MicroTCA™

Standard Device Model Design Guide

Guidelines for designing I/O access software for MTCA.4 systems

MTCA_DG.2 R1.0

August 30, 2017

Contents

Figures

Tables

Caveat

This design guide is not a specification. It contains additional detailed information but does not replace any applicable PICMG specifications.

For complete requirements on the design of MTCA.4 or MTCA.4.1 compliant boards and systems, refer to the full specification – do not use this design guide as the only reference for any design decisions.

1 Preface

1.1 About This Document

This guideline defines the functions and Application Programming Interfaces (APIs) for the Standard Device Model (SDM) developed for the MTCA.4 effort within PICMG. It is applicable to systems developed in conjunction with the standards released as part of that effort, and more generally to systems developed for instrumentation and machine control applications.

This guideline defines a standard operating model and Application Programming Interface (API) for code development to facilitate module re-use and portability. It is recommended, but not required, that applications developed for use with MTCA.4 or MTCA.4.1 systems make use of these guidelines to the greatest reasonable extent.

This design guide is not a specification. It contains additional detail information, but does not replace any applicable PICMG specifications.

For complete guidelines on the design of MTCA.4 or MTCA.4.1 compliant boards and systems, refer also to the full specification – do not use this design guide as the only reference for any design decisions.

1.2 Intended Audience

This design guide is intended for software engineers and programmers designing software for use with MTCA.4 or MATCA.4.1 systems.

1.3 No Special Word Usage

Unlike a PICMG specification, which assigns special meanings to certain words such as “shall”, “should” and “may”, there is no such usage in this document. That is because this document is not a specification; it is a non-normative design guide.

1.4 No Statements of Compliance

As this document is not a specification but a set of guidelines, there should not be any statements of compliance made with reference to this document.

1.5 Correctness Disclaimer

The code examples given in this document are believed to be correct but no guarantee is given. In most cases the examples come from designs that have been built and tested.

1.6 Name and Logo Usage

The PCI Industrial Computer Manufacturers Group's policies regarding the use of its logos and trademarks are as follows:

Permission to use the PICMG organization logo is automatically granted to designated members only as stipulated on the most recent Membership Privileges document (available at www.picmg.org) during the period of time for which their membership dues are paid. Nonmembers must not use the PICMG organization logo.

The PICMG organization logo must be printed in black or color as shown in the files available for download from the member's side of the Web site. Logos with or without the "Open Modular Computing Specifications" banner can be used. Nothing may be added or deleted from the PICMG logo.

The PICMG® name and logo are registered trademarks of The PICMG®. Registered trademarks must be followed by the ® symbol, and the following statement must appear in all published literature and advertising material in which the logo appears:

PICMG and the PICMG logo are registered trademarks of the PCI Industrial Computer Manufacturers Group.

1.7 Intellectual Property

The Consortium draws attention to the fact that implementing recommendations made in this document could involve the use of one or more patent claims ("IPR"). The Consortium takes no position concerning the evidence, validity, or scope of this IPR.

Attention is also drawn to the possibility that some of the elements of this specification could be the subject of unidentified IPR. The Consortium is not responsible for identifying any or all such IPR.

No representation is made as to the availability of any license rights for use of any IPR that might be required to implement the recommendations of this Guide. This document conforms to the Specification Development and does not contain any known intellectual property that is not available for licensing under Reasonable and Nondiscriminatory terms. In the course of Membership Review the following disclosures were made:

Necessary Claims (referring to mandatory or recommended features):

- No disclosures in this category were made during Adoption Ballot

Unnecessary Claims (referring to optional features or non-normative elements):

- No disclosures in this category were made during Adoption Ballot

Third Party Disclosures (Note that third party IPR submissions do not contain any claim of willingness to license the IPR.):

- No disclosures in this category were made during Adoption Ballot

THIS DOCUMENT IS BEING OFFERED WITHOUT ANY WARRANTY WHATSOEVER, AND IN PARTICULAR, ANY WARRANTY OF NONINFRINGEMENT IS EXPRESSLY DISCLAIMED. ANY USE OF THIS DOCUMENT SHALL BE MADE ENTIRELY AT THE IMPLEMENTER'S OWN RISK, AND NEITHER THE CONSORTIUM, NOR ANY OF ITS MEMBERS OR SUBMITTERS, SHALL HAVE ANY LIABILITY WHATSOEVER TO ANY IMPLEMENTER OR THIRD PARTY FOR ANY DAMAGES OF ANY NATURE WHATSOEVER, DIRECTLY OR INDIRECTLY, ARISING FROM THE USE OF THIS DOCUMENT.

1.8 Copyright Notice

PICMG has made every attempt to ensure that the information in this document is accurate, yet the information contained within is supplied "as-is".

Trademarks

Add references to trademarks for things mentioned within the guideline. Candidates:

ATCA and µTCA are registered trademarks of the PCI Industrial Computer Manufacturers Group (PICMG).

PCI Express is a registered trademark of Peripheral Component Interconnect Special Interest Group (PCI-SIG).

All product names and logos are property of their owners.

1.9 Acronyms and Abbreviations Used

Table 1-1: Acronyms and Abbreviations Used

Word(s)	Definition
A/D	Analog-to-Digital
ADC	Analog-to-Digital Converter
AIO	Analog Input/Output
API	Application Programming Interface
ATCA	Advanced Telecommunications Computing Architecture
BIOS	Basic Input/Output System
BSP	Board Support Package
D/A	Digital-to-Analog
DAC	Digital-to-Analog Converter
DAQ	Data Acquisition
DMA	Direct Memory Access
GPIB	General-Purpose Interface Bus
GPIO	General-Purpose (digital) Input/Output
HAL	Hardware Access Layer
HIU	Hardware I/O Unit
I2C	IIC
IANA	Internet Assigned Numbers Authority
IIC	Inter-Integrated Circuit (bus)
I/O	Input/Output
IP	Internet Protocol
IPMB	Intelligent Platform Management Bus
IPMI	Intelligent Platform Management Interface
Acronym	**Definition**

LAN	Local Area Network
µTCA	Micro-ATCA
OS	Operating System
PCI	Peripheral Component Interconnect (bus)
PCIe	PCI Express
PICMG	PCI Industrial Computer Manufacturers Group
RMCP	Remote Management Control Protocol
SDCM	Standard Device Configuration Manager
SDM	Standard Device Model
SDNR	Standard Device Name Resolver
SDTL	Standard Device Translation Layer
SHAPI	Standard Hardware API
SPI	Serial Peripheral Interface (bus)
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
UDP	Universal Datagram Protocol
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
SDM	Standard Device Model
SDNR	Standard Device Name Resolver
SDTL	Standard Device Translation Layer
SHAPI	Standard Hardware API
SPI	Serial Peripheral Interface (bus)
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver/Transmitter
UDP	Universal Datagram Protocol
USART	Universal Synchronous/Asynchronous Receiver/Transmitter
USB	Universal Serial Bus
WAN	Wide Area Network
xTCA	ATCA and/or µTCA

1.10 Model Terminology

For purposes of describing the Standard Device Model and its environment, the following terminology is applicable and is illustrated in Figure 1-1 using, as an example, creation of a virtual ADC device from a physical oscilloscope:

Table 1-2: Model Terminology

Term	Definition
Board Support Package (BSP)	A software module at the bottom layer of the software hierarchy which provides functional access to various hardware components, configurations, and platform services.
Device	General term for a path by which information is moved in or out of the platform
Device Driver	A device object which provides logical access, through some OS-specific access model, to a physical device.
Device Object	A software module that provides access to a device.
Element	The smallest unit in which data may be transferred through a device API; typically some integer number of bytes.
Logical Device	A physical device and its associated device drivers
Physical Device	A hardware I/O unit and its associated HAL software modules
Platform	The computer system on which the Standard Device Model operates
Remote HIU	A hardware I/O unit which is remote from the platform and accessed through a communication channel
Standard Device	A device object which provides logical access to a device using the Standard Device Model API
Translation Unit	A device object that translates between two different API models
Virtual Device	A device object which provides a logical model and API for an I/O device type that is not tied to a particular logical or physical device implementation.

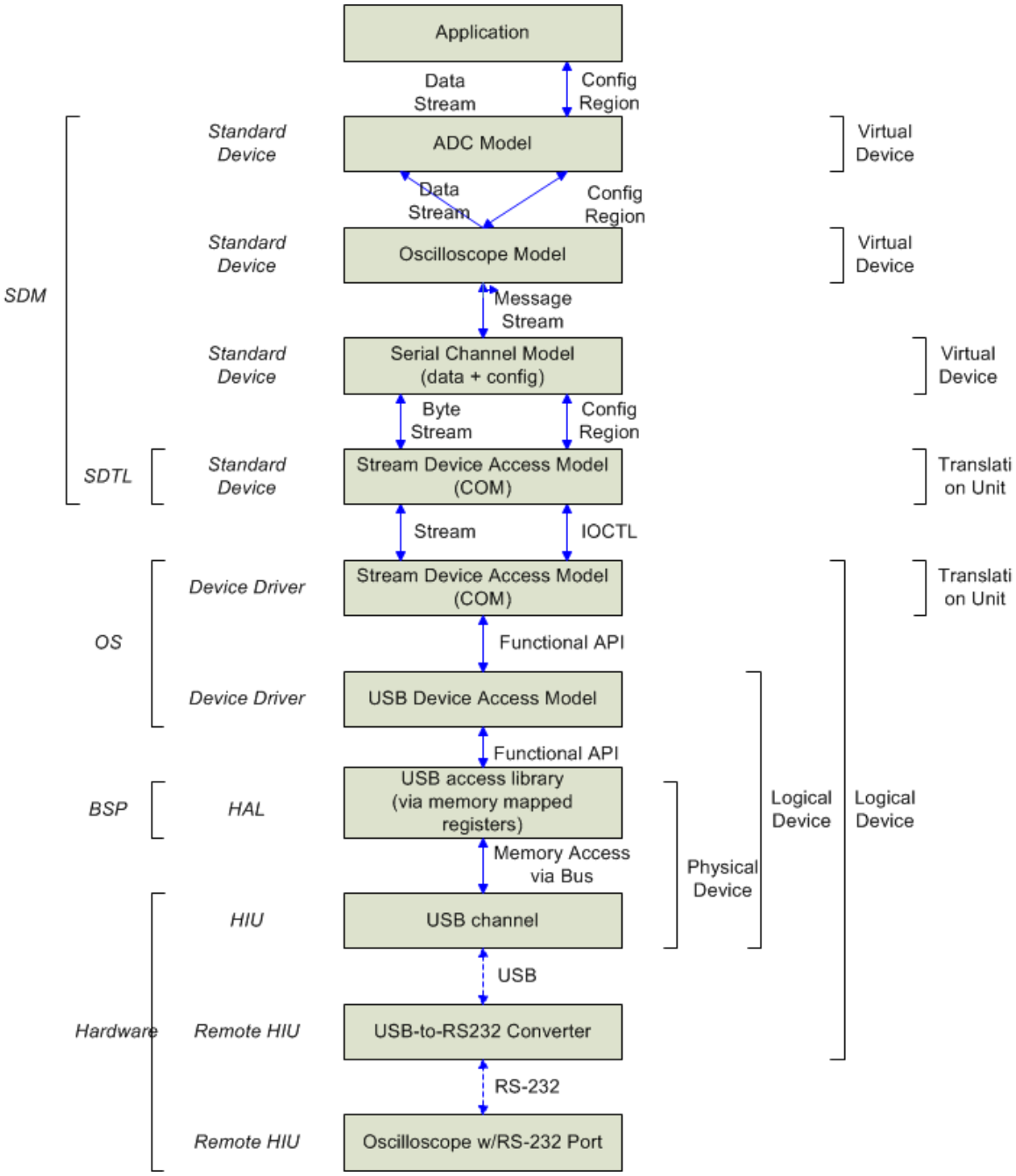

Figure 1-1: Device Model Nomenclature

2 Introduction

2.1 Overview

The Advanced Telecommunication Computing Architecture (ATCA) specification and related standards maintained by the PCI Industrial Computer Manufacturers Group (PICMG) define an infrastructure for development of distributed data processing systems for the telecommunications industry. At the urging of the experimental physics community a set of committees were formed under the auspices of PICMG to extend the ATCA family of standards for use in embedded instrumentation and machine control applications, with particular emphasis on scientific applications. These committees, known collectively as the MTCA.4 or MATCA.4.1 committees, were chartered to adapt and/or extend the existing ATCA family specifications to accommodate sensor and control I/O, including: support for analog signals; timing, synchronization, and interlock mechanisms required for instrumentation and control operations; and low latency data distribution protocols to support time-critical data processing and control constraints.

As part of the standardization effort the committees also defined common development architectures to encourage hardware and software component interoperability and portability among the various scientific centers. To that end, guidelines for various aspects of software development of MTCA.4 or MATCA.4.1 systems were developed and published.

This guideline defines the function and usage for a Standard Device Model (SDM) for use in MTCA.4 or MATCA.4.1 software applications.

2.2 Nomenclature

Recommendations are designated within this document by the words **shall**, **will, should", is,** and **are.** These terms are interchangeable and usage is driven entirely by context and stylistic considerations.

Although this is a guideline, rather than a standard, its intent is to define mechanisms and practices that facilitate interoperability of software modules across different projects and facilities. As such, it is useful to provide language that allows a discussion among implementers about the degree to which their applications do or do not conform to the recommendations presented here, and therefore the degree to which they can expect that goal of interoperability to be achieved. To that end, four levels of recommendation may be identified:

- **Mandatory**: These recommendations must be implemented to be fully conformant with this guideline.

- **Required:** These recommendations must be implemented to be fully conformant with this guideline, but may be excluded from a particular implementation provided the release notes for the implementation specifically identify them as areas of non-conformance. An implementation with exclusions may be considered "conditionally conformant".

- **Desired:** These recommendations comprise design goals, performance targets, and "nice to haves" that are desirable but not necessary for full conformance with this guideline. Application developers should not depend on availability of these items on all implementations.

- **Guidance**: These recommendations are provided for guidance and clarification to the designer regarding the intent of other specific recommendations, expected guideline usage, external interfaces and constraints, and preferred or potential design approaches, technologies, and practices. These items should be taken into account during design and deviations may be noted within the standard design documentation, but they are not binding for design.

Required, Desired, and Guidance recommendations must be specifically designated as such within the text of the specification. Any recommendation without such a designation is presumed to be *Mandatory*.

2.3 Background

2.3.1 Context and Rationale

The Standard Device Model (SDM) defined in this guideline is intended to apply to applications developed in conjunction with the MTCA.4 or MATCA.4.1 family of standards from PICMG. These applications typically involve large networks of distributed computing elements, sensors, actuators, and signal generators forming integrated control and data acquisition/analysis systems. Although such systems will often make use of large general-purpose computing platforms for system monitoring/control and offline analysis, a large fraction of software components in such systems operate within embedded environments, remote from the high-level control/analysis systems and with soft and hard real-time requirements.

Further, the operating environment for such systems is anticipated to be dynamic in terms of both hardware and software configuration as experiments evolve, and to involve a high degree of sharing, both of expertise and of actual hardware and software modules, among the various laboratories participating in the work. Both the dynamic nature of the environment and the degree of sharing between facilities leads to a strong desire for a software development infrastructure that facilitates rapid prototyping of new components,

efficient code development and validation, and code portability. That desire has been formalized by the PICMG MTCA.4 or MATCA.4.1 Software and Protocols committee as a set of guidelines for creating standard I/O interfaces, communication protocols, and other commonly used functional blocks and APIs. This standard defines the device model which supports I/O and communication functions.

The purpose of the Standard Device Model (SDM) is to provide a standardized and portable way for software applications to interact with physical I/O and communication devices.

Most operating systems provide a means to interact with hardware devices. The interface is generally provided via a software module called a Device Driver. Device Drivers are commonly integrated with the operating system and, in operating systems that erect barriers between application software and system-level resources to prevent unintended interactions between applications, operate at an elevated privilege level allowing them to interact directly with those resources and with the system hardware itself. While there are many common Device Driver interfaces, there is no single standard method or API for accessing devices, which makes it difficult to write application software that is portable across a wide range of platforms.

The SDM defines a standardized paradigm for creating an Application Programming Interface (API)[1] suitable for interacting with most types of hardware devices. As noted above, historically, hardware access mechanisms have been specific to a particular operating system. In any real implementation those operating-system-specific mechanisms will still be necessary but, to facilitate general code portability, the SDM provides a layer between the operating system and the application with a standard API at the application layer. If fully and properly implemented, porting of code from one platform to another would, therefore, require only an operating-system-specific adaptation of the SDM itself, with no modifications to application software modules.

Guidance: This guideline defines a structural and methodological approach to device access for purposes of increasing application robustness and portability and to facilitate rapid application development. In an effort to assist developers and encourage adoption of this guideline, various code libraries and example applications will be developed and made publicly available as part of the guideline development project; and it is assumed that further libraries and examples will be made available by the developer community as the guideline is adopted. However, although this guideline itself will be maintained by PICMG under version control, the various SDM code sets do not constitute a tool-set per se in the sense of a single integrated, managed, maintained, and version-controlled package. Rather,

[1] This guideline does not define an API per se, since the details of an API will be highly specific to the implementing language. Rather, it defines a paradigm for creating such an API – a standard set of functions, data structures, and naming conventions that any language-specific API must support.

the SDM should be considered a logical framework for application development, and the associated code sets should be considered open-source design resources available to developers for integration into, or as templates for creating, their own projects.

2.3.2 Guidance: Functional Requirements Summary

Figure 2-1 shows the relationship between a Standard Device Model and the Operating System/Application in a typical software environment hierarchy.

The SDM must provide a device access environment with the following features:

- A modular framework for creating access to customized devices
- A stackable structure to allow creation of complex device models from simpler ones.
- A standard device naming convention to support portability.
- Customizable interfaces that allow override of default behavior for API functions
- Thread-safe operation supported by interlocks and synchronization/ communication mechanisms from a Process Model
- Extensibility to multi-process operation.
- API models for both stream-oriented and addressable devices.
- API models for both transaction-oriented and register-based device configuration.

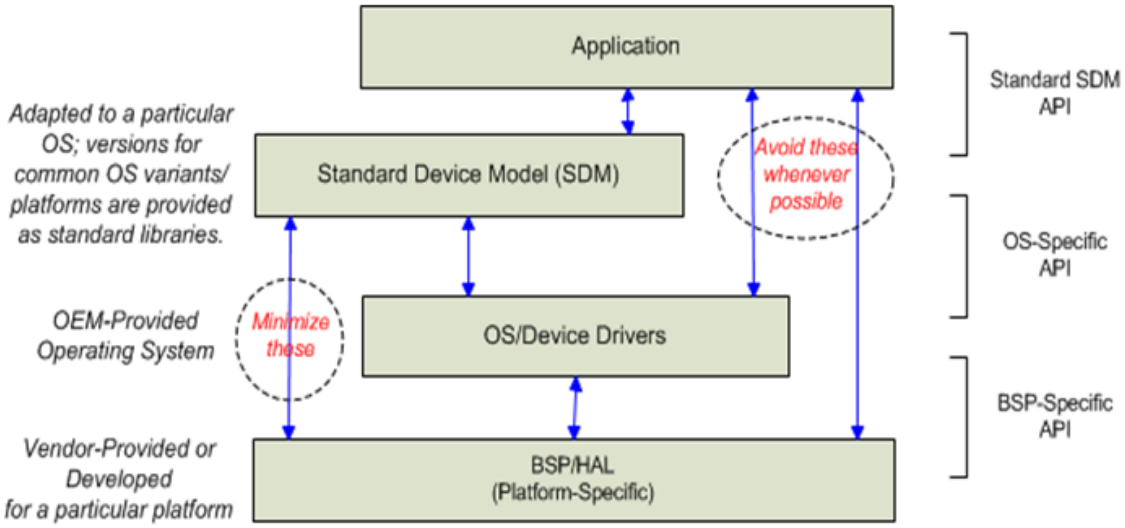

Figure 2-1: Device Model Context

For stream-oriented devices the SDM must provide the following functionality:

- Threaded wait for data with timeout
- Support for addressable streams such as random access files.
- Support for connection-oriented and connectionless transactions.
- Out-of-band and in-band communication support.
- Broadcast support.
- Ability to tunnel through a secondary interface

For addressable devices the SDM must provide the following functionality:

- Ability to provide access to multiple address spaces with different attributes and sizes.

The SDM is constrained by a need for basic compatibility with the widest possible range of pre-existing devices, from the simplest to the most complex. In particular, it cannot depend for its operation on any universal device structures or behavior, like a common set of access registers or a common functional API for querying device identity.

2.3.3 Technology and Operations Summary

As in any system, a Board Support Package (BSP) (or Basic Input/Output System (BIOS)) (perhaps a very simple one) and its associated Hardware Access Layer (HAL) provides hardware-specific access, including hardware memory management and I/O access, to the processor and its peripherals; and, as in any system, an operating system (perhaps a very simple one) and its associated device-drivers ("drivers") provide thread scheduling and logical access to I/O channels and storage devices. These elements are assumed to be standard components provided by the hardware vendor(s) and/or purchased separately as required to support high-level system requirements.

Since the BSP/BIOS and OS/driver API paradigms will vary widely among different systems, the SDM is provided to create a common Application Programming Interface (API) through which application programs can access the underlying hardware. Note that, in Figure 2-1, the ideal is for all application accesses to hardware to be mediated by the SDM.

Guidance: To the extent that developers of future board-support packages and/or OS APIs adopt the paradigms and APIs defined in this guideline as the top layer of their HAL/Device Driver models, the SDTL layer of the SDM could be eliminated entirely, with the top layers of the SDM operating directly on top of the BSP/OS. That is not the goal or intent of this guideline, but it would be a fortuitous development for application designers.

As shown in Figure 1-1 and in Figure 2-2, the SDM is designed to be stackable – that is, virtual devices can be constructed from other virtual devices and/or from logical or physical devices by organizing the device models hierarchically, with lower-level devices providing services to upper-level devices. To facilitate that, the SDM defines the same standard API at both the bottom and the top of its hierarchical connections, such that a device higher in the hierarchy connects to a device lower in the hierarchy in the same way that an application might connect to a physical hardware device.

At the bottom of the device hierarchy there must be a connection between the standard API and the system-specific BSP/BIOS and OS/driver. In Figure 1-1 and Figure 2-2, that connection is shown as the Standard Device Translation Layer (SDTL), which is a relatively simple adapter that must be provided for each system. Given an STDL properly adapted to the underlying BSP/BIOS and OS/driver combination, all the other SDM layers and the application layers would be fully portable; and to adapt an SDM/application stack to a new system, only the SDTL need be developed.

Guidance: The Standard Process Model (SPM), defined separately as part of the MTCA.4 or MATCA.4.1 development framework, provides a standard API for access to OS/driver functions. If the SPM is used as an intermediary between the SDTL and the OS then it provides the OS customization required for the platform and no customization of the SDTL may be required. However, the Standard Device Model is designed to be implementable through use of an appropriate translation layer independent of the availability or use of the Standard Process Mode.

The following sections describe the various types of device functions, access mechanisms, and operating paradigms a Standard Device Model and API must accommodate.

2.3.3.1 Address-Mapped I/O

At the fundamental hardware level, internal and external software-accessible hardware components connected to a processor core are almost always mapped in some fashion into the processor's address space and software access to these components operates by reading to or writing from the appropriate addressable elements. Address-Mapped I/O is the general term used to describe way a processor accesses these components – the processor reads data from (I="Input") and/or writes data to (O="Output") the hardware component, or "device", by reading from or writing to a location in its addressable space.

The SDM provides an API, the Address-Mapped API, to mimic that behavior. Although the Address-Mapped access model matches the underlying hardware access mechanism for direct processor-connected devices, it is also a generally useful paradigm for access to any device, physical or logical, which can be modeled as having some store of static data that may be changed from time to time.

The key point about an Address-Mapped access model is that it is well-suited to modeling the structure of data held within a device, with each datum accessible at its allotted place within the structure.

2.3.3.2 Busses and Operating Systems: Memory and I/O Models

At the processor level addressable spaces are often – but not always – separated into those designated for use with "memory" and those designated for use with "I/O" channels, with the different address spaces accessed using different processor instructions. Presumably the separation of I/O and memory address spaces was done historically to extend the amount of addressable space available on older processors with small address words, to allow for different types of optimization for memory and device accesses within the processor instruction set, and to simplify address decoding for memory and I/O devices on the external bus. However, not all processors separate memory and I/O spaces, and there is typically no fundamental difference at the hardware level between the two access mechanisms.

At the level of operating systems, however, memory and I/O accesses are typically treated very differently.

Memory is typically assumed to be allocated as a pool among the various software modules in a system, with each module being awarded exclusive access to a particular range of memory addresses out of the pool and returning that address range to the pool when it is done. Even virtual memory systems, which manage the swapping of data between physical memory and a data store on some non-volatile storage device like a disk, hide their operation from the software in such a manner that each software module appears to have exclusive access to its memory blocks. Hence, access to memory is typically done by mapping a pointer to the memory into the software module's address space and allowing it to access the memory directly using processor memory access instructions.

I/O devices, on the other hand, are typically assumed to be resources that must be shared by multiple software modules; it is not generally possible to grant one application exclusive access to a disk drive or a printer except over some very short intervals. Hence, the operating system assumes it must mediate accesses to these devices, and so it enforces an access model to the "I/O" space that operates through calls into the operating system rather than by allowing direct hardware access through processor instructions.

Hardware devices in modern computer systems operating over modern bus structures often blur the line between memory-accessed and I/O-accessed models, with some hardware I/O devices mapped into the physical memory space. Hence any standard device model must be able to accommodate – and hide the details of – either type of device access.

2.3.3.3 Configurable Bus Models

Modern processor bus structures, like PCI and PCIe, are designed for great flexibility and configurability to accommodate both a wide range of applications and on-the-fly detection and configuration of hardware components on the bus. That means that a typical bus interface will report not only multiple address ranges, mapped into both memory and I/O

spaces, for components connected to the bus but will also create a separate addressable bus configuration space that is accessed through special bus control mechanisms. A standard device model must be able to accommodate these multiple address ranges and separate configuration spaces within its overall device access paradigm.

2.3.3.4 Stream-Oriented I/O

A "stream" is a model for an ordered, sequential flow of data rather than for data statically arranged within a fixed structure. A stream can model a temporal flow (e.g. the samples reported by a data acquisition module) or it can model a geometric flow (e.g. the sequence of bytes within a file). In practice all stream APIs are temporal in nature because the software accesses are sequential in time, but the crucial aspect of a stream interface is ordering not timing; time is merely the by-product of sequence in any real-world system. In a stream API, the ordering of data passed into the interface is maintained such that when it appears for output at some other stream interface – say at the receiving end of a communication channel or when it is read back from a disk – the original ordering is intact.

Some streams are identified as "addressable" or as supporting "random access" – that is, they provide a mechanism for altering the ordering of data within the stream by moving the "location" at which the next element submitted through the interface will appear. This functionality is restricted to devices modeling geometric sequences, like disk drives, since a temporal sequence is ephemeral; you cannot rearrange the past. It might appear that an addressable stream model for device access is no different than an address-mapped model, but even an addressable stream enforces ordering: changing the stream "address" merely moves the starting point within the sequence for subsequent data transfers; the transfers themselves still proceed in sequential order from that starting point.

2.3.3.5 End-Points and Channels

Devices may be broadly grouped into two categories which describe their behavior and utility, End Points and Channels. An end-point is a source of or destination for data – it synthesizes or uses, stores or retrieves, or transforms the data in some way. A channel is a conduit through which data flows to/from somewhere else. Typically, a channel (or a chain of several channels) is used to connect two end-points.

The difference between an end-point and a channel is often a matter of context and perspective. For instance, a D/A converter that controls the speed of a fan may be an end-point when viewed from the perspective of an application that treats it, and the attached fan, as an integrated "cooling module". From a wider viewpoint, however, the D/A converter may be considered a channel through which speed control data is transmitted to the actual end-point, the fan. Or from a system perspective even the fan itself may be considered merely a conduit through which a specified amount of "cooling" is transferred to a heat source. All of these are legitimate perspectives in the appropriate context, and a device API might model the D/A (and all its connected components) in any one of those ways depending on what the designer is trying to achieve.

Further, the same device may present aspects of both a channel and an end-point: an RS-232 link is a channel through which data may flow to some external device, but it is an end-point for configuration and status information concerning operation of the channel itself.

2.3.3.6 Point-To-Point Channels

A Point-to-Point channel connects two end-points directly together. An RS-232 link between a computer and a printer is an example of a point-to-point channel. For a given point-to-point channel the end-points are pre-determined by the connection itself, so once a channel has been established no further identifying information needs to accompany the data to establish communication between the end-points.

Note that the "direct connection" between the endpoints may or may not be strictly physical. An RS-232 link comprises a unique physical link that involves only the two end-points and the cable in between; but, for instance, a point-to-point connection between two telephones through the public telephone network will typically involve many layers of switching, multiplexing and demultiplexing on shared physical channels, packetization/depacketization and re-formatting of the data among various transmission formats, and even dynamic re-routing of the data along different physical transmission paths as network conditions change or, in the case of mobile telephones, as the end-points themselves change location during the call.

2.3.3.7 Networked Channels

A Networked channel is a shared resource that connects some number of end-points to each other. The public telephone network, or an Ethernet LAN linking multiple computers through a switch, are examples of networked channels. Since an end-point on a network may send messages to any other end-point on the network some "addressing" mechanism must be provided to indicate the intended recipient for a given message.

2.3.3.8 Connection-Oriented and Connectionless Channels

A communication channel may be either Connection-Oriented or Connectionless. A Connection is logically similar to a point-to-point channel: it defines a pre-determined link between two end-points. A Connection-Oriented channel is one which establishes and maintains a connection during its period of operation; a Connectionless channel is one in which a connection must be established separately for every message that flows through the channel.

Connection-oriented and connectionless channels are the logical equivalents to the physical point-to-point and networked channels described above, but they do not necessarily map identically onto the corresponding physical channel structures. For example, one might establish a connection-oriented logical channel through a network by pre-defining within

the device API the end-point address for communications. Data submitted to the network through this connection-oriented channel would always be tagged with the address of that specific end-point, and at the device API the channel would appear to operate as a point-to-point link. Conversely, one might envision a connectionless device API representing all the RS-232 ports available in a system, with every message submitted to the API accompanied by an "address" indicating which of the RS-232 ports – and hence which end-point – it should be sent to.

2.3.3.9 Device Stacks

A generalized device model should support the interconnection of multiple devices into an operational hierarchy, or a "stack", as described in the following sections.

2.3.3.10 Conduits

Access to physical components is often done through one or more communication channels. For instance, many devices are now available with USB connections rather than direct backplane connections, and others are accessible via an intervening Ethernet or RS-232 link. Further, as noted before, even physical access to the communication channels themselves typically requires use of an address-mapped device model beneath the stream-oriented communication model. Hence device access often requires connection of different device models into a chain, with a logical device API connected to some channel API, which is in turn connected to a bottom-level register API.

2.3.3.11 Virtual Devices

A "virtual" device is an abstract model for some concrete I/O environment. The model is typically specified in terms of the functions it performs and the data it manages, rather than in terms of physical interfaces and implementation methods. For example, a "virtual" A/D Converter might be modeled at its application interface as a unit which reports a series of samples through a stream and can be configured for a certain sample rate by setting the rate into a register. The physical ADC might be at the other end of an I2C bus or an SPI channel or a USB link, and might require programming a complex series of flags and operating parameters into its internal configuration space; and communication with the physical ADC might, then, involve use not only of the ADC "device" itself but of a separate I2C or SPI or USB "device" as a conduit. But all that complexity would be encapsulated entirely within the "virtual" device model and invisible to the application using the device.

The functions of a "virtual" device representing a subsystem may also be constructed using several physical hardware components operating in tandem. For instance, a data acquisition system might be constructed from an A/D converter, a separate clocking circuit, and some cascade of configurable gain/signal-conditioning components. Each individual hardware component would be represented at the bottom level of the software hierarchy by a physical device which managed its specific functions, whereas the best device representation at the application level might be a single integrated "data acquisition" device. In that case the implementation could be constructed using a two-level hierarchy,

with the integrated device model operating through the lower-level device models rather than directly at the hardware level. Such an implementation would allow replacement of one or more of the lower-level components without requiring a re-implementation of the high-level integrated device model.

2.3.3.12 Direct Memory Access

Direct Memory Access, or DMA, is an implementation which optimizes the throughput rate between a physical device and the application using it by having the device move its data – either in or out – through one or more pre-defined memory buffers that are shared between the device and the application. The general operational model for DMA-driven transfers is that both the device and the application move data in or out of the memory independently of each other and without, therefore, requiring close interaction during the data transfer; the physical DMA management hardware provides some kind of signaling element that allows the application and the device to notify each other when the transfer must be initiated or is complete.

Note that the use of DMA is a performance optimization strategy, not a fundamentally different paradigm for device access. A device driver utilizing a DMA transfer mechanism could, for instance, appear to the application as a simple stream, with the mechanics of the DMA transfer hidden within the device driver and automatically initiated when the application transferred data into the stream output buffer. Such virtualization is the recommended way of structuring such interactions, if possible, because it decouples the fundamental function of the device – say the acquisition of a time-sequence of data – from the physical implementation of the device.

Nonetheless, there will be times when the application must be aware of, and be able to configure, such performance optimizations. Hence, a standard device model must be able to accommodate the configurations and controls required for DMA management.

2.3.3.13 Interrupts

Interrupts and interrupt handling are generally so tightly coupled to the platform hardware environment that creating a general interrupt model at the application level is nearly – or perhaps completely – impossible. For that reason, interrupts will generally be managed and serviced at the HAL and/or Device Driver level.

Nonetheless, an application-oriented device model must be able, at a minimum, to provide notification to the application when a device requires service.

2.3.4 A Simple Example

Figure 2-2 illustrates the use of a layered device model to implement access to an RS-232 communication channel via its registers through the HAL. It comprises a simple device stack-up consisting of an application program and two layers of Device Interfaces.

The USART hardware component contains 4 registers accessed via an addressable interface.

The lowest layer of the standard device model, StdDev_USART, reads and writes to those registers via the HAL in a manner dictated by the hardware platform and the target operating system. StdDev_USART provides a means of interacting with the USART by publishing an addressable interface via the Standard Device API. StdDev_USART implements a virtualized USART Device Interface by creating a common abstraction for USART devices. The register set published by the virtual interface is an abstraction that represents a set of features common to most USARTS. The software module handles the details of translating the virtual USART interface to the hardware specific registers to which it connects.

StdDev_SerialComm provides a stream interface for applications that perform serial communications. It handles the details of transmitting and receiving streams of data via the virtual USART interface. As such the application interface is relatively easy to use.

Because the application uses a generalized stream interface to communicate with a remote device, it is fairly trivial to port it to other types of serial communications interfaces, such as USB or Ethernet, provided appropriate Device Interfaces are created for the target platform.

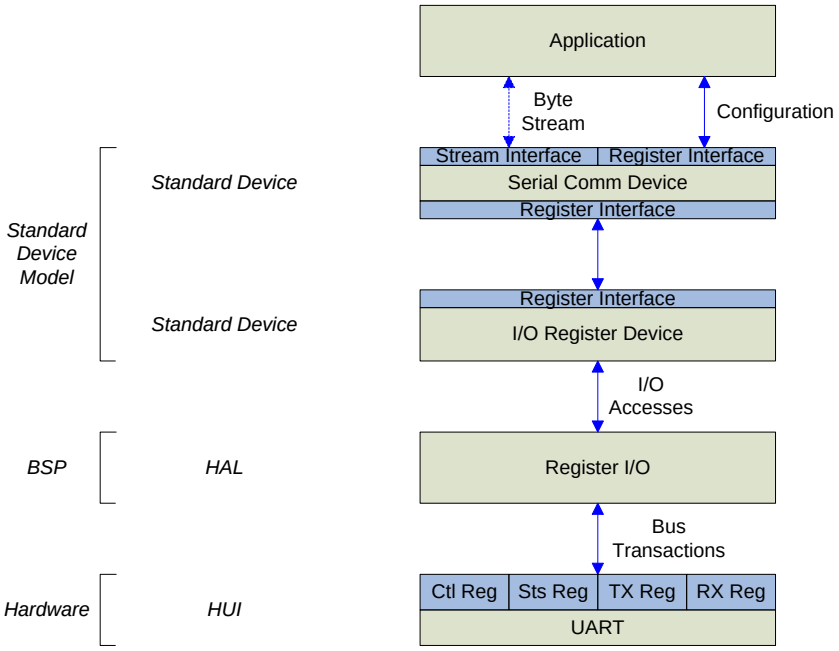

Figure 2-2: A Simple Device Model

3 Guidelines

3.1 SDM Architecture

The SDM architecture incorporates and standardizes elements commonly found in various commercially-available operating systems and board support packages in which those elements are implemented in various and disparate – and, therefore, incompatible – ways. The novelty of the SDM is not in the elements it provides but in the standardization of the API used to access those elements.

3.1.1 Conceptual/Behavioral Model

The SDM is designed to provide a common API paradigm that allows access to any device, including to communication channels and storage devices[2]. All device objects within the SDM will provide access using that common API, and that allows devices to be linked together through the API into a hierarchy, to create chains of devices that implement routing and/or format translation, and to create complex aggregated devices from simpler ones.

At root the SDM models all device interactions within three broad categories:

- I/O through a stream
- I/O through an addressable element (or "register" or "memory") space
- I/O through a command transaction.

I/O for these three mechanisms is managed through functional APIs using a small number of functions for 'reading' data from and/or 'writing' data to the associated I/O mechanism; data is moved between the application and the device through buffers passed as arguments to these functions.

The SDM is designed to operate within the context of a threaded application model. Synchronization and notification are done using mechanisms from the thread scheduler.

---

[2] Although storage devices and communication channels are included in this architecture and may be implemented within the Standard Device Model, the guideline does not stipulate that a particular application must do so. The conventions for access to files and simple streams has been fairly universalized within most programming languages and an application designer may prefer simply to use those universal mechanisms. Access to sockets has not been similarly standardized, but there are few enough flavors of sockets API that most people would consider that, also, to be universal and portable. The utility of including such "devices" within the general SDM framework is partly a matter of simple completeness, but mostly a facility for integrating those physical channels into more complex virtual devices developed within the SDM framework.

As a side-effect, it makes it generally easy to switch an application from using one type of communication channel to another. For example, if access to a "stream" device is adequately abstracted at the device API, then whether an application was receiving data from an RS-232 channel, a pipe, a file, a TCP socket, a USB channel, or (for example) a SRIO link would be completely transparent.

Guidance: The SDM should preserve any support for real-time thread scheduling and device interactions provided by the underlying operating system.

3.1.2 Hierarchical/Structural Model

A hierarchical/structural model of the standard device architecture is shown in Figure 3-1. It comprises two base layers, the Standard Device Translation Layer (SDTL) and the Standard Device Model (SDM), and some number of Custom Device Models (CDMs). The translation layer provides translation between the Standard Device Model API and the specific operating system and/or hardware access layer used for platform access; the Standard Device Model provides the framework for building up complex virtual device models from simpler devices. The Custom Device Models can be either generalized models for logical or physical device types (like a general-purpose ADC device) or adaptations of the general SDM framework for access to specific physical devices.

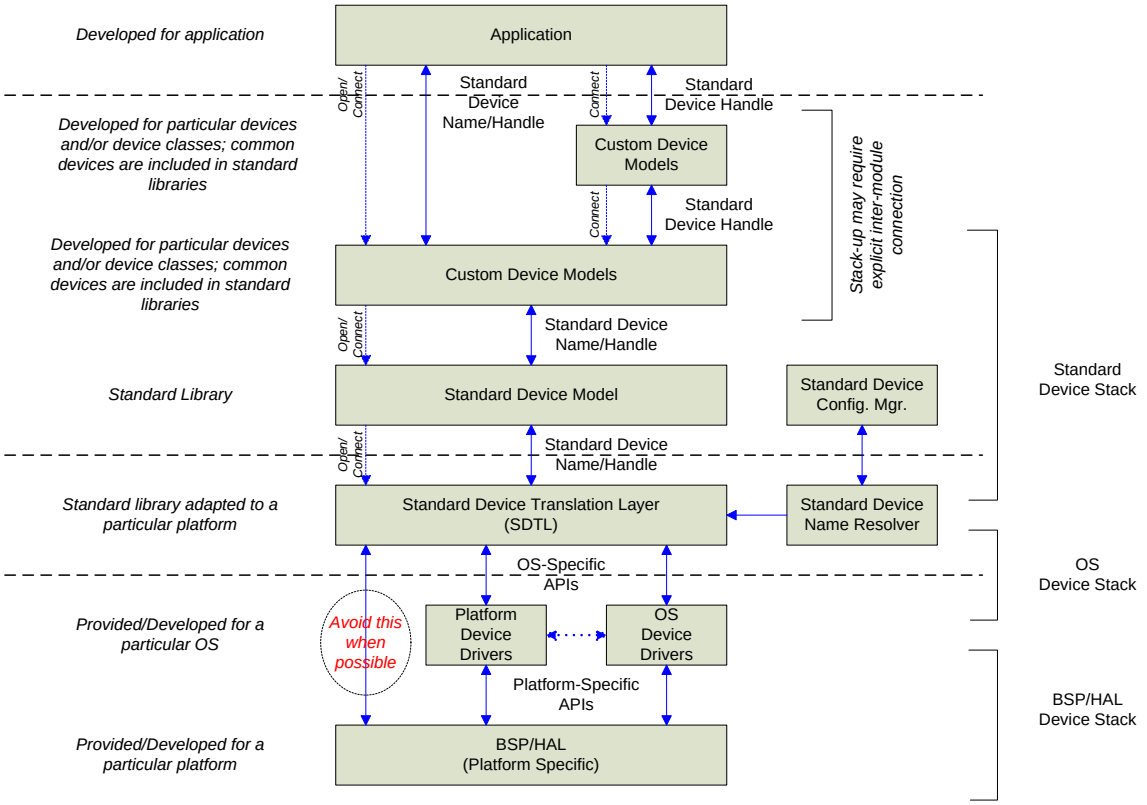

Figure 3-1: Standard Device Model Hierarchy

As shown, the Standard Device Model is designed specifically to facilitate the creation of abstract virtual devices by stacking of device objects into a hierarchy. To that end, at all layers from the top of the translation layer upward, the API is consistent at the top and bottom of the device object, allowing an SDM device object to connect with any other SDM device object in a device chain using the same, standardized, API. The translation layer, as the layer which maps the SDM to the underlying platform, specifically does not

provide the same API at the top and bottom. At the top it exposes the standard SDM API; at the bottom it is customized for the particular environment in which it operates.

Two utility modules, the Standard Device Name Resolver (SDNR) and the Standard Device Configuration Manager (SDCM), operate in parallel with the translation layer and device model, respectively, to manage name translation and mapping. The name resolver maps SDM device names and device name aliases to logical and physical device names that can be resolved by the operating system or hardware access layer. The configuration manager provides a functional library to manage that mapping by registering device aliases and name mappings within the name resolver database.

3.1.3 Interface Model

The Standard Device Model provides a functional API paradigm for device access, with the device construct itself referenced as an "object" (in object-oriented languages) or by a device handle (in procedural languages). It provides support for, and exposes, standard APIs for both stream-oriented and addressable devices as described in sections 2.3.3.1 and 2.3.3.4.

A particular device may be accessed through one or the other mechanism, but not both. Which it uses depends on its underlying functionality and on the logical model used to describe its operation. From an application standpoint, the device model is defined by its functional interfaces and behavior, not by its physical implementation. An ADC device might expose a simple register API, notwithstanding the fact that the actual ADC hardware must be accessed through a serial USB or RS-232 link using some command protocol (as illustrated in the example from Figure 1-1).

Guidance: An implementation that allowed a single device to export both Stream and Region APIs would be conformant to this guideline, but that functionality is not mandated.

A particular device will expose access to a single Stream or Region channel[3]. Stream APIs may be either connection-oriented or connectionless, as described in section 2.3.3.8, although whether connectionless device access is supported in a particular instance will depend on the device.

Guidance: A specific virtual device with complex state behavior and/or a more sophisticated API or device model may provide a device-specific functional API to replace and/or augment the stream/region API model defined here. Such a functional API is defined, for example, for configuration of stream channel buffer sizes as part of the SDM itself (see section 3.2.3.2). These functional APIs tend, by nature to be very specific to the

[3] Multiple 'channels' can be synthesized by modeling the device as a 'networked' component with separate "network addresses" for each channel, similar to the way various devices on an I2C bus are accessed (see section 9.5 for a description of I2C). This has proved, in practice, to be completely adequate for modeling, as an example, multi-channel ADCs: individual ADC channels are read from the single stream channel using the 'readFrom()' access method and specifying the specific channel from which the data is to be received; or a device can be dedicated to a specific ADC channel by pre-connecting it at the time the device is opened.

particular device model and are, therefore, not amenable to broad standardization. Hence, no standards for how to design such an API are defined in this guideline. They are, however, considered appropriate in some cases and should be considered an augmentation of, rather than in conflict with, the standard APIs defined in this guideline.

As in most operating environments, devices are identified for connection by logical names. The Standard Device Model defines both a standard naming convention to facilitate portability across platforms and a standard name mapping API to translate standard SDM device names into machine-specific device names required for connecting through the operating system device drivers.

The detailed functional API paradigm for the SDM is described in section 3.2.

To facilitate portability at the application and SDM level, the device model defines a standard naming convention for logical/physical devices and a facility for establishing platform-independent aliases for logical/physical device names. Mapping between application-level, SDM-level, and OS/HAL level device names is done automatically within the SDM. The SDM also provides an API paradigm for managing the name mapping database.

3.1.4 Transactional Model

The Standard Device Model defines a thread-aware synchronous device transaction model in which device accesses suspend thread operation until they can be completed rather than queuing some notification mechanism to be invoked later and asynchronously. All device data transfer functions allow three types of access:

Non-Blocking: The access request returns immediately and reports whether or not the access was completed.

Blocking: The access request suspends the calling thread indefinitely until the access can be completed.

Blocking with Timeout: The access request suspends the calling thread until either the access can be completed or a maximum wait time elapses; upon return, it reports whether or not the access was completed.

All device notifications – including interrupts – are mapped into this mechanism. In the case of an interrupt, or any other request for immediate attention, the "handler" must be in the form of a thread which suspends on the channel through which the interrupt will arrive by making a synchronous blocking call to a device alert function; the thread will then be suspended until the interrupt unblocks the alert request.

Guidance: Thread blocking/unblocking mechanisms should be fully compatible with any real-time thread management features provided by the underlying operating system.

Guidance: Interrupt handling that cannot tolerate the latency involved in a thread context switch must be done at the Device Driver or HAL level, below the level of the Standard Device Model. It is presumed that will not be an undue constraint since such time-critical code modules are probably best not implemented at the application level in any case. In general, it is assumed that an interrupt handler in the Device Driver or HAL will respond to the interrupt, perform whatever device transaction is requested, buffer the result, and wake up the thread waiting for that result by unblocking its SDM call.

3.1.5 Application Programming Interfaces (APIs)

The Standard Device Model APIs comprise the following function groups:

- Device connection and disconnection
- Device access management
- Device data transfer
- Device control transfer
- Device configuration/Control
- Device Name Mapping

The first five are associated with the device itself, while the last is a utility provided to support both internal SDM operation and application-level device access.

APIs are designed to support function overloading to allow for easier generation of customized device models by deriving them from standard device models. Details of that design feature will vary depending on the implementation language; example headers for 'C' and 'C++' language implementations will be made available as open-source libraries separate from this document; an abbreviated set of 'C++' headers, intended to illustrate implementation of the functional APIs, is provided as appendix C to this document.

The following subsections outline the functions provided by the API at an abstract level and are intended to be applicable to any choice of implementation language.

Guidance: APIs are described in terms of a procedural paradigm, but that is not intended as an implementation constraint. Adaptations to object-oriented implementations should provide equivalent functionality to what is defined here, but formatted as appropriate for the language being used. For example, procedural API functions to "create" and "destroy" a device object might be mapped to the constructor and destructor of a device object for a 'C++' or Java implementation, rather than defined explicitly as 'create' and 'destroy' functions. Similarly, language-specific features may make certain API functions unnecessary or redundant; for example, an error during construction of a 'C++' object results in an exception, rather than in the creation of an invalid object, so a function to verify object validity may not be applicable to an implementation in 'C++'.

Guidance: In terms of naming, the procedural functions are prefixed with a module identifier to minimize the potential for naming conflicts. However, in an object-oriented language like C++ or Java, the segmentation of name-spaces within objects does that for you automatically, so the use of the prefix is not necessary or desired.

Guidance: In keeping with the procedural description of the API, device access is described as being mediated through a device "handle". The use of the term "Device Handle" is not intended as an implementation constraint and should not be interpreted as such; in an object-oriented implementation, the "handle" would, instead, be a reference or a pointer to a device object.

3.2 Application API

3.2.1 Device Connection/Disconnection

A software device model object, or Virtual Device, must be created before software may interact with it, and it must be connected, through some chain of devices, to a physical device in order to move data in and out of the host platform[4]. During the creation process, a device object will allocate any resources, such as memory, required internally to perform functions on behalf of the application. When the process is completed, the creation function returns a Device Handle to the application for use when calling other functions provided by the device model.

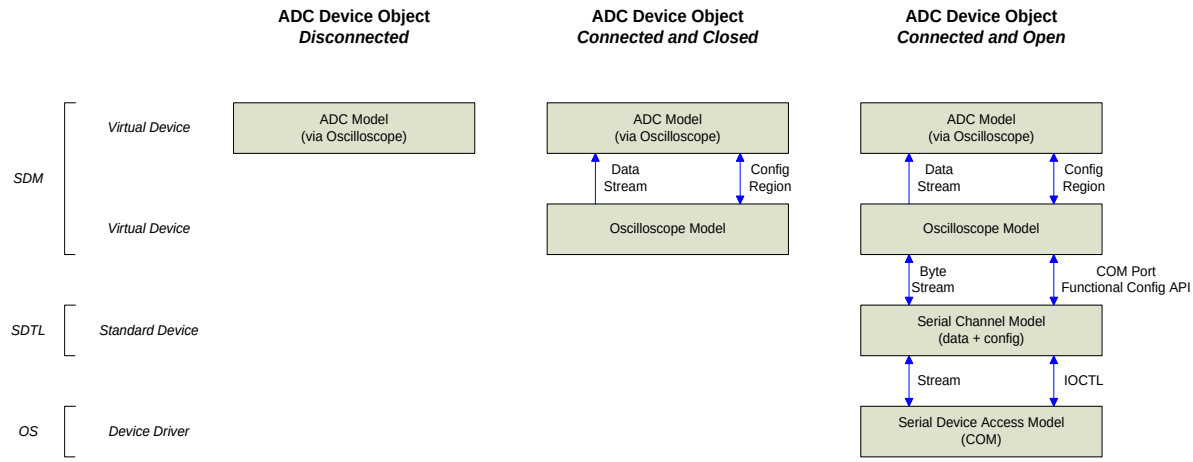

Figure 3-2: Virtual Device States

As illustrated in Figure 3-2, a virtual device object can exist in one of three states:

[4] This is a conceptual framework but it may not be literally true. It is possible to create a customized virtual device that behaves like a physical platform device – responding directly to an 'open' request, rather than relaying it to the SDTL, and acting as the end-point rather than as a conduit for data transfer requests. Such a device clearly cannot move data in and out of the platform without an actual connection to a physical device, but it could be used to create a device simulation for testing, or to build in-system structures like FIFOs that could be managed as devices through the SDM API.

- Disconnected
- Connected and Closed
- Connected and Open

A virtual device is *Disconnected* when it has no associations with other devices at the bottom of its hierarchical API; it is Connected when it has such associations.

A virtual device is *Closed* if the root of its hierarchical stack is not connected to a logical or physical device ready to accept I/O transactions; it is *Open* if there is a connection at its root to a logical or physical device which will accept I/O transactions.

The Standard Device Model provides the following instantiation and removal functions:

sdm_create	create and initialize a device object; does not connect to any other virtual, logical, or physical device
sdm_destroy	release a device object and its resources
sdm_isValid	report whether a device handle references a valid device object
sdm_open	connect a device object, possibly through a hierarchical stack of other device objects, to a logical or physical device by "opening" the device at the OS or HAL level.
sdm_close	disconnect a device object from the logical or physical device to which it is connected by "closing" the device at the OS or HAL level.
sdm_createAndOpen	create and open a device object in a single operation.
sdm_connect	connect a device object to another device object lower in the device hierarchy
sdm_disconnect	disconnect a device object from another device object lower in the device hierarchy

Guidance: In the context of an implementation in an object-oriented language, like C++, the 'create' and 'destroy' functions correspond to the class constructor and destructor; the 'createAndOpen()' function will typically be merely another variant of the constructor that takes as arguments the information required to open the device.

Guidance: Virtual device objects can, in general, be connected to more than one underlying device object at a time. Thus, it is relatively easy to build a complex virtual device which

manages the functions of several lower-level devices. Connection to multiple devices is presumed to be a function required when designing virtual devices, rather than when using them, and is, therefore, described as part of the Development API (section 3.3).

A logical or physical device is 'opened' through the translation layer by a call to the SDM 'open' function. If the virtual device is disconnected when the 'open' call is made, the SDTL creates a new virtual device object, attempts to open a platform device using the requested device name, connects the new virtual device to the platform device, and connects the pre-existing virtual device to the new virtual device. If a virtual device is already connected to one or more other devices when the 'open' is requested, the request is simply relayed to those other devices which, in turn, will either open a platform device through the SDTL or relay the open request down the hierarchy through their own connections.

The *'createAndOpen'* function invokes the SDTL directly to create a new virtual device object and connect it directly to a platform device. This provides the most direct – and therefore most efficient – connection between the application and a HUI through the SDM; access to the HUI through such a virtual device is nearly as efficient as a direct connection through the Device Driver/HAL since there is only one simple layer of indirection between them. However, a virtual device object that is connected directly to a platform device in this way is restricted in that it may not have more than that one connection – that is, its function is limited to being a simple accessor for a single physical or logical device via the SDM API.

3.2.2 Device Access Management

Since the SDM is designed to support device objects which operate across multiple threads, there must be some mechanism for protecting access to the device object(s) at critical times.

Use of the locks described below is optional on a device basis; if they will not be needed for a particular device instance (because, for example, the application design precludes access to the device from multiple threads), you can omit the locks and their associated overhead from that particular instance by passing the appropriate 'no lock' flag to the 'create' function. The SDM provides the following function to report the lock configuration for a specific device instance:

sdm_supportsLock	reports whether or not locking is supported by this device object

Internally, the SDM provides separate interlocks for read and write accesses to prevent multiple threads from simultaneously queueing read requests or write requests. 'Read' and 'Write' operations automatically acquire the appropriate interlock prior to starting and

release it when they are done; threads which cannot acquire the interlock immediately are suspended until the interlock becomes available. 'Transaction' requests – that is, requests that perform a 'write followed by a read' in a single operation – acquire both the read and write locks, as do 'control' operations (see section 3.2.5).

In addition to the internal interlocks, the SDM provides the following functions to manage a device-level interlock:

sdm_lockAccess	lock the specified device to prevent any other threads from accessing it.
sdm_unlockAccess	unlock the specified device.

Guidance: The purpose of the device-level interlock is to allow locking of the device across a series of device accesses that should not be interrupted. The individual read and write locks only lock the device for the duration of a single access.

Guidance: The device-level interlock is not managed automatically by the SDM – it is managed by the application if the application requires it. Hence, it is only effective if all application modules that utilize the device also utilize the lock!

3.2.3 Device Data Transfer

The core functions of the SDM API are the read and write operations which move data in and out of the platform. The SDM provides for two classes of device interfaces, stream devices and address-mapped (or region) devices, with unique forms for reading from and writing to the device. The SDM API provides functions for determining which type of data interface a particular device supports:

sdm_isRegion	reports whether or not the device exports a Region-oriented API
sdm_isStream	reports whether or not the device exports a Stream-oriented API

Both stream devices and address-mapped devices transfer data as "elements" with a size equal to some integer number of bytes. The element for a given stream or region may be of

any valid size, but once the size is established it will typically not change[5]. Multi-byte elements are transferred through the API in native host byte-order. The SDM API provides a function for querying the device about its element size:

sdm_getElementSize reports the size, in bytes, of an element within the region/transferred through the stream.

3.2.3.1 Address-Mapped Access

Address-Mapped devices model a register or memory space, with specific pieces of data assigned to locations at specific offsets within the address space. The data space of an address-mapped device may be accessed using Array or Block accesses, as described in the section 3.2.3.1.1. A special access model and functions are provided for access to memories mapped through a narrow window in the address space (see section 3.2.3.1.2).

An address-mapped access space in the SDM is called an address-mapped region, or a Region. The SDM provides the following functions to manage the regions within a device:

sdm_rg_getElementCount

reports the number of elements available in the region

sdm_rg_getFirst/LastOffset

reports the first and last valid element offset within the addressable region. The first offset will typically be zero, but that is not required.

sdm_rg_getBasePointer returns a pointer to the start of the memory space accessed by the region; ***this is only valid for regions which are mapped directly to an application-accessible memory space.***

Guidance: the last function is provided specifically to allow optimization of accesses to memory spaces within physical devices. It represents a compromise between a clean device encapsulation and performance optimization and, as such, creates an opportunity for wreaking havoc if not used carefully. If the device is accessed directly by pointer, then all of the standard device interlocks are bypassed; and accesses via the pointer after the device

[5] There are some devices for which the element size may be configurable during operation. For example, an SPI controller device that is connected externally to several different ICs, each connected to a different chip-select but sharing the clock and data lines, may need to perform transfers to each IC using a different word width.

has been closed will result in unpredictable behavior. Further, it is possible that such direct accesses may not be supported on all platforms, depending on the details of how the device functions within the HAL/OS paradigm, so such accesses may not be as portable as accesses through the region access functions. If direct access to memory via a pointer is not available for the specified region the function will return a NULL pointer and an error code.

The SDM provides the following functions for single-element accesses within the address-mapped region:

sdm_rg_read	read a single element from the specified address within the region
sdm_rg_write	write a single element to the specified address within the region
sdm_rg_update	do a read/modify/write from/to a single element at the specified address within the region; the update is defined by two bit masks specifying bits within the element to be set and cleared

3.2.3.1.1 *Array and Block Accesses*

An Array access writes to/reads from a contiguous set of elements within the address space starting at some specified offset. For example, a 16 element write starting at offset 8 would transfer one datum each to elements at addresses 8 through 23. This access mode might be used, for example, to transfer a sequence of bytes to a UART transmit FIFO through its transmit register. The SDM API provides the following array access functions:

sdm_rg_readArray	read a specified number of elements from the region using an array access starting at the specified address
sdm_ rg_writeArray	write a specified number of elements to the region using an array access starting at the specified address
sdm_ rg_transactArray	write a specified number of elements to the region using an array access starting at the specified address, and then read a (potentially) different number of elements back from the region starting at the same address

A Block access performs a sequence of writes or reads on the same addressable element at some specified offset. For example, a 16 element write starting at offset 8 would transfer 16 data in sequence to the element at address 8. This access mode might be used, for example, to transfer a sequence of bytes to a UART transmit FIFO through its transmit register.

The SDM API provides the following block access functions:

sdm_ rg_readBlock	read a specified number of elements from the region using a block access at the specified address
sdm_ rg_writeBlock	write a specified number of elements to the region using a block access at the specified address
sdm_ rg_transactBlock	write a specified number of elements to the region using a block access at the specified address, and then read a (potentially) different number of elements back from the same address

3.2.3.1.2 *Register-Mapped "Block" Memories*

As a utility, the SDM defines an access API for a common hardware structure, the Block Memory, which models the common hardware structure of a memory hidden behind a register pair, the "address" register (which sets the access location) and the "data" register (which accesses the memory cell), within a region. Addressing can be static or auto-incrementing, depending on the device. The SDM provides the following functions for access to Block Memories:

sdm_bm_create	allocate a block memory header to manage memory access and return a handle to it
sdm_bm_destroy	releases the block memory header and handle
sdm_bm_get/setAddress	report/set the next access address
sdm_bm_read	read a specified number of elements from the block memory starting at the specified address
sdm_bm_readNext	read a specified number of elements from the block memory starting at the address currently in the address register
sdm_bm_write	write a specified number of elements to the block memory starting at the specified address
sdm_bm_writeNext	write a specified number of elements to the block memory starting at the address currently in the address register

Guidance: In general, we assume the memory should behave as if it had an auto-incrementing address register for multi-element transfers. If the underlying HIU does not support that, then the virtual device object should synthesize that in software.

Guidance: It is recommended that implementations of the SDM provide this as a separate module that can be used in conjunction with a device, rather than as an integral part of the

device API. It is a common enough structure to be worth defining support for it, but not so common that it is worth burdening the general device API with it.

3.2.3.2 Stream Access

Stream devices closely model a communications channel or file access as commonly found in various operating systems and in the standard 'C' library[6]. The model is quite flexible and provides for a number of features to support various common and uncommon uses.

Stream device channels may have variable width. In general, communication channels like RS-232 or Socket channels are byte-wide. However, the general SDM API allows for wider channels. For example, a stream device might provide a 32-bit wide interface to a hardware FIFO or to an SPI bus.

A stream device may also be configured as a repositionable stream. File streams (as in disk files) are often configured as repositionable streams because files are often treated as random access resources for which the current read/write position may be moved backwards and forwards through the file.

The SDM provides the following functions to manage the streams within a device:

sdm_st_get/setBufferSize
: reports/configures the number of elements provided for buffering of transmitted and received data on the stream channel.

The SDM API provides 6 functions for reading from and writing to streams:

sdm_ st_read
: read a specified number of elements from a connection-oriented stream; no address is specified at the time of access.

sdm_ st_write
: write a specified number of elements to a connection-oriented stream; no address is specified at the time of access.

sdm_ st_transact
: write a specified number of elements to a connection-oriented stream and then read a (potentially) different number of elements back from the stream; no address is specified at the time of access.

[6] As noted earlier, the stream API is common enough across platforms that it could simply be used as-is, rather than re-implemented within this SDM paradigm. It is included within the SDM for two reasons: 1) to allow integration of stream-oriented and memory-mapped devices within a single API; and 2) to allow integration of standard stream APIs with other stream-oriented APIs, like sockets, USB channels, SPI busses, I2C busses, and so on that are not traditionally included in 'stream' support under most operating systems. As a consequence, the stream API has also been extended to be a bit more flexible, in that it supports streams with element sizes other than a single byte and integrates the common stream and socket access paradigms, along with other less common devices, within a single API.

sdm_ st_readFrom	read a specified number of elements from a connectionless stream; requires that an access address be specified at the time of access.
sdm_ st_writeTo	write a specified number of elements to a connectionless stream; requires that an access address be specified at the time of access.
sdm_ st_transactToFrom	write a specified number of elements to a connectionless stream and then read a (potentially) different number of elements back from the stream; requires that an access address be specified at the time of access.

The first three functions are used with connection-oriented streams in which the connection to a specific end-point has been established at some time prior to the access. This might be used, for instance, to move messages through an RS-232 channel, to move data to and from a disk file, or to move data through a socket interface that has already been 'connected' to a remote device.

The second three functions are provided specifically to support connectionless transactions in which a specific end-point for the transaction must be selected at the time the message is submitted. They might be used, for instance, to send a UDP packet through a network to a particular IP address, or to send a byte sequence through an I2C channel to device that responds to a specific device address.

The SDM also defines 2 functions to set or retrieve the current stream position:

sdm_ st_setPosition	set the current stream pointer
sdm_ st_getPosition	report the current stream pointer

Since not all streams support repositioning of the stream pointer, these functions may not be supported for all devices. If a device does not support these functions it should return an error code indicating that if/when the functions are called.

3.2.4 Device Control Transfer

In addition to relaying inbound data, some devices may need to notify an application that one or more 'events', unrelated to received data, have occurred. For example, an interval timer device may simply need to notify the application that the timer has expired.

The SDM provides an 'Alert Notification' API to allow such event notices.

Use of the alert mechanism described below is optional on a device basis; if it is not supported by a device class, it may be omitted from the implementation for that class; if will not be needed for a particular device instance, you can omit the alert and its associated overhead from that particular instance by passing the appropriate 'no alert' flag to the

'create' function. The SDM provides the following functions to report the alert configuration for a specific device instance:

sdm_supportsAlert	reports whether or not the device supports alerts

The SDM API provides the following functions to support alert notifications:

*sdm_testAle*rt	reports whether or not the specified alert(s) is/are currently being reported
sdm_waitAlert	suspends the current thread until the specified alert(s) is/are currently being reported

Each individual alert is identified by a bit within a 32-bit integer value. Hence, a given device can support up to 32 alerts.

3.2.5 Device Configuration/Control

It is virtually impossible to provide a common interface that allows for all possible forms that device configuration and control may take. Instead, the SDM provides two different functional APIs through which device-specific configurations/controls may be passed. A particular device may use one, both, or none, depending on its configuration and control requirements.

3.2.5.1 Control Region

Both Region-oriented and Stream-oriented device objects may expose a dedicated 'Control' region that can be used as a set of configuration registers. Control region elements are always 32 bits wide; if a device does not support use of a control region, it should report that the control region has a length of zero elements.

Guidance: This SDM guideline does not define any standard control region fields, but it is expected that, as common devices like serial transmission channels or ADCs are modeled using the SDM structure, common configurations will be defined for those particular devices.

The SDM API provides 3 functions for access to the control region:

sdm_getControlWordCount	report the number of control words (elements) available within the control region
sdm_readControl	read a specified number of control words from the control region using an array access starting at the specified address
sdm_writeControl	write a specified number of control words to the control region using an array access starting at the specified address

3.2.5.2 Control Transactions

Device drivers in various operating systems offer some kind of I/O control interface (often called IOCTL) that allows for data to be transferred into and out of the driver through opaque (no pre-defined or common structure) input and output buffers passed through the API. The SDM provides an equivalent function:

sdm_control

Guidance: The name of this function was specifically chosen to be unlike the term IOCTL so as to avoid confusion between this particular configuration mechanism and similar IOCTL functions available, with slight differences, in various operating systems.

The *sdm_control* function, like its cousin the IOCTL in its various guises, takes as arguments an opaque input buffer, an opaque output buffer, and a numerical code that can be used to indicate what function the transaction should perform.

Guidance: This SDM guideline does not define any standard sdm_control operations, but it is expected that, as common devices like serial transmission channels or ADCs are modeled using the SDM structure, common operations will be defined for those particular devices.

3.2.6 Device Name Mapping

3.2.6.1 Device Addressing Model

The general model for device access through the SDM presumes there is some kind of remote "device" which is accessed through a "channel" on the local system. Based on that model, a device "address" comprises two components:

An address for the local channel

An address for the remote device within the context of the channel

This accommodates the situation in which the "channel" is a conduit (e.g. a network connection or a bus) through which multiple devices may be accessed – it allows you to uniquely identify both the conduit, itself, and a particular device at the far end of the conduit. Access to particular device may or may not require specification of both address components, depending on the nature of the connection. In particular, for the class of local "point-to-point" channels that provide a 1:1 connection to a dedicated device (i.e. an RS-232 port), specifying the channel address fully identifies the device to be connected (because there can be only one device connected to the channel) and no "device" address is required. In that case, however, some aspects of the device address – say a "protocol" specification – may still be applicable and useful (see the discussion of Protocols, below).

The SDM device addressing model specifically operates using a "logical" view of the device connection environment and may, therefore, not reflect directly the physical mechanisms used for device connection. As a specific example, a USB channel is

physically a networked conduit that can connect to a variety of remote, addressable devices through a single electrical connection (via the mechanism of a distribution "hub"). However, its operating paradigm and API – created and enforced at the hardware and driver levels – maps that physical "networked" structure to a set of logical "point-to-point" connections in which each device appears to reside on a unique, locally-addressable "channel". Hence, the SDM addressing of a USB device is always defined a connection-oriented address.

3.2.6.1.1 *Interfaces*

We presume each local "channel" is accessible through a specific "interface" on the local system, where the "interface" defines the mechanism(s) (i.e. the API(s)) used by the system for access to the channel. Interfaces may reflect either physical or logical distinctions within the system operating environment. Examples of "physical" interfaces are:

socket, which is typically accessed through the standard socket API

file, which is accessed through the file system API

pipe, which is typically (but not necessarily) accessed as a special case through the file system API

com (aka "serial channel" – e.g. RS-232), which is typically accessed as a special case through the file system API but may, on some systems, be accessed directly through some BSP/HAL module

usb, which is accessed through some USB driver or directly through a BSP/HAL module

Examples of "logical" interfaces are:

rtc (real-time clock), which exposes a standard API for access to RTC functions but may utilize a physical device on a bus or on an I2C channel, or may utilize a logical device hidden behind an operating system API.

adc (analog-to-digital converter), which exposes a standard API for access to ADC functions but may utilize a physical device on a bus, on an I2C channel, on an SPI channel, at the far end of a USB or RS-232 or Ethernet channel, and so on.
The "channel" address identifies a specific channel within the interface, often (but not always) through use of an integer value (e.g. "com:1" might be identified by the channel address '1' within the com interface).

3.2.6.1.2 *Protocols*

Communication with a "device" through a "channel" may utilize some protocol for encoding/decoding the data as it passes through the channel. Further, a device, itself, may require that data submitted to it be encoded using some encoding protocol (for instance, it may be encrypted for security purposes).

In an ideal world, the channel encoding protocols (and, perhaps, the device encoding protocols) would be transparent to the application and would not need to be explicitly identified. In practice, however, both transmission channels and devices that support multiple protocols will need to be configured appropriately. Hence, the identification of channels and devices may include identification of the protocols to be used for connection.

In general, protocols specifically associated with the transmission channel will be identified as part of the "channel" address and protocols specifically associated with device operation will be identified as part of the "device" address; a typical use of a channel protocol would be identification of UDP or TCP as the transport protocol for a channel within the socket interface.

3.2.6.2 Device Naming

The SDM defines a common format for logical/physical device names to facilitate portability across platforms. To maximize compatibility with networked devices, the SDM naming convention is based loosely on the URI nomenclature delineated in RFC 3305 and used for identifying resources on the internet.

A URI has the general format:

<scheme name>:<scheme-specific identifier>

For the purposes of the SDM, *Standard Device Names* are URIs for which the scheme name is "sdm" and the scheme-specific identifier is a text string comprising three distinct fields separated hierarchically as follows:

[sdm://]<interface>/<channel address>[/<device address>]

Guidance: The formatting of device names with separate and discrete fields is designed to facilitate the mapping of SDM device names to platform-specific logical/physical device names. In particular, the "Interface" field will generally encode the logical/physical access mechanism that the SDTL must use to connect to the device through the OS/HAL API.

Within the context of the SDM implementation, the "scheme name" is optional since there is no ambiguity about what scheme is intended[7].

The fields within the Standard Device Name have the following meanings:

Interface — The logical mechanism/API on the local host through which the device will be accessed. For example, many operating systems provide access to devices through a generalized "file system", but

[7] If the name is to be interpreted in some more general context (e.g. within a browser application), then the scheme designator should be included to differentiate the name from names intended for interpretation using other schemes.

they differentiate between accesses to actual disk files and accesses to other types of devices by prefixing the device name with some special designator like "/./com" (serial port; interface="com") or "/./pipe" (pipe; interface="pipe"); a name without a reserved prefix is assumed to be a disk file (interface="file"). Other types of dedicated devices that are not accessed through the file system would have their own interface designators (e.g. "socket", "i2c", etc.).

This hides the details of how a particular platform BSP/HAL and/or operating system provide access to various types of devices from the application software. For example, it makes no difference at the application level whether the underlying access to a com port is through a file system stream or by direct connection to a BSP/HAL device API.

Channel Address The identifier for a particular communication channel on the specified interface through which the device will be accessed.

Device Address The identifier for a particular device connected to the communication channel.

Unless specifically identified otherwise, all name fields are case-insensitive.

Guidance: The term 'interface' is not intended to imply only physical interfaces. It is intended more broadly, to identify a logical interface that exposes a pre-defined set of data structures and/or functions for performing accesses. Hence, we can define "interface" types that correspond to both low-level physical interfaces (i.e. a serial port) and to high-level abstract interfaces (i.e. an interface to a device that exposes the SHAPI register set). As a rule, logical devices which are accessed through high-level, abstract interfaces will be connected to other devices using lower-level, less abstract interfaces through which they will perform their actual interactions with the corresponding physical devices.

As an example, suppose there are 3 devices that expose SHAPI registers, one that physically resides on a PCI bus, one at the far end of a USB link, and one on a remote device connected to a network. As part of the platform configuration, we might create three logical SHAPI devices, installed into the SDM as 'shapi/:1', 'shapi/:2', and 'shapi/:3', which internally connect to their respective physical devices through the appropriate channels. An application could then open a device through the SHAPI "interface" (i.e. 'shapi/:2') without needing to know where that particular device physically resided or what channel was used to talk to it.

For a remote device connected to a dedicated point-to-point communication channel (e.g. an instrument connected to an RS-232 port), the identification of the channel is entirely equivalent to identification of the device itself, so we identify the device by reference to the channel through which it is accessed and place that identification into the "channel address". For example, suppose there was an oscilloscope connected to an RS-232 port. The host computer only knows that it has an RS-232 port available for communication; it

has no knowledge of the fact that it is an oscilloscope, rather than a printer or another computer or something else, at the far end of the link. Hence, we don't even try to manage the identification of the oscilloscope as the remote device – we simply identify it as "the thing connected to COM2".

Similarly, for a typical locally-connected device that is accessed through a dedicated device-specific interface (e.g. a local real-time clock accessed using the interface "rtc), the channel itself is identified exclusively with the device so we, again, place the device identifier into the "channel address". As a more abstract example, if there is an interface for general-purpose timers, designated by the name "gpt", a variety of general-purpose timers might be mapped through that interface, some residing in the local address space as processor peripherals, some connected to a PCI bus, some connected to an I2C bus, and so on. If you are, however, connecting to those devices through the "gpt" interface (rather than connecting directly to the physical devices via the PCI bus or the I2C bus), the details of the channel through which you are accessing the device have been encapsulated within the "GPT" device abstraction. For purposes of identifying the device, it is simply "the thing on the GPT channel".

For channels that can connect to multiple devices (e.g. sockets, a GPIB bus, or an I2C bus), we use the "Device Address" to identify the specific device to be accessed through the channel.

Similarly, for files and pipes that could reside either on the local machine or on some remote machine, we place the pipe or file name in the "device address" and reserve the "channel address" to be able to identify the host machine on which the file resides.

Guidance: For purposes of determining how to format an address, we differentiate between the physical implementation of the channel and its logical API. For example, USB and PCI (and any similar technology) is, physically, a bussed interface with multiple remote devices connected to the bus. In practice, however, devices are always enumerated and connected to the driver at the time the device is opened so that the connection appears, logically, to be a point-to-point channel – you can't open a general connection to the "USB Bus" or the "PCI Bus" and then launch transactions aimed at arbitrary devices through that connection. Hence, USB and PCI device addresses treated as if they define discrete point-to-point channels.

A representative list of names associated with pre-defined interfaces are specified in Appendix C.

Guidance: PICMG shall maintain and publish a master list of interface names intended for general use under this guideline.

3.2.6.2.1 Channel and Device Addresses

Within the two 'address' fields ('channel address' and 'device address'), the address is further sub-divided into four discrete fields, as follows:

[<<host name>>[:<port>][;<protocol>][=<configuration>]

where the fields have the following meanings:

Host Name — Identifies the specific "channel" within the local machine or the specific device at the far end of the channel. For example, in the case of a file, this would be the file name; in the case of a socket, this would be the host name or IP address.

Note that this is separate from an "instance" number, which is identified in the "port" field.

In the case of a typical locally-attached device or communication port, there is only one "channel" available, so the "host name" field in the channel address may either be left blank or filled in with a special host name that identifies the "local host".

Note that the enclosing angle-brackets ('<' and '>') are required because the host name, itself, may contain arbitrary delimiting characters (including the URL hierarchy separator, '/'). We use the angle brackets to separate delimiting characters that are part of the host name from the delimiting characters we use to identify the other fields of the address.

Port — Identifies a particular 'instance' of the channel or device. For example, the RS-232 port "com:2" would be identified by port number 2; and the first instance of a real-time clock would be identified by port number 1.

Protocol — Identifies an encapsulation protocol to be applied to data moving through the channel. This is typically only applicable to communication channels that support different types of encapsulation; for example, a socket device name would need to identify whether the socket channel was to utilize UDP or TCP (or something else) for its transport encapsulation. Note that data-link layer encapsulations, like the asynchronous byte framing used by RS-232, are typically not the subject of a protocol specification because they are fixed implementations inherent to the physical channel.

Configurations that may be applied to either the transport or data-link protocols (e.g. asynchronous bit rate) should be included in the "configuration" field.

Configuration — A comma-delimited string that specifies channel/device-specific configurations. Other than the fact that it is comma-delimited, there are no pre-defined formats for this string; it will be highly specific to the particular channel or device. The string is passed as-is to the

device ‘open’ function for it to use (or not) as it sees fit. Note that device objects expose a ‘control’ API through which device configurations may be managed, so devices can generally be configured through that mechanism even if no configurations are specified as part of the device name.

For reasons of keeping the name parsing as simple as possible, the fields of the address should be specified in the order shown here.

A particular SDM implementation is not required to support use of configuration strings as part of the device name. The inclusion of the configuration string was for convenience only (see section 3.2.6.2.1.4 for a discussion); devices should be fully configurable through the control APIs defined in section 3.2.5. However, SDM implementations which to not support use of configuration strings should, nonetheless, accept and properly parse device names which include them.

3.2.6.2.1.1 Host Names

As described above, host name strings are placed between opening and closing angle-brackets (‘<’ and ‘>’) to ensure that SDM delimiter characters can be used within the host name without creating ambiguities. Host names are presumed to be case-insensitive and may include any character except a closing angle-bracket (‘>’) [8]. Note, however, that a file or pipe name may or may not be case-sensitive, depending on the underlying file system implementation.

The special host name strings, “.” and “localhost”, are reserved to identify the local host machine. Under most circumstances, it is not required that those be specified since the default host name, if none is specified, is the local host.

3.2.6.2.1.2 Ports

The address format allows association of a “port” with both the channel and the device. The two port identifiers should be used as follows:

Channel Port Identifies the specific instance of the channel/device on the local host.

Device Port Identifies an instance of the remote device.

For point-to-point communication channels and locally-connected devices, the “channel port” will identify the specific channel/device instance (e.g. com:2 = port 2); the “device port” will not typically be used unless a “device protocol” that requires identification of a

[8] To minimize the parsing complexity, we don’t define an ‘escape sequence’ for defining the closing angle-bracket as a valid name character; just don’t use it!

"port" (e.g. HDLC) is specified; in that case, the "device port" would specify the port (or device) number required by the protocol header.

For socket devices, the "port" corresponds directly to the port in the various IP protocol definitions. The Channel Port identifies the port number for the local end of the connection; the Device Port identifies the port number for the remote end of the connection.

For communication busses like I2C, GPIB, and SCSI, which identify devices on the bus by specification of a "device address", the "channel port" identifies the bus on the local host that is to be used for communication and the "device port" corresponds to the remote device address.

For a pipe, or any similar channel for which the two ends of the channel are individually accessible, the "port" defines which end of the channel you want to connect to.

Internally, ports are represented by numbers but, for purposes of defining them in the device name, ports may be identified either by number or by common port names; a representative list of names and numbers associated with pre-defined ports are specified in appendix C[9].

Guidance: PICMG shall maintain and publish a master list of port names intended for general use under this guideline.

Unless they are used to identify specific addresses defined by a particular standard or protocol (e.g. the device address on an I2C bus or the port number on a socket channel), indexed port numbers are assumed to begin with a value of 1; that is, index 1 is assumed to represent the "first" instance on a particular interface.

3.2.6.2.1.3 Encapsulation Protocols

The address format allows association of a "protocol" with both the channel and the device. The two protocol identifiers should be used as follows:

Channel Protocol Identifies encapsulation supported directly by the channel itself. For example, a socket channel would support encapsulation using various protocols like UDP, TCP, etc.

Device Protocol Identifies encapsulation applied to the data prior to submission to the channel. For example, data submitted to an RS-232 channel might be encapsulated using some datagram-oriented protocol like PPP.

In general, channel protocols are inherent to the channel implementation and will depend entirely on the underlying channel driver. They should be completely transparent to the application using the channel. We allow you to specify them because certain channel

[9] To avoid confusion, ports commonly used with sockets are assigned names/numbers corresponding to the standard IANA assignments

types, like sockets, require you to identify the protocol they will use as part of connecting to them.

Device protocols, however, generally apply to behavior of the device itself (for example, in the case of using PPP encapsulation for traffic on a serial channel, the device on the far end must support PPP encapsulation because the channel doesn't). The SDM Development API (section 3.3) provides a mechanism for installing application-defined protocol translators that can be used with devices that require them. In addition to using PPP encapsulation to create datagrams on stream-oriented channels, you could also use the "device protocol" to specify things like encryptions to be applied to a data stream.

Internally, protocols are represented by numbers but, for purposes of defining them in the device name, protocols may be identified either by number or by common protocol names; a representative list of names and numbers associated with pre-defined protocols are specified in appendix C[10]. The special protocol name, "raw", is reserved to indicate raw binary (no encapsulation). However, it will not generally be required that the "raw" protocol by specified explicitly, since that is the default protocol if no other is specified.

***Guidance**:* PICMG shall maintain and publish a master list of protocol names/numbers intended for general use under this guideline.

3.2.6.2.1.4 Configurations

Configurations are specific to a particular device. Other than the fact that they are formatted as a comma-delimited string, no details of what should or should not be included in a configuration string, or of how individual fields should be formatted, are defined by the SDM.

The "configuration" field is provided as a convenience to the developer for situations in which an application might need to use a variety of different channels, selected at run-time, to communicate with a device. Without the ability to specify a configuration as part of the device name, the application would be responsible for determining which of the various channels it was actually using and configuring it appropriately. With the configurations specified as part of the device name, the configurations may be automatically applied by the SDM when the channel is opened and the application can ignore those configuration requirements.

Guidance: To support transparent use of communication channels at the application level, configurable communication channels should define a format for an associated configuration string and support use of that string to configure the channel. Non-channel devices may or may not support configuration through a configuration string in the device name.

[10] To avoid confusion, protocols commonly used with sockets are assigned names/numbers corresponding to the standard IANA assignments.

As an example, consider the case in which an application might communicate with a remote device through either an RS-232 channel, a UDP-encoded Ethernet channel, a USB channel, or an I2C channel (e.g. an IPMB). The device names for the four possible connections could be identified as:

com/:2=8,1,none,115200/;ppp_pad32

usb/<0x0040:0x0001>:2/; ppp_pad32

socket/;udp/<192.168.51.4>:2147

i2c/:1=100000/:0x42

The "configuration" fields of the RS-232 and I2C channels specify that:

The RS-232 channel should be configured for 8 data bits/1 stop bit/no parity/115.2 kbaud

The I2C channel should be configured for a bit clock of 100 kbps

(Note: In addition, the RS-232 and USB channel names indicate that packets should be encapsulated using the PPP protocol with 32-bit padding. That creates logical datagrams on the stream-oriented RS-232 and USB channels to allow for packet identification and recovery; the UDP and I2C channels are already datagram-oriented, so no additional protocol encoding is required).

Since all the configuration information is in the device name, itself, and is applied automatically by the SDM when the device is connected, the application needs merely to open the device, as named, and use it – no knowledge of how the channel operates is required at the application level.

Guidance: In general, definition of configuration string formats for specific device types is beyond the scope of this guideline. It is expected that definitions for particular devices will be published as they are generated and that, over time, de facto common formats for general classes of devices will emerge. In that case, those common formats may be added to this guideline in future revisions. Unless specifically stated elsewhere in this guidance (specifically, in the following sections), configuration strings used in examples are exactly that – examples – and do not identify 'standardized' configurations.

3.2.6.2.1.4.1 Asynchronous Serial Configuration

The format of configuration strings for asynchronous serial ports, as illustrated in the example above, are simple and common enough that we have defined the following standard configuration string:

<# of data bits>,<# of stop bits>,<parity>,<baud rate>,<handshake>

Unused fields at the end of the configuration string may be omitted (as in the case of the "handshake" field in the example above); unused fields in the middle of a configuration string may be left empty but must still be separated from other fields with the appropriate comma-delimiter.

Valid values for the 'data bits' field are typically 7 and 8, though some implementations may support other data word widths.

Valid values for the number of stop bits are 1, 1.5, and 2.

Valid values for the 'parity' field are:

none	no parity
odd	odd parity
even	even parity
mark	'mark' parity
space	'space parity

Valid values for the 'handshake' field are:

none	no handshake
manual	RTS/CTS handshake (managed manually by the application)
rts/cts	RTS/CTS handshake (managed automatically)
xon/xoff	XON/XOFF (software) handshake (managed automatically)

Other handshaking methods (like DTR/DSR) are uncommon and not available through the configuration string; they must be configured (if available) by the application through the device control API.

Not all valid configuration string field values may be supported by a particular device; if a requested configuration is not supported, the device 'open' call should fail.

The default configuration (no configuration string specified) is:

8,1,none,9600,none

3.2.6.2.1.4.2 I2C Configuration

The format of configuration strings for I2C and SMBus channels (they are both identified using the 'i2c' interface specifier), as illustrated in the example above, are simple and common enough that we have defined the following standard configuration string:

<bit rate>

The maximum valid bit rate for an SMBus device is 100 kbps; the maximum valid bit rate for a standard I2C device is 400 kbps; and the maximum valid bit rate for a high-speed I2C device is 3.4 Mbps.

The default bit rate (no configuration string specified) is 100 kbps.

3.2.6.2.1.4.3 SPI Configuration

The format of configuration strings for SPI channels are simple and common enough that we have defined the following standard configuration string:

<# of data bits>,<bit rate>

The valid number of data bits for an SPI channel is typically an integer number of bytes (e.g. 8, 16, 24, 32), though some implementations may support other data word widths.

The default configuration (no configuration string specified) is:

16,1000000

3.2.6.2.2 *Guidance: Device Naming Examples*

The generalized form for a device name is illustrated by the following identification of a device on an I2C bus:

i2c/<host name>:2;master/:0x42;10

which specifies a connection as a bus master to device with the 10-bit address 42h that resides on the 2nd I2C bus on the machine identified by "host name". In this case:

The "channel" host name identifies the host on which the physical connection resides

The "channel" port identifies that it is the 2nd instance of the bus on that host

The "channel" protocol identifies that it operates as a bus master

The "device" port indicates the I2C address the device responds to

The "device" protocol indicates that the address specified by the port should be treated as a 10-bit address.

Since an I2C device isn't identified with a name, there is no entry in the "device" host name field.

If you wanted (as is most often the case) to specify an I2C bus on the local machine, you could either use one of the reserved host names to identify the local host:

i2c/<localhost>:2;master/:0x42;10
i2c/<.>:2;master/:0x42;10

or you could simply omit the "channel" host name entirely:

i2c/:2;master/:0x42;10

As a rule, the ‘protocol’ is fixed for the duration of the device connection. Hence, for example, an I2C channel opened as a bus master will always be a bus master (to simultaneously use the bus for receiving data as a bus slave, open a second device and specify the ‘slave’ protocol). For that reason (among others), things like the byte framing and baud rate for a serial communication channel will not be specified as a protocol but, rather, as a “configuration” in the configuration field.99

For many devices, there will either be no protocol customization available or there will be a default protocol that is applied if no configuration is specified. So, for example, we stipulate that the default protocol for an I2C bus is “master”, rather than “slave”, so:

i2c/:2/0x42;10

specifies that the connection should be a master (the default protocol). Similarly, for an I2C device, the default “device” protocol is 7-bit addressing, so:

i2c/:2/0x42

specifies the device address should be interpreted as the 7-bit address 42h.

Finally, an I2C device can be “unconnected” in the same sense that a socket can be “unconnected”, with no pre-defined device association. Hence:

i2c/:2

identifies an I2C bus through which you specify the device address at the time of the access using the ‘readFrom()’ and ‘writeTo()’ functions of the stream API.

Note: The general form for device naming allows us to connect to devices on remote machines by specifying the remote host name in the “channel address”. In many cases (e.g. for a “bus” device”) that may not make particular sense and, in most implementations on most platforms, connections to such physical devices (including things like an I2C channel or a serial port) on a remote machine will not be supported and the only valid “channel” host name specifier for those devices will be the local host. The fact that we allow specification of a remote host in the name for those devices does not imply that we intend all (or any) implementations to support that functionality. However, it does mean that the actual device specifier for those devices will be placed into the “device address” field of the overall device name rather than in the “channel address” field.

The following examples provide an illustration of how the device naming format is intended to be applied to range of common device types:

file/<host name>/<file path>

	A file on the machine “host name” with the path name “file path”; for the local machine, specify <.> or omit the local name entirely, as in the next examples.
file/<.>/<file path>	A file on the local machine with the path name “file path”.
file//<file path>	A file on the local machine with the path name “file path”.
pipe//<pipe name>	A pipe on the local machine identified by “pipe name”; create the server-side pipe by default
pipe/<.>:server/<pipe name>	A server-side pipe on the local machine identified by “pipe name”; note the special port name, “server”
pipe/:client/<pipe name>	A client-side pipe on the local machine identified by “pipe name”; note the special port name, “client”
pipe/<MyMachineName>:client/<pipe name>	A client-side pipe that originates on the remote machine “MyMachineName” and is identified by “pipe name”. Note that “MyMachineName” must be resolvable using the underlying name-resolution facilities use by the pipe subsystem.
vcomm[11] //<name>	A virtual communication channel on the local machine identified by “name”; connect to the “local” end by default
vcomm/:local/<name>	The “local” end of a virtual communication channel identified by “name”; note the special port name, “local”
vcomm/:remote/<name>	The “remote” end of a virtual communication channel identified by “name”; note the special port name, “remote”
loopback[12] //<name>	A loopback communication channel on the local machine identified by “name”\
com/:2	Asynchronous serial port com2 on the local machine
com/<remote host>:2	Asynchronous serial port com2 on a remote host.

[11] A ‘vcomm’, or “virtual communication” channel, is a logical communication channel, similar to a pipe that can be used for testing or for inter-thread communication. Unlike a pipe, it is “lightweight” in that it is designed for use only *within* a single process.

[12] A ‘loopback’ channel is a logical communication channel that can be used for testing or for unidirectional inter-thread communication. Any data written to a loopback channel becomes immediately available for reading from the same channel. A loopback channel is “lightweight” in that it is designed for use *only* within a single process.

com/:2/;hdlc_crc16:7	Asynchronous serial port com2; encapsulate packets using the HDLC protocol with a 16-bit CRC and the device address set to 7. Note that the encapsulation protocol is specified as part of the device address because it must be applied on top of the inherent channel encoding. It is done that way both because such encoding is not typically part of a 'com' channel and because it allows an equivalent encoding specifier to be applied to other stream-oriented channels, like a TCP link on a socket, for which the "channel address" protocol may already be used to specify a protocol for the channel, itself.
com/:2=8,1,even,115200	A synchronous serial port com2; configure for 8 data bits, 1 stop bit, even parity, and 115.2 kbaud.
gpt/:3	The 3rd GPT timer.
gpt/:3=timer,49152000	The 3rd GPT timer; configure as a free-running timer at 4.9152 MHz.
gpt/:3=one-shot,49152000,0.011	The 3rd GPT timer; configure as a retriggerable (one-shot) timer at a clock rate of 4.9152 MHz and with an 11 ms pulse when triggered.
gpio/:1	The 1st GPIO region.
gpib/:2/:20	The 2nd GPIB bus; permanently connected to remote device number 20, so all device accesses are addressed to that device.
spi/:1	The 1st SPI bus; the remote device number is left unspecified and must be specified at the time the device is accessed[13] .
bus//<0x100C0000:64>	A region on the local addressed bus starting at physical byte address 100C0000h and spanning 64 bytes; unless specified otherwise, we assume the primary memory bus of the local processor, which is designated as bus number 1 by default. However, on processors with multiple busses, you could use the instance designator to identify which bus is intended. In particular, on Intel processors that map separate memory and

[13] Because SPI links are so often shared in hardware by using common SCK/MOSI/MISO signals and a separate CS for each device, we model them as shared busses, with each CS selected by an index number. If the link is dedicated to a single device, then the only valid index number is 1.

I/O address spaces, you might (for example) identify bus 1 as the memory bus and bus 2 as the I/O bus.

bus/:2/<0x100C0000:64>

A region on the 2nd local addressed bus starting at byte address 100C0000h and spanning 64 bytes.

bus/<remote host>:2/<0x100C0000:64>

A region on the 2nd addressed bus of the remote host starting at byte address 100C0000h and spanning 64 bytes.

memory//<:64> A 64-byte long block of memory dynamically allocated in the local virtual address space; accessed as byte-wide elements by default.

memory//<:64:2> A 64-byte long block of memory dynamically allocated in the local virtual address space and accessed as 2-byte-wide elements.

memory//<@0xC0000000:64:2>

A 64-byte long block of memory at address C0000000h in the local virtual address space and accessed as 2-byte-wide elements.

memory//<MyName:64:4>

A 64-byte long block of memory dynamically allocated in the local virtual address space, accessed as 4-byte-wide elements, and associated with the name "MyName". Since, another request for a block of memory using the same name would return a pointer to the same block of memory; this implements a shared memory space.

pci//<0x0040:0x0001> A device on a local PCI bus with VID:PID=0040h:0001h; will connect using the addressing information in BAR 0; will take the next available instance.

pci/<remote host>/<0x0040:0x0001>

A device on a remote host's PCI bus with VID:PID=0040h:0001h; will connect using the addressing information in BAR 0; will take the next available instance.

pci/:2/<0x0040:0x0001>

Instance 2 of a device on a PCI bus with VID:PID=0040h:0001h; will connect using the addressing information in BAR 0.

pci//<0x0040:0x0001:2>

A device on a PCI bus with VID:PID=0040h:0001h will connect using the addressing information in BAR 2; will take the next available instance.

pci//<2@1> The device in slot 2 of PCI bus 1; will connect using the addressing information in BAR 0.

pci//<2@1:3> The device in slot 2 of PCI bus 1; will connect using the addressing information in BAR 3.

usb//<0x0040:0x0001> A device on a USB bus with VID:PID=0040h:0001h; will connect through interface 0 of configuration 0; will take the next available instance.

usb//<0x0040:0x0001:1:2>

A device on a USB bus with VID:PID=0040h:0001h; will connect through interface 2 of configuration 1; will take the next available instance.

usb/:2/<0x0040:0x0001>

The 2nd instance of a device on a USB bus with VID:PID=0040h:0001h; will connect through interface 0 of configuration 0.

usb/:"001"/<0x0040:0x0001>

The instance of a device on a USB bus with VID:PID=0040h:0001h and serial number "001"; will connect through interface 0 of configuration 0.

usb//<"string":1:2> A device on a USB bus configured with a name string matching "string"; will connect through interface 2 of configuration 1; will take the next available instance.

socket/;tcp/<MyMailHost>:pop3

A socket on the default local host NIC using the TCP protocol and the next available port number; connect to the remote host "MyMailHost" using the POP3 port. Note that "MyMailHost" must be resolvable using the underlying name-resolution facilities used by the socket API.

socket/<192.68.51.1>;udp

A socket on the local host NIC that responds to IP address 192.168.51.1 using the UDP protocol and the next available port number; the remote address is left unspecified and must be specified at the time the device is accessed.

socket/:2147;udp/<192.168.51.4>:2147
: A socket on the default local host NIC using the UDP protocol and local port number 2147; connect to remote IP address 192.168.51.4 on remote port 2147.

socket/;udp/<broadcast>:2147
: A socket on the default local host NIC using the UDP protocol and the next available port number; connect to the remote broadcast IP address on remote port 2147.

adc/:2
: The 2nd ADC[14] on the local host; the ADC channel[15] is left unspecified and must be specified at the time the ADC is read.

.

adc/:2/:3
: The 2nd ADC on the local host; pre-connected to ADC channel 3, so any reads from the ADC will come from that channel.

adc/:2=25000
: The 2nd ADC on the local host; configured for a 25 ksps sample rate; the ADC channel is left unspecified and must be specified at the time the ADC is read.

pwm/:2=1000
: The 2nd PWM device on the local host; configured for a 1 kHz pulse frequency; the PWM channel is left unspecified and must be specified at the time the PWM is written.

flash/:3
: The 3rd flash memory on the local host

[14] We create a logical interface for ADCs (and DACs and PWM devices and other things) to make access simpler. We assume specific ADCs on various communication channels may be mapped to an "ADC" interface either directly, in the low-level implementation, or indirectly by creating a virtual ADC device and installing it into the device manager. For example you could create a virtual ADC device that connected to the actual ADC through an I2C bus and then install that device as a device on the ADC interface. The application could then access the ADC by opening it as a discrete device through the ADC interface; the details of the I2C transactions required to read the channels would be hidden from the application by the virtual device implementation.

[15] An ADC is assumed to be a multi-channel device and access to a specific channel is treated as if the channel were a remote device number. If there is only one channel, then the only valid channel number is 1.

3.2.6.3 Name Aliasing

The SDM shall provide support for aliasing of device names at the application level to maximize portability. It shall maintain a database of mappings between application-level device aliases and SDM device names.

Guidance: The aliased device names may be passed directly to the SDM 'open' functions; they will be mapped internally to the corresponding SDM device names. Hence, the application is not required to use the SDM-formatted device names, provided an appropriate mapping is configured in the name mapping database.

3.2.6.4 Name Translation

Translation between the SDM device names and logical/physical device names used for connection to the OS/HAL shall be done within the SDTL and will, therefore, be transparent to the application.

3.3 Development API

The SDM exposes a separate API intended for use only in developing and installing customized virtual devices[16]. This development API comprises the following function groups:

- Device creation and configuration
- Device name management
- Standard Device Translation Layer (SDTL)
- Name Alias Management

3.3.1 Device Creation and Configuration

The SDM development API must allow creation of custom devices based on the generic device templates. The precise mechanisms for creating custom devices will, in general, vary depending on the implementation language. For example, in an object-oriented implementation language (like 'C++'), the customization may be done by creating a new device class by inheriting from a device base-class and overloading its virtual implementation functions; in a procedural language (like 'C'), it may require defining an explicit dispatch table, referenced via the device handle, and procedures for installing device-specific handler functions into that dispatch table.

For that reason, the SDM does not define any specific functional API for the creation and configuration of customized devices. It merely requires that some mechanism be provided within a given implementation for creating such devices.

[16] Although this is defined as a "development" API, it may be used by applications that define and install their own devices. It is defined here as a separate API both as an organizational aide and in recognition of the fact that these functions may delve more deeply into the operation of the SDM than would traditionally be necessary for an API used by an application only to connect to and move data through devices

The SDM does require that the following API functions be provided to install/uninstall platform-specific devices that have been created into the SDM framework and to find devices, either within the framework itself or within the underlying OS/HAL, by their address (as described in section 3.2.6.1):

sdm_install/removeDevice
: Install/remove a device into/from the SDM framework.

sdm_install/removeDeviceFinder
: Install/remove a function to search for a device by address.

Installed devices must be associated with a device address, per section **3.2.6.1**, by which they may be identified for purposes of opening the device.

The device finder installation function is provided to allow customization of the search for a valid device and, in particular, to allow installation of a platform-specific search function that identifies devices supported by the OS/HAL on a given platform.

The SDM provides the following API functions to allow installation of customized protocol support into the SDM framework:

sdm_install/removeProtocolTranslator
: Install/remove a protocol translator into/from the SDM framework.

A protocol translator is used with the 'Protocol' field of a device name to add support for a new encapsulation protocol. The translator function is associated with a protocol name and automatically invoked during the read/write operations on a stream device that has been associated with that protocol. For example, you could install a function that implemented an encryption algorithm and associate it with the protocol name, "myencryption". A device opened with the protocol field in the Device Name set to "myencryption" would then have that encryption applied to any data written to the device.

3.3.2 Device Name Management

The SDM development API must provide the following functions for installing/uninstalling extensions to the device naming capabilities in order to support the addition of application- and/or system-defined device interfaces, protocols, and ports:

sdm_install/removeAddressComparison
: Install/remove an extended address parsing function into/from the SDM framework

sdm_install/removeAddressTranslation
: Install/remove an extended address translation function into/from the SDM framework

The first function may be used to customize the comparison of a specified device address to the addresses of devices installed into the SDM framework. In general, it would be used to allow the addition of templates for classes of device names that could be recognized by their pattern(s) rather than by an element-by-element match.

The second function may be used to add translations between standard device names and platform-specific device names.

3.3.3 Standard Device Translation Layer (SDTL)

The Standard Device Translation Layer (SDTL) provides the interface between the SDM and the platform-specific device APIs. It also provides an interface between the SDM and the operating system thread management interlocks used to provide locking for the device/region/stream, and between the SDM and the operating system signals used to provide alert notifications.

The STDL API provides the following device management functions that are accessible to the development API:

sdtl_initialize/cleanupDevices
: Initialize/cleanup the devices accessible through the SDTL.

sdtl_deviceExists
: Reports whether a named device exists, either within the SDTL framework or within the underlying OS/HAL.

sdtl_createAndOpen
: Creates and opens a device, either from within the SDTL framework or from the underlying OS/HAL.

The SDTL also provides the following API for installing device objects that can be 'opened' at the SDTL level.

sdm_install/removeDevice
: Install/remove a device into/from the SDTL framework.

This is equivalent to the same functionality provided at the SDM level, except that devices installed within the SDTL appear to the rest of the system as if they were logical devices provided by the underlying platform, rather than as virtual devices supported by the platform-agnostic device model. Hence, the same restrictions apply: an SDTL-level device cannot be 'connected' to any lower-level devices.

In addition to the functions above, the SDTL must provide the following function and install it into the SDM when the SDTL is initialized:

sdtl_findDevice
: Searches for a device, by address, within the SDTL framework and in the underlying OS/HAL

The SDTL must provide the following APIs for requesting/releasing operating system interlocks and posting/querying operating system signals; these are used internally by other SDM modules, not directly by the application:

sdtl_mtx_create/Destroy Create/destroy a mutex.

sdtl_mtx_request/release Request/release a mutex.

sdtl_ilk_create/Destroy Create/destroy an interlock.

sdtl_ilk_request/releaseForRead
Request/release read access to an interlock.

sdtl_ilk_request/releaseForWrite
Request/release write access to an interlock.

sdtl_flg_create/Destroy Create/destroy an event flag list.

sdtl_flg_test Test for flag(s) available in a flag list.

sdtl_flg_wait Wait for flag(s) to become available in a flag list.

sdtl_flg_set/clear Set/clear flag(s) in a flag list.

3.4 Name Management API

The SDM will provide the following functions for managing the name mapping database:

sdm_install/removeDeviceAlias
Add/remove a device alias to/from the mapping database.

Actual storage/retrieval of the name mappings will generally be done within the SDTL, since the mechanism for storage is likely to be OS-dependent. Hence, the SDTL must provide an equivalent set of API functions for managing name aliasing:

sdtl_install/removeDeviceAlias
Add/remove a device alias to/from the mapping database.

A particular implementation of the SDTL must include a mechanism for establishing the mapping database as part of the development process and for placing names into persistent storage. However, a particular implementation may or may not allow dynamic updates to the mapping database from the application via the aliasing functions; whether that support is provided depends, in part, on whether the platform holds the database in re-writable or read-only storage.

Once an alias has been added to the name mapping database the standard SDM name mapping API function (see section **3.2.6.3**) may be used to extract the device name associated with the alias.

Guidance: As part of its operation, and as shown in Figure 3-1, the SDM will maintain a mechanism at the SDTL layer to map SDM device names to OS/HAL logical and physical device names. The nature of the mechanism will depend on the platform: in a Linux system it might reside in the file system as a configuration file or as a list of file names in a device directory; under Windows it might reside in the Registry; in an embedded system with some RTOS kernel it might reside as a record in a non-volatile memory; for networked resources it might reside (in part) in some external DNS server; or, in any implementation, it might be hard-coded as fixed logic into the application itself.

4 Operating Environment and Constraints

As specified in section **3.1.4**, the SDM is designed to operate in conjunction with a threaded application model and a supporting thread management system. It can operate within a single thread (control loop) application, but does not provide any mechanism for asynchronous notifications associated with devices other than blocking calls for device access. It is assumed that, within a single-thread application, the presence or absence of device-related events will be signaled by the return codes provided by device access function calls.

It is assumed that low-level interrupt handling will be done within the HAL or OS operating below the level of the SDM, and that device-related notifications will be translated at the boundary between the HAL/OS and the SDTL into unblocking operations within the thread management system.

5 Appendix A: Interface/Protocol/Port Assignments

This appendix provides a list of interface names, protocol names/numbers, and port names/numbers defined at the time this guideline was released for use in specifying device names. PICMG shall maintain and publish a master list of interface names, protocol names/numbers, and port names/numbers intended for general use under this guideline.

5.1 Interfaces

The following list defines common interface names for use with the SDM API. Not all SDM implementations are required to recognize and support all interfaces, but if an interface is supported it should be identifiable at the SDM API by the name in this list.

console	local text console
file	file in the file system
pipe	pipe
socket	socket
memory	local block of memory
bus	local addressable physical bus
pci/ pcie	device on a PCI/PCIe bus
loopback	virtual communication channel that loops its output to its input (available for testing)
vcomm	virtual communication channel (available for testing and/or for inter-thread communication)
serial *com* *async* *sync*	serial communication channel (e.g. RS-232)
parallel *lpt* *centronics*	parallel port (e.g. Centronics)
usb	device on a USB bus

firewire	device on a Firewire bus IEEE 1394
scsi	SCSI bus
gpib	GPIB bus
i2c	I2C bus
spi	SPI bus
rtc	real-time clock
timer	pre-configured free-running timer
gpt	general-purpose timer
gpio A	general-purpose digital I/O channel
aio	general-purpose analog I/O channel
adc	analog output channel
dac	analog input channel
pwm	pulse width modulation output channel
flash	flash memory device
ipmi	device that communicates using the IPMI protocol
shapi	device conforming to the PICMG Standard Hardware API guideline

5.2 Protocols

The following list defines protocol names for use with the SDM API. Not all SDM implementations are required to recognize and support all protocols, but if a protocol is supported it should be identifiable at the SDM API by the name in this list.

To avoid confusion, when assigning protocol names to protocol numbers within an SDM implementation, protocol numbers defined by the IANA for use with internet communications should be reserved and used as-is. Hence, any assignment of protocol names to numbers within an implementation should map IANA protocol names to IANA

protocol numbers, and any other protocol names should be mapped to numbers outside that range.

Only the most commonly-used subset of the IANA-assigned protocol names is explicitly defined in this list. For protocols that do not appear here, assume that the "name" associated with an IANA protocol corresponds to the common protocol abbreviation used in the IANA definition.

raw	raw binary (i.e. no encoding)
master	bus master (e.g. for I2C, SCSI, GPIB, etc.)
slave	bus slave (e.g. for I2C, SCSI, GPIB, etc.)
icmp	Internet Control Message Protocol, version 4 - icmpv4
ip	raw Internet Protocol, version 4 - Ipv4
tcp	Transmission Control Protocol
udp	Universal Datagram Protocol
ipv6	raw Internet Protocol, version 6
icmpv6	Internet Control Message Protocol, version 6
iptest1	available for testing
iptest2	available for testing
ppp_pad32	PPP with padding to create an integer number of 32-bit elements in the packet
ethernet	raw Ethernet packet protocol
tokenring	raw Token-Ring packet protocol
wifi	raw Wi-Fi packet protocol
bluetooth	raw Bluetooth packet protocol

5.3 Ports

The following list defines special port names for use with the SDM API. Not all SDM implementations are required to recognize and support all ports, but if a port is supported it should be identifiable at the SDM API by the name in this list.

To avoid confusion, when assigning port names to port numbers within an SDM implementation, port numbers defined by the IANA for use with internet communications should be reserved and used as-is. Hence, any assignment of port names to numbers within an implementation should map IANA port names to IANA port numbers, and any other port names should be mapped to numbers outside that range.

Only the most commonly-used subset of the IANA-assigned ports is explicitly defined in this list. For ports that do not appear here, you may assume that the "name" associated with an IANA port corresponds to the common port abbreviation used in the IANA definition.

For devices/channels (e.g. RS-232 ports) which use the port number as an index for the device instance on the local interface, index numbers begin at 1.

For devices/channels (e.g I2C, GPIB, SCSI, etc.) which use the port number to specify the address for a specific remote device, the port number corresponds directly to the remote device address.

client	client end of a pipe
remote	"remote" end of a virtual communication channel
server	server end of a pipe
local	"local" end of a virtual communication channel
dynamic	requests dynamic assignment of a port number
stdin	standard input channel on a console
stdout	standard output channel on a console
stderr	standard error channel on a console
systime	free-running 'system' timer; counts at system time-tick rate
highrestime	designated free-running 'high-resolution' timer
echo	message echo
ftp_data	data channel for FTP
ftp	control channel for FTP
ssh	secure shell

telnet	Telnet
smtp	SMTP
time	Time protocol
dns	DNS
bootp_server dhcp_server dhcpv4_server	BOOTP/DHCP version 4 server
bootp_client dhcp_client dhcpv4_client	BOOTP/DHCP version 4 client
gopher	Gopher protocol
finger	Finger protocol
http	HTTP
kerberos	Kerberos protocol
pop3	POP version 3
ident auth	Identification/Authorization protocol
sftp	Secure FTP
nntp	NNTP
ntp	NTP
netbios_ns	NetBIOS name service
netbios_dgm	NetBIOS datagram service
netbios_ssn	NetBIOS session service
imap	IMAP
snmp	SNMP
snmp_trap	SNMP trap

ipx	IPX
ptp_event	PTP (IEEE 1588) Event messages
ptp	PTP (IEEE 1588) General messages
ups	Uninterruptible Power Supply control/status
https	Secure HTTP
smtps	Secure SMTP
dhcpv6_client	DHCP version 6 client
dhcpv6_server	DHCP version 6 server
smtp_submit	SMTP submission port
nfs NFS	
ftps_data	data channel for Secure FTP
ftps	control channel for Secure FTP
telnets	Secure Telnet
imaps	Secure IMAP
pop3s	Secure POP version 3

6 Appendix B: Common Device APIs

This appendix defines SDM API extensions for commonly-used devices. These APIs are not part of the SDM per se, but are intended to define a common paradigm for access to these devices that simplifies application development. SDM libraries corresponding to these device API extensions are available as open-source development resources.

Guidance: These device APIs represent abstractions of the corresponding device behavior. In most cases there will be a physical device of the associated type – e.g. an RS-232 port or a socket – connected below the abstraction API to provide physical device access. However, that is not required. Rather, the device API can serve as an abstract wrapper for some entirely unrelated device, or even for some software module (like a simulator) that mimics a device without any physical connection. For instance, an abstracted asynchronous serial port device may be connected to some network tunneling protocol through a socket; or an abstracted socket API may be used for access to one or more serial ports using a network protocol.

6.1 File

A virtual device modeling a file in the file system exposes a single byte-wide stream through which data is transferred to/from the file system.

A file device will support the get/setPosition() functions of the stream API but will not support the readFrom() and writeTo() functions.

The name for a file includes the interface specification “file”; the device address defines the file name; the channel address, if specified, identifies the host computer on which the file resides. The following illustrates an example name for a physical file on the local machine:

file//<c:/users/data/MyFile.txt>

Information about the file, such as archive status, write protection, the save date, and so on, are accessible through the Control Region, as in able 6-1. Multi-word fields are presented in the API in host word order.

Table 6-1: File Control Registers

Offset	***Dir***	***Description***
0	RO	Flags: Bit 0: 1=read-only Bit 1: 1=hidden Bit 2: 1=archived Bit 3: 1=directory Bit 4: 1=symbolic link
3:1	n/a	Reserved
Offset	***Dir***	***Description***

5:4	RO	File Size
7:6	RO	Time of creation
9:8	RO	Time of last Modification
11:10	RO	Time of last status access

6.2 Digital I/O (GPIO)

A virtual device modeling a general-purpose digital I/O channel exposes a region, starting at index 0, through which I/O signals may be set/queried. Each element accessed through the region represents a different I/O "port". Hence, for example, you can write to port 1 using the standard Region API write function by specifying an offset of 1. To make things easier, ports in a GPIO device have a fixed width of 32 bits, matching the width of the standard Control Region element. If the actual port is narrower than 32 bits, then the port bits are mapped to the LSBs of the corresponding word and the MSBs are not used. Unused bits in the port element are ignored for writes and return zero on reads.

Guidance: Because the port elements are defined to always be 32 bits wide, this constrains the GPIO device model to supporting port widths of 32 bits or less. If you really need GPIO support for wider ports, you will need to create a special device model for that purpose.

Controls for the nth of N ports in a GPIO region, where n can range from 0 through N-1, are mapped into the device Control Region, as follows:

Table 6-2: GPIO Control Registers

Offset	***Dir***	***Description***
Capabilities Enumeration		
nx16 + 0	RO	Flags: signal is readable
nx16 + 1	RO	Flags: signal is writable
nx16 + 2	RO	Flags: signal output can be tri-stated
nx16 + 3	RO	Flags: signal output can be pulled up
nx16 + 4	RO	Flags: signal output can be pulled down
nx16 + 5	RO	Flags: signal can generate an alert
Pin Configuration		
nx16 + 6	RW	Flags: set pin 'use' enable
nx16 + 7	RW	Flags: set signal direction to output
nx16 + 8	RW	Flags: set signal output to tri-state
nx16 + 9	RW	Flags: set signal output to pull-up

Offset	*Dir*	*Description*
nx16 + 10	RW	Flags: set signal output to pull-down
Alert Configuration and Status		
nx16 + 11	RW	Flags: set alert to edge-triggered
nx16 + 12	RW	Flags: set alert to positive logic
nx16 + 13	RW	Flags: set alert to enabled
nx16 + 14	RO	Flags: alert is asserted
nx16 + 15		*reserved*

Each bit within a configuration element corresponds to the equivalent bit within the nth I/O port.

Any bit in the ports that is reported to be both readable and writable is presumed to be configurable as either an input or an output through the direction register.

Any bit in the ports that is reported to support alert generation can be configured to generate the alert through the alert configuration and enable registers.

Guidance: We don't report the details of what alert configurations are supported; read back alert configurations after you set them to verify you got what you wanted.

The name for a GPIO device includes the interface "gpio"; the instance identifier is specified as the port in the Channel Address.

The following is an example of a GPIO device name:

gpio/:1

6.3 Asynchronous Serial Port

A virtual device modeling an asynchronous communication channel exposes a byte-wide stream API. A serial port does not support the get/setPosition() functions or the readFrom() and writeTo() functions of the Stream API.

Channel configuration is managed through the Control Region, which supports the following registers:

Table 6-3: Async Channel Control Registers

Offset	*Dir*	*Register*	*Description*
0	RO	Feature Set	Reports what channel features are supported by the device
1	RO	Maximum Baud Rate	Maximum available baud rate
2	RO	Link Status	Link status flags
3	RW	Link Control	Manual link control flags
4	RW	Baud Rate/TX Baud Rate	Transmit baud rate; both transmit and receive baud rates if they are not separately configurable.
5	RW	RX Baud Rate	Receive baud rate
6	RW	Data Format	Data format (framing)
7	RW	Flow Configuration	Configuration flags for flow-control modes
8	RW	XON character	Character value to be used for XON in software flow control
9	RW	XOFF character	Character value to be used for XOFF in software flow control

Guidance: Not all combinations of configuration are supported on all platforms; a write of an incorrect value to a configuration register should be rejected by the device implementation.

Within the Feature Set register, feature configurability is indicated by the feature configuration bit being set (1); if the bit is not set, the feature is not supported/may not be configured. The feature set bits are:

Table 6-4: Async Channel Feature Set Register Fields

Field	*Bits*	*Description*
not used	31:30	
Loopback	29	Loopback mode is supported
Parity	28	Parity is supported
Data Bits	27	# of data bits is configurable
Stop Bits	26	# of stop bits is configurable
RX Rate	25	RX baud rate is independently configurable
[TX] Rate	24	Baud rate is configurable
not used	23:22	

Field	*Bits*	*Description*
Xon/Xoff In	21	Use of XON/XOFF for RX is supported
Xon/Xoff Out	20	Use of XON/XOFF for TX is supported
DSR	19	Use of DSR is supported
DTR	18	Use of DTR is supported
CTS	17	Use of CTS is supported
RTS	16	Use of RTS is supported
not used	15:10	
Break Out	9	Break assertion is supported
Break In	8	Break detection is supported
not used	7:0	

Within the Link Status register, an “on” or “in effect” status is indicated by a flag bit being set (1); if the bit is not set, the item is “off” or “not in effect”. The status bits are:

Table 6-5: Async Channel Link Status Register Fields

Field	*Bits*	*Description*
not used	31:30	
Loopback	29	Loopback is enabled
not used	28:22	
XOn In	21	Xon has been received (input)
XOn Out	20	Xon has been sent (output)
DSR	19	DSR (input) is asserted
DTR	18	DTR (output) is asserted
CTS	17	CTS (input) is asserted
RTS	16	RTS (output) is asserted
not used	15:10	
Break Out	9	A break condition is asserted on TX
Break In	8	A break condition was detected on RX
not used	7:3	
Corrupted Data	2	Corrupted data (i.e. parity error) was detected
Framing Error	1	A framing error was detected
Overrun	0	A receive overrun was detected

Within the Link Control register, an “on” or “in effect” status is indicated by a flag bit being set (1); if the bit is not set, the item is “off” or “not in effect”. The control bits are:

Table 6-6: Async Channel Link Control Register Fields

Field	***Bits***	***Description***
not used	31:30	
Loopback	29	Enable loopback mode
not used	28:21	
XOn Out	20	Send Xon (1) or Xoff (0) (manual mode only)
not used	19	
DTR	18	Assert DTR (manual mode only)
not used	17	
RTS	16	Assert RTS (manual mode only)
not used	15:10	
Break Out	9	Assert a break condition on TX
not used	8:0	

Guidance: Xon/Xoff messages are sent when the device is in manual control mode (not automatic ‘Xon Out’ generation) and the state of the control bit changes. At device initialization, the state of the control bit will normally be set to 1 (reception enabled) but no Xon character will be sent. From that point on, toggling the state of the control bit will result in transmission of the appropriate character.

The data framing configuration is controlled by 3 fields within the Data Format register, as follows:

Table 6-7: Async Channel Data Format Register Fields

Field	***Bits***	***Description***
not used	31:19	
Parity	18:16	Parity mode (enumerated)
not used	15:11	
Stop Half Bits	10:8	# of half-bits for a stop condition (e.g. 2=1bit)
not used	7:4	
Data Bits	3:0	# of data bits/byte

The enumerated values for the parity configuration field are:

Table 6-8: Async Channel Parity Configuration Codes

Value	*Configuration*
0	No parity
1	Odd parity
2	Even parity
3	Mark parity
4	Space parity

The handshaking configuration is controlled by flags within the Flow Configuration register, as follows:

Table 6-9: Async Channel Flow Configuration Register Fields

Field	*Bits*	*Description*
not used	31:6	
Xon/Xoff In	21	Enable use of Xon/Xoff reception
Xon/Xoff Out	20	Enable automatic Xon/Xoff transmission
DSR	19	Enable use of DSR
DTR	18	Enable automatic DTR generation
CTS	17	Enable use of CTS
Auto RTS	16	Enable automatic RTS generation
not used	15:0	

The name for an asynchronous serial device includes the interface identifier “com”; the instance identifier is specified as the port in the Channel Address. In addition, the device name may include the following ordered configuration parameters:

1. Baud Rate: Specified as an integer in bits/second
2. # of Data Bits: Specified as an integer in bits; typically limited to 5, 6, 7, or 8.
3. # of Stop Bits: Specified as a floating-point number; typically limited to 1, 1.5, or 2.
4. Parity: Specified using one of the following letter codes:
   N = no parity
   O = odd parity
   E = even parity
   M = mark parity
   S = space parity

The configuration parameters are not required, since all configurations may be updated by the application through the device Control Region. Default configurations are platform-specific. The following illustrates an example name for an asynchronous serial device:

com/:2=8,1,e,19200

6.4 Socket

A virtual device modeling a socket exposes a byte-wide stream API. A socket does not support the get/setPosition() functions, but does support the readFrom() and writeTo() functions of the Stream API.

If a socket device is 'connected' to a remote address, then the 'read()' and 'write()' functions of the stream API will automatically address their operations to that remote address, even if the device is configured for a connectionless protocol like UDP. If the device is not connected to a remote address, then the application must identify the remote address at the time of the transaction by using the 'readFrom()' and 'writeTo()' functions of the stream API.

Guidance: If the remote address specified in the 'readFrom()' function identifies a valid remote host, then only messages from that host will be accepted and returned; if the remote address specified in the 'readFrom()' function does not identify a specific remote host, then the function will accept any inbound message and return the address of the host from which it was received.

The Socket device utilizes the Control Region API of the general device for channel configuration. **Table 12** lists the available Control Region registers:

Table 6-10: Socket Channel Control Registers

Offset	*Dir*	*Register*	*Description*
0	RO	Interface Number	reports the index number of the NIC interface through which this device is connected

Guidance: Buffer sizing is managed using the standard stream API 'get/setBufferSize()' functions and timeouts are managed on an access-by-access basis through the 'timeout' arguments to the access functions. Hence, there are no controls defined for configuring those. The mechanism used for establishing blocking behavior (e.g. simply using blocking calls for the internal socket send()/recv() functions or using the 'select' mechanism) is established internally by the device implementation; no facility is exposed for allowing the application to configure that choice.

Guidance: In the interest of simplicity, other socket configuration options have been omitted from the API; it is presumed that the device implementation, itself, will choose and set up appropriate configurations when it is initialized. If an implementer wishes to include

those more esoteric configuration options, they may add them at the end of the standard command region register set.

The Socket device utilizes the ‘control()’ function API of the general device for out-of-band control activities. The following control functions are defined for managing device access:

0	bind	bind the socket to a local address
1	connect	connect the socket to a remote address
2	listen	configure the socket to listen for incoming connection requests
3	accept	wait for an incoming connection request

Device addresses are passed in and out of the control function using the address structures defined by the implementation for specifying device addresses; the content of the structures must be appropriate for the socket device. The queue depth (number of connections that can be queued) for the ‘listen’ command is passed into the control function as an integer, sized as the native integer type for the platform/development environment. The new connection is returned from the ‘accept’ function as a device handle which references a new device object created to manage the connection.

The name for a socket device uses the interface specification “socket” and must include the encapsulation protocol (e.g. “tcp” or “udp”) as the protocol specifier in the Channel Address. If a host name and/or port is included in the Channel Address, it is interpreted as the host name (or IP address) and/or port for the local network interface; the host name/IP address and/or port for a remote system may be specified as the host name and/or port in the Device Address if you want to pre-establish the connection address at the time the device is opened. If the host name is omitted from the Channel Address, or if it specifies the local host, the SDM assumes it should connect to the default network interface on the local platform.

The following illustrates some example names for socket connections:

socket/;udp:49152

connects to the default network interface on the local host using port 49152 and the UDP protocol; the remote address remains unspecified, so accesses must use the ‘readFrom()’ and ‘writeTo()’ functional API

socket/;tcp/<myhost.com>:pop3

connects to the default network interface on the local host on a system-assigned port and establishes a connection to the remote host ‘myhost.com’ using the TCP protocol on the POP3 service port; the

remote address is specified at the time of connection, so accesses must use the ‘read()’ and ‘write()’ functional API.

6.5 I2C Bus and I2C Device

A virtual device modeling a general I2C bus or a specific device on an I2C bus creates a byte-wide stream API. An I2C device does not support the get/setPosition() functions, but does support the readFrom() and writeTo() functions of the Stream API.

An I2C bus slave will also support the alert API to receive notifications that an external master is requesting attention.

Guidance: If the device is used to implement an I2C slave, it can’t ACK a request for a data read unless/until the application thread using the device acknowledges the request and posts its response. Hence, the I2C bus will be held in a wait state until that happens. That means such threads generally ought to run at an elevated priority so as to minimize the latency between receiving a read request and responding to it.

Transactions from an I2C bus master include a device address which must be transferred on the bus as part of the transaction. If the device object was opened by specifying a particular device address, the physical device is accessed using the ‘read()’, ’write()’, and ’transact()’ functions of the API and no address is provided at the time of the transaction. If the device object was opened without specifying a device address, then the physical device is accessed using the ‘readFrom()’, ‘writeTo()’, and transactToFrom()’ functions of the API and a device address must be specified with each transaction.

Responses to transaction requests on an I2C slave channel should simply use the ‘read()’ or ‘write()’ functions.

Channel configuration is managed through the Control Region, which supports the following registers:

Table 6-11: I²C Channel Control Registers

Offset	*Dir*	*Register*	*Description*
0	RO	Max Bit Rate	Maximum available bit rate
1	RO	Link Status	Link status flags
2	RW	Bit Rate	Transaction bit rate (master only)

Within the Link Status register, an “on” or “in effect” status is indicated by a flag bit being set (1); if the bit is not set, the item is “off” or “not in effect”. The status bits are:

Table 6-12: I²C Channel Link Status Register Fields

Field	*Bits*	*Description*
not used	31:8	
Bus Busy	7	The I^2C bus is busy
not used	6:3	
Device Busy	2	The remote device is busy
NACK	1	The remote device NACKed the transaction
Lost Arbitration	0	A receive overrun was detected

The following are examples of I2C bus master device names:

i2c/:1

Connects to the first I2C bus as a master without specifying a physical device; accesses must use the 'readFrom()' and 'writeTo()' functional API

i2c/;master:1

Same as previous; if no protocol is specified, the channel is assumed to be a bus master

i2c/:2/:66

Connects to the second I2C bus as a master and specifies a pre-connection to the physical device with address 66; accesses must use the 'read()' and 'write()' functional API

i2c/:2/;10:66

Connects to the second I2C bus as a master and specifies a pre-connection to the physical device with the 10-bit address 66 (the device protocol "10" indicates the 10-bit address); accesses must use the 'read()' and 'write()' functional API

The following is an example of an I2C bus slave device name:

i2c/;slave:2/:66

Connects to the second I2C bus as a slave that responds to device address 66.

Device masters merely initiate transactions by making a read/write/transact call through the stream API.

Device slaves should pend on the 'waitAlert()' function of the general device API, which will wake them when an external master has requested a transaction. The following flag values are used to signal what is being requested:

00000001h a read request has been received

00000002h a write request has been received

For a write request, the application should respond by calling the 'read()' function of the stream API to receive the incoming data;

For a read request, the application should respond by calling the 'write()' function of the stream API to post the outbound data.

Guidance: Unless there is only one data set that the slave application can write, the 'read' request will generally be preceded by a 'write' request that indicates what data should be written in response to the next 'read' request.

Guidance: If the application must be able to access the bus as both a master and a slave – as might be the case for a device using an I2C channel as an IPMB – the device implementation must support that usage by allowing the device to be opened in both modes simultaneously by separate master and slave processing threads.

6.6 SPI Channel

A virtual device modeling a general SPI bus or a specific device on an SPI bus creates a stream API with a configurable element size; the element size for an SPI master may be configured through a register in the Control Region; the element size for an SPI slave should be pre-configured when the device is created.

Guidance: Given the typical transmission rate of SPI busses, and the fact that there is no mechanism for holding off transmission while preparing responses, an SPI slave must either have pre-queued whatever data is going to be provided or be able to respond to a request for data within one bit clock cycle (i.e. typically on the order of 1µs or less). Given that, it would generally be unrealistic to expect that slave device functionality would be useful at the application level and this API is not designed to provide it. All SPI devices are opened as SPI bus masters.

An SPI device does not support the get/setPosition() functions of the Stream API. Since SPI transactions always include a simultaneous read and write, the SPI device doesn't support the readxxx() and writexxx() functions of the Stream API; it only supports transact() and transactToFrom().

Guidance: Although SPI itself does not specify a device address as part of the protocol, the SPI bus is often shared in hardware by providing discrete chip-select signals to different devices. The logical model for the SPI device designates each individual chip-select as an indexed device address, starting at address 0, and supports that by treating them as remote addresses.

Transactions through an SPI bus include an indexed device address which must be transferred on the bus as part of the transaction. If the device object was opened by

specifying a particular device address, the physical device is accessed using the 'trascact()' function of the API and no address is provided at the time of the transaction.

If the device object was opened without specifying a device address, then the physical device is accessed using the 'transactToFrom()' function of the API and a device address must be specified with each transaction.

Channel configuration is managed through the Control Region, which supports the following registers:

Table 6-13: SPI Channel Control Registers

Offset	*Dir*	*Register*	*Description*
0	RO	Max Bit Rate	Maximum available bit rate
1	RO	Transfer Word Bytes	With of the transfer word, in bytes
2	RW	SPI Mode	The SPI transfer mode (0-3)
3	RW	Transfer Word Bits	With of the transfer word, in bits
4	RW	Bit Rate	Transaction bit rate (master only)

The following are examples of SPI bus master device names:

spi/:1

Connects to the first SPI bus without specifying a physical device; accesses must use the 'transactToFrom()' functional API

spi/:2/:3

Connects to the second SPI bus and specifies a pre-connection to the physical device with indexed address 3; accesses must use the 'transact()' functional API

6.7 USB Device

A virtual device modeling a USB-connected device exposes a byte-wide stream API. A USB device does *not* support the get/setPosition() functions or the read() and write() functions of the Stream API; it uses the readFrom()/writeTo()/transactToFrom() functions of the Stream API to access individual endpoints within the USB device. Hence, access to a USB end-point requires specifying the desired endpoint as part of the transaction.

Guidance: Although USB is physically a bus, the driver API establishes a 1:1 association between the driver and a specific device on the bus at the time of connection. Hence, for access purposes, a USB connection is properly modeled as a point-to-point link. However, within that single device, there will generally be multiple endpoints connected to individual streams. Hence, the use of the functional API readFrom()/writeTo()/transactToFrom().

Guidance: Each USB endpoint supports a single USB transfer type, so access to a given endpoint automatically utilizes that transfer type.

The USB device is configured through the Control Region API of the general device. The following control region registers are defined:

Table 6-14: USB Channel Control Registers

Offset	*Dir*	*Register*	*Description*
0	RW	Configuration	Currently-selected configuration
1	RW	Interface	Currently-selected interface

The USB remote device identification information is extracted through the 'control()' function API of the general device. The following control functions are defined:

0	getDeviceDesc	get the device descriptor
1	getConfigDesc	get a configuration descriptor
2	getInterfaceDesc	get an interface descriptor
3	getEndpointDesc	get an endpoint descriptor
4	getStringDesc	get a string descriptor

The returned data for the 'getXXX' functions are data structures containing the specified descriptors as defined by the USB standard:

- 'get device descriptor' takes no input and returns a USB device descriptor structure

- The input to 'get configuration descriptor' is a 1-element array of 16-bit integers that specifies the configuration number; it returns a USB configuration descriptor structure.

- The input to 'get interface descriptor' is a 2-element array of 16-bit integers containing, in order, the configuration number and the interface number; it returns a USB interface descriptor structure.

- The input to 'get endpoint descriptor' is a 3-element array of 16-bit integers containing, in order, the configuration number, the interface number, and the endpoint number; it returns a USB endpoint descriptor structure.

- The input to 'get string descriptor' function is a 1-element array of 16-bit integers containing the string index; it returns a USB string descriptor structure.

The following are examples of USB device names:

usb//<0x0040:0x0001>

Connects to the next available instance of a USB device with VID:PID=0040h:0001h; will connect through interface 0 of configuration 0 by default

usb//<0x0040:0x0001:2:3>

Connects to the next available instance of a USB device with VID:PID=0040h:0001h; will connect through interface 3 of configuration 2

usb/:2/<0x0040:0x0001>:2

Connects to the 2nd instance of a USB device with VID:PID=0040h:0001h; will connect through interface 0 of configuration 0 by default

usb/:"001"/<0x0040:0x0001>

Connects to an instance of a USB device with VID:PID=0040h:0001h and a serial number string that matches the string "001"; will connect through interface 0 of configuration 0 by default

usb//<"string">

Connects to next available instance of a USB device with a device name string matching "string"; will connect through interface 0 of configuration 0 by default

6.8 PCI/PCIe Device

A virtual device modeling a PCI/PCIe-connected device exposes a region API with a width and length defined by the physical device interface.

Guidance: Although PCI/PCIe is physically a bus, the driver API establishes a 1:1 association between the driver and a specific device on the bus at the time of connection. Hence, for access purposes, a PCI/PCIe connection is properly modeled as a point-to-point link.

The PCI/PCIe device maps the PCI/PCIe control register set to the Control Region of the SDM API, with each element in the Control Region corresponding to a single register in the PCI/PCIe control space. Since PCI/PCIe control register are 16 bits wide, they are mapped to the LSBs of the Control Register space.

In general, each BAR within the PCI/PCIe device that defines a separate address space must be mapped to a separate device object instance.

The PCI/PCIe device name uses the interface specifier “pci” and includes the identification information for the device within the host name of the Channel Address. The following are examples of PCI/PCIe device names:

pci//<0x0040:00001>

Connects to the next available instance of a device on the PCI/PCIe bus with VID:PID=0040h:0001h; will connect using the addressing information in BAR 0 by default

pci/:2/<0x0040:00001>

Connects to the 2nd instance of a device on the PCI/PCIe bus with VID:PID=0040h:0001h; will connect using the addressing information in BAR 0 by default

pci//<0x0040:00001:2>

Connects to the next available instance of a device on the PCI/PCIe bus with VID:PID=0040h:0001h, using the addressing information in BAR 2

pci/<2@1>

Connects to the device at slot 2 of PCI/PCIe bus 1; will connect using the addressing information in BAR 0 by default

pci//<2@1:3>

Connects to the device at slot 2 of PCI/PCIe bus 1, using the addressing information in BAR 3

6.9 Free-Running Timers

A virtual device modeling a pre-configured free-running timer device exposes a single-element region API with a 64-bit element width. The current time is reported through the single element within the region; the device only supports the read() function from the Region API.

By convention, the first two timers available on any system should be:

timer/:1	Reports the time in system (thread) time-ticks
timer/:2	Reports the time in high-resolution time-ticks

Either timer may be unavailable if the system does not support it, but the instance numbering should be maintained. Other timers, if any, should be added using instance numbers beginning with 3.

The Timer device exports configuration information using the Control Region from the general device. Configuration information is available at the following offsets:

Table 6-15: Timer Control Registers

Offset	*Dir*	*Register*	*Description*
1:0	RO	Frequency	Count rate for the timer
3:2	RO	Maximum Count	The maximum count reported by the timer

The 64-bit fields are accessed through the 32-bit Control Region elements in native host word order, so one can pass a pointer to a 64-bit integer to the 'read()' function.

If the underlying timer implementation provides a count of less than 64 bits, the maximum count will reflect that.

6.10 IPMI devices

A virtual device modeling a remote component that communicates using the IPMI protocol may be used to hide the details of the physical channel from the application. For example, if a device is created as an "IPMI" device, the application will not necessarily know whether it resides on the local IPMB or is accessed remotely via RMCP through a network link.

A virtual IPMI device creates a byte-wide stream API that supports the read()'/'write() and readFrom()/WriteTo() API functions. The former should be used if the remote endpoint was established when the device was created; the latter should be used to dynamically address different endpoints at the time of access.

Guidance: It is assumed that BMCs will connect without a pre-established endpoint so they can listen for incoming requests from the various remote processors, while other processors will connect with the BMC as a pre-established endpoint.

Guidance: It is anticipated that a virtual IPMI device implementation may provide an auxiliary functional API that is customized for IPMI transactions to remove the burden of parsing/formatting IPMI messages from the application. That is highly recommended, but the details are beyond the scope of this guideline.

An IPMI device name uses the interface specifier "ipmi" and identifies a logical channel using an index number in the port field of the Channel Address.

Guidance: It is not possible to completely hide the details of the logical channel from the application if it needs to pre-establish a connection to a remote address, since the form for the address will depend on the channel through which it operates. For example, a device on the local IPMB is addressed by its slot number, whereas a device on a network is identified

by its IP address. However, a baseboard controller that listens for inbound traffic would not need to worry about that because the inbound address would be passed to it by the 'readFrom()' function and accepted by the 'writeTo()' function without it needing to examine the details of the address itself. For other devices, it is assumed there will be only one applicable channel – the one that connects to the BMC – that can be configured using an alias to hide the details from the application.

The following are examples of IPMI device names:

ipmi/:2

Connects to the 2nd instance of an IPMI communication channel without a pre-established connection to a remote address; accesses must use the 'readFrom()'/'writeTo()' function API.

ipmi/:1/<slot:7>

Connects to the 1st instance of an IPMI communication channel and establishes a connection to the device at slot 7 on that bus (assumes it's a local IPMB) – note that the host name "slot:" is reserved to indicate "a slot on the local IPMB"; accesses must use the 'read ()'/'write ()' function API.

ipmi/:1/<bmc>

Connects to the 1st instance of an IPMI communication channel and establishes a connection to the BMC for that bus – note that the host name "bmc" is reserved to indicate "the BMC for this bus"; accesses must use the 'read ()'/'write ()' function API.

ipmi/:1/<MyHostName>

Connects to the 1st instance of an IPMI communication channel and establishes a connection to the device at the specified host address (assumes a host name that can be resolved on the associated channel); accesses must use the 'read ()'/'write ()' function API.

6.11 Standard Hardware API (SHAPI)

A virtual device modeling a hardware API that implements the Standard Hardware API (SHAPI) defined by PICMG exposes a 32-bit wide region, starting at index 0, through which the standard SHAPI register set may be accessed. Each element in the region represents a corresponding register in the SHAPI address space.

The following is an example of a SHAPI device name:

shapi/:1

Guidance: SHAPI devices will generally be accessed through some other physical channel, so devices accessible through a 'shapi' interface must encapsulate the details of the physical access through that channel. Figure 6-1 shows an example of a 'typical' SHAPI device stack.

The dark boxes on the right side of the figure represent the physical SHAPI implementation within a device (e.g. inside an FPGA on some remote board).

The dark boxes on the left side of the figure represent a "logical" SHAPI device within the SDM, which is an element of the overall "Device Model Stack". The lower box is the logical device that exposes the SHAPI register set; the upper box uses that register set to create a logical model of an ADC that happens to live behind the SHAPI registers in the physical device.

As illustrated, there is a "stack" of device models that interact:

- At the bottom of the stack is a device that interacts with the physical communication channel

- The next block up is a device that manages interpretation/formatting of the message packets in whatever protocol is being used to encapsulate the communications; it maps the information in the message packets to an addressable 'register space'.

- The next block up is the "SHAPI" device, which understands the meaning and use of the specific SHAPI register set and incorporates any required SHAPI-specific logic.

- At the top of the stack is a device that models the behavior of the ADC that has been mapped behind the SHAPI registers.

The SDM would allow you, for example, to create those various blocks as devices that you would install into the device manager (or, in the case of the lower-most block, they may already exist within the device manager standard implementation). In particular, if the "SHAPI" device were installed into the SDM using the identifier "shapi/:1", then it could be opened as a named device representing the first device instance on the "shapi" interface; and the device handle that was returned from the 'open()' call would expose a SHAPI-conformant addressable region through which the SHAPI registers could be accessed.

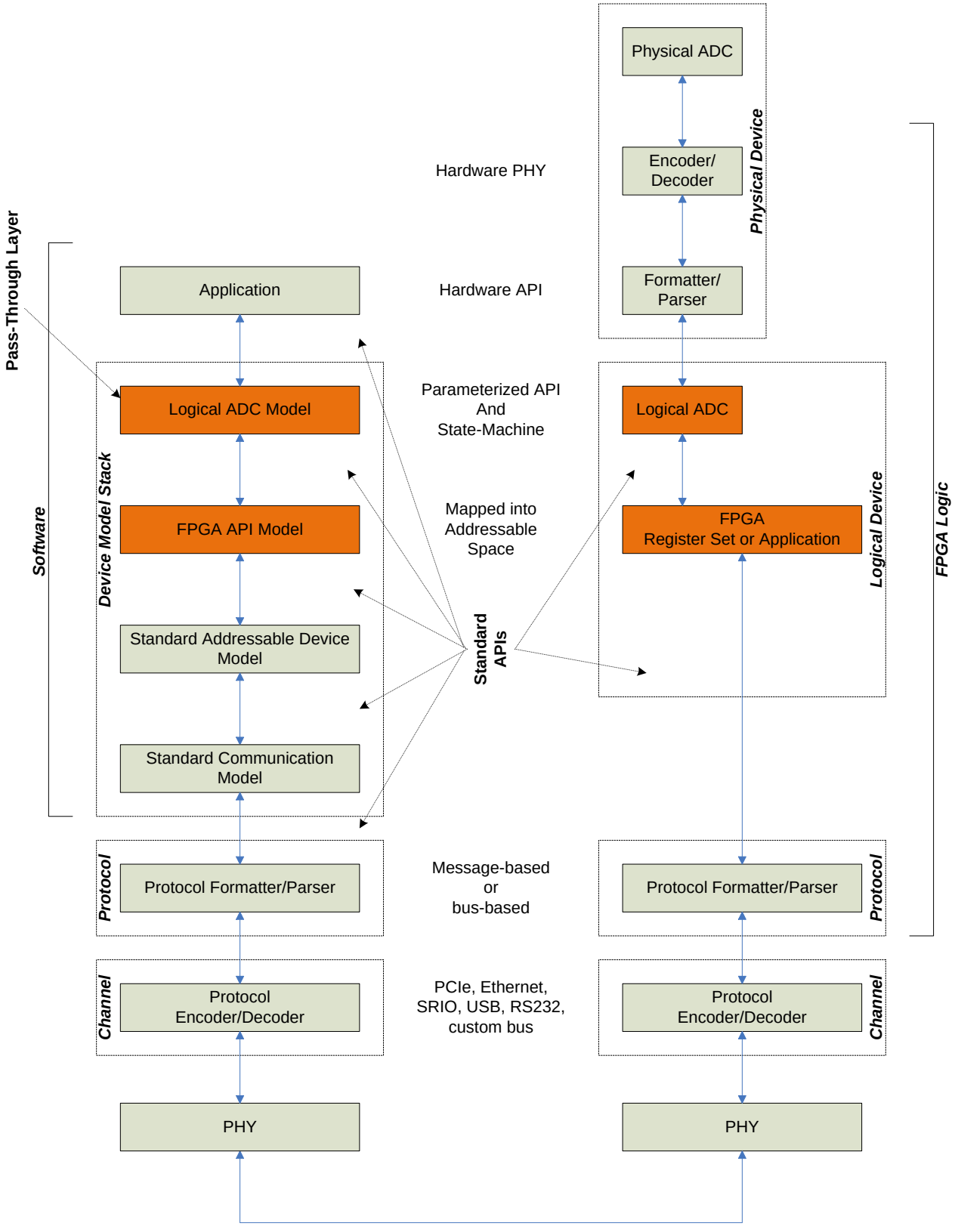

Figure 6-1: Example SHAPI Device Stack

NOTE: From the standpoint of the application software, whether the physical SHAPI device was connected directly to an addressable bus or through a communication channel – and the details of the communication protocols used to carry the SHAPI information – would be invisible and irrelevant. Different SHAPI devices connected using different physical channels and/or different communication protocols could be treated, at the application layer, as if they were the same.

7 Appendix C: C++ Implementation Headers

This appendix provides example headers, written in the C++ language, to illustrate an implementation of the required SDM API functionality. These headers have been simplified to exclude implementation details in order to focus on the templates for implementation of the required public APIs.

7.1 Device Manager API

The following header defines the API for creating/opening/closing devices available within system. It provides a generalized way of connecting to a device by specifying its name; the device object that is returned is of the type (region or stream) appropriate for the device.

Application-level device manager

```
class CUSDM {
    // public device API -- available to applications
public:
    // methods
    // construction/destruction/initialization
    /*
     NOTE:    constructor is not public because this is only intended for use as a bse-class!
     */
    virtual ~CUSDM() { cleanup(); }

    // operations
    // search for a device by its specification or name
    static bool deviceExists ( Addressing::SUDevSpec const& Address );
      /*
      NOTE:    This is left out if device name parsing is not supported; in that case,
      use the device specification to identify a device.
      */
    static bool deviceExists ( _Char const *pstrDeviceName );
    // open a connection to a device by its specification or name
    static CUDevice* open ( UNativeInt uOpenOptions, UNativeInt uAccessOptions,
                            Addressing::SUDevSpec const& Address,
                            _Char const* pstrConfigChannel =_NULL, _Char const*
pstrConfigDevice =_NULL );
    static CUDevice* open ( UNativeInt uOpenOptions, UNativeInt uAccessOptions,
                            _Char const* pstrDeviceName );
    // search for a device name alias
    static _LResult findAliasedDeviceName (char strDevName[Addressing::DEVICE_NAME_CHARS_MAX],
                                           char const* pstrAlias );
    // implementation
    // report whether two addresses match
    static _LResult deviceSpecsMatch ( Addressing::SUDevSpec const& Spec1,
                                       Addressing::SUDevSpec const& Spec2 );
    static _LResult addressesMatch ( Addressing::SUGeneric const& Addr1,
                                     Addressing::SUGeneric const& Addr2 );
    // create a device specification from a device name
    static _LResult createDeviceSpecFromName ( Addressing::SUDevSpec& Spec,
                                               _Char const* pstrDeviceName );
    static _LResult createNameFromDeviceSpec ( _Char
strDeviceName[Addressing::DEVICE_NAME_CHARS_MAX],
                                               Addressing::SUDevSpec const& Spec );
    static UInt32/*interface #*/ extractInterfaceNumber ( _Char const* pstrInterface );
    static _Char const* /*name string*/ formatInterfaceName (
                                              _Char
strInterface[Addressing::IF_NAME_CHARS_MAX],
                                              UInt32 uInterfaceNumber );
    static UInt32/*port #*/ extractPortNumber ( _Char const* pstrPort );
    static _Char const* formatPortName ( _Char strPort[Addressing::PORT_NAME_CHARS_MAX],
                                         UInt32 uPortNumber );
    static UInt32/*protocol #*/ extractProtocolNumber ( _Char const* pstrProtocol );
    static _Char const* formatProtocolName ( _Char
```

```
strProtocol[Addressing::PROT_NAME_CHARS_MAX],
                                        UInt32 uProtocolNumber );
    static bool isLocalHost ( _Char const* pstrHostName );
};
```

Device-level device manager

```
class CUSDM_Impl : public CUSDM {
    // public API -- accessible to the device developer
public:
    // type definitions
    typedef _LResult        FNFINDDEVICE (CUDevice* &pDevice,
                                Addressing::SUDevSpec const& Spec,
                                bool bCreateDevice );
    typedef FNFINDDEVICE*   PFNFINDDEVICE;
    typedef CUStream*       FNCREATEPROTOCOLTRANSLATOR ( void );
    typedef FNCREATEPROTOCOLTRANSLATOR* PFNCREATEPROTOCOLTRANSLATOR;
    typedef _LResult        FNADDRESSCOMPARE (Addressing::SUGeneric const& Addr1,
                                Addressing::SUGeneric const& Addr2 );
    typedef FNADDRESSCOMPARE* PFNADDRESSCOMPARE;
    typedef _LResult        FNCREATEDEVICESPEC (
                                Addressing::SUDevSpec& Spec,
                                _Char const* pstrInterface,
                                _Char const* pstrChannelPort,
                                _Char const* pstrChannelProtocol,
                                _Char const* pstrChannelAddress,
                                _Char const* pstrDevicePort,
                                _Char const* pstrDeviceProtocol,
                                _Char const* pstrDeviceAddress );
    typedef FNCREATEDEVICESPEC* PFNCREATEDEVICESPEC;
    typedef _LResult        FNEXTRACTDEVICESPEC ( Addressing::SUDevSpec const& Spec,
                                Char strInterface[Addressing::IF_NAME_CHARS_MAX],
                                _Char strChannelPort[Addressing::PORT_NAME_CHARS_MAX],
                                _Char strChannelProtocol[Addressing::PROT_NAME_CHARS_MAX],
                                _Char strChannelAddress[Addressing::HOST_NAME_CHARS_MAX],
                                _Char strDevicePort[Addressing::PORT_NAME_CHARS_MAX],
                                _Char strDeviceProtocol[Addressing::PROT_NAME_CHARS_MAX],
                                _Char strDeviceAddress[Addressing::HOST_NAME_CHARS_MAX] );
    typedef FNEXTRACTDEVICESPEC* PFNEXTRACTDEVICESPEC;
    typedef UInt32/*#*/     FNEXTRACTELEMENTNUMBER (
                                _Char    const* pstrElementName );
    typedef FNEXTRACTELEMENTNUMBER* PFNEXTRACTELEMENTNUMBER;
    typedef _Char const* /*name string*/ FNFORMATELEMENTNAME (
                                _Char* pstrElementName,
                                UInt32 uElementNumber );
    typedef FNFORMATELEMENTNAME* PFNFORMATELEMENTNAME;

    // methods
    // construction/destruction/initialization
    CUSDM_Impl ( UNativeInt uPreAllocatedEntries = DEFAULT_PRE_ALLOCATED_ENTRIES );
    virtual ~CUSDM_Impl();

    // install a device finder into the device manager
    static void installDeviceFinder ( PFNFINDDEVICE pfnFinder );
    // install a protocol translator into the device manager
    static _LResult installProtocolTranslator ( UInt32 uProtocolNumber,
                                        PFNCREATEPROTOCOLTRANSLATOR pfnCreate );
    static _LResult removeProtocolTranslator ( UInt32 uProtocolNumber );
    // install an address comparator into the device manager
    static _LResult installAddressComparison ( PFNADDRESSCOMPARE pfnsCompare );
    static _LResult removeAddressComparison ( PFNADDRESSCOMPARE pfnCompare );
    // install a device name translator into the device manager
    static _LResult installAddressTranslation ( PFNCREATEDEVICESPEC pfnCreate,
                                        PFNEXTRACTDEVICESPEC pfnExtract );
    static _LResult removeAddressTranslation ( PFNCREATEDEVICESPEC pfnCreate,
                                       PFNEXTRACTDEVICESPEC pfnExtract );
    // install an interface name translator into the device manager
    static _LResult installInterfaceTranslation ( PFNEXTRACTELEMENTNUMBER pfnExtractNumber,
                                         PFNFORMATELEMENTNAME pfnFormatName );
```

```
    static _LResult removeInterfaceTranslation ( PFNEXTRACTELEMENTNUMBER pfnExtractNumber,
                                                 PFNFORMATELEMENTNAME pfnFormatName );
    // install a port name translator into the device manager
    static _LResult installPortTranslation ( PFNEXTRACTELEMENTNUMBER pfnExtractNumber,
                                             PFNFORMATELEMENTNAME pfnFormatName );
    static _LResult removePortTranslation ( PFNEXTRACTELEMENTNUMBER pfnExtractNumber,
                                            PFNFORMATELEMENTNAME pfnFormatName );
    // install a protocol name translator into the device manager
    static _LResult installProtocolTranslation ( PFNEXTRACTELEMENTNUMBER pfnExtractNumber,
                                                 PFNFORMATELEMENTNAME pfnFormatName );
    static _LResult removeProtocolTranslation ( PFNEXTRACTELEMENTNUMBER pfnExtractNumber,
                                                PFNFORMATELEMENTNAME pfnFormatName );
    // install a device into the device manager
    static _LResult installDevice ( CUDevice* pDevice,
                                    Addressing::SUDevSpec const& Spec, bool bCopyDevice );
    static _LResult removeDevice ( Addressing::SUDevSpec const& Spec );
    static _LResult removeDevice ( CUDevice* pDevice );
    // install/remove a device name alias into the device manager
    static _LResult installDeviceAlias ( char const* pstrDeviceName, char const* pstrAlias
);
    static _LResult removeDeviceAlias ( char const* pstrAlias );
};
```

7.2 Device APIs

The following headers illustrate the APIs for access to devices.

In general, it is anticipated that devices will be created, opened, and closed through the Device Manager defined in section 3.2.1. However, device objects may be created directly through the device classes.

Base-class for all devices; defines the Control Region, Control, and Alert APIs.

```
class CUDevice {
    // public device API -- available to applications
public:
    // methods
    // construction/destruction/initialization
    CUDevice ( bool bSupportLock = true, bool bSupportAlert = false );
    CUDevice ( UNativeInt uOpenOptions, UNativeInt uAccessOptions,
               Addressing::SUDevSpec const& Spec,
               _Char const* pstrConfigChannel, _Char const* pstrConfigDevice,
               bool bSupportLock = true, bool bSupportAlert = false );
    virtual ~CUDevice();
    // report the type of device (region, stream, neither, or both)
    bool isRegion ( void ) const {
               return ( 0 != ( CAP_REGION_DEVICE & enumerateCapabilities() ) );
               }
    bool isStream ( void ) const {
               return ( 0 != ( CAP_STREAM_DEVICE & enumerateCapabilities() ) );
               }
    // report whether the device supports locking
    inline bool supportsLock ( void ) const {
               return ( _NULL != m_hAccessLock );
               }
    // report whether the device supports alerts
    inline bool supportsAlert ( void ) const {
               return ( _NULL != m_hAlertFlags );
               }
    // report whether we are connected to at least one valid lower-level device
    inline bool isConnected ( void ) const { return ( 0 != connections() ); }
    UNativeInt/*# of connections*/ connections ( void ) const { return m_uConnections; }
    UNativeInt/*connected index*/ connect ( CUDevice& Device, bool bOpenedDevice = false,
                                            bool bPlatformDevice = false );
    UNativeInt/*disconnected index*/ disconnect ( void );
    // make a copy of this device object
    virtual CUDevice* copy ( void ) const;
    static void releaseCopy ( CUDevice& Device, bool bClosing = false );
    // accessors
```

```
    // report what capabilities the device supports
    UNativeInt/*capabilities flags*/ enumerateCapabilities ( void ) const;
    // report what access types the device supports
    UNativeInt/*access flags*/ enumerateAccess ( void ) const;
    // report the device element size
    UNativeInt/*bytes*/ getElementSize ( void ) const;
    // report the number of active device users
    inline UNativeInt connectedUsers ( void ) const { return users(); }
    // report the current channel and device addresses
    inline Addressing::SUGeneric const& getChannelAddress ( void ) const {
                                                    return channelAddress();
                                                    }
    inline Addressing::SUGeneric const& getDeviceAddress ( void ) const {
                                                    return deviceAddress();
                                                    }
    // operations
    // open/close a connection to a device
    void open ( UNativeInt uOpenOptions, UNativeInt uAccessOptions,
                Addressing::SUDevSpec const& Spec,
                _Char const* pstrConfigChannel, _Char const* pstrConfigDevice );
    static void close ( CUDevice& Device );
    // read from/write to the control region
    UInt32/*# of elements in control region*/ getControlWordCount ( void ) const {
                                                      return controlRegionWords();
                                                      }
    UInt32/*# ofelements read*/ readControl ( UInt32 uTimeout_ms, UInt32 uStartOffset,
                                              CUTypedBuffer<UInt32>& Data,
                                              UInt32 uRequestedElements = 0 );
    UInt32/*# of elements written*/ writeControl ( UInt32 uTimeout_ms,
                                                   UInt32 uStartOffset,
                                                   CUTypedBuffer<UInt32>& Data );
    inline UInt32/*# ofelements read*/ readControl ( UInt32 uTimeout_ms,
                                                     UInt32 uStartOffset,
                                                     CUTypedBuffer<SInt32>& Data,
                                                     UInt32 uRequestedElements = 0 ) {
                                                           readControl (uTimeout_ms,
                                                           uStartOffset,
                                                           *(CUTypedBuffer<UInt32>*)&Data,
                                                           uRequestedElements );
                                                     }
    inline UInt32/*# of elements written*/ writeControl ( UInt32 uTimeout_ms,
                                                          UInt32 uStartOffset,
                                                          CUTypedBuffer<SInt32>& Data ) {
                                                          return writeControl (
                                                           uTimeout_ms, uStartOffset,
                                                           *(CUTypedBuffer<UInt32>*)&Data );
                                                          }
    // perform a control transaction through the control API
    UInt32/*# of bytes consumed*/ control ( UInt32 uTimeout_ms, SNativeInt nControlID,
                                            CUBuffer& SubmittedData,
                                            CUBuffer& ReturnedData );
    // wait for an alert
    // check for one or more alert flags; returns immediately
    UInt32/*flags accepted*/ testAlert ( UInt32 uFlagsRequested,
                                         bool bTestForAll = false );
    // wait for one or more alert flags; blocks until the flags are asserted
    UInt32/*flags accepted*/ waitAlert ( UInt32 uTimeout_ms,
                                         UInt32 uFlagsRequested,
                                         bool bTestForAll = false );
    // manage device access controls (application-level!)
    bool lockAccess ( UInt32 uTimeout_ms );
    void unlockAccess ( void );
    // general device address matching
    static _LResult addressesMatch ( Addressing::SUGeneric const& Addr1,
                                     Addressing::SUGeneric const& Addr2 );
};
```

Base-class for stream-oriented devices; defines the stream access API

```
class CUStream : public CUDevice {
    // public API -- available to applications
public:
    // enumerations
    enum EUPosRef {
        REF_FROM_START,               // start of stream
        REF_FROM_CURRENT,               // current location in stream
        REF_FROM_END                   // end of stream
    };
```

```
    // type definitions
    typedef SInt64              POSITION;
    typedef POSITION*           PPOSITION;
    typedef enum EUPosRef       POSITIONREFERENCE;
    typedef POSITIONREFERENCE*  PPOSITIONREFERENCE;
    // methods
    // construction/destruction/initialization
    CUStream (bool bSupportFormattedIO = true,
              bool bSupportLock = true, bool bSupportAlert = false );
    CUStream ( UNativeInt uOpenOptions, UNativeInt uAccessOptions,
               Addressing::SUDevSpec const& Spec,
               _Char const* pstrConfigChannel = _NULL,
               _Char const* pstrConfigDevice = _NULL,
               bool bSupportFormattedIO = true,
               bool bSupportLock = true, bool bSupportAlert = false );
    virtual ~CUStream();
    virtual CUDevice* copy ( void ) const;
    bool supportsFormattedIO ( void ) const { return (_NULL != m_pstrWriteFormatBuffer); }
    // manage the transmit and receive buffers
    void getBufferSize ( UInt32& uRXBufferBytes, UInt32 uTXBufferBytes );
    void setBufferSize ( UInt32 uRXBufferBytes, UInt32 uTXBufferBytes );
    // operations
    // move data through a pre-addressed channel
    UInt32/*# of elements read*/ read ( UInt32 uTimeout_ms,
                                        CUBuffer& Data, UInt32 uRequestedElements = 0 );
    UInt32/*# of elements written*/ write ( UInt32 uTimeout_ms,
                                            CUBuffer& Data );
    UInt32/*# of elements written*/ transact ( UInt32 uTimeout_ms,
                                               CUBuffer& SubmittedData,
                                               CUBuffer& ReturnedData,
                                               UInt32 uRequestedElements = 0 );
    // move data through an addressable channel
    UInt32/*# of elements read*/ readFrom ( UInt32 uTimeout_ms,
                                            Addressing::SUGeneric& Address,
                                            CUBuffer& Data, UInt32 uRequestedElements = 0 );
    UInt32/*# of elements written*/ writeTo ( UInt32 uTimeout_ms,
                                              Addressing::SUGeneric const& Address,
                                              CUBuffer& Data );
    UInt32/*# of elements written*/ transactToFrom ( UInt32 uTimeout_ms,
                                                     Addressing::SUGeneric const& Address,
                                                     CUBuffer& SubmittedData,
                                                     CUBuffer& ReturnedData,
                                                     UInt32 uRequestedElements = 0 );
    // move the access position (pre-addressed channels only!)
    POSITION/*current position*/ getPosition ( void );
    POSITION/*new position*/ setPosition ( POSITION nInterval,
                                           POSITIONREFERENCE eFromReference );
    // implementation
    static _LResult addressesMatch ( Addressing::SUGeneric const& Addr1,
                                     Addressing::SUGeneric const& Addr2 );

};
```

Base-class for region-oriented devices; defines the region access API

```
class CURegion : public CUDevice {
    // public API -- available to applications
public:
    // methods
    // construction/destruction/initialization
    CURegion ( UInt32 uIndexLast = 0, UInt32 uFirstIndex = 1,
               bool bSupportLock = true, bool bSupportAlert = false );
    CURegion ( UNativeInt uOpenOptions, UNativeInt uAccessOptions,
               Addressing::SUDevSpec const& Spec,
               _Char const* pstrConfigChannel = _NULL,
               _Char const* pstrConfigDevice = _NULL,
               bool bSupportLock = true, bool bSupportAlert = false );
    virtual ~CURegion();
    virtual CUDevice* copy ( void ) const;
    // accessors
    // number of element addresses within the region
    inline UInt32 getElementCount ( void ) const {
            return ((firstIndex() > lastIndex()) ? 0 : (lastIndex() - firstIndex() + 1) );
           }
    // first and last valid element addresses
    inline UInt32 getFirstOffset ( void ) const { return firstIndex(); }
    inline UInt32 getLastOffset ( void ) const { return lastIndex(); }
    // single register access functions
```

```
    CUElement read ( UInt32 uTimeout_ms, UInt32 uAtOffset );
    void write ( UInt32 uTimeout_ms, UInt32 uAtOffset, CUElement& Datum );
    void update ( UInt32 uTimeout_ms, UInt32 uAtOffset, CUElement& ClearMask,
                  CUElement& SetMask );
    // array access functions
    UInt32/*# of elements read*/ readArray ( UInt32 uTimeout_ms, UInt32 uStartOffset,
                                             CUBuffer& Data,
                                             UInt32 uRequestedElements = 0 );
    UInt32/*# of elements written*/ writeArray ( UInt32 uTimeout_ms,
                                                 UInt32 uStartOffset, CUBuffer& Data );
    UInt32/*# of elements written*/ transactArray ( UInt32 uTimeout_ms,
                                                    UInt32 uStartOffset,
                                                    CUBuffer& ReturnedData,
                                                    UInt32 uRequestedElements,
                                                    CUBuffer& SubmittedData );
    // block access functions
    UInt32/*# of elements read*/ readBlock ( UInt32 uTimeout_ms, UInt32 uAtOffset,
                                             CUBuffer& Data,
                                             UInt32 uRequestedElements = 0 );
    UInt32/*# of elements written*/ writeBlock ( UInt32 uTimeout_ms, UInt32 uAtOffset,
                                                 CUBuffer& Data );
    UInt32/*# of elements written*/ transactBlock ( UInt32 uTimeout_ms, UInt32 uAtOffset,
                                                    CUBuffer& ReturnedData,
                                                    UInt32 uRequestedElements,
                                                    CUBuffer& SubmittedData );
    static _LResult addressesMatch ( Addressing::SUGeneric const& Addr1,
                                     Addressing::SUGeneric const& Addr2 );
};


class CUBlockMemory {
    // public API -- available to applications
public:
    // methods
    // construction/destruction/initialization
    CUBlockMemory ( CURegion& Device, UInt32 uAddressOffset, UInt32 uDataOffset,
                    bool bDeviceSupportsAutoSequence = true, bool bUseLock = false );
    ~CUBlockMemory();
    // accessors
    inline UNativeInt/*bytes*/ getElementSize ( void ) const {
                                     return m_pDevice->getElementSize();
                                     }
    // operations
    // address management
    void setAddress ( UInt32 uTimeout_ms, CUElement& Address );
    CUElement getAddress ( UInt32 uTimeout_ms );
    // memory access
    UInt32/*# of elements read*/ read ( UInt32 uTimeout_ms, CUElement& StartAddress,
                                        CUBuffer& Data, UInt32 uRequestedElements = 0 );
    UInt32/*# of elements read*/ readNext ( UInt32 uTimeout_ms, CUBuffer& Data,
                                            UInt32 uRequestedElements = 0 );
    UInt32/*# of elements written*/ write ( UInt32 uTimeout_ms, CUElement& StartAddress,
                                            CUBuffer& Data );
    UInt32/*# of elements written*/ writeNext ( UInt32 uTimeout_ms, CUBuffer& Data );
};
```

7.3 SDTL API

The following headers define the API adapting the SDM to the underlying platform through the Standard Device Translation Layer (SDTL).

In addition to the API provided specifically for device access, the SDTL also provides translations for interlocks and thread signaling.

Device access class

```
class CUSDTL {
    // public APIs
public:
    // methods
    // construction/destruction/initialization
```

```
    CUSDTL ( void ) : m_pInstalledDeviceLock ( _NULL ), m_pInstalledDeviceList ( _NULL )
                                                                        { initialize(); }
    ~CUSDTL() { cleanup(); }
    static _LResult initializeDevices ( CUSDM_Impl& SDM/*device manager*/ );
    static void cleanupDevices ( CUSDM_Impl& SDM/*device manager*/ );
    // operations
    static _LResult deviceExists ( Addressing::SUDevSpec const& Spec );
    static CUDevice* createAndOpen ( UNativeInt uOpenOptions, UNativeInt uAccessOptions,
                                     Addressing::SUDevSpec const& Spec,
                                     _Char const* pstrConfigChannel,
                                     _Char const* pstrConfigDevice );
    static CUSDM_Impl::FNFINDDEVICE findDevice;
    static _LResult installDevice ( CUDevice* pDevice, Addressing::SUDevSpec const& Spec );
    static _LResult uninstallDevice ( Addressing::SUDevSpec const& Spec );
    static _LResult findAliasedDeviceName (
                                     char strDeviceName[Addressing::DEVICE_NAME_CHARS_MAX],
                                     char const* pstrAlias );
    static _LResult installDeviceAlias ( char const* pstrDeviceName,
                                          char const* pstrAlias );
    static _LResult removeDeviceAlias (char const* pstrAlias );
    // implementation
    static CUSDM_Impl::FNADDRESSCOMPARE addressesMatch;
    static CUSDM_Impl::FNCREATEDEVICESPEC createDeviceSpecFromElements;
    static CUSDM_Impl::FNEXTRACTDEVICESPEC extractElementsFromDeviceSpec;
    static UInt32/*interface #*/ extractInterfaceNumber ( _Char const* pstrInterface );
    static _Char const* formatInterfaceName (
                                     _Char strInterface[Addressing::IF_NAME_CHARS_MAX],
                                     UInt32 uInterfaceNumber );
    static UInt32/*port #*/ extractPortNumber ( char const* pstrPort );
    static _Char const* formatPortName ( _Char strPort[Addressing::PORT_NAME_CHARS_MAX],
                                         UInt32 uPortNumber );
    static UInt32/*protocol #*/ extractProtocolNumber ( _Char const* pstrProtocol );
    static _Char const* formatProtocolName (
                                     _Char strProtocol[Addressing::PROT_NAME_CHARS_MAX],
                                     UInt32 uProtocolNumber );
    static bool isLocalHost ( char const* pstrHostName );
};
```

Interlock and Signal classes

```
class CUMutex {
    // public API
public:
    // methods
    // construction/destruction/initialization
    CUMutex ( void ) { initialize(); }
    ~CUMutex() { cleanup(); }
    // operations
    bool request ( UInt32 uTimeout_ms );
    void release ( void );
};

class CUInterlock {
    // public API
public:
    // methods
    // construction/destruction/initialization
    CUInterlock ( void ) { initialize(); }
    ~CUInterlock() { cleanup(); }
    // operations
    bool requestForRead ( UInt32 uTimeout_ms );
    void releaseForRead ( void );
    bool requestForWrite ( UInt32 uTimeout_ms );
    void releaseForWrite ( void );
};

class CUFlagList {
    // public API
public:
    // methods
    // construction/destruction/initialization
    CUFlagList ( void ) { initialize(); }
    ~CUFlagList() { cleanup(); }
    // operations
    UInt32/*flags accepted*/ test ( UInt32 uFlagsRequested,
                                    bool bTestForAll = false, bool bAutoClear = true );
    UInt32/*flags accepted*/ wait ( UInt32 uTimeout_ms, UInt32 uFlagsRequested,
                                    bool bWaitForAll = false, bool bAutoClear = true );
    void set ( UInt32 uFlagsToSet );
```

```
    void clear ( UInt32 uFlagsToClear );
};
```