verify the presence of a new device in hardware and return a device handle back to the user when `cometqAdd` is invoked.

When a COMET or COMET-Quad device is added to the module, we read the Revision/Chip ID register on the device use its contents to determine whether the user is adding a COMET or COMET-Quad device. The device data block stores the device type (COMET or COMET-Quad) in the `cometqFlag` field. We are then able to check the `cometqFlag` field in the DDB in subsequent API calls before performing an operation on the device to determine whether it is a COMET or COMET-Quad device.

Each API function in the device driver takes a framer number parameter. This framer number parameter is relevant only for COMET-Quad devices as the COMET has only a single framer. We use the `cometqFlag` member of the DDB to determine whether or not the framer number parameter is to be used. On COMET-Quad devices, there are four channels and we require the user to indicate which channel they would like to perform the operation on by providing a number from 0 to 3.

The user provides us with the base address of the device when it is added to the driver. The base address is a memory pointer to an eight bit address space, `unsigned char*`, because the COMET and COMET-Quad devices are eight bit devices.

The user must also provide a parameter called the user context. The user context is a pointer to a structure provided by the application that we report back to the application so that the it can identify the device. The user context is explained in detail in Section 8.5.

4.3.3.2 Initializing a Device: cometqInit

This function performs basic initialization on a device. We apply a software reset to the COMET or COMET-Quad device and we then configure the device according to the initialization information provided by the user.

When calling this function, the user provides a device initialization vector (DIV) or a profile number. The device initialization vector is shown in Table 6. An initialization profile uses the same structure as a DIV, `sCMQ_DIV`, but allows a user to store this structure internally within the device driver and then access it using a profile number. The details of initialization profiles are given in Section 5.

Table 6 – The device initialization vector (`sCMQ_DIV`).

Field Name	Field Type	Field Description
`pollISR`	`unsigned int`	Indicates type of interrupt processing to use: ISR mode (0) or polling (1)
`cbackIntf`	`Function Pointer`	Callback function pointer for interface interrupt events (Refer to Section 8.5)
`cbackFramer`	`Function Pointer`	Callback function pointer for framer interrupt events (Refer to Section 8.5)
`cbackAlarmInBand`	`Function Pointer`	Callback function pointer for inband code and bit oriented code interrupt events (Refer to Section 8.5)
`cbackSigInsExt`	`Function Pointer`	Callback function pointer for signaling insertion and extraction interrupt events (Refer to Section 8.5)
`cbackPMon`	`Function Pointer`	Callback function pointer for performance monitoring interrupt events (Refer to Section 8.5)
`cbackSerialCtl`	`Function Pointer`	Address of the callback function for serial control interrupt events (Refer to Section 8.5)
`initDevice`	`unsigned int`	Flag to indicate whether or not to apply hardware configuration (analogInit, framerInit, backplaneInit) to the device. If this flag is not set, the device is left in its default power up state. By setting this flag, analogInit, framerInit, and backplaneInit are all applied to the device.
`analogInit`	`struct sCMQ_ANALOG_INIT`	Initialization configuration structure for the analog interfaces.
`framerInit`	`struct sCMQ_FRAMER_INIT`	Initialization configuration structure for the transmit and receive framers.
`backplaneInit`	`struct sCMQ_BACKPLANE_INIT`	Initialization configuration structure for the transmit and receive backplane interfaces.

The device initialization vector provides function pointers that we use to perform interrupt event callbacks (Section 8.5) as well as the information required for hardware configuration. We use DIV member `initDevice` to specify whether or not to configure the device with the three hardware configuration sub-structures, `analogInit`, `framerInit`, and `backplaneInit`, after the device has been reset and we have set the callback function pointers provided by the user in the DDB. By including this flag in the DIV, we provide the user with flexibility as to how they would like to configure a device. When `initDevice` is false, the user must configure the device using the configuration API. The process of initializing a device is described in detail in Section 5.

4.3.3.3 Activating a Device: cometqActivate

This function activates a device by enabling interrupt processing. After we activate the device, the interrupt service routine will handle all enabled interrupts or the user can poll the device for any interrupts indicators. We use the device data block member, `mask`, in Table 3 to represent whether each interrupt enable on the device is active or inactive.

4.3.3.4 Deactivating a Device: cometqDeActivate

We allow the user to deactivate interrupt processing using the function `cometqDeActivate`. When disabling interrupt processing, we disable all hardware interrupt enables and allow no interrupts to be for the device by either polling or execution of the interrupt service routine.

4.3.3.5 Resetting a Device: cometqReset

When a user invokes this function, we perform a software reset on the COMET or COMET-Quad device by toggling the RESET bit in the Software Reset register on the device. Upon reset of the device, the device registers return to their default values and configuration is lost. T1/E1 traffic passing through the device will immediately be terminated.

4.3.3.6 Deleting a Device: cometqDelete

We define the function `cometqDelete` to allow a user to remove a device from the module data block. We free the associated device data block by returning its state to `CMQ_START`. The user may invoke `cometqDelete`, for example, when a quad T1 line card has been removed in a hot swappable bus architecture. In this case, the application code would invoke `cometqDelete` once to remove a COMET-Quad device or four times to remove four COMET devices.

5 Device Initialization

We designed the COMET and COMET-Quad device driver to provide multiple ways to initialize a device. Most users will use the device in a standard configuration and rely on the device driver to hide the many detailed configurations allowed by the device hardware. However, a small fraction may be using some of the more advanced features of the COMET or COMET-Quad. These users will require more detailed configurations. These users still desire, however, that we perform the mapping from abstract variables in data structures to actual registers in hardware so that they need not use the device at the bit level.

We provide the user with the following different device initialization techniques:

- Device initialization vector passed via `cometqInit`
- Device initialization vector passed via initialization profiles mechanism and applied using `cometqInit`
- All hardware configuration performed through API calls

5.1 DIV Based Initialization

The device initialization vector structure can be passed when initializing a device with the function call to `cometqInit` or stored internally within the driver as an initialization profile and applied with a call to `cometqInit`. We designed the device initialization vector for users who are interested in configuring the device into a relatively standard operating mode without a detailed study of the COMET or COMET-Quad register set. The device initialization vector is described in Table 6 on page 35.

The hardware initialization members of the DIV correspond to the three main sections of the device: the analog line interface, the framers, and the backplane interface. Within each of these sections, the minimum required configuration is available in order to keep the data structure simple to use. All other configuration features that are not specified within the DIV data structure remain in their default value. The default values for the more obscure and non-standard features within COMET and COMET-Quad devices are such that they are the value that a user who is not interested in them would want them to be.

The DIV sub-structures are given below in Table 7, Table 8, and Table 9.

Table 7 – The device initialization sub-structure for analog line side initialization (`sCMQ_ANALOG_INIT`).

Field Name	Field Type	Field Description
`txEnable`	`unsigned int`	Selects analog transmitter high impedence (0) or transmitter enable (1).
`txLineBuildOut`	`enum eCMQ_TX_LBO`	Selects the analog transmitter line transmit waveform. The user selects from the predefined tables stored within the device driver.
`rxEqualizerTable`	`enum eCMQ_RX_LINE_EQ`	Selects one of the predefined receiver equalizer settings. The user can select a predefined equalizer coefficient set that is stored within the device driver.
`csuClkMode`	`enum eCMQ_CSU_SVC_CLK`	Selects the clock synthesis unit operational mode for the device based on the required transmit frequency and the input clock to the device.

Table 8 – The device initialization sub-structure for framer initialization (`sCMQ_FRAMER_INIT`).

Field Name	Field Type	Field Description
txFramerMode	enum eCMQ_FRAME_MODE	Selects a T1 or E1 framing format for the transmit framer from the framing formats provided by the device.
rxFramerMode	enum eCMQ_FRAME_MODE	Selects a T1 or E1 framing format for the receive framer from the available framing formats provided by the device.

Table 9 – The device initialization sub-structure for backplane side initialization (sCMQ_BACKPLANE_INIT).

Field Name	Field Type	Field Description
backplaneTxMode	eCMQ_BACKPLANE_TX_MODE	Selects the configuration of the transmit backplane interface from the transmit backplane configurations available within the device driver.
backplaneRxMode	eCMQ_BACKPLANE_RX_MODE	Selects the configuration of the receive backplane interface from the available receive backplane available within the device driver.

For each of these three initialization structures, we defined only a few abstract members. We allow the user to select the configuration they want and the we then map the configuration parameter into the corresponding register configurations. These configurations are outlined within the data sheets for the COMET and COMET-Quad so that the user can find out exactly what we are configuring in hardware when the user selects one of these modes.

When a user would like to initialize a device with a DIV, they create the structure and pass it as a parameter to `cometqInit` when they are initializing a device. If the user

would like to store the structure within the driver so that it can be applied to the device again, they should use the initialization profile mechanism described in Section 5.2.

If a user wants to initialize a device but does not want any hardware to be configured, they can set the `initDevice` flag to zero so that only the callbacks function pointers are set within the device data block when `cometqInit` is called. With this mechanism, we allow the user to leave the hardware in its reset configuration and perform initialization using the configuration API functions.

5.2 Initialization profiles

The initialization profile mechanism is based on the device initialization vector structure. However, in this case the device initialization vector is stored within the device driver as an initialization profile. The user adds a DIV and the device driver validates it to ensure that the hardware configuration selected within the DIV hardware initialization sub-structures is valid.

We provide the basic initialization profile flow in Figure 14 and describe the initialization profile API functions in the proceeding subsections.

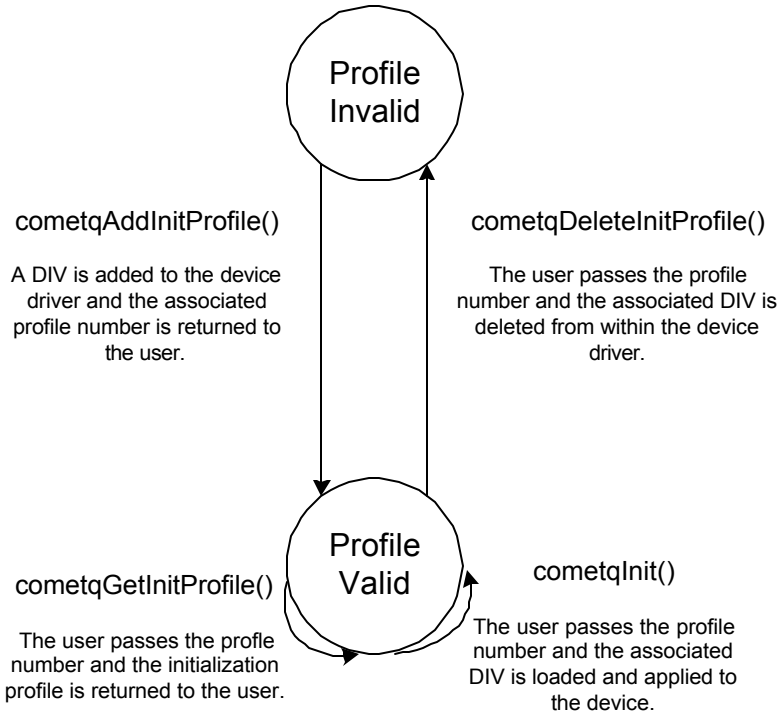

Figure 14 – Initialization profile flow.

5.2.1 cometqAddInitProfile

We defined this function to allow a user to register a device initialization vector within the device driver so that they can access it again at a later time. When the DIV is added, the device driver assigns a profile number to the DIV that is being added and in subsequent calls that access initialization profiles, the user can then reference this initialization profile by providing the profile number.

After the profile has been added, we allow the user to apply it to a device by calling `cometqInit` and specifying the profile number.

5.2.2 cometqGetInitProfile

We provide this routine to allow a user to retrieve a copy of an initialization profile that has been added to the device driver by specifying the profile number.

5.2.3 cometqDeleteInitProfile

The `cometqDeleteInitProfile` function provides the application with a method of deleting an intialization profile from the device driver. We mark the associated initialization profile as free within the module data block so that it can be reassigned later to another initialization profile.

6 Device Configuration

The device configuration functions allow the user to perform detailed configuration of each block within the device. We divide the device configuration section into three areas: line, backplane, and framer configuration. The line side and backplane sections of the device together form the interface configuration section of the device driver. The application calls these functions when initialization of the device is performed with the `initDevice` flag cleared within the device initialization vector or in addition to the device initialization vector configuration when additional configuration is required.

6.1 Interface Configuration

We designed the interface configuration section of the device driver to encapsulate both the line side and backplane side sections of the device because these two device blocks provide the interface between the device and the system in which it is operating. We describe both components of the interface configuration application programming interface functions in this section.

6.1.1 Line Side Interface

The line side section of the device interfaces with the analog T1 or E1 signal and is responsible for both recovering a clean digital signal to pass on to the framers and generating an analog signal to transmit across the line. We provide the user with functions to perform the following functions with this section of the device driver:

- digital to analog conversion on the transmitted signal based on a user specified analog pulse waveform template

- analog to digital conversion of the received signal
- line encoding of the transmit signal
- clock recovery and line decoding of the received signal

These different items map onto a set of application programming interface functions that are described in the following sections.

6.1.1.1 Analog Transmitter Configuration: cometqLineTxAnalogCfg

We provide this function to configure the analog transmitter on a COMET or COMET-Quad. The following can be performed with this function:

- Selection of transmitter high impedance state or enabling the transmitter. In certain applications, the COMET or COMET-Quad device may only be used as a receiver or there may be multiple drivers for a single line and thus the user may desire to leave the transmitter in high impedance state.
- Analog transmitter pulse waveform. The user can select a pulse waveform template by specifying an enumerated type value that corresponds to a pulse waveform template stored within the device driver. The pulse waveform templates stored within the device driver specify the line type (short haul or long haul) and specify the line length that the transmitter is going to drive. We took these templates from the COMET and COMET-Quad data sheets and they are standard pulse waveforms for T1 and E1 transmission. They are stored internally within the driver so the user does not need to convert the

values from the data sheet into C code and then write the lengthy code that programs the device. The following templates are stored within the device driver:

- T1 short haul transmission (< 660 feet) for line lengths between 0 and 110 feet, 110 to 220 feet, 220 to 330 feet, 330 to 440 feet, 440 to 550 feet, and 550 feet to 660 feet.
- T1 long haul transmission (> 660 feet) with 0 dB, 7.5 dB, 15 dB, or 22.5 dB signal attenuation
- E1 short haul transmission for 120 Ohm or 75 Ohm cables

- Analog transmitter fuse values. The user can provide values for both positive and negative current control in the digital to analog converter to fine-tune the transmitter amplitudes. We perform the required sequence of steps to program the fuses as outlined in the device data sheets.

6.1.1.2 Analog Receiver Configuration: cometqLineRxAnalogCfg

This API function provides the user with an interface to configure the analog receiver on a COMET or COMET-Quad device. We designed this function to incorporate all of the analog receiver characteristics including the analog loss of signal definition.

The analog receiver declares loss of signal when it determines that there is no valid signal input. This alarm condition is valuable because it allows the user to detect if the input line has been severed. The user would then insert an alarm indication signal, AIS, in

response to the alarm condition. The loss of signal alarm generates an interrupt that we configure and handle through the interrupt servicing architecture described in Section 8. We allow configuration of the analog loss of signal within this function because it is one of the key functions of the receiver, in addition to performing equalization and analog to digital conversion of the received signal.

The basic configuration parameters of this API function are as follows:

- The analog receiver on the device performs equalization on the received signal to reduce inter-symbol interference. The user can specify whether the equalizer is to be programmed for T1 or E1 and we then load the corresponding equalizer coefficients stored internally within the driver and program the equalizer. By storing the coefficients internally within the driver, we do not require the user to convert the text from the data sheet into C code.

- Parameters for declaring loss of signal. The user can specify the signal power threshold and the time threshold for declaring loss of signal. We configure the receiver with the user provided thresholds so that when the receiver determines that the signal power is below the power threshold for the specified time threshold, it will declare loss of signal.

- Parameters for clearing loss of signal. The user can specify the signal power threshold and the time threshold for clearing loss of signal once the loss of signal alarm has been declared. We configure the receiver with the user provided thresholds so that when the receiver determines that the signal power

is above the power threshold for the specified time, it will clear the impending loss of signal alarm.

6.1.1.3 Line Coding Configuration: cometqLineTxEncodeCfg and cometqLineRxEncodeCfg

We defined these API functions to configure the device for the line coding format to transmit and the and receive. The T1 transmission standard specifies line coding in alternate mark inversion (AMI) and bipolar eight zero substitution (B8ZS). The E1 standard allows AMI and high density bipolar (HDB3) coding. We examine the device data block operational mode flag when the user calls one of these functions with the line coding mode and we limit the user to the allowed coding schemes based on whether the device is operating in T1 or E1.

6.1.2 Backplane Configuration

In the backplane configuration section of the application programming interface, we allow the user to specify exactly the format of how the data should be added or removed from the backplane. The transmit backplane takes data from the backplane and passes it onto the transmit framer while the receive backplane interface inserts data from the receive framer onto the backplane on the device. These API functions configure how the device performs these actions.

Configuration of the backplane is an involved process and requires the configuration of many registers that specify how the waveforms appear on the backplane and how the data is clocked. The COMET and COMET-Quad data sheets outline a number of basic configurations for both the transmit and receive backplane configurations and the

majority of application developers who use the COMET and COMET-Quad devices will use one of these basic configurations.

We allow the user to specify one of the standard backplane configurations or for the user to configure each of the backplane parameters individually. The user can select a standard backplane configuration through our device initialization vector in the backplane configuration sub-structure defined in Table 9 on page 40.

We defined the backplane configuration application programming interface for application developers who have precise non-standard requirements for backplane configuration. The API functions provide access to all of the individual configurations within the backplane configuration registers on the COMET and COMET-Quad device and allow the user to determine how each of these configurations should be set.

We separated the backplane configuration API functions into transmit and receive configuration for both COMET and COMET-Quad devices and added an additional function for H-MVIP interface configuration for COMET-Quad devices only. For each of the transmit and receive configurations, we created two functions: one function to specify the data configuration for the backplane interface and another function to specify the detailed configuration of the frame pulse waveforms and backplane parity settings. The H-MVIP interface configuration function configures both the transmit and receive H-MVIP interfaces for backplanes that are going to support the 8.192 Mbps H-MVIP standard [6]. We only allow the user to invoke this function for COMET-Quad devices because the COMET device does not support H-MVIP.

6.1.2.1 Backplane Transmit Interface Configuration: cometqBTIFAccessCfg and cometqBTIFFrmCfg

We defined these API functions to configure the backplane transmit interface (BTIF). We divided configuration of the BTIF into configuration dealing with the backplane data mode and configuration dealing with the format of the framing waveform.

We provide an application developer with the following functionality with the routine `cometqBTIFAccessCfg`:

- A flag that specifies whether the device is the clock master or if it is the slave
- An enumerated type selecting one of the COMET and COMET-Quad supported backplane data formats (full frame, 64 kbps channelized, or 56 kbps channelized).
- An enumerated type selecting one of the supported backplane data rates (1.544 Mbps, 2.048 Mbps, or 8.192 Mbps).

We designed `cometqBTIFFrmCfg` function to configure the detailed waveform and parity configuration on the backplane when the device default values for these parameters are not acceptable for the application.

6.1.2.2 Backplane Receive Interface Configuration: cometqBRIFAccessCfg and cometqBRIFFrmCfg

The backplane receive interface (BRIF) configuration functions, `cometqBRIFAccessCfg` and `cometqBRIFFrmCfg`, are analogous to those for the backplane transmit interfce

configuration. The only difference is that they configure the register set for the BRIF interface.

6.1.2.3 Backplane H-MVIP Interface Configuration: cometqHMVIPCfg

This API function configures the H-MVIP interface on COMET-Quad devices and we return an error code if it is invoked on a COMET device. We check the device type flag in the device data block, `cometqFlag`, to ensure that the user is trying to use this function on a COMET-Quad device. We abstract the H-MVIP hardware features into a few function parameters:

- Enumerated type specifying the receive H-MVIP mode. The user can select one of: H-MVIP mode disable, H-MVIP data highway, or H-MVIP data highway with common channel signaling. We translate the configuration into the appropriate register values.

- Enumerated type specifying the transmit H-MVIP mode. The user can select one of: H-MVIP mode disable, H-MVIP data highway, or H-MVIP data highway with common channel signaling. Again, we translate the configuration into the appropriate register values.

6.2 Framer Configuration

The COMET and COMET-Quad devices contain a single operating mode bit that is located in the Global Configuration registers. The operating mode bit selects between T1 or E1 mode for the device. Both the receive and transmit framers must operate in T1 or

E1 mode and on a COMET-Quad device, all four quadrants operate in the same mode. These are all hardware features of the devices.

We allow the user to configure the characteristics of the T1 or E1 framers using framer configuration set of API functions. We provide a mechanism for the user to configure detailed operations of the T1 or E1 framer when the default values are not suitable for the application. When a user selects a specific T1 or E1 framing format for the receive and transmit framers through a device initialization vector, we perform the following sequence of events:

- Set the global T1 or E1 operating mode select bit on the COMET or COMET-Quad device
- Configure the transmit and receive T1 or E1 framing formats
- Configure all blocks within the device that require configuration based on the transmit or receive framing format

When a user initializes a device with the `initDevice` flag in the DIV off, the user must configure the framers using our framer configuration API. The user must first call the `cometqSetOperatingMode` function. We configure the device for T1 or E1 mode with this function as indicated by the user. When we configure the operating mode, we configure the global device operating mode bit in hardware and we then set our operating software mode flag within the device data block to reflect whether the device is operating in T1 or E1 mode. Once the operating mode of the device has been configured, we can perform detailed configuration of the transmit and receive framers.

We designed the device driver so that the user must first configure the operating mode via our function, and then invoke our specific T1 or E1 framer configuration functions. We defined two framer configuration functions for the receive and transmit framers: one function for T1 mode and another for E1 mode. We used two distinct sets of functions for T1 and E1 because the parameters for these two framing modes are completely different.

To configure the device in T1 mode, for example, the application must first call our function `cometqSetOperatingMode` specifying T1, and then invoke our function `cometqT1TxFramerCfg` to configure the transmit T1 framer and then our function `cometqT1RxFramerCfg` to configure the receive T1 framer. The user must perform an analogous approach to configure the E1 framers.

We separated the operating mode configuration function and the specific T1 or E1 framer configuration functions to ensure that a user does not attempt to configure the transmit framer for T1 and the receive framer for E1. We check the device data block operating mode in each of the E1 and T1 framer configuration API functions to ensure consistency between the function type and the current operating mode set by the user.

6.2.1 T1 Framer Configuration: cometqT1TxFramerCfg and cometqT1RxFramerCfg

We allow a user to define the configuration of both the transmit and receive T1 framers using these two functions. When the device is operating in T1 mode, several blocks must be configured within the device to correspond to the actual T1 framing format. Although these blocks are not part of the framing block on the device, they must be configured to

be consistent with the framing mode set in the framer. As a convenience to the user, we program all of the required blocks within the device outside of the T1 framers to reflect the configuration of the receivers whenever `cometqT1TxFramerCfg` or `cometqT1RxFramerCfg` are invoked.

We designed the transmit framer API function to take the following configuration parameters:

- An enumerated type specifying the T1 framing format. The user can specify which T1 specific framing format (extended superframe, superframe, Japanese J1) the transmit stream should generate and we set the appropriate bits in hardware. In addition to configuring the transmit framer, we configure all blocks in the transmit data path that must be configured for the appropriate T1 specific framing format.

- An enumerated type that specifies the type of zero code suppression, the replacement of excess strings of zeroes according to a defined standard, to perform on the transmit data stream. Again, by using an enumerated type, we alleviate the user from dealing with what bits must be set for each zero code suppression format.

We designed the receive T1 framer configuration to take the following parameters:

- An enumerated type specifying the T1 framing format. We allow the user to specify which T1 specific framing format (extended superframe, superframe, Japanese J1) the framer should align to and we then set the appropriate bits in

the device. In addition to configuring the receive framer, we configure all blocks in the receive data path that must be configured for the appropriate T1 specific framing format.

- An enumerated type that indicates the criteria for declaring loss of frame. The T1 framer hardware generates interrupts indicating that it has lost frame alignment via the interrupt processing architecture described in Section 8. We allow the user to specify the ratio of consecutive framing bit errors that will indicate loss of frame and we configure the T1 framer hardware accordingly.

6.2.2 E1 Framer Configuration: cometqE1TxFramerCfg and cometqE1RxFramerCfg

We defined the E1 framer configuration API similar to the T1 framer configuration to promote consistency within the device driver and between T1 and E1 modes of operation. The actual parameters that the user passes to these functions, however, differ from those for the T1 framer configuration functions. When in E1 mode, there are no blocks within the device that must know the specific E1 framing mode except for the E1 framer itself.

We perform the following configuration in the transmit E1 framer configuration function `cometqE1TxFramerCfg`:

- A flag to indicate whether or not to generate CRC-4 multiframes
- A flag to indicate whether or not to insert signaling into DS0 16
- Flags to indicate whether or not to insert data into the E1 extra bits, national bits, and the international bits

After the user calls the function with the parameters, we configure the registers on the E1 transmit framer to reflect the user settings.

We implemented a similar approach in the receive framer function, `cometqE1RxFramerCfg`:

- A flag indicating whether or not to align to CRC-4 multiframes
- A flag indicating whether or not to align to signaling multiframes
- Enumerated types specifying the criteria the framer should apply to the data stream to generate the following alarm conditions:
    - Loss of signaling multiframe alignment
    - Alarm indication signal detect
    - Remote alarm indication signal detect

7 Device Operation

We designed the operational application programming interface to provide the application developer with access to features that will be used while active data is passing through the device. We can can divide the device operation application programming interface into three sections: signal extraction, alarm control and inband communications, and serial control.

7.1 Signal Extraction

We provide a means for the user to examine signaling data in the receive T1 or E1 streams in the signal extraction API. Signaling formats for T1 and E1 are explained in Section 2.2.2 and Section 2.3.2 respectively. We outline the functions that comprise the signal extraction API in this section.

7.1.1 Retrieving Change of Signaling State: cometqExtractCOSS

For each of the 24 DS0s in a T1 stream there are four signaling bits in T1 ESF and two signaling bits in T1 SF. For the 30 DS0s in an E1 stream, there are a total of four signaling bits. In both T1 and E1 data streams, these signaling bits form a single signaling state for each DS0 channel. The COMET and COMET-Quad devices maintain the current signaling state internally for each DS0. Whenever a signaling value changes, an interrupt is generated to indicate that a change of signaling state (COSS) event has occurred. There is only a single change of signaling state interrupt for each T1 or E1 stream. Thus, we note that a user requires knowledge of the precise DS0 within the T1 or E1 stream that has changed signaling states.

We designed the API function `cometqExtractCOSS` to work in conjunction with the change of signaling state interrupt by providing the user with knowledge of the DS0 in the T1 or E1 stream that has a new signaling state. If the user invokes this function immediately upon a change of signaling state interrupt, we indicate to the user which DS0s have experienced a change signaling state.

In the routine `cometqExtractCOSS`, we return a 32-bit integer that is a "bitmap" of whether a change in signaling state event has occurred for each DS0. For T1, only the lower 24 bits are used because there are only 24 DS0s and for E1, all 32 bits are used with 0s inserted for DS0 0 and DS0 16 as these channels do not contain signaling data.

7.1.2 Retrieving Signaling State Information: cometqSigExtract

The API function `cometqSigExtract` was designed to work in conjunction with `cometqExtractCOSS`. After searching the bitmap provided by `cometqExtractCOSS`, we provide the user with the new signaling information for the DS0s whose signaling values have changed.

This function provides the values of the four signaling bits, "ABCD", in a single bitmap that contains these signaling values in its lower four bits. The function `cometqSigExtract` provides signaling values for a single DS0 at a time rather than all 24 or 30 DS0s in a single function invocation because we designed it to be a natural extension to `cometqExtractCOSS.`

7.2 Alarm Control and Inband Communications

The COMET and COMET-Quad devices provide features to transmit alarm and to relay information to the remote end within the communications band without disturbing user data. We provide support for these hardware features in the alarm control and inband communications API part of the device driver. We describe the functions that allow the user to insert alarms and receive and transmit loopback and bit oriented codes in this section.

7.2.1 Alarm Insertion: cometqInsertAlarm

The COMET and COMET-Quad devices allow a user to insert alarms into the transmit or the receive data stream. We designed the alarm insertion API as a single function, `cometqInsertAlarm`, that allows a user to enable or disable the insertion of one or all five different alarms that can be generated by a COMET or COMET-Quad device. The user provides us the alarm type as an enumerated type parameter and an enable/disable flag as the parameters to this function. We then clear or set the alarm activation bit in the appropriate device register corresponding to the alarm type.

T1 and E1 alarm concepts are explained in Section 2.2.4 and Section 2.3.4. We support the following alarms in this function:

- AIS insertion into the receive stream. We allow the user to insert an alarm indication signal into the receive stream. When this alarm is enabled, we program the device to insert a stream of unframed 1's onto the receive backplane interface. A user may invoke this type of alarm when a loss of signal occurs at the device input.

- Transmission of AIS. Upon instruction by the user, we program the device to transmit alarm indication signal (AIS) in the transmit stream to the remote end. An application developer may use this feature when there is no valid data being received on the transmit backplane interface.

- Transmission of Yellow alarm (E1 Remote Alarm Indication).

- Transmission of E1 Y-bit alarms. When the user selects this type of alarm, we check the DDB for the specified device to ensure that it is operating in E1 mode. If it is not, we return an error.

- Transmission of E1 DS0 16 AIS alarms. When the user selects this type of alarm, we again check the DDB to ensure that the indicated device is operating in E1 mode. If it is not, we return an error code.

7.2.2 Bit Oriented Code Transmission and Reception

The COMET and COMET-Quad devices allow a user to transmit and receive bit oriented codes (BOCs) to be transmitted in a T1 extend superframe facilities data link, as explained in Section 2.2.5. We designed the COMET and COMET-Quad device driver to allow a user to access the bit oriented code interface in an easy to use manner. Through our BOC application programming interface routines, a user can transmit a bit oriented code and retrieve the most recent bit oriented code received.

7.2.2.1 Transmitting a Bit Oriented Code: cometqBOCTxCfg

We designed the function `cometqBOCTxCfg` to allow a user to transmit a bit oriented code in the T1 ESF FDL. We made this function very simple to use by only requiring the user to provide two parameters: the 6-bit codeword to transmit in bit oriented code format and the number of times to repeat the code. We write the 6-bit code to the correct register on the device and the device then automatically encodes the code into BOC format, as described in Section 2.2.5, and the BOC is transmitted in the outgoing facilities data link.

7.2.2.2 Receiving a Bit Oriented Code: cometqBOCRxGet

The bit oriented code receive hardware on the COMET and COMET-Quad devices scans the facilities data link when the receive framer is configured for T1 ESF mode. Whenever the hardware detects that a valid bit oriented code has been received, an indication bit in a register is set and an interrupt will be generated if the user has enabled the correct interrupt.

After a user determines that a valid BOC code has been received, either by interrupt notification or by polling the indicator flag bit, our driver function, `cometqBOCRxGet`, allows the user to retrieve the six bit message. The user must provide a pointer to a byte into which the BOC code value is to be copied and we read the appropriate register and return the BOC as the least significant six bits of the byte.

7.3 Per DS0 Serial Control: cometqTPSCPCMCtl and cometqRPSCPCMCtl

The COMET device contains two per-DS0 serial controller blocks and the COMET-Quad device contains two per-DS0 serial controller blocks for each of the four channels. One of the blocks is located in the receive data path, immediately before the backplane receive interface. The other block is in the transmit data path, immediately after the backplane transmit interface. Each of these blocks allows the user to manipulate the T1 or E1 stream on a DS0 basis.

The hardware interface for each of these serial controllers consists of a sequential memory. Each memory location corresponds to a T1 or E1 DS0 and there are three memory addresses associated with each DS0 of a T1 or E1 stream: a control byte, a trunk conditioning byte, and a signaling data byte. Each bit within the control byte for a given DS0 performs a specific operation on the DS0 allowing the user to invert bits, overwrite data with the trunk conditioning data, and force signaling data to come from the signaling data byte.

We support these hardware features in the device driver in the serial controller API. We designed the transmit per DS0 serial controller (TPSC) interface function, `cometqTPSCPCMCtl`, and the receive per DS0 serial controller (RPSC) interface function `cometqRPSCPCMCtl`, to both read and write each of the three bytes associated with a DS0. We free the user from the process of interfacing with the controller memory interface by generating the memory address for the control byte, signaling byte, and trunk conditioning byte given the DS0 number provided by the user.

We do not provide the user with abstraction of each of the individual bits within the control byte. We take the value of the control byte as a user parameter to the function and write it directly to the appropriate address in the hardware RAM.

7.4 Status and Counts

In the status and counts section of the device driver, we provide the user with access to the status indicator bits and the count registers on a COMET or COMET-Quad device.

7.4.1 Retrieving Performance Monitoring Statistics: cometqForceStatsUpdate and cometqGetStats

The COMET and COMET-Quad devices contain a performance monitoring hardware block that examine the receive T1 or E1 data stream and maintains counts of the number of framing errors, line code violations, and CRC errors. We provided the user with a convenient interface to retrieve these hardware counters in the functions `cometqForceStatsUpdate` and `cometqGetStats`.

The devices keep the hardware counts in internal holding registers. Whenever the current counter values are to be accessed, an update register must be written to force the hardware to retrieve the internal hardware counters and transfer their values to the registers accessible by the microprocessor. In the function `cometqForceStatsUpdate`, we perform the write to the update register to force the transfer of the counter values. We designed this function as a convenience routine for the user so that the application does not have to perform the register write access directly. With the use of the API function, the details of the counts update procedure are encapsulated within the device driver.

After the counter values have been transferred, the device sets an indicator bit and generates an interrupt if enabled. The user can then service the interrupt using an interrupt service routine or poll the indicator bit using our device driver polling interface. After the user has verified that the hardware counters are now available, we retrieve the available counter values in the function `cometqGetStats`.

7.4.2 Retrieving Device Status: cometqGetStatus

Within the various blocks on the device, there are a number of status bits that indicate the current state of the device. We designed the function `cometqGetStatus` to provide users the most important status bits on the device in a single call. As these bits are not all located within the same hardware block, this function provides abstraction of the internal hardware of the device.

The user passes our function a pointer to a structure that we then fill with the values from the appropriate hardware bits on the COMET or COMET-Quad device. We provide the following status indicators that are read directly from the corresponding hardware bits:

- Loss of signal. Set when there is no signal detected by the device at the input of the receiver.
- Loss of frame. Persists while the T1 or E1 framer does not have frame alignment on the incoming data stream. For E1, the device reports this condition when the framer does not have frame alignment on the 256 bit basic frame unit.
- E1 Loss of Signaling Multiframe and Loss of CRC Multiframe.

- Alarm indication signal.
- Yellow alarm (E1 RAI).
- E1 DS0 16 yellow alarm.

8 Interrupt Servicing Architecture

The COMET and COMET-Quad devices provide interrupts throughout each of the internal hardware blocks corresponding to error, alarm, and event conditions that a software application can be notified of immediately. We provide the user with a mechanism of handling interrupt events by using an interrupt driven mode or an event polling mode in the interrupt servicing architecture of the device driver.

8.1 Interrupt Hardware Bits

For each interrupt on the COMET and COMET-Quad device, there are at least two bits in hardware: the interrupt enable bit and the interrupt status bit. The interrupt enable bit determines whether or not the associated interrupt condition will assert the interrupt line connected to the microprocessor. The interrupt status bit indicates whether or not the interrupt event has occurred. When the interrupt enable bit is on and the interrupt status bit transitions from off to on, the interrupt line is asserted. The interrupt line will remain asserted until the interrupt is acknowledged. For the COMET and COMET-Quad devices, the interrupt is acknowledged by reading the register containing the interrupt status bit. The interrupt status bit is then cleared after the register is read.

In the interrupt service routine, we read the interrupt status registers to determine what interrupt events have occurred. In reading an interrupt status registers, the device clears interrupt events in hardware and the interrupt line becomes de-asserted after we have read and thus cleared all pending interrupts.

When an interrupt enable bit is not set, the interrupt status bit is asserted whenever the interrupt event occurs, however, the interrupt line is not asserted. We exploit this feature

in interrupt polling mode. With all interrupt enables off, we are able to detect interrupt status at any time by reading the interrupt status registers.

Some interrupts contain additional status information. For example, the AIS detection state interrupt hardware contains three bits: AIS interrupt enable bit, AIS interrupt status bit, and AIS status bit. The AIS status bit indicates whether or not AIS is detected on the incoming data stream. Whenever the AIS status bit changes state, indicating that an AIS signal has been detected or is no longer detected, the interrupt status bit is asserted indicating that the AIS status has changed. If the AIS interrupt enable bit is on, the interrupt line will be asserted and the processor will perform a context switch to the interrupt service routine. For interrupts with a status bit related to a device condition, we capture this status value in addition to capturing the interrupt event itself.

8.2 The Device Data Block Interrupt Mask Structure

For each device that is registered with the device driver, we maintain an interrupt mask structure within the device data block. The interrupt mask structure is the `mask` member of the device data block shown in Table 3. We capture the interrupt enable setting of every interrupt on a COMET or COMET-Quad device with an `integer` flag. For COMET devices, we use only a subset of the members of the structure because the device contains only a subset of the interrupts on the COMET-Quad device.

Upon device initialization, all interrupts on a device are disabled. Accordingly, we initialize the interrupt mask in the device data block to all zero values when a device is added to the device driver.

8.3 Selecting the Interrupt Mode: cometqISRConfig

We provide the user with a method of selecting the interrupt mode after the device has been initialized with the `cometqISRConfig` function. The user must pass the function the device handle and the enumerated type value specifying the interrupt mode: polling or ISR.

If the device is in ISR mode and the user selects polling mode, we perform the following:

- Disable task preemption and disable interrupts
- Disable all interrupt enable bits on the COMET or COMET-Quad device
- Clear all pending interrupts by reading all interrupt status registers
- Re-enable task preemption and enable interrupts
- Set the device interrupt mode in the DDB to polling mode

When the user calls `cometqISRConfig` to change from polling mode to ISR mode we the following:

- Disable task preemption and disable interrupts
- Clear all pending interrupts by reading all interrupt status registers
- Apply the interrupt mask in the DDB to the interrupt enable registers
- Enable task preemption and enable interrupts
- Set the device interrupt mode in the DDB to ISR mode

When changing interrupt modes, we must disable task preemption and interrupts to ensure that the transition to the new mode is atomic. By making the process of updating the hardware interrupt enables atomic, we can be assured that the ISR will not execute and modify any registers during the process of updating hardware.

We note that a switch between the two interrupt modes affects the interrupt enable bits in the device, but we retain the interrupt enable mask within the DDB. If a user is in ISR mode and has a set of interrupts enabled on the device, we disable all interrupts in hardware when switching to polling mode but retain the set of interrupt enables within the DDB. When the user changes back to ISR mode, we apply the interrupt mask in the device data block to the interrupt enable registers, restoring the original interrupt enable status.

8.4 ISR Mode

If the device driver is configured to operate in interrupt driven mode and the interrupt line is asserted because of a COMET or COMET-Quad interrupt event, an interrupt service routine (ISR) is invoked. The interrupt service routine captures the interrupt information on the device for deferred processing, and clears all interrupt conditions. We then pass the interrupt information to a task dedicated to processing interrupt conditions.

Our interrupt service routine performs the following sequence of events for every registered device when an interrupt occurs on the system:

- Read each interrupt event indicator bit on the device to determine which interrupt events have occurred. In the process, the pending interrupt events are cleared.

- Capture the interrupt event indicators in a structure called the interrupt service vector.

- Capture any state information that is relevant to the interrupt events that have occurred in the interrupt service vector.

- Disable the interrupt enables for each of the interrupts that have occurred.

- Send the interrupt service vector to the deferred processing routine using a message queue.

The interrupt service routine was designed to perform the minimal required processing and offload the event specific processing to the deferred processing (DPR) task. An interrupt service routine cannot be preempted when executing, and thus if the ISR takes an excessive amount of time to execute, the deterministic scheduling bounds guaranteed by the real time operating system may no longer apply. In the case of the COMET and COMET-Quad device driver, we only read the interrupt status registers and save this information in a structure in the ISR.

In addition to capturing what interrupts have occurred, we designed the ISR to read and save any additional required information for any events that have occurred. We capture this information within a structure called the interrupt service vector (ISV), along with the flags indicating which interrupts have actually occurred.

The DPR task is a high priority task that performs the required processing for a given interrupt event. We pass the interrupt service vector from the interrupt handler to the DPR task using a message queue. We can perform detailed processing of the interrupt

conditions using the information provided by the ISV in the deferred processing task, outside of the interrupt context.

Figure 15 summarizes how an interrupt is handled in our architecture. The sequence of events and the interactions between the interrupt handler, the deferred processing task, and the user application when the interrupt line is asserted is indicated by the following:

1. COMET or COMET-Quad device generates an interrupt and the interrupt service routine on the controlling CPU is invoked.

2. ISR reads the interrupt indicator bits on the COMET or COMET-Quad device and determines which interrupt events have occurred. By reading the interrupt indicator bits, we clear the interrupt. We disable the interrupt enables on the device for the interrupts that have occurred.

3. ISR creates an interrupt service vector (ISV) structure that records the interrupt events that have occurred. We then send this structure from the ISR to the deferred processing task using a message queue. The ISR terminates and control returns to the operating system.

4. The deferred processing task is waiting for a message and is signaled by the operating system that an ISV message has arrived.

5. The deferred processing calls the appropriate user application callback functions for each event that is flagged in the ISV. The user callback functions perform the processing of the interrupt event defined by the user application. We note that the user interrupt callback function is invoked in a

task context, opposed to an ISR context, and specifically, the interrupt callback function executes in the context of the deferred processing task.

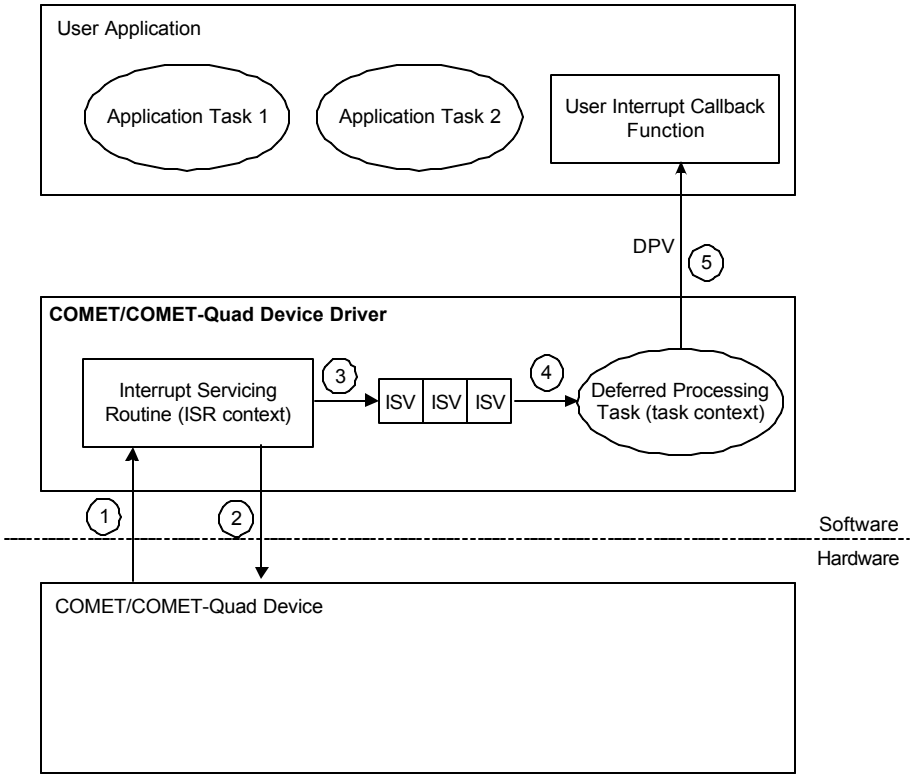

Figure 15 – Interrupt servicing architecture.

8.5 User Callback Functions and the User Context

As we have stated previously, the deferred processing task performs detailed interrupt processing required for an interrupt event. We examine the fields within the ISV and then invoke callback functions within the DPR task. The user registers the callback functions with a device upon initialization, as shown in Table 6.

We define the user callback function prototype to take two parameters: the user context and the deferred processing vector. From the context of our driver, the user context

parameter is a `void` pointer, or a pointer to data of a type that is application specific. As we mentioned in discussion of `cometqAdd` in Section 4.3.3.1, the user context for a device is defined when the device is added and we store the user context within the device data block. We then pass this pointer back to the application in the invocation of the callback function as a parameter so that the user can identify the device on which the interrupt occurred.

The second callback parameter is a pointer to a structure called the deferred processing vector (DPV). We have defined this structure to contain event indicator fields that consist of a series of bit definitions within a number of four byte integer values. We defined each bit to represent one of the interrupts on a COMET or COMET-Quad device. We set these bits to reflect the interrupts that have occurred in hardware in the deferred processing task before invoking the callback. The user can then use the DPV to determine what interrupts have occurred in the callback function.

8.6 Polling Mode: cometqPoll

We provide a means to poll interrupt conditions for those users who do not want to handle interrupts. We support this feature by allowing a user to configure the device driver for polling mode using our driver function, `cometqISRCfg`.

When a device is in polling mode, we disable all interrupt enables on the device. However, when an interrupt is disabled on the device and the interrupt condition occurs, the interrupt indicator is still set despite the interrupt line not being asserted. Thus, we can poll the indicator bits in the device driver to determine what interrupt conditions currently exist.

The sequence of operations we perform when a user invokes `cometqPoll` to poll for interrupt events is shown in Figure 16. When in polling mode, we do not use the DPR task. We read the status registers and then invoke the DPR task's main routine directly, bypassing the message queue transmission. We perform application callbacks in the same context as the application task that calls `cometqPoll`. Each of the numbered sequences correspond to the following actions:

1. A user application task initiates polling of the device by calling the `cometqPoll API` function, indicating which COMET or COMET-Quad device to poll.

2. We read the interrupt indicator bits on the COMET or COMET-Quad device and determine which interrupt events have occurred. For an interrupt that is active and has been enabled in the device data block, we save the indicator bit information in an interrupt service vector.

3. We directly invoke the deferred processing routine main function, passing the interrupt service vector structure as a parameter.

4. Executing in the user task context, we perform the identical operations as the DPR task when in ISR mode. We invoke user application callbacks for the active conditions as required for the fields in the interrupt service vector structure.

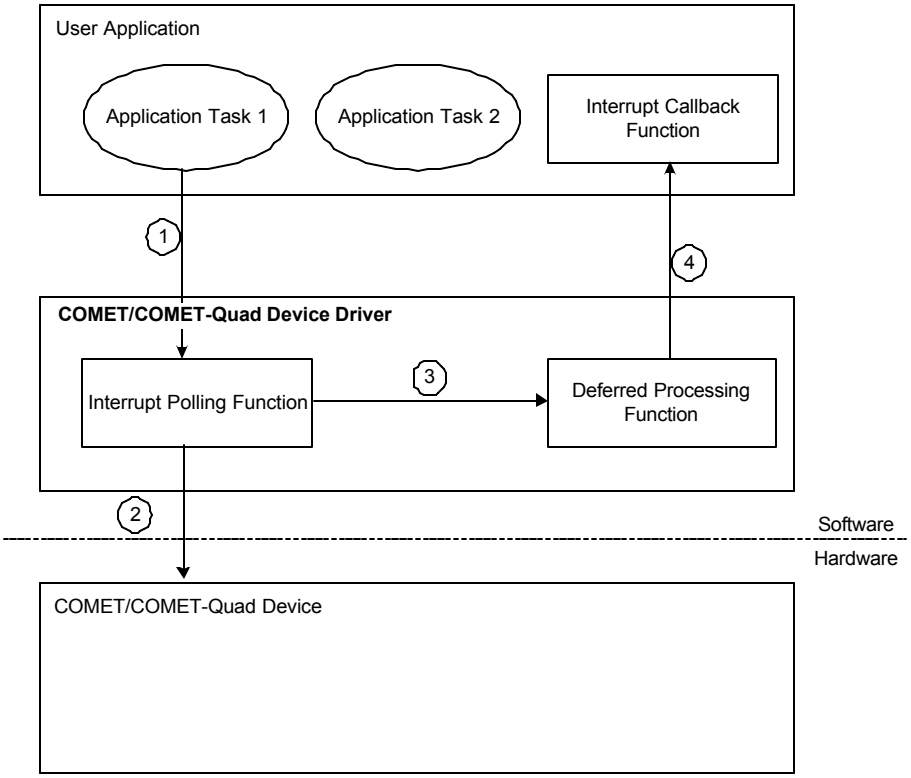

Figure 16 – Interrupt polling architecture.

9 Diagnostics Interface

We provide two types of diagnostics in the device driver: register write and readback tests and data loopbacks. The register tests provide a means to verify the integrity of the COMET or COMET-Quad device registers while the data loopback modes allow the user to verify their own system.

9.1 Register Write and Readback Test: cometqTestReg

In this function, we verify basic integrity of the COMET or COMET-Quad device. We write the sequence "01010101", "10101010", "10100101", and "01011010" to the registers on the device and read the values back immediately. If the registers are functioning and the device is alive, we will read the same value as was written. If the values are not the same, we return an error code and identify the corrupt register address to the user.

9.2 Configuring Loopback Modes: cometqLoopFramer

The hardware supports three loopback modes: line loopback, payload loopback, and system side digital loopback. We support all three of these loopback modes in the device driver. Both line loopbacks and payload loopbacks operate on the line interface side of the device that interfaces with the analog line. System side digital loopbacks are used to loopback data on the backplane. To configure a loopback or to disable a loopback, the user must pass our function, `cometqLoopFramer`, an enumerated value indicating whether to enable one of the three loopback modes or to disable an already active loopback.

When a COMET or COMET-Quad device is in line loopback, the device performs analog to digital conversion of the input signal and decodes the line code. The device then

performs line coding and forwards data to the analog transmitter and transmits an analog signal back to the user. Line loopback does not modify the data in any way and serves as a diagnostic tool for the configuration of the analog transmitter and receiver on a COMET or COMET-Quad device when using a T1 or E1 traffic generator or as a diagnostic tool for the far end equipment.

Payload loopback aligns to the framing of the received signal and extracts the payload. When the device is configured for payload loopback, the device removes the payload from the received signal and then reinserts the payload into a T1 or E1 frame to transmit. With payload loopback, the T1 or E1 DS0 data is not modified by the device, however, the device regenerates the T1 framing bit or the E1 DS0 0 and/or DS0 16 framing. Payload loopback can be used to verify the configuration of the framers and data path on the COMET or COMET-Quad device or as a diagnostic tool for the far end equipment.

When the device is configured for system side digital loopback, the device takes data received on the backplane transmit interface and reinserts it onto the backplane using the backplane receive interface. System side loopback can be used to verify the configuration of the backplane on the COMET or COMET-Quad device or to verify that the remote end backplane hardware is functioning.

We provide the user with access to each of these three hardware features by setting the diagnostics bits on the COMET or COMET-Quad device. The device then performs the loopback function. When the user parameter indicates that no loopback is to be active on the device, we disable the bits that activate line, payload, and system side loopback.

10 Testing the Device Driver

In order to verify that the device driver code was bug free, we developed and executed a comprehensive test plan. The test plan implemented tests for each API function within the device driver. The test plan consisted of two types of tests: parameter and state checking tests and functional tests.

Each API function validates the parameters that are passed by the user when invoking the function. If an invalid parameter is discovered, the function fails and returns an invalid argument error code. In order to verify that each API function is validating parameters properly, we included tests for each API function where it was called with invalid parameters. We ensured that the API function returned an error for these cases.

In addition to validating the parameters, each API function must ensure that the device driver has been initialized and the device data block is in the correct device state for the given function. Each API function can only be invoked for a subset of the three states: present, active, and inactive. As such, we included tests in the test plan where each function was invoked in each state and we ensured that an error code was returned when a function was called in an invalid state.

In the functional set of tests, we verify that each API function is working properly. We tested each API function to ensure that each register is being configured properly and that the COMET or COMET-Quad device is operating as expected.

10.1 API Function Testing

Functional testing of the API functions verified that each API function was configuring the intended registers properly and that the device was functioning as expected. In the

functional testing of the device driver, we performed tests with T1/E1 traffic running through the device and tests where an API function was executed to configure a section of the device into a certain mode. We then read the relevant registers back directly to ensure that the device driver was configuring the registers properly.

10.1.1 Basic Device Driver Test

The basic device driver test was where we began initial testing of the device and this test involved executing the minimal required series of our API functions to perform a line and payload loopback on a COMET and a COMET-Quad device in T1 and E1 mode. The first step was to successfully perform a line loopback in T1 and E1 mode. In doing so, we were able to confirm that our device driver was configuring the transmitter and receiver properly. Upon verifying that we were configuring the line interface on the device properly, we began testing our framer configuration API by performing a payload loopback. We used the Fireberd 6000A T1/E1 tester for this test because we only required basic generation of a framed T1 or E1 data stream. In addition to establishing data flow through each device in both T1 and E1 mode, we performed the tests on both COMET and COMET-Quad devices to ensure that our device driver was operating correctly on both devices.

We invoked the following sequence of API functions to perform a line loopback:

- `cometqInit` – enabled the device for the inactive state. We set the `initDevice` flag to zero in the DIV to manually configure the device.

- `cometqSetOperatingMode` – selected T1 or E1 mode

- `cometqLineTxAnalogCfg` – configured analog transmitter
- `cometqLineRxAnalogCfg` – configured analog receiver
- `cometqLineTxEncodeCfg` – configured transmit line coding
- `cometqLineRxEncodeCfg` – configured receive line coding
- `cometqLoopFramer` – enabled line loopback mode

When we established payload loopback on the COMET and COMET-Quad devices, we performed the following sequence of API functions:

- `cometqInit` – enabled the device for the inactive state. We set the `initDevice` flag to zero in the DIV to manually configure the device.
- `cometqSetOperatingMode` – selected T1 or E1 mode
- `cometqLineTxAnalogCfg` – configured analog transmitter
- `cometqLineRxAnalogCfg` – configured analog receiver
- `cometqLineTxEncodeCfg` – configured transmit line coding
- `cometqLineRxEncodeCfg` – configured receive line coding
- `cometqT1TxFramerCfg`, `cometqT1RxFramerCfg` – configured framing mode when testing in T1 mode

- `cometqE1TxFramerCfg`, `cometqE1RxFramerCfg` – configured framing mode when testing in E1 mode

- `cometqLoopFramer` – enabled payload loopback mode

10.1.2 Testing the Initialization Profile

By performing the basic device driver test first, we were able to verify that our basic configuration algorithms were correct. When we initialize a device using a device initialization vector or initialization profile, the same code is used in the API functions as is used in `cometqInit` to apply the DIV hardware configuration to the device.

We tested the initialization profile in two stages. The first stage was to test that the analog configuration and the framer configuration were working as expected. We configured the device for payload loopback by creating and initializing a DIV structure and then applying it to the device. We then invoked `cometqLoopFramer` to enable payload loopback. On the COMET-Quad, we performed data loopback on all four quadrants to ensure correctness in programming each quadrant.

In the second stage of the initialization testing, we verified that the backplane configuration was correct. Because there was no hardware available to interface to the backplane and to thus test the backplane configuration by passing data, we verified the configuration by applying a DIV and then verified that the backplane registers contained the correct values. We tested the backplane transmit and receive configuration functions in the device driver in this manner.

10.1.3 Interface Configuration API Testing

10.1.3.1 Line Side API

We tested the line side configuration functions for basic correctness implicitly when the basic device driver test was performed. We then tested each possible option in this API by calling each function and verifying that the registers on the device contained the correct values.

10.1.3.2 Backplane Configuration API

As mentioned in the initialization profile testing section, there was no hardware available to interface with the backplane interface on a COMET or COMET-Quad device. As a result, we tested the backplane configuration API functions using direct register reads. We called each API function with different configurations and read the values back from the relevant registers to ensure that the configurations were being applied properly.

10.1.4 Device Operation API Testing

10.1.4.1 Signal Extraction API

We tested the signal extraction API by passing T1 and E1 data through the device and used the HP Cerjac tester for these tests. This test equipment allowed us to select the values of the four signaling bits to transmit in each DS0 within the T1 or E1 data stream.

In the signaling test, we initially configure the tester to transmit a certain sequence in the signaling bits for a given DS0 in the data stream. In this case, `cometqExtractCOSS` reports no change of signaling state events and `cometqSigExtract` returns the signaling values as configured in the tester. After the initial test, we changed the signaling state

generated on the tester for the given DS0 and the subsequent call to `cometqExtractCOSS` reported the correct DS0 indicating a change of signaling state. We then called `cometqSigExtract` and retrieved the correct new signaling value.

10.1.4.2 Alarm Insertion API

We tested the function `cometqInsertAlarm` using the HP Cerjac tester. We generated transmit AIS alarms and yellow alarms for T1 data and transmit AIS, yellow alarms, E1 DS0 16 AIS, and E1 Y-Bit alarms for E1 data. We then inserted these into the transmit stream individually and the HP Cerjac tester then correctly verified that they were present. We disabled each of the alarm conditions using the API to ensure that the function correctly disables the alarm condition.

We did not test AIS insertion in the receive data stream using the test equipment because this alarm is inserted onto the backplane. We did verify, however, that we were turning the correct bits on in hardware by reading the registers.

10.1.4.3 Bit Oriented Code and Inband Code API

We were able to test the bit oriented code and inband code API by invoking the functions with parameters and verifying that the correct values were written to the registers. We did not test these API functions using any test equipment because the test equipment lacked the required features to monitor the T1 ESF facilities data link. Because each of these functions operate on only a few bits within a register, this testing was sufficient.

10.1.5 Per DS0 Serial Control API

Using the HP Cerjac test equipment, we were able to examine an individual DS0 and insert a given sequence into a DS0. During payload loopback, we used the tester to insert the bit sequence "01101001" into a DS0. We inverted the contents of the DS0 by calling `cometqTPSCPCMCtrl` and then verified the inverted sequence on the tester. Thus, we were able to verify the correct operation of `cometqTPSCPCMCtrl`.

We could not test `cometqRPSCPCMCtrl` because it modifies data immediately before it enters the backplane. We tested this by writing control bytes to the device using the API function and then performing direct register read accesses to verify that the values were being written to the correct registers.

10.1.6 Status and Counts API

In testing this API section, we used the HP Cerjac tester. We tested the device statistics API, `cometqForceStatsUpdate` and `cometqGetStats`, by using the tester to insert framing errors and line code violations into the data stream. We invoked these two API functions to clear the internal COMET or COMET-Quad device counts. We used the tester to insert errors and then retrieved counts again from the device. The second time, we found that the counts had incremented as expected, indicating that the device driver was correctly reading the appropriate registers.

We tested the function `cometqGetStatus` by configuring the device for payload loopback with data running through the device without any alarms. We called this function with this state and no error conditions were reported, as expected. We then inserted alarm conditions individually and with each call to `cometqGetStatus`, the

driver reported the alarm conditions to be active. We disconnected the tester from the device to generate the major alarm conditions loss of frame and loss of signal and found that our device driver reported both conditions correctly.

10.2 Interrupt Testing

Our testing of the interrupt architecture was one of the most important parts of the test plan. We initially verified that the interrupt architecture was behaving as expected and later ensured stability of the architecture by testing interrupts from multiple COMET and COMET-Quad devices in the system.

Initially, when we tested the basic interrupt service routine and deferred processing task functionality, we used the loss of signal interrupt. We configured the device for payload loopback with traffic running smoothly through the device. We enabled the loss of signal interrupt using `cometqISRSetMask` and then removed the input signal from the tester into the device. We implemented the application callbacks to display strings to the user console on the target to indicate the current interrupt event.

Our initial testing verified that a single interrupt could be generated and handled successfully by the interrupt service routine. We also verified communication between the interrupt service routine and the deferred processing task was operating correctly and that the deferred processing task was performing callbacks properly. The next stage was to test the different interrupt events that could be generated using the given test equipment. We inserted error conditions and alarm conditions using the tester and found that the correct callback strings were reported to the console.

We initially tested interrupts using a COMET device. Once we determined that we could successfully handle interrupts on a COMET device, we proceeded to test the interrupt architecture on a COMET-Quad device. On COMET-Quad devices, our interrupt handling routine must correctly detect and report the framer number on which the interrupt event occurred. We generated interrupts on the different framers to verify that there were no bugs in handling different framers or multiple framers simultaneously.

After we verified that the interrupt architecture was functional for an individual COMET or COMET-Quad device, we generated interrupts with both devices in the system simultaneously. Our goal was to ensure that the interrupt architecture was stable when multiple devices were in the system. We generated a constant stream of interrupts by enabling the line code violation interrupt and selecting an error insertion rate of 10^{-5} on the tester. In this manner, we proved that the interrupt architecture was stable when it was under stress.

We tested the polling mode interface by verifying that the ISR mode was stable and bug free. We then configured the device for polling mode using `cometqISRConfig` and then generated all of the interrupt conditions in that we had used for testing in ISR mode. We invoked `cometqPoll` after generating the event and verified that the user callback function reported the correct event to the console.

10.3 Summary of Test Results

Upon completion of the test plan, we found that all API were functioning correctly. We found there were some minor bugs in the code where we may have been toggling the

wrong bits or accessing the incorrect register. However, there were no design flaws, as we did not have to restructure any of the device driver API functions.

Given that our device driver is for framer devices, there was very little data processing and we primarily performed configuration of the COMET and COMET-Quad devices. As a result, there is no relevant performance metric for the device driver.

11 Conclusions

Our device driver for PMC-Sierra, Inc.'s COMET and COMET-Quad T1 and E1 transceiver integrated circuits is a hardware abstraction layer that encapsulates the many features of the devices in an easy to use application programming interface. To design, implement, and test the COMET and COMET-Quad device driver, we required a thorough understanding of T1 and E1 networking concepts as well as the underlying hardware. We extended this knowledge to develop a software interface for an application developer that was user-friendly, that provided sufficient encapsulation of device features, and that was highly portable.

Central to our goal of a functional and bug free device driver was the test plan. We tested each function in the application programming interface of the device driver using a methodological test plan. We ensured that each function was not only functional correct, but that possible user errors were handled gracefully. After we completely executed the test plan, we found that our device driver was performing all functions correctly after minor bug fixes; the design of the device driver and the structure of the application programming interface functions were correct.

Possible future enhancements could include a performance monitoring application that uses the raw counts in the status and counts interface. In the future, our software architecture developed for the COMET and COMET-Quad device driver could easily scale to support COMET devices of higher density.

12 List of Acronyms

AIS	Alarm Indication Signal
AMI	Alternate Mark Inversion
ANSI	American National Standards Institute
API	Application Programming Interface
B8ZS	Bipolar with 8 Zero Substitution
BOC	Bit Oriented Code
BRIF	Backplane Receive Interface
BTIF	Backplane Transmit Interface
COMET	Combined E1 and T1 Transceiver
COSS	Change of Signaling State
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DDB	Device Data Block
DIV	Device Initialization Vector
DPV	Deferred Processing Vector
DS1	Digital Signal Level 1
ESF	Extended Superframe
FAS	Frame Alignment Signal
FDL	Facilities Data Link
FEBE	Far End Block Error
GO-MVIP	Global Organization for High Density Multi-Vendor Integration Protocol
HDB3	High Density Bipolar Order 3
HDLC	High Level Data Link Control
H-MVIP	High Density Multi-Vendor Integration Protocol
ISR	Interrupt Service Routing
ISV	Interrupt Service Vector
ITU-T	International Telecommunications Union
LAN	Local Area Network
LAPD	Link Access Procedure, D-Channel
MDB	Module Data Block
MIV	Module Initialization Vector
NFAS	Non-Frame Alignment Signal
OAM	Operation, Administration, and Maintenance
PAM	Pulse Amplitude Modulation
PCI	Peripheral Component Interconnect
PCM	Pulse Code Modulation
RAI	Remote Alarm Indication
RPSC	Receive per Channel Serial Controller
RTOS	Real Time Operating System
SF	Superframe
SNR	Signal to Noise Ratio
TPSC	Transmit per Channel Serial Controller
WAN	Wide Area Network

13 References

1. ANSI - T1.107-1995 - American National Standard for Telecommunications - Digital Hierarchy - Formats Specification.

2. ANSI - T1.231-1993 - American National Standard for Telecommunications - Layer 1 In-Service Digital Transmission Performance Monitoring

3. ANSI - T1.403-1995 - American National Standard for Telecommunications - Carrier to Customer Installation - DS-1 Metallic Interface Specification.

4. J. Bellamy. *Digital Telephony*, 2nd Edition. New York, NY, John Wiley: 1991.

5. W. Flanagan. *T-1 Networking*, 5th Edition. New York, NY, Telecom Books: 1997.

6. GO-MVIP – H-MVIP Standard, Release 1.1a, 1997.

7. ITU-T - Recommendation G.704 - Synchronous Frame Structures Used at Primary Hierarchical Levels, July 1995.

8. ITU-T - Recommendation G.706 - Frame Alignment and CRC Procedures Relating to G.704 Frame Structures, 1991.

9. ITU-T - Recommendation G.711 – Pulse Code Modulation (PCM) of voice frequencies, 1988.

10. B. P. Lathi. *Modern Digital and Analog Communication Systems*, 3rd Edition. New York, NY, Oxford University Press: 1998.

11. PMC-Sierra, Inc. – COMET-Quad Long Form Data Sheet. http://www.PMC-Sierra.com/products/details/pm4354/

12. PMC-Sierra, Inc. – COMET Long Form Data Sheet. http://www.PMC-Sierra.com/products/details/pm4351/

13. PMC-Sierra, Inc - COMET and COMET-Quad Device Driver User's Manual. http://www.PMC-Sierra.com/products/details/pm4354/